A General Approach to Define Binders using Matching Logic

XIAOHONG CHEN, University of Illinois at Urbana-Champaign, USA
GRIGORE ROŞU, University of Illinois at Urbana-Champaign, USA and Runtime Verification Inc., USA

We propose a novel definition of binders using matching logic, where the binding behavior of object-level binders is directly inherited from the built-in $\exists$ binder of matching logic. We show that the behavior of binders in various logical systems such as λ-calculus, System F, π-calculus, pure type systems, can be axiomatically defined in matching logic as notations and logical theories. We show the correctness of our definitions by proving conservative extension theorems, which state that a sequent/judgment is provable in the original system if and only if it is provable in matching logic, in the corresponding theory. Our matching logic definition of binders also yields *models* to all binders, which are deductively complete with respect to formal reasoning in the original systems. For λ-calculus, we further show that the yielded models are representationally complete, a desired property that is not enjoyed by many existing λ-calculus semantics. This work is part of a larger effort to develop a logical foundation for the programming language semantics framework $\mathbb{K}$ (http://kframework.org).

CCS Concepts: • **Theory of computation** → **Logic**; *Lambda calculus.*

Additional Key Words and Phrases: binders, matching logic, conservative extension, completeness

ACM Reference Format:
Xiaohong Chen and Grigore Roşu. 2020. A General Approach to Define Binders using Matching Logic. *Proc. ACM Program. Lang.* 4, ICFP, Article 88 (August 2020), 32 pages. https://doi.org/10.1145/3408970

1 INTRODUCTION

In this paper, we propose a novel definition of binders using *matching logic* [Chen and Roşu 2019; Roşu 2017], where the binding behavior of object-level binders is directly inherited from the built-in $\exists$ binder of matching logic. An appealing aspect of our definition is that it automatically yields *models* to all binders. Therefore, it is interesting and motivating to define a logical system that features binding in matching logic, because it allows us to study the resulting model theory and properties, in addition to the proof theory. We define λ-calculus [Church 1941], System F [Girard 1972; Reynolds 1974], pure type systems [Barendregt 1993], and π-calculus [Milner et al. 1992] in matching logic as *theories* and prove the correctness of definitions as *conservative extension theorems* (Theorems 36 and 49). We also show that the models that matching logic yields for these theories are *deductively complete* with respect to formal reasoning in each of the respective systems (Sections 7 and 9.2). For λ-calculus, we show that the corresponding matching logic models are also *representationally complete* for all λ-theories, a desired property that is not known to hold for many existing λ-calculus semantics [Berry 1978; Bucciarelli and Salibra 2004; Engeler 1981; Girard 1986; Krivine 1993; Plotkin 1972; Schellinx 1991; Scott 1972, 1975a,b] (see discussion in Section 8.2.2).

Authors' addresses: Xiaohong Chen, xc3@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, Illinois, USA, 61801; Grigore Roşu, grosu@illinois.edu, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, Illinois, USA, 61801, Runtime Verification Inc., 102 E Main St #500, Urbana, Illinois, USA, 61801.

2475-1421/2020/8-ART88
https://doi.org/10.1145/3408970

We use λ-calculus as an example to illustrate our definition of binders in matching logic. We define λ-abstraction, $\lambda x.\, e$, as the following matching logic formula (called *pattern*; see Definition 2):

$$\lambda x.\, e \equiv \mathsf{lambda}\,(\mathsf{intension}\,\exists x\colon \mathit{Var}.\,\langle x, e\rangle) \tag{1}$$

Intuitively, $\langle x, e\rangle$ builds an argument-value pair; $\exists$ is the built-in binder in matching logic that thus creates the binding of x to e; $\exists x\colon \mathit{Var}.\,\langle x, e\rangle$ builds the set-theoretic union of all argument-value pairs $\langle x, e\rangle$, as x ranging over all variables of sort *Var*; this union set is called the *graph* of the function $x \mapsto e$, which is then "packed" by the operator intension into an object and passed to lambda. Finally, lambda decodes/retracts the packed object and returns the intended interpretation of $\lambda x.\, e$. Binders in the other systems may require different retracts other than lambda, but all take the same packed object as argument, which for convenience we write $[x\colon \mathit{Var}]\; e \equiv \mathsf{intension}\,\exists x\colon \mathit{Var}.\,\langle x, e\rangle$.

The main goal of this paper is to show that the matching logic definition of binders as illustrated in Eq. (1), is mathematically interesting and can serve as a foundation of binders in language frameworks. In Section 2, we start with a discussion on the major existing approaches to dealing with binders and we compare them with our approach. Then we make the following contributions:

- We propose a novel functional variant of matching logic that is more suitable to capture binders, and we comprehensively study its model theory (Section 3); we demonstrate the expressiveness of this functional variant of matching logic by defining several important mathematical instruments (such as equality and sorts) as theories and notations (Section 4);
- We define λ-calculus (Section 5) as a theory in matching logic (Section 6), as an illustrative case study. Then we prove the conservative extension theorem for λ-calculus and show that matching logic yields complete models, in terms of deduction, for λ-calculus (Sections 7-8). We also discuss the representability problem in λ-calculus and show that matching logic yields models that are representationally complete, in Section 8.2.2;
- We generalize our method to arbitrary binders (Section 9).

Finally, we conclude the paper with future work in Sections 10-11.

This paper marks an important step towards formalizing the logical foundation of the $\mathbb{K}$ semantic framework (http://kframework.org), which has been used to define complete formal semantics of several real-world languages [Bogdănaş and Roşu 2015; Dasgupta et al. 2019; Hathhorn et al. 2015; Hildenbrandt et al. 2018; Park et al. 2015]. Prior attempts have been made to propose a logical foundation of $\mathbb{K}$ using formalisms like rewriting logic [Meseguer and Roşu 2013; Roşu and Şerbănuţă 2010] and graph rewriting [Şerbănuţă and Roşu 2012], but none of them were satisfactory. Recently, matching logic has been proposed as an alternative [Chen and Roşu 2019; Roşu 2017]. The main idea is that arbitrarily complex programming languages and calculi defined in $\mathbb{K}$ become theories in matching logic, and all the tools offered by $\mathbb{K}$, such as execution engines, symbolic reasoning, and even full functional correctness verification of program or language properties, become proof search heuristics in matching logic, which admits a small proof system and thus a small trust base. Several important logical systems have been defined in matching logic, but none where binders play a major role, like λ-calculus or type systems. On the other hand, the current $\mathbb{K}$ implementations already provide built-in support for user-defined binders of certain restricted forms (Remark 43). Thus, this paper fills this gap by giving the theoretical results about how to define logical systems that feature binders in matching logic and thus in $\mathbb{K}$, without any foundational compromise.

All proof details can be found in the companion technical report [Chen and Roşu 2020].

2 RELATED WORK: EXISTING APPROACHES TO DEFINING BINDERS

We discuss some existing approaches to defining binders and compare them with our approach using matching logic. These approaches include: (1) *de Bruijn* techniques [de Bruijn 1972], which give α-equivalent terms identical encodings; (2) *combinators* [Church 1941], which translate terms

with binders to binder-free combinator terms; (3) *nominal logic* [Pitts 2003], which uses first-order logic (FOL) to axiomatize *name-swapping* and *freshness*, and uses them to axiomatize object-level binding; (4) *higher-order abstract syntax* [Pfenning and Elliott 1988] (abbreviated HOAS), which uses fixed binders in the meta-language, often a variant of typed λ-calculus, to define arbitrary binders in the object-level systems; (5) *explicit substitution* [Abadi et al. 1991], which uses customized calculi where the meta-level operation of capture-free substitution is incarnated in an object-level operation as part of the calculi; (6) *term-generic logic* [Popescu and Roşu 2015] (abbreviated TGL), which is a FOL variant parametric in a generic term set, defined axiomatically and not constructively, which can be *instantiated* by a concrete binder syntax. We discuss how these approaches handle binders and binding behavior using the following λ-expression as an example (a closed expression with distinct bound variables, which requires α-renaming during reduction to avoid variable-capture):

$$(\lambda z.\,(zz))(\lambda x.\,\lambda y.\,(xy)) \tag{$\dagger$}$$

De Bruijn encodings eliminate bound variables by replacing them with indexes that denote the number of (nested) binders that are in scope between them and their corresponding binders.[1] For example, the de Bruijn encoding of ($\dagger$) is $(\lambda(11))(\lambda\lambda(21))$, where 1 means that it is bound by the closest binder and 2 means that it is bound by the second closest binder. Bound variables are eliminated so α-equivalent expressions have the same de Bruijn encoding. However, substitution requires index shifting, to adjust the indexes. De Bruijn techniques are used as the internal representations of terms in several theorem provers, but the encoding is not human readable, implementations are often tricky to get right, and efficiency problems can still appear on large terms.

Combinators translate binders to binder-free terms, which are built with constants like k and s, and application. This translation is called *abstraction elimination*, and can be implemented using term rewriting [Klop 1993]. It may cause exponential growth in the translated term size. Reduction of combinatory terms is done using equations like $kxy = x$ and $sxyz = (xz)(yz)$ regarded as rewrite rules. Combinatory terms are not human readable; for example, (one of) the equivalent combinator term of ($\dagger$) is $s(skk)(skk)s(s(ks)(s(kk)(skk)))(k(skk))$. Using combinators, the binding behavior of λ is captured *implicitly* through abstraction elimination.

Nominal logic refers to a family of FOL theories whose signatures contain a *name-swapping* operation $(x\ y)\cdot e$ that swaps all (free and bound) occurrences of x and y in e, and a *freshness* predicate $x \# e$ stating that x has no free occurrences in e. The notions of α-equivalence and capture-free substitution are then axiomatized using additional FOL axioms on top of the axioms of name-swapping and freshness. As an example, the following is an axiom in [Pitts 2003, Appendix A.3] that states that swapping two fresh names that do not occur free in a term has not effect:

$$\text{(F1)}\quad \forall x{:}\mathit{Var}.\,\forall y{:}\mathit{Var}.\,\forall e{:}\mathit{Exp}.\,x \# e \wedge y \# e \rightarrow (x\ y)\cdot e = e$$

where *Var* and *Exp* are the sorts of variables (also called atoms) and expressions, respectively. Nominal logic also defines a new sort $[\mathit{Var}]\mathit{Exp}$ and a FOL binary function $\_.\_ : \mathit{Var} \times \mathit{Exp} \rightarrow [\mathit{Var}]\mathit{Exp}$ for binding, whose properties such as α-equivalence are axiomatized. Then, β-reduction in λ-calculus, e.g., can be defined in the following way [Pitts 2013, pp. 251, Eq. (12.17)]:

$$(\beta \text{ in Nominal Logic})\qquad \forall x{:}\mathit{Var}.\,\forall e{:}\mathit{Exp}.\,\forall e'{:}\mathit{Exp}.\,\mathit{app}(\mathit{lam}(x.e), e') = \mathit{subst}((x.e), e')$$

where $\mathit{subst}(\_,\_)$ is a binary function defined by four axioms (see [Pitts 2003, pp. 8]), in accordance to the four possible forms that e can take (i.e., the variable x; a variable distinct from x; application; or abstraction). E.g., the following is the substitution axiom for abstraction [Pitts 2013, Eq. (12.20)]:

$$\forall x{:}\mathit{Var}.\,\forall y{:}\mathit{Var}.\,\forall e{:}\mathit{Exp}.\,\forall e'{:}\mathit{Exp}.\,y \# e' \rightarrow \mathit{subst}(x.\,\mathit{lam}(y.\,e), e') = \mathit{lam}(y.\,\mathit{subst}(x.\,e, e'))$$

[1] Other de Bruijn encodings count the binders from the top of the terms.

Besides nominal logic and its metatheory [Cheney 2006, 2014; Gabbay and Cheney 2004], there is a wider range of research on *nominal techniques* in general, including studies on using Fraenkel-Mostowski sets [Gabbay and Pitts 1999], nominal sets [Pitts 2005] or similar set-theoretic structures [Urban 2008] as well as category-theoretic notions [Gabbay and Gabbay 2017] to formalize and reason about binders and operations on them, and have resulted in practical implementations that support complex recursive and inductive reasoning over terms with bindings as well as algorithms for unification [Ayala-Rincón et al. 2018] and narrowing [Ayala-Rincón et al. 2016]. These nominal approaches deal with variable names and bindings *directly*, treat variable names as normal data that can be manipulated, quantified, and reasoned about, and give explicit definitions to operations such as free variables and capture-free substitution (via name-swapping and freshness).

Nominal approaches can be directly exploited in matching logic, because FOL is a methodological fragment of matching logic. Indeed, [Roşu 2017, Section 7] shows how matching logic *symbols* (see Definition 2) can be used to uniformly represent both FOL predicates and FOL functions (Sections 3.2.1 and 3.2.2), in a way where FOL theories become matching logic theories as are, without any translations. Therefore, nominal logic variants can be defined as theories in matching logic straightforwardly, via the FOL capability of matching logic. Future research shall reveal more direct methods that capture the essence of nominal techniques (e.g., nominal sets) within matching logic, without going through FOL. In this paper, however, we explore a different, more HOAS-style treatment of binders using matching logic, where the built-in $\exists$ binder is used to define binders in object-languages (explained below and revisited in Remark 1).

Higher-order abstract syntax (HOAS) is a design pattern where some expressive higher-order calculus, usually one of the variants of typed λ-calculus [Felty and Momigliano 2012; Gacek et al. 2012; Harper et al. 1993; McDowell and Miller 2002; Paulson 1989; Pfenning and Elliott 1988] or second-order equational logic [Felty and Momigliano 2012; Fiore and Mahmoud 2010], is used as a foundation to define object-level binders. As an example, we show (part of) the HOAS-style definition of (untyped) λ-calculus in the Twelf system [Pfenning and Schürmann 1999]:

```
exp : type.                    // the type for λ-expressions
app : exp -> exp -> exp.       // application is defined as a constant of a function type
lam : (exp -> exp) -> exp.     // lambda is defined as a constant of a function type whose
                               //    argument also has a function type; e.g., the encoding of (†)
                               //    is app (lam ([z] (app z z))) (lam ([x] lam ([y] (app x y))))
red : exp -> exp -> type.      // reduction relation (its type result makes it a binary predicate)
red-beta : red (app (lam ([x] (F x))) E) (F E).   // β-reduction, discussed below
```

where [x] _ is the built-in binder of (the HOAS variant underlying) Twelf; E is a variable of type exp; F is a variable of the function type exp -> exp; and (F x) is the (metalevel) application of F to x. Higher-order matching is needed when red-beta is applied, and the internal substitution mechanism of Twelf is triggered when F is applied to E. The binding behavior of λ is obtained from the binding behavior of the built-in binder [x] _, via a constant lam; specifically, $\lambda x.e$ is encoded as lam ([x] e). Object-level substitution is avoided, but clearly this is not how β-reduction is usually defined (for the usual definition, see (β, Reduction) below). Application in λ-calculus is defined by a simple desugaring to the builtin application, using a different constant app; that is, $e_1\ e_2$ is defined as app $e_1\ e_2$ (rather than $e_1\ e_2$). Thus, the definition needs to be justified by proving *adequacy theorems* that establish a bijection between the expressions and formal proofs of λ-calculus, and the HOAS terms and type derivations, which is a tedious and nontrivial task [Cheney et al. 2012].

Explicit substitution turns the implicit meta-level substitution operation into more explicit and atomic steps, in order to provide a better understanding of the operational semantics and execution models of higher-order calculi (see [Kesner 2009, pp. 1–2]; see also [Bloo 1997, pp. 4] for historical

remarks). By doing so, it bridges the gap between higher-order formalisms and their implementations, and has resulted in several practical tools. For example, [Stehr 2000] proposes a calculus for explicit substitution whose implementation allows us to define executable formal representations of many logical systems featuring binders with a close-to-zero representational distance.

Term-generic logic (TGL) is a FOL variant, where the set of terms T is generic and given as a *parameter* that exports two operations—free variables and capture-free substitution—satisfying certain properties [Popescu and Roşu 2015, Definition 2.1]. TGL formulas are then defined constructively as in FOL, from predicates $\pi(e_1, \ldots, e_n)$ and equations $e_1 = e_2$, to compound formulas built using $\wedge$, $\neg$, and $\exists$, with the important exception that $e_1, \ldots, e_n$ are not constructive terms like in FOL, but generic terms in T. In the case of λ-calculus, the set of λ-expressions Λ can be proved to satisfy the definition of a generic term set in TGL, so we can *instantiate* TGL by Λ. The binding behavior of λ is inherited automatically, through the T instance. The metalevel of λ-calculus can be defined by TGL axioms. For example, β-reduction is captured either as an equation or as a relation:

$$(\beta, \textsc{Equation}) \quad (\lambda x.\, e)\, e' = e'[e/x] \qquad\qquad (\beta, \textsc{Reduction}) \quad \mathit{reduces}\big((\lambda x.\, e)\, e', e'[e/x]\big)$$

where *reduces* is a binary predicate; $(\lambda x.\, e)\, e', e'[e/x] \in \Lambda$ are generic terms (schemas) that represent all the concrete instances. TGL has been used to define various systems featuring bindings. In this paper, we use TGL as an intermediate to capture other systems with binders within matching logic.

Our Approach Using Matching Logic. Our matching logic encoding of binders is inspired by the key observation that the meaning of a term with binders, say $\lambda x.\, e$, can be given on top of the function that maps x to e, which can be encoded as its *graph*: the set of argument-value pairs $\bigcup_x \{(x, e)\}$. This set is then packed as an object and passed to a retraction function lambda that retracts/decodes the intended meaning of the term. We recall the encoding of $\lambda x.\, e$ in Eq. (1) below:

$$\lambda x.\, e \equiv \mathsf{lambda}\ (\mathsf{intension}\ \exists x\colon \mathit{Var}.\ \langle x, e\rangle)$$

Note that by introducing the following notation

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

the encoding of $\lambda x.e$ becomes lambda $([x\colon \mathit{Var}]\ e)$, where *Var* is the sort for λ-calculus variables and thus a subsort of *Exp* for expressions (see Section 6). Note that our matching logic encoding of binders is reminiscent of both the nominal encoding $\mathit{lam}(x.e)$ and the HOAS encoding lam ([x] E).

An important aspect of our approach is that it yields *models*. We will give a comprehensive study on the *model theory* of matching logic, by which every theory is associated with default models that can be used to give *semantic interpretations* of all matching logic formulas (called *patterns*) of that theory. In particular, the matching logic theory of λ-calculus will also yield a precise and insightful description of how $\lambda x.\, e$ is interpreted (semantically) in matching logic models.

Models are insightful. They help us understand a logical system better, from a different angle. It is not unusual that more than one notion or class of models are proposed for one logic, because each has its unique merit in helping us understand the logic from a certain perspective. Since matching logic has a built-in notion of models, by defining a logical system as a matching logic theory we can immediately study its resulting model theory and properties. For example, in Section 8.2.2, we show how by defining λ-calculus in matching logic, we obtain a new semantics of λ-calculus that is representationally complete for all λ-theories.

The importance of models has also been recognized by several HOAS approaches. For example, [Fiore et al. 1999] proposes presheaf models of variable binding in a second-order syntax of binding terms, where the initial model is used to define recursive/inductive operations; this work also yields an explicit connection to the scope-safe variant of De Bruijn approaches. [Fiore and Hur 2010; Fiore and Mahmoud 2010] propose for the same binding syntax yet another category of models, called second-order universal algebras, together with completeness and conservative extension results w.r.t. first-order universal algebras; however, the conservative extension w.r.t.

the original logical systems that feature binding and their formal reasoning is not investigated at our knowledge, and not known if it holds. In our work using matching logic, we shall prove the conservative extension for all logical systems that feature binders considered in the paper, but will not cover the topics of inductive reasoning and/or initial models (although a special initial algebra will be discussed in Section 8 for λ-calculus); this topic is left as future work (see Section 10).

As a logic that features binding, we expect matching logic to be definable within HOAS. Such a definition will likely work fine in capturing the syntax and binding behavior of matching logic formulas/patterns as well as its proof theory, but it will not capture the semantics or models of matching logic; see related discussion in Remark 52. In this paper, we will discuss the other direction, that is to capture HOAS by matching logic. We will do that indirectly, by firstly capturing term-generic logic (TGL) and then re-using the existing TGL definitions of HOAS (see [Popescu and Roşu 2013]). This indirect approach has the advantage that we will be able to examine how the very general TGL models are translated and preserved when defined in matching logic.

Remark 1. Dealing with binders has been and still is an active research topic. The variety of proposals and approaches has occasionally caused heated arguments. We conclude this section by reminding the reader that matching logic was designed to serve as a unified logical foundation for the $\mathbb{K}$ framework, which is intended to support all languages and all definitional styles as logical theories. That is, when looked at through the matching logic lenses, the various approaches to binders above become different methodologies for how to define matching logic theories.

3 FUNCTIONAL VARIANT OF MATCHING LOGIC

Matching logic has been recently proposed in its full generality in [Chen and Roşu 2019; Roşu 2017]. In this paper, we will use a variant of matching logic that has a more similar representation to functional programming languages, where the main constructs are function application and constants. Since matching logic is relatively new, we will not assume the reader familiar with it. Therefore, this section has a dual goal: to introduce the reader to the basic intuitions and notations of matching logic, and to propose and present in detail a functional variant of it. Section 3.1 defines its syntax and Section 3.2 its models and semantics. We define matching logic theories in Section 3.3.

3.1 Matching Logic Syntax

Matching logic is parametric in a *signature* that includes *variables* and *constant symbols*:

Definition 2. A *signature* is a tuple $\mathbb{\Sigma} = (EV, SV, \Sigma)$, where $EV \cap SV = \emptyset$ and

(1) EV is a countably infinite set of *element variables* denoted $x, y, \ldots$;
(2) SV is a countably infinite set of *set variables* denoted $X, Y, \ldots$;
(3) Σ is an at most countable set of *(constant) symbols*, or just *symbols*, denoted $\sigma, \sigma_1, \sigma_2, \ldots$.

Matching logic formulas, called $\mathbb{\Sigma}$*-patterns* or simply *patterns*, are inductively defined as follows:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\, \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \tag{2}$$

where $\varphi_1\, \varphi_2$ is called an *application* and is assumed associative to the left; $\exists x.\, \varphi$ is the built-in *binder* in matching logic that binds x within φ. Note that $\exists$ only binds element variables and not set variables. We use $\textsc{Pattern}(\mathbb{\Sigma})$, or simply $\textsc{Pattern}$, to denote the set of all $\mathbb{\Sigma}$-patterns.

Remark 3. The syntax of the original matching logic has sorts and multiary many-sorted operations [Chen and Roşu 2019; Roşu 2017]. Our functional variant syntax in Definition 2 is much simpler: it has no sorts and contains only one binary operation, the application, and constants. Yet, as seen in this paper, this simpler variant has the same expressiveness and reasoning capability.

As a convention, we assume the scope of $\exists$ goes as far as possible to the right, so for example, $\exists x.\, y \to x$ should be understood as $\exists x.\, (y \to x)$. In addition, we assume the standard notions of *free*

free variables:

$$\mathrm{FV}(x) = \{x\} \qquad \mathrm{FV}(X) = \{X\} \qquad \mathrm{FV}(\sigma) = \emptyset \qquad \mathrm{FV}(\varphi_1\,\varphi_2) = \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2)$$
$$\mathrm{FV}(\bot) = \emptyset \qquad \mathrm{FV}(\varphi_1 \to \varphi_2) = \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2) \qquad \mathrm{FV}(\exists x.\, \varphi) = \mathrm{FV}(\varphi) \setminus \{x\}$$

α-renaming:

$$\exists x.\, \varphi \equiv \exists y.\, \varphi[y/x], \text{ for } y \notin \mathrm{FV}(\varphi)$$

capture-free substitution (where y distinct from x and z is fresh):

$$(\exists x.\, \varphi)[\psi/x] \equiv \exists x.\, \varphi \qquad (\exists x.\, \varphi)[\psi/y] \equiv \exists z.\, \varphi[z/x][\psi/y]$$

---

derived constructs defined as syntactic sugar:

$$\neg\varphi \equiv \varphi \to \bot \qquad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$
$$\top \equiv \neg\bot \qquad \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$$

Fig. 1. Above line: standard notions of free variables, α-equivalence, and capture-free substitution for $\exists$ in matching logic. Below line: usual derived constructs defined as syntactic sugar. Standard precedence assumed.

variables $\mathrm{FV}(\varphi) \subseteq EV \cup SV$, *$\alpha$-equivalence* $\varphi_1 \equiv_\alpha \varphi_2$, and *capture-free substitution* $\varphi[\psi/x]$, which are all summarized in Fig. 1. We regard α-equivalent patterns as *syntactically identical* patterns; in other words, $\varphi_1 \equiv_\alpha \varphi_2$ implies that $\varphi_1 \equiv \varphi_2$. A set of common derived constructs are also included in Fig. 1 in the usual way as syntactic sugar, and we assume the standard precedence among them.

The matching logic syntax of patterns given in Eq. (2) is similar to the FOL syntax of terms and formulas, except that we drop the distinction between terms and formulas, and unify them as patterns.[2] Also, we drop the multiary functions/predicates in FOL, and replace them with a set of constant symbols that can be *applied* to other patterns using the built-in application $\varphi_1\,\varphi_2$. This simpler syntax of matching logic makes it easier to develop its metatheory, and yet, as we will show in Section 4, we do not lose any specification or reasoning power, and can still define important and necessary mathematical instruments as theories and notations in matching logic.

By unifying the syntax of terms and formulas, we can *bind variables in terms*, using the built-in matching logic binder $\exists$. A minimal example is $\exists x.\, x$, where x is bound by $\exists x$, so $\mathrm{FV}(\exists x.\, x) = \emptyset$. While $\exists x.\, x$ is a well-formed matching logic pattern, it is neither a well-formed term nor a well-formed formula in FOL. As we will see in Section 6, being able to build terms *and* create bindings over them is what makes our encoding of various binders in matching logic possible, and novel.

3.2 Matching Logic Semantics

Matching logic patterns are interpreted on an underlying carrier set of elements, and each pattern is then interpreted as a *set of elements*, which are those that *match* the pattern. This is called the *pattern matching semantics* of matching logic, and is what inspired the name "matching logic".

Intuitively, the pattern $\bot$ (called *bottom*) is matched by no elements, while $\top$ (called *top*, defined in Fig. 1) 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 7). 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. 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.

Definition 4. Given $\Sigma = (EV, SV, \Sigma)$, a *$\Sigma$-model* (or just *model*) is a tuple $(M, \_\bullet\_, \{\sigma_M\}_{\sigma\in\Sigma})$, where

[2]The syntax of a logic should be in harmony with its semantics. FOL distinguishes terms and formulas because their interpretations are different: terms are interpreted as elements and formulas are interpreted as truth values. As we will see in Section 3.2, the matching logic semantics interprets patterns uniformly to the sets of elements that match them, so there is no need to distinguish terms and formulas. Other such examples include modal logic [Blackburn et al. 2001] (which abandons terms entirely) and separation logic [Reynolds 2002] (which merges the syntax for memory heaps with formulas).

$$\text{(Curry.1)} \quad k = s(s(ks)(s(kk)k))(k(skk))$$
$$\text{(Curry.2)} \quad s = s(s(ks)(s(k(s(ks)))(s(k(s(kk)))s)))(k(k(skk)))$$
$$\text{(Curry.3)} \quad s(s(ks)(s(kk)(s(ks)k)))(kk) = s(kk)$$
$$\text{(Curry.4)} \quad s(ks)(s(kk)) = s(kk)(s(s(ks)(s(kk)(skk)))(k(skk)))$$
$$\text{(Curry.5)} \quad s(k(s(ks)))(s(ks)(s(ks))) = s(s(ks)(s(kk)(s(ks)(s(k(s(ks)))s))))(ks)$$
$$\text{(Meyer-Scott)} \quad \forall x.\, \forall y.\, (\forall z.\, xz = yz) \rightarrow s(k(skk))x = s(k(skk))y$$

Fig. 2. Five axioms of Curry and the Meyer-Scott axiom for λ-models [Barendregt 1984, pp. 94] ($\bullet_A$ is omitted).

(1) M is an underlying carrier set, required to be non-empty ($M \neq \emptyset$);
(2) $\_\bullet\_ : M \times M \rightarrow \mathcal{P}(M)$ is called the *interpretation of application*, where $\mathcal{P}(M)$ is the powerset;
(3) $\sigma_M \subseteq M$ is a subset, called the *interpretation of* σ, defined for every $\sigma \in \Sigma$.

We often use the same letter M to denote the above model and refer to $\mathbb{\Sigma}$ as the *signature of* M.

Let us compare matching logic and FOL, w.r.t. models. Both logics require their models to have nonempty carriers, so they agree on (1). For (3), however, FOL models interpret constants to elements, while matching logic models interpret constants to any carrier subsets. Similarly, for (2), FOL models interpret application (regarded as a binary function) as a function of $M \times M \rightarrow M$ that returns one element, while matching logic models interpret application to a function that returns a set. We use the terminology *functional interpretation* to refer to how FOL interprets functions and terms. Functional interpretation is in harmony with the syntax of FOL terms, which represent elements. Similarly, the *set-theoretic interpretation* of matching logic application and symbols is in harmony with its syntax of patterns, which represent sets of elements.

Note that the FOL functional interpretation can be seen as a special instance of the matching logic set-theoretic interpretation, due to the bijection between an element a and the singleton $\{a\}$: for any set M, the set of all singletons of M is isomorphic to M itself. This justifies our abuse of notation (used often in this paper) in which $\{a\}$ is written as a when there is no confusion. We will use two examples to illustrate how the functional interpretation is a special instance of the set-theoretic interpretation. These examples are also related to the model theory of λ-calculus, so we will re-visit them later; for now, we only use them as examples of matching logic models.

Example 5. Let $(A, \_\bullet_A\_)$ be an *applicative structure* [Barendregt 1984, Definition 5.1.1], where A is a nonempty carrier set and $\_\bullet_A\_ : A \times A \rightarrow A$ is an application function. Let matching logic signature $\mathbb{\Sigma}^{\emptyset}$ contain no symbols. We define a $\mathbb{\Sigma}^{\emptyset}$-model $(M, \_\bullet\_, \{\})$, where $M = A$ and $a \bullet b = \{a \bullet_A b\}$ for all $a, b \in A$. Then, M is isomorphic to A under the bijection between elements and singletons.

Example 6. Let $(A, \_\bullet_A\_, k, s)$ be a *combinatory algebra* [Barendregt 1984, Definition 5.1.7], where $(A, \_\bullet_A\_)$ is an applicative structure and $k, s \in A$ are distinguished elements such that $k \bullet_A a \bullet_A b = a$ and $s \bullet_A a \bullet_A b \bullet_A c = (a \bullet_A c) \bullet_A (b \bullet_A c)$, for all $a, b, c \in A$. A is called a *λ-model* [Barendregt 1984], if it additionally satisfies the five axioms of Curry [Barendregt 1984, Theorem 5.2.5] and the Meyer-Scott axiom [Barendregt 1984, Definition 5.2.7], shown in Fig. 2. Let $\mathbb{\Sigma}^{ks}$ be the matching logic signature $\mathbb{\Sigma}^{ks} = \{\mathrm{k}, \mathrm{s}\}$ and define a $\mathbb{\Sigma}^{ks}$-model $(M, \_\bullet\_, \{\mathrm{k}_M, \mathrm{s}_M\})$, where $M = A$, $\mathrm{k}_M = \{k\}$, $\mathrm{s}_M = \{s\}$, and $a \bullet b = \{a \bullet_A b\}$ for all $a, b \in A$. Then M is isomorphic to A under the element-singleton bijection.

Examples 5 and 6 show that the functional interpretation (of application and constants) is a special instance of the set-theoretic interpretation of matching logic, and that applicative structures, combinatory algebras, and λ-models are special instances of matching logic models. In Section 4, we will show how to enforce functional interpretation in matching logic models, *axiomatically*.

We continue with the semantics of matching logic and define the interpretation of patterns.

Definition 7. Let M be a matching logic model like in Definition 4. We extend the interpretation of application $\_\cdot\_$ *pointwisely*, from over elements to over sets, as $A \cdot B = \bigcup_{a\in A, b\in B} a \cdot b$ for any $A, B \subseteq M$. An *M-valuation* (or simply *valuation*), written $\rho\colon (EV \cup SV) \to M \cup \mathcal{P}(M)$, is a function that maps element variables to elements and set variables to sets, i.e., $\rho(x) \in M$ for $x \in EV$ and $\rho(X) \subseteq M$ for $X \in SV$. It yields a *pattern valuation*, written $|\_|_\rho : \textsc{Pattern} \to \mathcal{P}(M)$, defined as:

(1) $|x|_\rho = \{\rho(x)\}$ for $x \in EV$;
(2) $|X|_\rho = \rho(X)$ for $X \in SV$;
(3) $|\sigma|_\rho = \sigma_M$ for $\sigma \in \Sigma$;
(4) $|\varphi_1\, \varphi_2|_\rho = |\varphi_1|_\rho \cdot |\varphi_2|_\rho$, where $\_\cdot\_$ is pointwisely extended to sets;
(5) $|\bot|_\rho = \emptyset$;
(6) $|\varphi_1 \to \varphi_2|_\rho = M \setminus (|\varphi_1|_\rho \setminus |\varphi_2|_\rho)$, where "$\setminus$" denotes set difference;
(7) $|\exists x.\, \varphi|_\rho = \bigcup_{a\in M} |\varphi|_{\rho[a/x]}$, where $\rho[a/x]$ is the valuation ρ' such that $\rho'(x) = a$, $\rho'(y) = \rho(y)$ for all $y \in EV$ distinct from x, and $\rho'(X) = \rho(X)$ for all $X \in SV$.

Remark 8. The above semantic rules should not be unexpected. Rules (1) and (2) interpret variables according to ρ. Rules (3) and (4) interpret symbols and application according to M. For rules (5)-(7), if we regard $\emptyset$ as "false" and M as "true", then these rules become precisely the FOL semantic rules of bottom, implication, and $\exists$-quantification, respectively.

We can prove that the derived constructs in Fig. 1 have the expected semantics:

Proposition 9. *The following propositions hold:*

(1) $|\neg\varphi|_\rho = M \setminus |\varphi|_\rho$;
(2) $|\varphi_1 \vee \varphi_2|_\rho = |\varphi_1|_\rho \cup |\varphi_2|_\rho$;
(3) $|\varphi_1 \wedge \varphi_2|_\rho = |\varphi_1|_\rho \cap |\varphi_2|_\rho$;
(4) $|\top|_\rho = M$;
(5) $|\varphi_1 \leftrightarrow \varphi_2|_\rho = M \setminus (|\varphi_1|_\rho \triangle |\varphi_2|_\rho)$, *where "$\triangle$" denotes set symmetric difference;*
(6) $|\forall x.\, \varphi|_\rho = \bigcap_{a\in M} |\varphi|_{\rho[a/x]}$.

Proof. We only prove (1) and (6), as the others are similar. For (1), we have $|\neg\varphi|_\rho = |\varphi \to \bot|_\rho = M\setminus(|\varphi|_\rho \setminus |\bot|_\rho) = M\setminus(|\varphi|_\rho \setminus \emptyset) = M\setminus|\varphi|_\rho$. For (6), we have $|\forall x.\, \varphi|_\rho = |\neg\exists x.\, \neg\varphi|_\rho = M\setminus|\exists x.\, \neg\varphi|_\rho = M \setminus \bigcup_{a\in M} |\neg\varphi|_{\rho[a/x]} = M \setminus \bigcup_{a\in M}(M \setminus |\varphi|_{\rho[a/x]}) = M \setminus (M \setminus \bigcap_{a\in M} |\varphi|_{\rho[a/x]}) = \bigcap_{a\in M} |\varphi|_{\rho[a/x]}$. □

Remark 10. Definition 7 and Proposition 9 show that there is a close connection between the matching logic pattern constructs and the set operations in set theory: conjunction corresponds to intersection of two sets; disjunction corresponds to union of two sets; negation corresponds to set complement; top ($\top$) corresponds to the total set; bottom ($\bot$) corresponds to the empty set; $\exists$-quantification corresponds to the (big) union of a collection of sets; and $\forall$-quantification corresponds to the (big) intersection of a collection of sets. This connection to the set-theoretic operations can be useful to understand the intuitive meaning of complex matching logic patterns.

3.2.1 Predicate Patterns. A difference between FOL formulas and matching logic patterns is that the former can only be interpreted as either true or false, while the latter can be interpreted as any subsets of the carrier set. Following up on Remark 8, we identify two special sets, M and $\emptyset$, and use them to represent (logical) true and false, respectively. Obviously, not all patterns are interpreted as M or $\emptyset$. Given a model M, we call φ an *M-predicate*, if $|\varphi|_\rho \in \{\emptyset, M\}$ for all ρ. We call φ a *predicate* (or *predicate pattern*), if it is an M-predicate in all M. Predicate patterns can be built from $\bot$, $\top$, and matching logic logical constructs, e.g., $\forall x.\, (\sigma\, x) \wedge \neg(\sigma\, x)$. More interesting patterns can be built from symbols and application. For example, $\sigma\, x_1 \cdots x_n$ is a predicate pattern if the underlying *matching logic theory* (discussed in Section 3.3) enforces the models to interpret σ as a predicate

(i.e., either $\emptyset$ or M). We will see more predicate patterns in Section 4 and throughout the paper. Roughly speaking, predicate patterns are the matching logic counterparts of FOL formulas. They make "statements", and can take only two possible values: M if the statements are facts, and $\emptyset$ if the statements are not facts. Note that except the application, all matching logic constructs (primitive or derived) preserve the predicate-ness of patterns. We can then use application to build FOL-style predicates, and this way regard predicate logic as a methodological fragment of matching logic.

3.2.2 Functional Patterns. Examples 5 and 6 emphasized that any set M is isomorphic to the set of singletons of M, and that functional interpretation is a special instance of set-theoretic interpretation. Formally, given M, we call φ an *M-functional pattern* if $|\varphi|_\rho$ is a singleton for all ρ. We call φ a *functional pattern*, if it is an M-functional pattern for all M. Roughly speaking, functional patterns are the matching logic counterparts of FOL terms. A functional pattern denotes exactly one element; e.g., x is the simplest functional pattern. More interesting functional patterns can be built by symbols and application; e.g., $\sigma\, x_1\, \cdots\, x_n$ is a function pattern if the underlying matching logic theory (discussed in Section 3.3) enforces the models to interpret σ as a function. We will show many examples of functional patterns in Section 4 and throughout the paper.

3.3 Matching Logic Theories

Examples 5 and 6 show that we sometimes want to consider only a subclass of matching logic models, those that satisfy certain properties. This can be achieved by defining a matching logic *theory*—a set of patterns regarded as *axioms*—and considering only the satisfying models. Formally:

Definition 11. For M and φ, we say M *validates* φ, or φ *holds* in M, written $M \vDash \varphi$, iff $|\varphi|_\rho = M$ for all ρ. For a pattern set Γ, we say M *validates* Γ, written $M \vDash \Gamma$, iff $M \vDash \psi$ for all $\psi \in \Gamma$. We write $\Gamma \vDash \varphi$, iff $M \vDash \Gamma$ implies $M \vDash \varphi$ for all M. A matching logic *theory* (Σ, Γ) is a pair, where Σ is a signature and Γ is a set of Σ-patterns. We often abbreviate (Σ, Γ) as Γ, if Σ is understood.

Note that φ holds in M if it represents a "logical truth", i.e., its interpretation is the total set M.

Remark 12. The axiom set Γ may contain patterns that have free variables. By Definition 11, free (element and set) variables are effectively *universally quantified*, as we need to check the validity of each axiom on all possible valuations. Free element variables in an axiom can be eliminated using $\forall$-quantification, defined in Fig. 1, as in FOL. However, free set variables in an axiom *cannot be eliminated*, because $\forall$-quantification is not applicable to set variables. Allowing free set variables in axioms to be effectively universally quantified, makes matching logic more expressive (in terms of capturing models) than FOL (see Section 4.4), and comparable to the fragment of *monadic second-order logic* [Courcelle and Engelfriet 2012; Virtema et al. 2013] where all quantifiers over sets are universal quantifiers and only appear at the top.

We will define various matching logic theories in the rest of the paper. To define a theory, we need to define its sets of element variables, set variables, symbols, and axioms. We often omit explicit definitions of the variable sets and only specify the symbol and axiom sets. For readability, we mix the definitions of the symbol and axiom sets in our narrative texts. For example, when we say "we consider/define a symbol $\sigma \in \Sigma$", we mean to add σ to the symbol set of the theory we are defining. Similarly, when we say that "we define/assume an axiom ψ", we mean to add ψ to the axiom set of the theory we are defining. We will often define a theory Γ' by building it upon another more basic theory Γ. In that case, Γ' is assumed to include all components of Γ.

4 IMPORTANT MATHEMATICAL INSTRUMENTS

In this section, we (axiomatically) define several important mathematical instruments, like functions and equality, that are required in order to define binders as *theories* within matching logic (as opposed to extensions of the logic). We also propose appropriate notations for them. In Section 4.1,

we define the *definedness symbol* and use it to define equality, membership, set-theoretic inclusion, and functional constants. In Section 4.2, we define the *inhabitant symbol* and use it to define sorts, subsorting, and many-sorted functions and partial functions. This allows us to reason about sorts and to capture logical systems with sorts, in the unsorted matching logic. In Sections 4.3 and 4.4, we define matching logic theories that completely capture the models of *product sets* and *powersets*.

4.1 Definedness Symbol and Related Instruments

Recall the pattern matching semantics of matching logic: the interpretation of pattern φ is the set of elements that match it. When φ is matched by at least one element, we say that φ is *defined*. The definedness symbol (Definition 13) takes any pattern φ, and builds a new *definedness pattern* $\lceil\varphi\rceil$, which is a predicate pattern stating that φ is defined. Many important mathematical instruments such as equality and membership, can be derived from the definedness symbol as syntactic sugar.

Definition 13. Let us consider a (constant) symbol written $\lceil\_\rceil \in \Sigma$, which we call the *definedness symbol*. We write $\lceil\varphi\rceil$ to mean $\lceil\_\rceil\,\varphi$, obtained by applying $\lceil\_\rceil$ to φ. We define the following axiom:

$$\text{(Definedness)} \quad \lceil x\rceil \qquad \text{// or, equivalently, } \forall x\,.\,(\lceil\_\rceil\, x)$$

We define totality $\lfloor\_\rfloor$, equality $\_=\_$, membership $\_\in\_$, and set inclusion $\_\subseteq\_$ as derived constructs:

$$\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil \qquad \varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor \qquad x \in \varphi \equiv \lceil x \wedge \varphi\rceil \qquad \varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \rightarrow \varphi_2\rfloor$$

Intuitively, (Definedness) states that every individual element x is defined. This is clearly true with our intended meaning of $\lceil\_\rceil$, because x is matched by *exactly one element* to which it evaluates; this intended meaning is precisely what the (Definedness) axiom captures. Specifically, in any model that validates (Definedness), $\lceil x\rceil$ is interpreted as the total set, according to matching logic validity (Definition 11). Now, consider any pattern φ that is defined, and that φ is matched by one element, say x. By *pointwise extension* (Definition 7), the interpretation of $\lceil\varphi\rceil$ must include the interpretation of $\lceil x\rceil$, which we know is the total set. Therefore, $\lceil\varphi\rceil$ is also interpreted as the total set, as intended. On the other hand, if φ is *undefined*, its interpretation is the empty set, and by pointwise extension, $\lceil\varphi\rceil$ is also interpreted as the empty set. This intuition is formalized below.

Proposition 14. *For any model M, patterns $\varphi, \varphi_1, \varphi_2$, element variable x, and valuation ρ, we have*

(1) $\lceil a\rceil_M = M$ *for any* $a \in M$, *where* $\lceil a\rceil_M$ *means* $\lceil\_\rceil_M \cdot a$ *and* $\lceil\_\rceil_M$ *is the interpretation of* $\lceil\_\rceil$;
(2) $|\lceil\varphi\rceil|_\rho = M$ *if* $|\varphi|_\rho \neq \emptyset$; *otherwise,* $|\lceil\varphi\rceil|_\rho = \emptyset$;
(3) $|\lfloor\varphi\rfloor|_\rho = M$ *if* $|\varphi|_\rho = M$; *otherwise,* $|\lfloor\varphi\rfloor|_\rho = \emptyset$;
(4) $|\varphi_1 = \varphi_2|_\rho = M$ *if* $|\varphi_1|_\rho = |\varphi_2|_\rho$; *otherwise,* $|\varphi_1 = \varphi_2|_\rho = \emptyset$;
(5) $|x \in \varphi|_\rho = M$ *if* $\rho(x) \in |\varphi|_\rho$; *otherwise,* $|x \in \varphi|_\rho = \emptyset$;
(6) $|\varphi_1 \subseteq \varphi_2|_\rho = M$ *if* $|\varphi_1|_\rho \subseteq |\varphi_2|_\rho$; *otherwise,* $|\varphi_1 \subseteq \varphi_2|_\rho = \emptyset$; *note that* $|x \subseteq \varphi|_\rho = |x \in \varphi|_\rho$;

Note that all the above patterns in (2)-(6) are predicate patterns (Section 3.2.1).

Not all models validate (Definedness). Indeed, as said in Section 3.3, the purpose of axioms and theories is to restrict models under consideration. A model whose interpretation of application is a function that always returns the empty set does not validate (Definedness), as it fails to satisfy Proposition 14(1). Models that satisfy (Definedness) are also easy to come by. A canonical example is a model M with one distinguished element #def such that $\#\text{def} \cdot a = M$ for all $a \in M$, and let $\lceil\_\rceil_M$, the interpretation of $\lceil\_\rceil$, to be $\{\#\text{def}\}$. Then we have $|\lceil x\rceil|_\rho = \lceil\_\rceil_M \cdot \rho(x) = \{\#\text{def}\} \cdot \{\rho(x)\} = \#\text{def} \cdot \rho(x) = M$, and thus M validates (Definedness). In fact, any model can be extended into one that validates (Definedness) by adding an element like #def above to it and letting $\lceil\_\rceil_M$ be $\{\#\text{def}\}$. Since definedness is so useful, we assume it in all subsequent theories defined in this paper, and hereby we do not consider the models that do not satisfy the axiom (Definedness).

Remark 15. We explain why defining equality needs the definedness symbol, when there is already the logical biconditional construct $\varphi_1 \leftrightarrow \varphi_2$, given in Fig. 3. It is *not always the case* that $|\varphi_1 = \varphi_2|_\rho = |\varphi_1 \leftrightarrow \varphi_2|_\rho$ for all ρ. By Proposition 14, $\varphi_1 = \varphi_2$ is a *predicate* stating that φ_1 and φ_2 are matched by the same set of elements, while by Proposition 9, $\varphi_1 \leftrightarrow \varphi_2$ is a pattern (not necessarily a predicate) that is matched by the elements a, such that a matches φ_1 iff a matches φ_2. If $|\varphi_1|_\rho = |\varphi_2|_\rho$, then both $\varphi_1 \leftrightarrow \varphi_2$ and $\varphi_1 = \varphi_2$ are interpreted as the total set, but if otherwise, $\varphi_1 = \varphi_2$ is interpreted as the empty set, while $\varphi_1 \leftrightarrow \varphi_2$ is the complement of set difference. The fact that we can define equality axiomatically, i.e. without extending the logic, to mean *precise* identity in models is particularly useful in our subsequent developments, albeit surprising. Indeed, it is well-known that equality cannot be defined in FOL (which justifies the extension of FOL *with equality*), while in second-order logic it requires quantification over sets.

As a simple example, we can use the definedness symbol (and derived constructs) to axiomatize *functional constants*, which are matching logic symbols whose interpretations are singletons.

Example 16. Let $\sigma \in \Sigma$ be a matching logic symbol. Let us consider the following axiom

$$\text{(Functional Constant)} \quad \exists x.\, \sigma = x$$

Then for any model M that validates this axiom, we have $|\exists x.\, \sigma = x|_\rho = \bigcup_{a \in M} |\sigma = x|_{\rho[a/x]} = M$. By Proposition 14, $|\sigma = x|_{\rho[a/x]}$ is either $\emptyset$ or M, so there exists $a \in M$ such that $|\sigma = x|_{\rho[a/x]} = M$, which implies that $\sigma_M = |x|_{\rho[a/x]} = \{a\}$, i.e., σ is interpreted as a singleton in M.

4.2 Inhabitant Symbol and Related Instruments

Matching logic is an unsorted logic, but we can capture sorts by defining a set of functional constants (Example 16) that represent the *names* of the sorts, and define a special symbol, which we call the *inhabitant symbol*, to get the actual *inhabitant set* of each sort. This intuition is made formal below. From now on, we will always assume the definedness symbol and the (Definedness) axiom.

Definition 17. A *sort constant* (or simply *sort*) is a symbol $s \in \Sigma$, which is a functional constant, as defined in Example 16. Let us consider another symbol $[\![\_]\!] \in \Sigma$, which we call the *inhabitant symbol*. We write $[\![s]\!]$ to mean $[\![\_]\!]\, s$, obtained by applying $[\![\_]\!]$ to s, and call it the *inhabitant of* s.

In other words, the pattern s is matched by the sort name s itself, while $[\![s]\!]$ is matched by the actual elements of sort s. For example, for two sorts *Nat* and *Int* of natural and integer numbers, *Nat* is matched by one element—the sort name *Nat*; *Int* is matched by one element—the sort name *Int*; $[\![Nat]\!]$ is matched by all natural numbers; and $[\![Int]\!]$ is matched by all integer numbers. Note that Definition 17 does not enforce any particular axioms about sorts or the inhabitant symbol. Their interpretations are determined by the models and can be constrained by axioms. For example, subsorting $s_1 \leq s_2$ is a partial ordering on sorts that enforces the subset relation between the inhabitants of s_1 and s_2. In matching logic, subsorting can be axiomatically captured:

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

which states that the inhabitant of s_1 is included in the inhabitant of s_2. In this paper we use subsorting to define the syntax of λ-calculus and other logical systems that feature bindings. In Section 6 we define a sort *Var* for λ-calculus variables and a sort *Exp* for λ-expressions, and we define the *subsorting axiom* $[\![Var]\!] \subseteq [\![Exp]\!]$ to specify that λ-calculus variables are also λ-expressions.

4.2.1 Sorted Quantification. The meaning of $\exists x.\, \varphi$ is the set-theoretic (big) union of the interpretations of φ, with x ranging over all elements in the carrier set (see Remark 10). Now that we have defined sorts, we will want to *restrict* x to range over not all elements, but only those having sort s.

For that, we define the following self-explanatory derived constructs, called *sorted quantification*:

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

4.2.2 Many-Sorted Functions. Given sorts $s, s_1, \ldots, s_n$, we call a (constant) symbol $f \in \Sigma$ a *many-sorted function* from $s_1, \ldots, s_n$ to s, written $f\colon s_1 \times \cdots \times s_n \rightarrow s$, if it satisfies the axiom:

$$\text{(Function)} \quad \forall x_1{:}s_1.\, \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, f\, x_1\, \cdots\, x_n = y \tag{3}$$

Application is left-associative (Definition 2), so $f\, x_1\, \cdots\, x_n$ means $(\cdots (f\, x_1)\, \cdots\, x_n)$. Intuitively, (Function) requires that $f\, x_1\, \cdots\, x_n$ consist of exactly one element, y, which is an inhabitant of s, given that $x_1, \ldots, x_n$ are inhabitants of $s_1, \ldots, s_n$, respectively. Note that while f, $f x_1$, $f x_1 x_2$, ..., $f x_1 \cdots x_{n-1}$ are all well-formed patterns, they are not required to consist of exactly one element.

4.2.3 Many-Sorted Partial Functions. The axiom (Function) above is not unusual; it translates to matching logic a standard encoding of many-sorted functions using an unsorted logic (see [Nelson et al. 2010, pp. 8] for a related discussion). What is a lot harder problem is how to capture *partial functions*, which can be *undefined* in some arguments. Capturing partial functions in a formal system is not just of theoretical interest. It is also a practical concern that has arisen in the formal verification of programs with *exceptional expressions*, such as division by zero or the head of an empty list, and has resulted in work on *partial algebras* [Burmeister 1993], *exception algebras* [Bernot et al. 1986], *error algebras* [Gogolla et al. 1984], *order-sorted algebras* [Goguen and Meseguer 1992], and various *logics for partial functions* [Abdallah 1995; Lucio-Carrasco and Gavilanes-Franco 1989].

On the other hand, it is surprisingly easy to capture partial functions in matching logic. We take the axiom (Function) and change the equality $\_ = \_$ to set inclusion $\_ \subseteq \_$:

$$\text{(Partial Function)} \quad \forall x_1{:}s_1.\, \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, f\, x_1\, \cdots\, x_n \subseteq y \tag{4}$$

Intuitively, (Partial Function) requires $f\, x_1\, \cdots\, x_n$ to consist of *at most* one element. The *undefinedness* of f on $x_1, \ldots, x_n$ is captured, by $f\, x_1\, \cdots\, x_n$ returning the empty set $\emptyset$. For notional simplicity, we will write $f\colon s_1 \times \cdots \times s_n \rightharpoonup s$ to mean that f is a partial function from $s_1, \ldots, s_n$ to s.

The reason why partial functions can be directly defined using (Partial Function), without needing to extend or modify matching logic, is due to the pattern matching semantics of matching logic, where patterns are not restricted to a functional interpretation, and are given a more general, set-theoretic interpretation, which unifies (both syntactically and semantically) total functions and FOL terms, predicates and FOL formulas, and partial functions and partial terms.

4.3 Product Sorts

In this and the next sections, we assume the definedness symbol, the inhabitant symbol, and all the related instruments that are given in Sections 4.1 and 4.2. Our goal in this section is to axiomatize the *product sort* $s_1 \otimes s_2$, whose (intended) inhabitant is the (set-theoretic) product of the inhabitants of s_1 and s_2, up to isomorphism. Formally:

Definition 18. Given two sorts s_1, s_2, we consider a functional constant $s_1 \otimes s_2 \in \Sigma$, which we call the *product (sort) of s_1 and s_2*. We define a function $\langle \_, \_ \rangle\colon s_1 \times s_2 \rightarrow s_1 \otimes s_2$, called *pairing*, where the function notation was introduced in Section 4.2.2. We write $\langle \varphi_1, \varphi_2 \rangle$ to mean $\langle \_, \_ \rangle\, \varphi_1\, \varphi_2$, obtained by applying $\langle \_, \_ \rangle$ to φ_1, and then to φ_2. We define the following two axioms:

$$\text{(Product)} \quad [\![s_1 \otimes s_2]\!] = \exists x_1{:}s_1.\, \exists x_2{:}s_2.\, \langle x_1, x_2 \rangle$$
$$\text{(Injectivity)} \quad \forall x_1{:}s_1.\, \forall x_2{:}s_2.\, \forall y_1{:}s_1.\, \forall y_2{:}s_2.\, \langle x_1, x_2 \rangle = \langle y_1, y_2 \rangle \rightarrow x_1 = y_1 \wedge x_2 = y_2$$

Intuitively, $\langle x_1, x_2 \rangle$ denotes the pair consisting of x_1 and x_2. (Product) states that the inhabitant of $s_1 \otimes s_2$ is the product of the inhabitants of s_1 and s_2. (Injectivity) states that $\langle \_, \_ \rangle$ is injective.

Proposition 19. *For any model M validating the axioms in Definition 18, we have $M_{s_1 \otimes s_2} \cong M_{s_1} \times M_{s_2}$, where we use $M_s = [\![\_]\!]_M \bullet s_M$ to denote the inhabitant of s in M, for any sort s.*

4.4 Power Sorts

Our goal in this section is to axiomatize the power sort 2^s, whose (intended) inhabitant is the powerset of the inhabitant of s, up to isomorphism. Formally:

Definition 20. Given a sort s, let us consider a functional constant $2^s \in \Sigma$, which we call the *power (sort) of* s. For clarity, we use the Greek letters $\alpha, \beta, \ldots$ for element variables whose intended range is in sort 2^s. Let us define a (constant) symbol extension $\in \Sigma$, called the *extension symbol* (explained later), and define the following axioms:

(Arity)	$\forall \alpha{:}2^s.\ (\mathsf{extension}\ \alpha) \subseteq [\![s]\!]$
(Powerset)	$X \subseteq [\![s]\!] \longrightarrow \exists \alpha{:}2^s.\ (\mathsf{extension}\ \alpha) = X$
(Extensionality)	$\forall \alpha{:}2^s.\ \forall \beta{:}2^s.\ (\mathsf{extension}\ \alpha) = (\mathsf{extension}\ \beta) \longrightarrow \alpha = \beta$

Note that set variable X is free in (Powerset). By Remark 12, it is effectively universally quantified.

Definition 20 needs some explanation. Let us consider an intended model M, where the inhabitant of s is M_s and the inhabitant of 2^s is $M_{2^s} = \mathcal{P}(M_s)$, i.e., the *powerset* of M_s. We use $a, b, \cdots \in M_s$ to denote elements in M_s and $A, B, \cdots \in M_{2^s}$ to denote elements in M_{2^s}, i.e., subsets of M_s. Note that α is an element variable of sort 2^s, so let us assume it evaluates to some $A \in M_{2^s}$. Then, the intended, intuitive meaning of $(\mathsf{extension}\ \alpha)$, is that it is a pattern (of sort s) that is matched by all elements a in A. Please note the difference between α and $(\mathsf{extension}\ \alpha)$. On one hand, α is an element variable of sort 2^s, so it is matched by one "element" A. On the other hand, $(\mathsf{extension}\ \alpha)$ is a pattern of sort s, so it is matched by all elements in the set A. In other words, A is regarded as an individual "element" in sort 2^s but a real "set" in sort s, on which the pointwise extension (Definition 7) can apply. Thus, the matching logic symbol "extension" takes A as an element and returns A itself as a set. This has a similar meaning to the term "extension" in logic and philosophy—an extension of a concept consists of the things to which it applies. Here, we regard the element A of the powerset as an intensional concept and the set A of its elements as its extension.

With the above intuition, the axioms in Definition 20 are self-explanatory. (Arity) states that $(\mathsf{extension}\ \alpha)$ has sort s whenever α has sort 2^s. (Powerset) states that any subset of the inhabitant of s, ranged by X, has a corresponding "element" denoted α whose extension is X. Therefore, the inhabitant of 2^s is *at least* as large as the powerset of the inhabitant of s. On the other hand, (Extensionality) states that α and β are equal whenever their extensions are equal, so the inhabitant of 2^s is *at most* as large as the powerset of the inhabitant set s. Putting the arguments together, we show that the inhabitant of 2^s is the powerset of the inhabitant of s, up to isomorphism:

Proposition 21. *For any model M validating the axioms in Definition 20, we have $M_{2^s} \cong \mathcal{P}(M_s)$.*

The reverse of extension, called *intension*, can be defined as the following syntactic sugar:

$$\mathsf{intension}\ \varphi \equiv \exists \alpha{:}2^s.\ \alpha \wedge (\mathsf{extension}\ \alpha = \varphi)$$

Intuitively, φ has sort s; $(\mathsf{intension}\ \varphi)$ has sort 2^s, and is matched by the *unique* element α of sort 2^s such that $\mathsf{extension}\ \alpha = \varphi$; the uniqueness is guaranteed by the axiom (Extensionality).

Remark 22. Proposition 21 shows that powersets can be completely, finitely axiomatized in matching logic. This result is known to *not hold* in FOL, because by the Löwenheim-Skolem theorem [Löwenheim 1915], if a FOL theory has infinite models, then it has a countable model. However, using powersets, we can enforce uncountable models by first enforcing an infinite model and considering its powerset. As an example, we define natural numbers *Nat* using *zero* and *suc*, and define the standard injectivity axioms $zero \neq suc(x)$ and $suc(x) = suc(y) \longrightarrow x = y$ to enforce *Nat* to be infinite, as it must contain *zero*, $suc(zero)$, $suc(suc(zero))$, etc., which are all distinct. If powersets could have been completely axiomatizable in FOL, then we could define the powerset of natural numbers 2^{Nat} that is uncountable, contradicting the Löwenheim-Skolem theorem.

free variables:

$$\mathrm{FV}(x) = \{x\} \qquad \mathrm{FV}(e_1\, e_2) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \qquad \mathrm{FV}(\lambda x.\, \varphi) \equiv \mathrm{FV}(\varphi) \setminus \{x\}$$

α-renaming:

$$\lambda x.\, \varphi \equiv \lambda y.\, \varphi[y/x], \text{ for } y \notin \mathrm{FV}(\varphi)$$

capture-free substitution (where y distinct from x and z is fresh):

$$(\lambda x.\, \varphi)[\psi/x] \equiv \lambda x.\, \varphi \qquad (\lambda x.\, \varphi)[\psi/y] \equiv \lambda z.\, \varphi[z/x][\psi/y]$$

Fig. 3. Meta-properties about binder λ, similar to those for the binder $\exists$ in matching logic (Fig. 1).

4.5 Matching Logic Proof System

There is a Hilbert-style proof system for matching logic that defines the provability relation $\Gamma \vdash \varphi$ for matching logic theory Γ and pattern φ. The proof system is not needed in order to understand the technical results discussed in this paper (see [Chen and Roşu 2019]). We only review some meta-theorems about the proof system, which are needed in order to prove the subsequent results, mentioning that any (sound) proof system that has these properties would be equally suitable:[3]

Proposition 23. *If Γ contains the definedness symbol and the axiom (Definedness), then*

(1) $\Gamma \vdash \varphi$, *if φ is a propositional tautology over patterns;*
(2) $\Gamma \vdash \varphi_1$ *and* $\Gamma \vdash \varphi_1 \longrightarrow \varphi_2$ *imply* $\Gamma \vdash \varphi_2$;
(3) $\Gamma \vdash \varphi[y/x] \longrightarrow \exists x.\, \varphi$;
(4) $\Gamma \vdash \varphi_1 \longrightarrow \varphi_2$ *and* $y \notin FV(\varphi_2)$ *imply* $\Gamma \vdash (\exists y.\, \varphi_1) \longrightarrow \varphi_2$;
(5) $\Gamma \vdash \varphi = \varphi$;
(6) $\Gamma \vdash \varphi_1 = \varphi_2$ *and* $\Gamma \vdash \varphi_2 = \varphi_3$ *imply* $\Gamma \vdash \varphi_1 = \varphi_3$;
(7) $\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \varphi_2 = \varphi_1$;
(8) $\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \psi[\varphi_1/x] = \psi[\varphi_2/x]$, *known as the Leibniz characterization of equality.*

Proposition 23 essentially states that FOL with equality reasoning is supported by the proof system of matching logic, where patterns are conveniently regarded as either "predicates" or "terms", depending on the context. We require Γ to contain the definedness symbol and axiom, because they are needed to define equality $\varphi_1 = \varphi_2$, as discussed in Definition 13.

We review the following *soundness theorem* of the matching logic proof system:

Theorem 24 (Soundness Theorem). $\Gamma \vdash \varphi$ *implies* $\Gamma \vDash \varphi$.

While several *(deductive) completeness results* (i.e., $\Gamma \vDash \varphi$ implies $\Gamma \vdash \varphi$) have been proved for some theories Γ in [Chen and Roşu 2019; Roşu 2017], it is incomplete in general for all Γ and φ. Fortunately, it does not affect this paper. Instead, we prove a *new completeness result* as a corollary of the conservative extension theorem of λ-calculus (Theorem 36), where Γ is the matching logic theory that captures λ-calculus and φ is an equation between λ-expressions; see Section 5.

5 λ-CALCULUS PRELIMINARIES

The syntax of λ-calculus [Church 1941] is parametric in a set of variables V^λ, whose elements are written $x, y, \ldots$. The set Λ of *λ-expressions* is inductively defined by the following grammar:

$$e ::= x \mid e_1\, e_2 \mid \lambda x.\, e$$

Free variables $\mathrm{FV}(e)$, α-equivalence $e_1 \equiv e_2$, and capture-free substitution $e[e'/x]$ are defined as usual, shown in Fig. 3. We regard α-equivalent λ-expressions as identical expressions.

In λ-calculus, we are interested in proving equations of the form $e_1 = e_2$, for $e_1, e_2 \in \Lambda$. Equational reasoning in λ-calculus includes the standard reflexivity, symmetry, transitivity, and congruence

[3] Note that Γ is different from typing contexts in type systems (see, e.g., [Cardelli 1996]) that share variables with judgment φ. Here, Γ has variables independent from φ and its axioms are implicitly universally quantified; see also Remark 12.

proof rules, and the distinguished (β) *axiom schema* that specifies the result of function application:

$$(\beta) \quad (\lambda x.\, e)\, e' = e[e'/x] \quad \text{for all } x \in V^\lambda \text{ and } e, e' \in \Lambda$$

We write $\vdash_\lambda e_1 = e_2$ to mean that $e_1 = e_2$ is provable in λ-calculus.

5.1 Our Goal and the Main Challenges

Our first goal is to define a matching logic theory Γ^λ that faithfully captures λ-calculus, in the sense that λ-expressions are well-formed matching logic patterns and λ-reasoning is captured by matching logic reasoning. Formally, our goal is to prove the *conservative extension* theorem:

$$\Gamma^\lambda \vdash e_1 = e_2 \quad \underset{\text{extensiveness}}{\overset{\text{conservativeness}}{\rightleftharpoons}} \quad \vdash_\lambda e_1 = e_2 \text{ for all } e_1, e_2 \in \Lambda \tag{5}$$

which says that we can safely reduce λ-calculus reasoning to matching logic reasoning, without proving fewer or more equations between λ-expressions. Specifically, the *extensiveness* direction means that all provable equations between λ-expressions can also be proved in Γ^λ, which is thus an *extension* of λ-calculus, while the *conservativeness* direction says that no additional equations between λ-expressions can be proved. Note that we are only concerned with equations *between* λ*-expressions*. Since matching logic has a richer syntax than λ-calculus, of course there are equations, e.g. $\bot = \bot$, which are provable in matching logic but do not even exist in λ-calculus.

Main Challenges. There are two main challenges. The first challenge is to capture the binding behavior of λ, that is, to define $\lambda x.\, e$ as *syntactic sugar* in matching logic such that it satisfies the properties about free variables, α-equivalence, and capture-free substitution in Fig. 3. The key observation is that λ plays two important roles: (i) it builds a *term* $\lambda x.\, e$, and (ii) it builds a *binding* of x into e. Matching logic allows us to separate these two roles, where we define terms using symbols and application as shown in Section 4 and bindings using matching logic's built-in binder $\exists$.

The other challenge is to prove the conservative extension theorem shown as Eq. (5). The extensiveness direction is easy, because equational reasoning is supported in matching logic (Proposition 23). We only need to include all instances of (β) in Γ^λ. The conservativeness direction is more involved and is a major technical contribution of this paper. Indeed, matching logic has a richer syntax and a more complex proof system than λ-calculus; we need to show that this more complex infrastructure cannot be used to prove more equations between λ-expressions.

5.2 Our Plan

We will give two different proofs for the conservativeness of Γ^λ, each providing a unique insight about the construction of Γ^λ. The first is based on a model theory of λ-calculus, discussed in Section 7. It considers a special class of λ-calculus models, called concrete Cartesian closed category models, or simply concrete ccc models, which are known to be complete with respect to λ-calculus reasoning (Lemma 26). This model-based proof is easier to understand due to its close connection to the models, and is what inspired our encoding of the λ binder in matching logic (see Eq. (1)). However, it does not generalize to other logical systems with binders that do not have well-established models. Hence, in Section 8 we give an alternative conservativeness proof, based on the syntax and proof derivations of λ-calculus, and not on models. The syntax-based proof does not depend on the existence of a complete class of models, and is thus easier to generalize to other logical systems.

5.3 Concrete ccc Models of λ-Calculus

We review the *concrete Cartesian closed category (ccc) models of* λ*-calculus* [Barendregt 1984, Definition 5.5.9]. They will be used in the model-based proof of the conservativeness of Γ^λ.

Definition 25 ([Berline 2000, Definition 57]). Given an applicative structure $(A, \_\bullet_A\_)$, its set of *representable functions* is $R(A) = \{f : A \to A \mid \text{there is a } b \in A \text{ such that } f(a) = b \bullet_A a \text{ for all } a \in A\}$.

$$\Gamma^\lambda \vdash e_1 = e_2 \quad \Longrightarrow_1 \quad \Gamma^\lambda \vDash e_1 = e_2 \quad \Longrightarrow_2 \quad M \vDash e_1 = e_2 \text{ for all matching logic models } M \vDash \Gamma^\lambda$$
$$\Downarrow_3$$
$$\vdash_\lambda e_1 = e_2 \quad \Longleftarrow_5 \quad \vDash_\lambda e_1 = e_2 \quad \Longleftarrow_4 \quad A \vDash_\lambda e_1 = e_2 \text{ for all concrete ccc models } A$$

Fig. 4. The main proof steps of the model-based conservativeness proof of Γ^λ.

A *pre-model* is a triple $(A, \_\bullet_A\_, \mathbb{L})$, where $\mathbb{L}\colon R(A) \to A$ is a *retraction function* such that $\mathbb{A} \circ \mathbb{L}$ is the identity on $R(A)$, where $\mathbb{A}\colon A \to R(A)$ is defined as $\mathbb{A}(b)(a) = b \bullet_A a$ for all $b, a \in A$. A pre-model A is called a *concrete ccc model*, if the following definition of $|e|^\lambda_\rho$ is well-defined for every $\rho\colon V^\lambda \to A$:

(1) $|x|^\lambda_\rho = \rho(x)$;
(2) $|e_1 e_2|^\lambda_\rho = |e_1|^\lambda_\rho \bullet_A |e_2|^\lambda_\rho$;
(3) $|\lambda x.\, e|^\lambda_\rho = \mathbb{L}(f^\rho_{e,x})$ where $f^\rho_{e,x}(a) = |e|^\lambda_{\rho[a/x]}$ for $a \in A$, and that $f^\rho_{e,x} \in R(A)$.

Given a concrete ccc model A, we write $A \vDash_\lambda e_1 = e_2$ iff $|e_1|^\lambda_\rho = |e_2|^\lambda_\rho$ for all ρ. We write $\vDash_\lambda e_1 = e_2$ iff $A \vDash_\lambda e_1 = e_2$ for all concrete ccc models A. In the latter, we say $e_1 = e_2$ is valid in λ-calculus.

We review two important results about concrete ccc models in the model-based conservativeness proof, whose main proof steps are shown in Fig. 4. The first result is that concrete ccc models are a special instance of matching logic models. In other words, Γ^λ includes all concrete ccc models as its validating models. This result will be used in Step 3, from matching logic validity to λ-calculus validity. The second result is that concrete ccc models are *complete* with respect to λ-calculus reasoning, i.e., all valid λ-calculus equations can be proved.[4] This known completeness result is restated in Lemma 26. It will be used in Step 5 in Fig. 4, from λ-calculus validity to provability.

Lemma 26 ([Koymans 1982]). $\vDash_\lambda e_1 = e_2$ *implies* $\vdash_\lambda e_1 = e_2$ *for any* $e_1, e_2 \in \Lambda$.

Other λ-Calculus Models. We discuss the other relevant notions of λ-calculus models and discuss why we choose the concrete ccc models in our conservativeness proof (given in Section 7).

There are three main notions of models in λ-calculus; see [Manzonetto 2008] for a survey. Firstly, there are *λ-models* [Barendregt 1984, Section 5.2], which are combinatory algebras that provide coherent interpretations to all λ-expressions. Secondly, there are *categorical models* [Barendregt 1984, Section 5.5], which are given as the reflexive objects of a Cartesian closed category (ccc), where λ-expressions are interpreted as morphisms. Thirdly, there are *Hindley-Longo models* [Hindley and Longo 1980], which form an alternative presentation of λ-models and interpret λ-expressions directly, without translating them to combinatory terms. The concrete ccc models (Definition 25) in this paper belong to the categorical models, where the underlying categories are *strictly concrete categories* (see, e.g., [Barendregt 1984, Definition 5.5.8]).

We choose concrete ccc models because they have a non-categorical set-theoretical presentation (Definition 25) that fits well with the pattern matching semantics of matching logic. In concrete ccc models, the interpretation of a λ-expression is inductively defined from the interpretation of its sub-expressions, so it is more natural to turn concrete ccc models into matching logic models, needed for the conservativeness proof. In contrast, λ-models and Hindley-Longo models interpret *all* λ-expressions *at the same time*. For example, in Hindley-Longo models, $|\lambda x.\, e|^\lambda_\rho$ is defined as some *unspecified element* that satisfies that $|\lambda x.\, e|^\lambda_\rho \bullet_A a = |e|^\lambda_{\rho[a/x]}$ for all a. In concrete ccc models, instead, $|\lambda x.\, e|^\lambda_\rho$ is interpreted explicitly by $|\lambda x.\, e|^\lambda_\rho = \mathbb{L}(f^\rho_{e,x})$, using a given (by the model) retraction function to encode functions into elements. Therefore, it is more convenient in our context to consider concrete ccc models, as they provide an explicit, constructive interpretation of $\lambda x.\, e$.

[4] Here we use the term "completeness" to mean deductive completeness, as given in Lemma 26. In the literature on λ-calculus, *representability completeness* (of λ-calculus models) is also considered; see related discussion in Section 8.2.2.

6 DEFINING λ-CALCULUS IN MATCHING LOGIC

In this section we define the matching logic theory Γ^λ that captures λ-calculus. Our definition is inspired by the concrete ccc models of λ-calculus discussed in Section 5.3. The key ingredient is the retraction function $\mathbb{L}$ that encodes representable functions into elements. Therefore, we first define representable functions and the retraction function.

Recall that $f^\rho_{e,x}$ is the representable function as defined in Definition 25, which corresponds to the interpretation of $\lambda x.\, e$ under ρ in the concrete ccc model. We can capture $f^\rho_{e,x}$ by defining its *graph*:

$$\mathsf{graph}(f^\rho_{e,x}) = \left\{ \left(a, |e|^\lambda_{\rho[a/x]} \right) \mid \text{for all elements } a \text{ in the concrete ccc model } A \right\} \tag{6}$$

which contains all the argument-value pairs of $f^\rho_{e,x}$. Note that this graph is an element in $\mathcal{P}(A \times A)$, the powerset of $A \times A$, but not every element in $\mathcal{P}(A \times A)$ is the graph of a representable function. Therefore, the retraction function $\mathbb{L}$ is captured as a *partial function* from $\mathcal{P}(A \times A)$ to A (see Remark 27) which is defined only on the graphs of representable functions, and undefined elsewhere.

Now we start to define Γ^λ following the above intuition. Firstly, we include all λ-calculus variables in V^λ as element (and not set) variables in Γ^λ. Then, we define four sorts: *Var* as the sort of λ-calculus variables; *Exp* as the sort of λ-expressions; $\mathit{Var} \otimes \mathit{Exp}$ as the *product sort* of *Var* and *Exp* (Definition 18); and $2^{\mathit{Var}\otimes\mathit{Exp}}$ as its *power sort* (Definition 20). Intuitively, $2^{\mathit{Var}\otimes\mathit{Exp}}$ is the sort of all binary relations, including non-functions, over *Var* and *Exp*, because the inhabitant of $2^{\mathit{Var}\otimes\mathit{Exp}}$ is the powerset of the Cartesian product of the inhabitants of *Var* and *Exp*, by Propositions 19 and 21.

Next, we define the subsorting axiom (Section 4.2), $[\![\mathit{Var}]\!] \subseteq [\![\mathit{Exp}]\!]$, to specify that all variables are well-formed λ-expressions. We define a partial function (Section 4.2.3), $\mathsf{lambda} : 2^{\mathit{Var}\otimes\mathit{Exp}} \rightharpoonup \mathit{Exp}$, to represent the retraction function $\mathbb{L}$ in Definition 25, although the partial function requirement is included only for clarity and is technically unnecessary, because it will be automatically validated by the intended canonical models that we construct in Sections 7 and 8.

Remark 27. We include both sorts *Var* and *Exp* in theory Γ^λ so as to be completely faithful w.r.t. the λ-calculus syntax defined in Section 5, which has two syntactic categories: V^λ for variables and Λ for expressions. As a result, lambda is a partial function with the power domain $2^{\mathit{Var}\otimes\mathit{Exp}}$. A valid alternative is to use $2^{\mathit{Exp}\otimes\mathit{Exp}}$ as the domain. The conservative extension theorem (Theorem 36) still holds, and its model-based proofs shown in Section 7 are still valid, because the models we will construct there interpret both *Var* and *Exp* to the same inhabitant set.

Now, we define λ-expressions as syntactic sugar in matching logic. The λ-calculus variables and application are already well-formed matching logic patterns, where $x \in \mathit{Var}$ is represented by the element variables x and $e_1\ e_2$ is represented by the built-in matching logic application $e_1\ e_2$. Abstraction $\lambda x.\, e$ is defined as the following syntactic sugar, where we extract the general *binding notation* $[x{:}\,\mathit{Var}]\ e$ for clarity and because it can be used to define any other binders, not only λ:

$$[x{:}\,\mathit{Var}]\ e \equiv \mathsf{intension}\ \exists x{:}\,\mathit{Var}.\ \langle x, e\rangle \qquad \text{// the binding notation} \tag{7}$$

$$\lambda x.\, e \equiv \mathsf{lambda}\ [x{:}\,\mathit{Var}]\ e \qquad \text{// } \lambda\text{-abstraction} \tag{8}$$

We assume that $[x{:}\,\mathit{Var}]\ e$ binds the tightest, so $\mathsf{lambda}\ [x{:}\,\mathit{Var}]\ e$ is parsed as $\mathsf{lambda}\ ([x{:}\,\mathit{Var}]\ e)$.

Eq. (8) is a logical incarnation of the semantics of $\lambda x.\, e$ in the concrete ccc models (Definition 25), into matching logic. Recall that in a concrete ccc model, $|\lambda x.\, e|^\lambda_\rho = \mathbb{L}\left(f^\rho_{e,x}\right)$, where $f^\rho_{e,x}(a) = |e|^\lambda_{\rho[a/x]}$. By Remark 10, $\exists x{:}\,\mathit{Var}.\ \langle x, e\rangle$ denotes the union set $\bigcup_x \{(x, e)\}$, namely the graph of $f^\rho_{e,x}$. (Note that $\forall x{:}\,\mathit{Var}.\ \langle x, e\rangle$ also yields the correct binding behavior, but it does not have the right semantic meaning of a graph.) The binding notation $[x{:}\,\mathit{Var}]\ e$ takes this graph as a *set* of pairs and *packs* them into one object in the power sort $2^{\mathit{Var}\otimes\mathit{Exp}}$. Then, this packed object is passed to lambda, which decodes/retracts it into the intended interpretation of $\lambda x.\, e$. For now, we do not know any property

Variables:	
$x, y, \ldots$	element variables, including all λ-calculus variables in V^λ
Symbols:	
Var	a sort constant
Exp	a sort constant
lambda	the retraction symbol, used to capture λ
Axioms:	
(Subsorting)	$[\![Var]\!] \subseteq [\![Exp]\!]$
(β)	$\forall x_1 : Var. \cdots \forall x_n : Var. (\lambda x.\, e)\, e' = e[e'/x]$ where $x_1, \ldots, x_n$ are all the free variables in $\mathrm{FV}((\lambda x.\, e)\, e')$.

Fig. 5. Summary of the matching logic theory Γ^λ that captures λ-calculus (infrastructure definitions omitted)

about lambda, except that it is a partial function from $2^{Var \otimes Exp}$ to *Exp*. Its intended behavior will be axiomatized by the axiom schema (β)—*the* axiom schema that characterizes λ-abstraction and the semantics of λ.

We emphasize that the encoding of $\lambda x.\, e$ in Eqs. (7)-(8) is only possible because matching logic treats terms and formulas uniformly as patterns, and it allows (FOL-style) quantification to be built on terms. A similar definition will immediately fail in FOL, because FOL enforces a clear distinction between terms and formulas at the syntax level and quantification only applies to formulas.

Remark 28. Under the above notations, all λ-expressions are well-formed matching logic patterns. Particularly, the syntactic sugar $\lambda x.\, e$ in Eqs. (7)-(8) satisfies all binding properties about λ in Fig. 3.

Definition 29. Let Γ^λ be the matching logic theory that contains all the axioms and notations that we have defined in this section, and all instances of the (β) axiom schema, as shown in Fig. 5.

Remark 30. Remark 28 holds, not because of the axioms in Γ^λ, but because of the syntactic sugar definition in Eqs. (7)-(8) and the binding behavior of $\exists$. In other words, the binding behavior of λ is directly inherited from from the binding behavior of the built-in binder $\exists$ in matching logic, and is *not* specified by axioms. The axioms specify the semantic behavior of λ, not its binding behavior.

We finish this section by proving the extensiveness theorem for λ-calculus.

Theorem 31. $\vdash_\lambda e_1 = e_2$ *implies* $\Gamma^\lambda \vdash e_1 = e_2$, *for all* $e_1, e_2 \in \Lambda$.

Proof. By Proposition 23, because Γ^λ contains all instances of (β). □

7 MODEL-BASED CONSERVATIVENESS PROOF

Here we prove the conservativeness of Γ^λ, making use of the concrete ccc models of λ-calculus discussed in Section 5.3. The main proof steps have been discussed in Section 5 and summarized in Fig. 4. The only nontrivial one is Step 3, which requires to show that $M \models e_1 = e_2$ for all matching logic models $M \models \Gamma^\lambda$ implies $A \models_\lambda e_1 = e_2$ for all concrete ccc models A. The following is the key lemma establishing the connection between concrete ccc models and matching logic models of Γ^λ:

Lemma 32. *For any concrete ccc model* A *and any valuation* ρ *into* A, *there exists a matching logic model* $M^A \models \Gamma^\lambda$ *and a valuation* ρ^A *into* M^A *such that* $|e|_{\rho^A} = \left\{ |e|^\lambda_\rho \right\}$ *for every* $e \in \Lambda$.

Proof. We give the high-level proof idea. Let us fix a concrete ccc model $(A, \_\bullet_A\_, \mathbb{L})$, where $R(A)$ is its set of representable functions and $\mathbb{L} : R(A) \to A$ is its retraction function. Let the carrier set M_A include A. Recall that Γ^λ defines sorts *Var* and *Exp*, and partial function lambda from $2^{Var \otimes Exp}$ to *Exp* (Fig. 5). Since A is the domain of both variable valuations and expression interpretations in

the concrete ccc model, in M_A we let A be the inhabitants of both *Var* and *Exp* (see Remark 27), validating axiom (Subsorting). We define lambda_{M^A} accordingly to the retraction function $\mathbb{L}$; i.e., $\mathsf{lambda}_{M^A} \cdot P = \{\mathbb{L}(f)\}$ whenever $P = \mathsf{graph}(f)$ and $f \in R(A)$, and $\mathsf{lambda}_{M^A} \cdot P = \emptyset$, otherwise.

We define ρ^A as $\rho^A(x) = \rho(x)$, for every $x \in V^\lambda$, and prove that $|e|_{\rho^A} = \{|e|^\lambda_\rho\}$ for every $e \in \Lambda$. The proof is based on structural induction on e and the only nontrivial case is when e is $\lambda x.\, e_1$. In this case, we have $|\lambda x.\, e_1|_{\rho^A} = |\mathsf{lambda}\ (\mathsf{intension}\ (\exists x{:}\mathit{Var}.\ \langle x, e_1\rangle))|_{\rho^A} = \mathsf{lambda}_{M^A} \cdot |\mathsf{intension}\ (\exists x{:}\mathit{Var}.\ \langle x, e_1\rangle)|_{\rho^A} = \mathsf{lambda}_{M^A} \cdot |\exists x{:}\mathit{Var}.\ \langle x, e_1\rangle)|_{\rho^A} = \mathsf{lambda}_{M^A} \cdot \bigcup_{a\in A}\{(a, |e_1|_{\rho^A[a/x]})\} = \mathsf{lambda}_{M^A} \cdot \bigcup_{a\in A}\{(a, |e_1|^\lambda_{\rho[a/x]})\} = \mathsf{lambda}_{M^A} \cdot \mathsf{graph}(f^\rho_{e_1,x}) = \{\mathbb{L}(f^\rho_{e_1,x})\} = \{|\lambda x.\, e_1|^\lambda_\rho\}$.

Finally, we show that M^A validates (β). Using the above result, for any $x \in V^\lambda$, $e, e' \in \Lambda$, and ρ, we have that $|(\lambda x.\, e)e'|^\lambda_\rho = |e[e'/x]|^\lambda_\rho$ in A implies $|(\lambda x.\, e)e'|_{\rho^A} = |e[e'/x]|_{\rho^A}$ in M^A. Noting that ρ^A is arbitrary (as ρ is arbitrary), M^A validates (β). □

Remark 33. The operations, intension and lambda, have been crucial in the proof. Without them, the pattern $\exists x{:}\mathit{Var}.\ \langle x, e\rangle$ itself is merely the graph set and is not even a functional pattern (in the sense discussed in Section 4.2.2), and thus cannot be directly used to interpret $\lambda x.\, e$.

Using Lemma 32, we can immediately prove Step 3 in Fig. 4:

Lemma 34. *If $M \models e_1 = e_2$ for all models $M \models \Gamma^\lambda$, then $A \models_\lambda e_1 = e_2$ for all concrete ccc models A.*

Proof. Let A be any concrete ccc and ρ be any valuation. By Lemma 32, there exists a matching logic model $M^A \models \Gamma^\lambda$ and a valuation ρ^A such that $|e|_{\rho^A} = \{|e|^\lambda_\rho\}$ for any $e \in \Lambda$. Since $M^A \models e_1 = e_2$, we have $|e_1|_{\rho^A} = |e_2|_{\rho^A}$, and thus $|e_1|^\lambda_\rho = |e_2|^\lambda_\rho$. Since ρ is any valuation, we have $A \models_\lambda e_1 = e_2$. □

Theorem 35. *$\Gamma^\lambda \vdash e_1 = e_2$ implies $\vdash_\lambda e_1 = e_2$, for all $e_1, e_2 \in \Lambda$.*

Proof. See Fig. 4, where Step 1 is by Theorem 24; Step 2 is by Definition 11; Step 3 is by Lemma 34; Step 4 is by Definition 25; and Step 5 is by Lemma 26. □

Theorem 35 together with Theorem 31 show that Γ^λ is a conservative extension of λ-calculus. In fact, we prove the following equivalence theorem (for $e_1, e_2 \in \Lambda$):

Theorem 36. *These are equivalent: (1) $\Gamma^\lambda \vdash e_1 = e_2$; (2) $\Gamma^\lambda \models e_1 = e_2$; (3) $\models_\lambda e_1 = e_2$; (4) $\vdash_\lambda e_1 = e_2$.*

Proof. (1) $\Longrightarrow$ (2) is by Theorem 24. (2) $\Longrightarrow$ (3) is by Lemma 34. (3) $\Longrightarrow$ (4) is by Lemma 26. (4) $\Longrightarrow$ (1) is by Theorem 35. Note: Conservative extension theorem is the equivalence (1) $\Longleftrightarrow$ (4). □

Remark 37. The equivalence (2) $\Longleftrightarrow$ (4) shows the *(deductive) completeness* of the matching logic models of Γ^λ with respect to λ-calculus. By defining λ-calculus in matching logic, we automatically obtain, from the model theory of matching logic, models that are complete to λ-calculus.

8 SYNTAX-BASED CONSERVATIVENESS PROOF

In this section we show an alternative conservativeness proof of Theorem 35 that is entirely based on the syntactic structure of λ-expressions, and thus is easier to generalize to other logical systems and binders, especially those which do not have well-established models. This syntax-based proof also shows that Γ^λ is *representationally complete* for λ-calculus; see Section 8.2.2.

8.1 Proof Overview: Using the Term Model to Prove the Conservativeness Theorem

We build a special matching logic model $T \models \Gamma^\lambda$, which we call the *term model* of λ-calculus,[5] and follow the term algebra technique [Bell and Machover 1977; Hasenjaeger 1953; Quackenbush 1988]:

[5]In the literature on λ-calculus, term models have a different meaning. For example, in [Barendregt 1984], term models are special λ-calculus models constructed based on the combinatory algebra semantics; see Section 8.2.1 for a comparison.

T has as elements the equivalence classes of λ-expressions modulo $\alpha\beta$-equivalence, and each $e \in \Lambda$ is interpreted in T as the equivalence class containing itself, $[e]$. Formally, we will prove this:

Theorem 38. *Let* $[e] = \{e' \in \Lambda \mid \vdash_\lambda e = e'\}$ *be the equivalence class of* e *modulo* $\alpha\beta$*-equivalence. Let* $[\Lambda] = \{[e] \mid e \in \Lambda\}$ *be the set of all these classes. Then, there is a matching logic model* $T \vDash \Gamma^\lambda$*, called* term model*, and a valuation* ρ_T*, called* term valuation*, such that* $|e|_{\rho_T} = \{[e]\}$ *for all* $e \in \Lambda$.

Remark 39. For distinct variables $x, y \in V^\lambda$, we have $[x] \neq [y]$ [Barendregt 1984, Fact 2.1.37]. Clearly, $x \in [x]$, but $[x]$ also includes infinitely many expressions: $(\lambda y.\, y)x$, $(\lambda y.\, y)((\lambda y.\, y)x)$, etc.

We will construct T in Section 8.2. For now, we show how to prove Theorem 35 using Theorem 38:

Syntax-Based Proof of Theorem 35. We need to prove $\Gamma^\lambda \vdash e_1 = e_2$ implies $\vdash_\lambda e_1 = e_2$:

$\Gamma^\lambda \vdash e_1 = e_2$	implies	$\Gamma^\lambda \vDash e_1 = e_2$	by Theorem 24
	implies	$T \vDash e_1 = e_2$	by Definition 11
	implies	$\lvert e_1\rvert_{\rho_T} = \lvert e_2\rvert_{\rho_T}$	by Proposition 14
	implies	$[e_1] = [e_2]$	by Theorem 38
	implies	$\vdash_\lambda e_1 = e_2$	by Definition of $[e]$ in Theorem 38. □

8.2 Construction of the Term Model T and the Term Valuation ρ_T

In this section we construct T and show that $T \vDash \Gamma^\lambda$. Like for the matching logic model of Γ^λ in the proof of Lemma 32, we need to give interpretations to the sorts *Var* and *Exp*, as well as to the retraction function lambda. For *Var* and *Exp*, we define their inhabitants as $T_{Var} = [V^\lambda]$ and $T_{Exp} = [\Lambda]$, where $[V^\lambda]$ and $[\Lambda]$ are the set of equivalence classes of variables and λ-expressions. Clearly, we have $[V^\lambda] \subseteq [\Lambda]$, which validates the axiom (Subsorting) $[\![Var]\!] \subseteq [\![Exp]\!]$. We define the interpretation of application on λ-expressions as the application in λ-calculus, i.e., $[e_1]\cdot[e_2] = [e_1\, e_2]$ for any $e_1, e_2 \in \Lambda$. Note that this definition is well-defined, because $\vdash_\lambda e_1\, e_2 = e_1'\, e_2'$ whenever $\vdash_\lambda e_1 = e_1'$ and $\vdash_\lambda e_2 = e_2'$. Finally, we define the interpretation lambda_T such that

$$\mathsf{lambda}_T \cdot \left(\bigcup_{z \in V^\lambda} ([z], [e[z/x]]) \right) = \{[\lambda x.\, e]\}, \quad \text{for any } x \in V^\lambda \text{ and } e \in \Lambda. \tag{9}$$

and $\mathsf{lambda}_T \cdot P = \emptyset$, if P is not a graph of the above form.

The construction of T, especially Eq. (9), is critically depending on the matching logic encoding $\lambda x.\, e \equiv \mathsf{lambda}\,(\mathsf{intension}\, \exists x{:}\, Var.\, \langle x, e\rangle)$. The α-equivalence $\lambda x.\, e \equiv \lambda z.\, (e[z/x])$ is captured, both syntactically and semantically, by collecting the pairs $\langle z, e[z/x]\rangle$ for all z, using the matching logic pattern $\exists x{:}\, Var.\, \langle x, e\rangle$ (see Remark 10 for the connection between the $\exists$-patterns and the set-theoretic unions). Therefore, $\exists x{:}\, Var.\, \langle x, e\rangle$ encapsulates all the information about $[\lambda x.\, e]$, which is *packed* by intension and passed to lambda, and then *retracted* to restore the original expression $\lambda x.\, e$. The following proposition shows that the condition in Eq. (9) on lambda_T is not inconsistent:

Proposition 40. $[\lambda x.\, e] = [\lambda x'.\, e']$, *whenever*

$$\bigcup_{z \in V^\lambda} ([z], [e[z/x]]) = \bigcup_{z \in V^\lambda} ([z], [e'[z/x']]) \tag{10}$$

Proof. Assume the opposite, i.e., $[\lambda x.\, e] \neq [\lambda x'.\, e']$. Let $z^* \in V^\lambda$ be a fresh variable that does not occur in $\lambda x.\, e$ or $\lambda x'.\, e'$. Then we have $\lambda x.\, e \equiv \lambda z^*.\, e[z^*/x]$ and $\lambda x'.\, e' \equiv \lambda z^*.\, e'[z^*/x']$. By the assumption, we have $[\lambda z^*.\, e[z^*/x]] \neq [\lambda z^*.\, e'[z^*/x']]$, and thus $[e[z^*/x]] \neq [e'[z^*/x']]$. Noting that $[z_1] = [z_2]$ iff $z_1 = z_2$, for every $z_1, z_2 \in V^\lambda$ (Remark 39), we have that the pair $([z^*], [e[z^*/x]])$ is in the LHS of Eq. (10) but not its RHS, which is a contradiction. □

So far, we have constructed the term model T. We now define the term valuation ρ_T. Let $\mathit{VarVal} = \{\rho \mid \rho(x) \in [V^\lambda] \text{ for all } x \in V^\lambda\}$ be the set of valuations that map λ-calculus variables (which have been taken as matching logic element variables; see Section 6) to the equivalence classes of λ-calculus *variables*, and not any λ-expressions. We define the term valuation ρ_T, as $\rho_T(x) = [x]$ for every $x \in V^\lambda$. Clearly, $\rho_T \in \mathit{VarVal}$.

Proposition 41. *$|e|_{\rho_T} = \{[e]\}$, and $|e|_{\rho[\rho(z)/x]} = |e[z/x]|_\rho$ for all $\rho \in \mathit{VarVal}$.*

Proof. We prove both properties simultaneously by induction on the *λ-depth* $d(e)$ of e, the maximum number of nested λ binders in e. If $d(e) = 0$ then e is a variable or is built from only application and has no λ abstraction. In this case, both properties can be proved by another structural induction on e. If $d(e) \geq 1$ then e has either the form $e_1\, e_2$ where $d(e_1), d(e_2) \leq d(e)$, or the form $\lambda x.\, e_1$ where $d(e_1) \leq d(e) - 1$. Then another structural induction on e proves both properties. □

Proposition 42. *If $\vdash_\lambda e = e'$, then $|e|_\rho = |e'|_\rho$ for any $\rho \in \mathit{VarVal}$.*

Proof. Note that the interpretation of a λ-expression relies on its free variables. Suppose $\mathrm{FV}(e) \cup \mathrm{FV}(e') = \{x_1, \ldots, x_n\}$ and $\rho(x_i) = [y_i]$ for $i \in \{1, \ldots, n\}$. By Remark 39, y_i is the unique variable that is in $[y_i]$. Since ρ equals to $\rho_T[[y_1]/x_1] \cdots [[y_n]/x_n]$ restricted on $x_1, \ldots, x_n$, we have $|e|_\rho = |e|_{\rho_T[[y_1]/x_1]\cdots[[y_n]/x_n]}$. By Proposition 41, $|e|_{\rho_T[[y_1]/x_1]\cdots[[y_n]/x_n]} = |e[y_1/x_1] \cdots [y_n/x_n]|_{\rho_T} = \{[e[y_1/x_1] \cdots [y_n/x_n]]\}$; similarly $|e'|_\rho = \{[e'[y_1/x_1] \cdots [y_n/x_n]]\}$. Then, $\vdash_\lambda e[y_1/x_1] \cdots [y_n/x_n] = e'[y_1/x_1] \cdots [y_n/x_n]$, i.e., $[e[y_1/x_1] \cdots [y_n/x_n]] = [e'[y_1/x_1] \cdots [y_n/x_n]]$. Hence, $|e|_\rho = |e'|_\rho$. □

The only thing left is to prove Theorem 38. We have shown that $|e|_{\rho_T} = \{[e]\}$ for every $e \in \Lambda$, in Proposition 41. It remains to show that T validates (β), i.e., $|(\lambda x.\, e)\, e'|_\rho = |e[e'/x]|_\rho$ for all $\rho \in \mathit{VarVal}$, which follows immediately from Proposition 42. Note that we only need to consider valuations in *VarVal* because all variables in (β) are quantified over the sort *Var*.

8.2.1 Comparing Our Term Model T to the Classical Notion of Term Models in λ-Calculus. In the literature on λ-calculus, a *term model* [Barendregt 1984, Definition 5.2.11] is a λ-model (Example 6), where the underlying carrier set A is $[\Lambda]$, the application function is the application function over equivalence classes, and the two special constants are $k = [\lambda x.\, \lambda y.\, x]$ and $s = [\lambda x.\, \lambda y.\, \lambda z.\, (xz)(yz)]$; we will denote this λ-model as A and call it a *classical term model*, to not confuse it with our term model T. Clearly, T and A represent different approaches to capture λ-expressions. While A uses the name-free, combinators approach, where λ is handled by *abstraction elimination*, our term model T gives an explicit and constructive interpretation to λ, as shown in Eq. (9).

8.2.2 The Representabiltiy Problem. There has been a long-standing, concerning and open problem in the study of λ-calculus, called the *representability problem* [Berline 2006, pp. 8], which asks if a given class of λ-calculus models is *representationally complete*, in the sense that there exists a model in the given class such that any two expressions e_1 and e_2 are provably equal if and only if they are interpreted as the same element/value in that model. Representability completeness indicates that a class of λ-calculus models is sufficient in capturing the formal reasoning in λ-calculus, so one may *reduce* the study of formal reasoning in λ-calculus to the study of models, where more mathematical tools and techniques can be applied. Hence, *reduction* is the main motivation.

λ-calculus models are broadly divided into *syntactic models* and *non-syntactic models* [Manzonetto 2008, pp. 13], depending on whether their construction is based on the syntax and provability of λ-calculus or not. All the classical term models in λ-calculus, as well as our particular matching logic term model in Section 8.2, are syntactic models. Syntactic models are often representationally complete, but studying them tends to be as hard as studying the syntax and formal reasoning directly, and thus the reduction to syntactic models usually does not help simplify the study of

λ-calculus. Thus, for decades researchers have been searching for and studying sub-classes of non-syntactic concrete ccc models, hoping they are also representationally complete. So far, three main such sub-classes have been identified, known as the *main semantics* of λ-calculus: Scott's continuous semantics [Scott 1972], Berry's stable semantics [Berry 1978; Girard 1986], and Bucciarelli-Ehrhard strongly stable semantics [Bucciarelli and Ehrhard 1993]. The representability problem for the main semantics (and their sub-classes) has remained largely open as of today, except for some negative results proved for some sub-classes (e.g., graph models [Bucciarelli and Salibra 2004]).

Theorem 38 shows that the class of matching logic models of Γ^λ is representationally complete, positively answering the representability problem for our matching logic semantics of λ-calculus. Our proof does not rely on any known results about the representational completeness of any existing semantics; instead, it is entirely based on the model theory of matching logic, which is not specific to λ-calculus but which allows for an appropriate axiomatization of λ-calculus as a theory that is hereby endowed with the desired representationally complete models automatically. We can push Theorem 38 even further to any equational extensions of λ-calculus, known as *λ-theories*. Indeed, the definition of the equivalence class $[e]$ as the set of $\alpha\beta$-equivalent expressions of e, has not been critical in the proof of Theorem 38, and the conclusion still holds if we consider any equivalence class $[e]$ that includes the basic $\alpha\beta$-equivalence. Therefore, we conclude that the matching logic definition of λ-calculus is representationally complete for *all λ-theories*.

Although we do not solve any of the existing open problems, our work suggests the matching logic can be a viable alternative to the existing λ-calculus models within the main semantics. The matching logic models are as good as the existing models for λ-calculus in terms of theoretical properties w.r.t. formal reasoning and semantics, yet unlike the existing models, they are general in the sense that they are not crafted specifically for λ-calculus, but are obtained from the matching logic theory Γ^λ. We give a general solution for all the binders, which for λ-calculus is as good as the state of the art, considering *both* the proof-theoretic *and* the model-theoretic aspects.

9 DEFINING BINDERS IN OTHER LOGICAL SYSTEMS USING MATCHING LOGIC

We showed how to capture the binder λ in matching logic as the following notation (Eqs. (7)-(8)):

$$\lambda x.\, e \equiv \mathsf{lambda}\ [x{:}\mathit{Var}]\ e \tag{11}$$

We defined a matching logic theory, Γ^λ (shown in Fig 5), and proved the conservative extension theorem for λ-calculus, Eq. (5). In this section we show that our approach is not specific to λ-calculus. We provide evidence that matching logic can serve as a general approach to dealing with binders. We will show how to use patterns similar to Eq. (11) to define the binders in a variety of logical systems, including System F [Girard 1972; Reynolds 1974], pure type systems [Barendregt 1993], π-calculus [Milner et al. 1992], and more, and prove a corresponding conservative extension theorem for each of them. To do that, several challenges need to be solved.

A first challenge is that binders can have more complex binding behavior than in λ-calculus; see Fig. 6. For example, $\lambda x{:}e_1.\, e_2$ in System F binds x within e_2, but not in e_1; $\mathsf{Inp}(x, y, e)$ in π-calculus has the binding variable in the second position (i.e., y), and not the first position. We deal with this binding behavior by desugaring to binders whose binding variable is their first argument and is bound within the second argument only; that is, we desugar an arbitrary binder to a binder of the form $b(x, e_1, \ldots, e_n)$, where x is bound in e_1 but not in $e_2, \ldots, e_n$. Clearly, this desugaring process is just a sequence of argument swappings. Then, we further desugar $b(x, e_1, \ldots, e_n)$ to $b'(b''(x, e_1), e_2, \ldots, e_n)$, where b' is a (binding-free) symbol and b'' is a binder that binds x to e_1, just like λ in λ-calculus. Finally, we define $b''(x, e_1)$ as the following syntactic sugar:

$$b''(x, e) \equiv \mathsf{retraction}_b\ [x{:}\mathit{Var}]\ e \tag{12}$$

Constructs	Binding Behavior	Meaning	Origins
$\lambda x.\, e$	binding x into e	function abstraction	λ-calculus
$\lambda x{:}e_1.\, e_2$	binding x into e_2	function abstraction	System F
$\lambda t.\, e$	binding t into e	type abstraction	System F
$\Pi t.\, e$	binding t into e	Π-type constructor	System F
$\lambda x{:}e_1.\, e_2$	binding x into e_2	function abstraction	Pure type system
$\pi x{:}e_1.\, e_2$	binding x into e_2	type abstraction	Pure type system
$\mathsf{Inp}(x, y, e)$	binding y into e	input process	π-calculus
$\nu y.\, e$	binding y into e	new process name creation	π-calculus
$\mathsf{Bout}(e_1, x, y, e_2)$	binding y into e_2	bound output transition	π-calculus
$\mathsf{Inp}(e_1, x, y, e_2)$	binding y into e_2	input transition	π-calculus

Fig. 6. Some example binding constructs and their binding behavior in logical systems.

in the same way as in Eq. (11), except that here we use a new retraction symbol $\mathsf{retraction}_b$ that is specific to the binder b. Each binder has its own retraction symbol, but the other infrastructure symbols, such as products, powersets, and the binding notation $[x{:}\mathit{Var}]\, e$, are the same. From now on, we will only consider binders $b(x, e)$ that bind x within e, for technical convenience.

A second challenge is that logical systems featuring bindings are very different from each other, in terms of the kinds of *logical reasoning* that is carried out in them. For example, System F derives *typing judgments* $\Gamma \triangleright e_1{:}e_2$ to mean that e_1 has type e_2 under typing environment Γ; π-calculus derives *transitions* $e_1 \xrightarrow{act} e_2$ to mean that process e_1 transits by action act to process e_2. It is tedious and non-systematic to consider these logical systems *separately*, because we would need to capture their specific logical reasoning and prove the conservative extension theorem for each of them, more or less similarly to the syntax-based proof in Section 8.

Remark 43. The current $\mathbb{K}$ framework implementation provides a "binder" attribute, which allows one to define a language construct that binds all variables occurring in its first argument within its other arguments. The results demonstrated in this paper, particularly this section, will be used to improve $\mathbb{K}$ and let it support binders with more complex binding behaviors. The reader who is interested in seeing examples about the current $\mathbb{K}$ support for binders may look at [K Team 2020], where the "binder" attribute is used to define the syntax of λ-calculus.

To capture the various logical systems featuring bindings more systematically, we employ a parametric framework for binders, called *term-generic logic* [Popescu and Roşu 2015] (TGL). TGL is a parametric variant of FOL, whose syntax is parametric on a set of (generic) terms that are not constructed from constants and functions, but defined axiomatically. When we instantiate TGL with the term syntax of a given system (e.g., λ-calculus, System F, π-calculus, etc), it becomes a (first-order) *meta-logic* of that system and can be used to specify and reason about its meta-properties. Using TGL, we give a systematic treatment of binders in the various logical systems. We will capture TGL in matching logic and prove a conservative extension theorem for TGL, from which the conservative extension theorems for the other logical systems follow as corollaries.

Why not use TGL directly then, but instead use matching logic? There are two reasons. Firstly, TGL in its full generality is not implementable, because it does not deal with any concrete syntax of binders. Its notion of (generic) terms is given axiomatically and needs to be instantiated, which is what we will do in Section 9.1, where we instantiate TGL to bridge matching logic and other logical systems with binders. The second reason is that TGL is a logic specifically designed for binders, while matching logic serves as the unifying logical foundation for the $\mathbb{K}$ framework, as discussed in Section 1 and other places in the paper. Therefore, matching logic supports reasoning in many mathematical domains other than binders, and thus it is more practical than TGL.

We next first introduce TGL in Section 9.1 and then its matching logic definition in Section 9.2.

9.1 Term-Generic Logic (TGL) Preliminaries

TGL [Popescu and Roşu 2015] is a variant of many-sorted FOL whose syntax is parametric in a (generic) term set that is defined axiomatically. In TGL, any set T exporting two operations—free variables $\mathrm{FV}(e)$ and capture-free substitution $e[e'/x]$—and satisfying the conditions in [Popescu and Roşu 2015, Definition 2.1], forms a generic term set. TGL formulas are built like in FOL, from predicates $\pi(e_1, \ldots, e_n)$, equations $e_1 = e_2$, and standard connectives $\wedge, \neg, \exists$, except that $e_1, \ldots, e_n$ are generic terms, that is, arbitrary elements in T. The metatheory of TGL, including its semantics and models, terms/formulas interpretation, proof system, and, importantly, a soundness and completeness theorem, have been studied and presented in detail in [Popescu and Roşu 2015].

For concreteness, we will not introduce TGL in its full generality. Instead, we instantiate TGL with a concrete, constructive term syntax with binders (defined below) and introduce the metatheory of that TGL instance. From the discussion at the beginning of Section 9, this term syntax is sufficient to capture the binders in various logical systems with more complex bindings (Fig. 6).

Definition 44. A *binder syntax* is a tuple (S, V, F, B), where

(1) S is a set of *sorts* denoted s, r, possibly with subscripts; we use $\bar{s} \in S^*$ to mean a list of sorts;
(2) $V = \{V_s\}_{s \in S}$ is a sort-wise disjoint family of *variables* denoted $x{:}s$, $y{:}s$, etc;
(3) $F = \{F_{\bar{s},r}\}_{\bar{s} \in S^*, s \in S}$ is a family of *many-sorted operations* of argument sorts $\bar{s}$ and result sort r;
(4) $B = \{B_{s,s',r}\}_{s,s',r \in S}$ is a family of *binders*, where $b(x{:}s, e)$ binds $x{:}s$ to e (of sort s') and returns a term of sort r, for each $b \in B_{s,s',r}$.

We use *TGLTerm* to denote the set of terms generated by the above syntax, where free variables, α-equivalence, and capture-free substitution are defined in the usual way. We omit sorts when they can be inferred. Note that when $B = \emptyset$, rules (1)-(3) generate the standard FOL terms.

Remark 45. *TGLTerm* forms a TGL generic term set in [Popescu and Roşu 2015, Definition 2.1].

TGL formulas, interpretations, validity, and provability are defined in the standard way, (almost) identical to FOL, except that terms are interpreted simultaneously instead of constructively. Specifically, the interpretation of compound term $f(e)$ is not defined from the interpretation of its sub-term e,[6] but instead we have a Henkin-style definition for term interpretations:

Definition 46 ([Popescu and Roşu 2015, Section 2]). For a given set of *many-sorted predicates* $\Pi = \{\Pi_{\bar{s}}\}_{\bar{s} \in S^*}$, we define the set *TGLForm* of *TGL formulas* by the following grammar:

$$\varphi ::= e_1 = e_2 \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \exists x{:}s'.\, \varphi \mid \pi(e_1, \ldots, e_n) \text{ for } \pi \in \Pi_{s_1 \cdots s_n} \text{ and } e_i \text{ has sort } s_i \text{ for all } i$$

Let $A = \{A_s\}_{s \in S}$ be an S-indexed carrier set. A *TGL valuation* $\rho : V \to A$ is a function such that $\rho(x{:}s) \in A_s$ for every $s \in S$ and $x{:}s \in V_s$. Let *TGLVal* be the set of all TGL valuations. A *TGL model* $(\{A_s\}_{s \in S}, \{A_e\}_{e \in TGLTerm}, \{A_\pi\}_{\pi \in \Pi})$ has a Henkin-style definition as follows:

(1) $A_s \neq \emptyset$ for every $s \in S$.
(2) $A_e : TGLVal \to A_s$, where s is the sort of e, such that for any $x{:}s, e, e', \rho$:
   (a) $A_{x:s}(\rho) = \rho(x{:}s)$.
   (b) $A_{e[e'/x:s]}(\rho) = A_e(S_{e',x:s}(\rho))$, where $S_{e',x:s}(\rho)$ is the TGL valuation such that $S_{e',x:s}(\rho)(x{:}s) = A_{e'}(\rho)$ and $S_{e',x:s}(\rho)(y{:}s') = A_{y:s'}(\rho)$ for any $y{:}s' \not\equiv x{:}s$.
(3) $A_\pi \subseteq A_{s_1} \times \cdots \times A_{s_n}$ for every $\pi \in \Pi_{s_1 \ldots s_n}$.

We let $A_\varphi \subseteq TGLVal$ for $\varphi \in TGLForm$ be the set of valuations under which φ holds, defined as:

(1) $\rho \in A_{e_1 = e_2}$ iff $A_{e_1}(\rho) = A_{e_2}(\rho)$;

[6] TGL in its full generality as in [Popescu and Roşu 2015] does not even have a notion of compound terms or sub-terms.

(2) $\rho \in A_{\pi(e_1,\ldots,e_1)}$ iff $(A_{e_1}(\rho),\ldots,A_{e_n}(\rho)) \in A_\pi$;
(3) $\rho \in A_{\varphi_1 \wedge \varphi_2}$ iff $\rho \in A_{\varphi_1}$ and $\rho \in A_{\varphi_2}$;
(4) $\rho \in A_{\neg\varphi}$ iff $\rho \notin A_\varphi$;
(5) $\rho \in A_{\forall x:s.\,\varphi}$ iff $\rho[a/x{:}s] \in A_\varphi$ for every $a \in A_s$.

TGL has a sound and complete Gentzen proof system [Popescu and Roşu 2015, Figs. 1-2], which derives sequents of the form $E \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ for $E, \Delta_1, \Delta_2 \subseteq \mathit{TGLForm}$, which intuitively means that under TGL theory E, the *conjunction* of the formulas in Δ_1 implies the *disjunction* of the formulas in Δ_2. It is required that E contains formulas without free variables, and Δ_1, Δ_2 are finite sets containing formulas with finitely many free variables; these requirements are needed for TGL's completeness theorem and all TGL sequents considered in this paper satisfy these requirements.

Definition 47 ([Popescu and Roşu 2015, Sections 2-3]). For a TGL model A and $\varphi \in \mathit{TGLForm}$, we write $A \models_{\mathsf{TGL}} \varphi$ iff $A_\varphi = \mathit{TGLVal}$. We write $A \models_{\mathsf{TGL}} E$ iff $A \models_{\mathsf{TGL}} \varphi$ for all $\varphi \in E$. *TGL validity* $E \models_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ is defined as $\bigcap_{\varphi\in\Delta_1} A_\varphi \subseteq \bigcup_{\varphi\in\Delta_2} A_\varphi$, for all $A \models_{\mathsf{TGL}} E$. *TGL provability* $E \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ is defined by the Gentzen proof system of TGL in the usual way.

Theorem 48 ([Popescu and Roşu 2015, Theorem 3.1]). *Under the above requirements about E, Δ_1, Δ_2, we have $E \models_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$ if and only if $E \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$.*

9.2 Defining Term Generic Logic in Matching Logic

In this section we define a matching logic theory Γ^{TGL} and introduce notations such that all TGL terms and formulas are well-formed matching logic patterns. We show that Γ^{TGL} is a conservative extension of TGL, by proving the following equivalence theorem.

Theorem 49. *Under the notations in Theorem 48, the following are equivalent: (1) $(\Gamma^{\mathsf{TGL}} \cup E) \vdash \bigwedge \Delta_1 \longrightarrow \bigvee \Delta_2$. (2) $(\Gamma^{\mathsf{TGL}} \cup E) \vDash \bigwedge \Delta_1 \longrightarrow \bigvee \Delta_2$; (3) $E \models_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$; (4) $E \vdash_{\mathsf{TGL}} \Delta_1 \triangleright \Delta_2$; Here, $\bigwedge \Delta_1$ is the conjunction of patterns in Δ_1 and $\bigvee \Delta_2$ is the disjunction of patterns in Δ_2.*

Thanks to the mathematical instruments and notations that we have introduced in Section 4, the definition of Γ^{TGL} is straightforward. The many-sorted binder syntax (Definition 44) and TGL terms are captured by defining sorts and many-sorted functions as in Section 4.2, and defining binders as in Eq. (12). TGL formulas, except $\pi(e_1,\ldots,e_n)$, are captured by matching logic's derived connectives (Fig. 1) and equality (Definition 13). Predicate $\pi(e_1,\ldots,e_n)$ for $\pi \in \Pi_{s_1\cdots s_n}$, is captured by defining a matching logic symbol π and the following axiom:

$$\text{(Predicate)} \quad \forall x_1{:}s_1.\ \ldots \forall x_n{:}s_n.\ (\pi\, x_1\ \cdots\ x_n = \top) \vee (\pi\, x_1\ \cdots\ x_n = \bot) \tag{13}$$

which specifies that π returns either $\top$ or $\bot$, i.e., it indeed builds predicate patterns. Without such axioms, $\pi\, x_1 \cdots x_n$ could be any subset. Let Γ^{TGL} contain all the above definitions and notations.

Remark 50. Under the above notations and axioms, all TGL terms are matching logic functional patterns (Section 3.2.2) and all TGL formulas are matching logic predicate patterns (Section 3.2.1).

Theorem 49 is proved using a model-based approach similar to Fig. 4. Here we explain the only nontrivial proof step, which is (2) $\Longrightarrow$ (3). This is proved by constructing a matching logic model M^A from any given TGL model A, such that all TGL terms and formulas are interpreted the same in M^A and A, i.e., $|e|_\rho = \{A_e(\rho)\}$ for every $e \in \mathit{TGLTerm}$; $|\varphi|_\rho = M^A$ whenever $\rho \in A_\varphi$, and $|\varphi|_\rho = \emptyset$, whenever $\rho \notin A_\varphi$, for every $\varphi \in \mathit{TGLForm}$.

Remark 51. Using TGL and Theorem 49, we obtain a systematic proof of the conservative extension theorems and deductive completeness theorems for all logical systems that have been defined in TGL and studied in [Popescu and Roşu 2015, Section 4] and [Popescu and Roşu 2013, Section 4],

including System F [Girard 1972; Reynolds 1974] (both the typing and reduction versions), λ-calculus (including the untyped [Church 1941], sub-typed [Cardelli et al. 1994], illative [Barendregt 1984], and linear versions [Girard 1987; Lincoln and Mitchell 1992]), pure type systems [Barendregt 1993], and π-calculus [Milner et al. 1992]. The systematic proof works as follows. For each logical system L, its set of terms $Term_L$ can be captured by a binder syntax using the desugaring discussed at the beginning of Section 9. The proof/type system of L that derives sequents of the form $\vdash_L \Phi$ is captured by a set of TGL axioms E^L, where each axiom corresponds to one type/proof rule of L [Popescu and Roşu 2015]. An *adequacy theorem* is also proved there for each L, stating that $\vdash_L \Phi$ iff $E^L \vdash_{\mathsf{TGL}} \Phi^{\mathsf{TGL}}$, where Φ^{TGL} (of the form $\Delta_1^\Phi \triangleright \Delta_2^\Phi$) is the corresponding TGL encoding of the L-sequent Φ. Let $\Gamma^L = \Gamma^{\mathsf{TGL}} \cup E^L$ be the matching logic theory that captures L, and $\Phi^{\mathsf{ML}} = \bigwedge \Delta_1^\Phi \to \bigvee \Delta_2^\Phi$ be the matching logic encoding of Φ. By Theorem 48, we have that $\vdash_L \Phi$ in L, iff $E^L \vdash_{\mathsf{TGL}} \Phi^{\mathsf{TGL}}$ in TGL, iff $\Gamma^L \vdash \Phi^{\mathsf{ML}}$ in matching logic, iff $\Gamma^L \vDash \Phi^{\mathsf{ML}}$ in matching logic. Hence, Γ^L is a conservative extension of L and the class of matching logic models of Γ^L is complete with respect to L.

Remark 52. Note that the term "consistency" has different meanings in different contexts. In type systems, inconsistency means the ability to prove any typing judgments $t : \tau$. Similarly, in λ-calculus or other equational logic theories, inconsistency means the ability to prove any equations $e_1 = e_2$. However, in matching logic (and also FOL), inconsistency means the ability to prove logical false $\bot$. Thus, inconsistency for classical logics such as matching logic is stricter than that for type systems and λ-calculus. For example, if T is a PTS that contains the typing axiom $Type : Type$, then T is inconsistent [Martin-Löf 1998], but its matching logic theory Γ^T is still a consistent matching logic theory and has a model that interprets the typing relation $\_:\_$ as the total relation on all PTS terms.

10 FUTURE WORK

Inductive Reasoning. An important direction for future work is to investigate *inductive reasoning* on terms with binders. We use λ-calculus as an example but the discussion applies to all binders.

The set of λ-expressions Λ is an inductive structure. This means that Λ is the smallest set closed under variables, application, and abstraction, and it admits the *principle of inductive reasoning*, which can be intuitively expressed by the following formula (this should be understood informally; in particular, the inductive hypothesis for $\lambda x.\, e$ in (‡) takes various forms in the literature; e.g., [Pitts 2003, pp. 21] uses the ν-quantifier on x, meaning that there exists $x : Var$ such that x is not free in e, while [Aydemir et al. 2008, pp. 5] uses $\forall$-quantifier to quantify all $x : Var$ that are not free in e):[7]

$$\begin{aligned}
&\forall P.\ (\forall x : Var.\ x \in P) \\
&\quad \wedge\ (\forall e : Exp.\ \forall e' : Exp.\ e \in P \wedge e' \in P \to (e\ e') \in P) \\
&\quad \wedge\ (\forall e : Exp.\ e \in P \to \forall x : Var.\ \lambda x.\, e \in P) \qquad (\ddagger) \\
&\quad \to \forall e : Exp.\ e \in P
\end{aligned}$$

where $P \subseteq \Lambda$ is a property of λ-expressions. Inductive reasoning on terms with binders is known to be hard when the binding behavior of λ yields bindings in the meta-language, making it difficult to write pattern-matching style recursive definitions and reasoning (see, e.g., [Gabbay 2000]). For example, if we try to parse the above inductive principle as a matching logic pattern, we will notice that $\forall x : Var$ in (‡) binds *nothing*—x is already bound in $\lambda x.\, e$.

There is relevant research on this topic, e.g., [Chlipala 2008; Despeyroux et al. 1995; Schürmann et al. 2001] for HOAS approaches and [Pitts 2013; Urban 2008] for nominal induction and recursion, which we will investigate and reconcile within matching logic. We believe that matching logic is particularly suitable for defining such inductive principles. Indeed, matching logic allows set variables, which are effectively universally quantified in formulas. Therefore, the second-order

[7] [Aydemir et al. 2008] gives credits to [McKinna and Pollack 1993] and mentions that it can be used in many other logics.

quantification $\forall P$ in the inductive principle above can be effectively captured in matching logic by simply dropping the $\forall P$ quantifier and letting the set variable P stay free in the formula.

Replacing Axiom Schemas with Axioms. The matching logic theory Γ^λ for λ-calculus (Section 6) includes *axiom schema* (β) with meta-variables x, e, e', the same as the original λ-calculus. Thus, Γ^λ is a faithful definition of λ-calculus that captures it *as is*. This was intended and desired, because we believe that as a unifying logic for semantic frameworks (like $\mathbb{K}$), matching logic should allow us to define logics, calculi and languages as a mirror of the original, without any encodings or translations except for defining the necessary mathematical instruments and convenient notations. For practical reasons, it is also useful to define λ-calculus (and other binders) using axioms (not schemas) and normal variables (not meta-variables), as in nominal logic axiom (β in Nominal Logic) and HOAS (e.g., Twelf definition red-beta), both shown in Section 2. Thus, one way to eliminate schemas and meta-variables is to follow nominal and/or HOAS approaches methodologically, as explained in Remark 1; that is, we define nominal logic or HOAS in matching logic as theories and notations, and then define binders through them. However, matching logic also gives us an opportunity for alternative definitions. Below, we will show at a high level one example. Studying such alternative encodings of calculi is interesting and practical, but will be addressed in other places.

Recall that $\lambda x.\, e \equiv \mathsf{lambda}\ (\mathsf{intension}\ \exists x{:}\, Var.\, \langle x, e\rangle)$, where $(\mathsf{intension}\ \exists x{:}\, Var.\, \langle x, e\rangle)$ denotes the graph of $x \mapsto e$ as an element of sort $2^{Var \otimes Exp}$. As pointed out in Section 6, not all elements of sort $2^{Var \otimes Exp}$ represent a graph, so we may identify and axiomatize a subsort *Graph* of $2^{Var \otimes Exp}$ that includes precisely all graphs. And thus, the schema (β) can be replaced by the following axiom:

$$(\beta, \text{Not a Schema}) \quad \forall g{:}\, Graph.\, \forall e'{:}\, Exp.\, (\mathsf{lambda}\ g)\ e' = \mathsf{graph\text{-}lookup}\ g\ e'$$

where g and e' are normal variables and graph-lookup is axiomatized as the graph lookup operation.

11 CONCLUSION

In this paper, we used (a functional variant of) matching logic to define binders in various logical systems. The binding behavior of binders in the object-level systems is directly inherited from the built-in binder $\exists$ in matching logic. We demonstrated our approach directly by defining λ-calculus as a matching logic theory, and indirectly by capturing term-generic logic (TGL); the latter yields matching logic definitions for many logical systems that feature bindings that were previously defined as TGL theories, including System F, pure type systems, π-calculus, etc. We proved the conservative extension theorems for all of these. We illustrated two proof methods: one based on models that is suitable for object-level systems that come equipped with models, and another based on syntax and proof derivations that is more involved but available even when the system lacks models. Our approach also yields *models* for the defined systems. For the systems discussed in the paper, the obtained models are complete w.r.t. logical reasoning, which follows from the conservative extension theorems. For λ-calculus, the models are representationally complete for all λ-theories, suggesting that matching logic is a promising alternative semantics for λ-calculus.

ACKNOWLEDGMENTS

We warmly thank the $\mathbb{K}$ Team for invaluable and continuous feedback on matching logic and its role as a foundation of $\mathbb{K}$, as well as for their creative yet hard work on turning theoretical results into practical tools. We also warmly thank James Cheney, Maribel Fernández, Andrei Popescu, and Thomas Tuegel for many insightful comments and concrete suggestions. We are indebted to the four anonymous reviewers, whose wit and dedication helped us improve the presentation. This work was supported in part by NSF CNS 16-19275. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.

REFERENCES

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

Areski Nait Abdallah. 1995. Partial first-order logic. In *The Logic of Partial Information*. Springer, Berlin, Heidelberg, Chapter 14, 425–452. https://doi.org/10.1007/978-3-642-78160-5_14

Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. 2018. Nominal C-unification. In *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'17) (Lecture Notes in Computer Science)*, Vol. 10855. Springer International Publishing, Namur, Belgium, 235–251.

Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. 2016. Nominal narrowing. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD'16) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:17. https://doi.org/10.4230/LIPIcs.FSCD.2016.11

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)* (San Francisco, California, USA). ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Henk Barendregt. 1984. *The lambda calculus, its syntax and semantics*. College Publications, King's College London, Strand, London WC2R 2LS, UK. https://doi.org/10.2307/2274112

Henk Barendregt. 1993. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Vol. 2, background: computational structures. Oxford University Press, UK, Chapter 2, 117–309. https://doi.org/10.5555/162552.162561

John Bell and Moshe Machover. 1977. *A course in mathematical logic*. North Holland, Amsterdam, Netherlands.

Chantal Berline. 2000. From computation to foundations via functions and application: the λ-calculus and its webbed models. *Theoretical Computer Science* 249, 1 (2000), 81–161. https://doi.org/10.1016/S0304-3975(00)00057-8

Chantal Berline. 2006. Graph models of λ-calculus at work, and variations. *Mathematical Structures in Computer Science* 16, 2 (2006), 185–221. https://doi.org/10.1017/S0960129506005123

Gilles Bernot, Michel Bidoit, and Christine Choppy. 1986. Abstract data types with exception handling: An initial approach based on a distinction between exceptions and errors. *Theoretical Computer Science* 46 (1986), 13–45. https://doi.org/10.1016/0304-3975(86)90019-8

Gérard Berry. 1978. Stable models of typed λ-calculi. In *Automata, Languages and Programming*. Springer, Berlin, Heidelberg, 72–89. https://doi.org/10.5555/646232.682069

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal logic*. Cambridge University Press, One Liberty Plaza, New York, NY.

C. J. Bloo. 1997. *Preservation of termination for explicit substitution*. Ph.D. Dissertation. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR499858

Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, Mumbai, India, 445–456.

Antonio Bucciarelli and Thomas Ehrhard. 1993. A theory of sequentiality. *Theoretical Computer Science* 113, 2 (1993), 273–291. https://doi.org/10.1016/0304-3975%2893%2990005-E

Antonio Bucciarelli and Antonino Salibra. 2004. The sensible graph theories of lambda calculus. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. IEEE, Turku, Finland, 276–285. https://doi.org/10.1109/LICS.2004.1319622

Peter Burmeister. 1993. Partial algebras—an introductory survey. In *Algebras and orders*. NATO ASI Series, Vol. 389. Springer, Dordrecht, Netherlands, 1–70. https://doi.org/10.1007/978-94-017-0697-1_1

Luca Cardelli. 1996. Type systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 263–264.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1 (1994), 4–56. https://doi.org/10.1006/inco.1994.1013

Xiaohong Chen and Grigore Roşu. 2019. Matching μ-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*. IEEE, Vancouver, Canada, 1–13. https://doi.org/10.1109/LICS.2019.8785675

Xiaohong Chen and Grigore Roşu. 2020. *A general approach to define binders using matching logic*. Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/106608

James Cheney. 2006. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic* 71, 1 (2006), 299–320.

James Cheney. 2014. A simple sequent calculus for nominal logic. *Journal of Logic and Computation* 26, 2 (2014), 699–726. https://doi.org/10.1093/logcom/exu024

James Cheney, Michael Norrish, and René Vestergaard. 2012. Formalizing adequacy: a case study for higher-order abstract syntax. *Journal of Automated Reasoning* 49, 2 (2012), 209–239. https://doi.org/10.1007/s10817-011-9221-6

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. ACM, British Columbia, Canada, 143–156.

Alonzo Church. 1941. *The calculi of lambda-conversion.* Princeton University Press, Princeton, New Jersey, USA. https://doi.org/10.2307/2267126

Bruno Courcelle and Joost Engelfriet. 2012. *Graph structure and monadic second-order logic: a language-theoretic approach.* Vol. 138. Cambridge University Press, England, UK. https://doi.org/10.1017/CBO9780511977619

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19).* ACM, Phoenix, Arizona, USA, 1133–1148.

Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. 1995. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications.* Springer, Berlin, Heidelberg, 124–138. https://doi.org/10.1007/BFb0014049

Erwin Engeler. 1981. Algebras and combinators. *Algebra Universalis* 13, 1 (1981), 389–392. https://doi.org/10.1007/BF02483849

Amy Felty and Alberto Momigliano. 2012. Hybrid, a definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning* 48, 1 (2012), 43–105. https://doi.org/10.1007/s10817-010-9194-x

Marcelo Fiore and Chung-Kil Hur. 2010. Second-order equational logic (extended abstract). In *Computer Science Logic*, Anuj Dawar and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–335.

Marcelo Fiore and Ola Mahmoud. 2010. Second-order algebraic theories. In *Mathematical Foundations of Computer Science 2010*, Petr Hliněný and Antonín Kučera (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–380.

M. Fiore, G. Plotkin, and D. Turi. 1999. Abstract syntax and variable binding. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158).* IEEE, Trento, Italy, 193–202.

Murdoch Gabbay and James Cheney. 2004. A sequent calculus for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04).* IEEE, Washington, DC, USA, 139–148. https://doi.org/10.5555/1018438.1021850

Murdoch J. Gabbay and Michael J. Gabbay. 2017. Representation and duality of the untyped λ-calculus in nominal lattice and topological semantics, with a proof of topological completeness. *Annals of Pure and Applied Logic Volume* 168, 3 (Oct. 2017), 501–621. https://doi.org/10.1016/j.apal.2016.10.001

M. Gabbay and A. Pitts. 1999. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Symposium on Logic in Computer Science (LICS'19).* IEEE, Trento, Italy, 214–224. https://doi.org/10.1109/LICS.1999.782617

Murdoch J. Gabbay. 2000. *A theory of inductive definitions with α-equivalence: semantics, implementation, programming language.* Ph.D. Dissertation. DPMMS and Trinity College, Cambridge University.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2012. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning* 49, 2 (2012), 241–273. https://doi.org/10.1007/s10817-011-9218-1

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Paris Diderot University, Paris, France.

Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159–192. https://doi.org/10.1016/0304-3975(86)90044-7

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

M. Gogolla, K. Drosten, U. Lipeck, and H.-D. Ehrich. 1984. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Computer Science* 34, 3 (1984), 289–313. https://doi.org/10.1016/0304-3975(84)90056-2

Joseph Goguen and José Meseguer. 1992. Order-sorted algebra, part I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 2 (1992), 217–273. https://doi.org/10.1016/0304-3975(92)90302-V

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *J. ACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

Gisbert Hasenjaeger. 1953. Eine bemerkung zu Henkin's beweis für die vollständigkeit des prädikatenkalküls der ersten stufe. *The Journal of Symbolic Logic* 18, 1 (1953), 42–48. https://doi.org/10.2307/2266326

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).* ACM, Portland, OR, 336–345.

Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18).* IEEE, Oxford, UK, 204–217. http://jellopaper.org.

Roger Hindley and Giuseppe Longo. 1980. Lambda-calculus models and extensionality. *Mathematical Logic Quarterly* 26, 4 (1980), 289–310. https://doi.org/10.1002/malq.19800261902

K Team. 2020. K tutorials—λ-calculus. https://github.com/kframework/k/tree/master/k-distribution/tutorial/1_k/1_lambda/lesson_2.

Delia Kesner. 2009. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science* 5, 3 (2009), 1–29.

Jan Willem Klop. 1993. Term rewriting systems. In *Handbook of Logic in Computer Science*. Vol. 2, Background: computational structures. Oxford University Press, Inc., USA, Chapter 1, 1–116.

C. P. J. Koymans. 1982. Models of the lambda calculus. *Information and Control* 52 (1982), 306–332.

Jean Louis Krivine. 1993. *Lambda-calculus, types and models*. Ellis Horwood, USA.

Patrick Lincoln and John Mitchell. 1992. Operational aspects of linear lambda calculus. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS'92)*. IEEE, California, USA, 235–246. https://doi.org/10.1109/LICS.1992.185536

Leopold Löwenheim. 1915. Über möglichkeiten im relativkalkül. *Math. Ann.* 76, 4 (1915), 447–470.

Francisca Lucio-Carrasco and Antonio Gavilanes-Franco. 1989. A first order logic for partial functions. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*. Springer, Paderborn, Germany, 47–58. https://doi.org/10.1007/BFb0028972

Giulio Manzonetto. 2008. *Models and theories of lambda calculus*. Ph.D. Dissertation. Università Ca' Foscari di Venezia. https://tel.archives-ouvertes.fr/tel-00715207

Per Martin-Löf. 1998. *Twenty five years of constructive type theory*. Oxford Logic Guides Book, Vol. 36. Oxford University Press, Oxford, UK, Chapter An intuitionistic theory of types, 127–172. http://www.cse.chalmers.se/~peterd/papers/MartinLöf1972.pdf

Raymond C. McDowell and Dale A. Miller. 2002. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic* 3, 1 (2002), 80–136. https://doi.org/10.1145/504077.504080

James McKinna and Robert Pollack. 1993. Pure type systems formalized. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.). Springer, Berlin, Heidelberg, 289–305. https://doi.org/10.1007/BFb0037113

José Meseguer and Grigore Roşu. 2013. The rewriting logic semantics project: a progress report. *Information and Computation* 231 (Oct. 2013), 38–69. https://doi.org/10.1016/j.ic.2013.08.004 Invited paper at FCT 2011.

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes (part 1). *Information and Computation* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Timothy Nelson, Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. *On the finite model property in order-sorted logic*. Technical Report. Worcester Polytechnic Institute, Brown University.

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, Portland, OR, 346–356.

Lawrence C. Paulson. 1989. The foundation of a generic theorem prover. *Journal of Automated Reasoning* 5, 3 (1989), 363–397. https://doi.org/10.1007/BF00248324

Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf—a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE 99)*. Springer, Trento, Italy, 202–206. https://doi.org/10.1007/3-540-48660-7_14

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X

Andrew M. Pitts. 2005. Alpha-structural recursion and induction. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–34.

Andrew M. Pitts. 2013. *Nominal sets: names and symmetry in computer science*. Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/CBO9781139084673

Gordon Plotkin. 1972. *A set-theoretical definition of application*. Technical Report. University of Edinburgh.

Andrei Popescu and Grigore Roşu. 2013. *Term-generic logic (extended technical report)*. Technical Report. Technische Universitat Munchen, University of Illinois at Urbana-Champaign.

Andrei Popescu and Grigore Roşu. 2015. Term-generic logic. *Theoretical Computer Science* 577 (2015), 1–24. https://doi.org/10.1016/j.tcs.2015.01.047

Robert W. Quackenbush. 1988. Completeness theorems for universal and implicational logics of algebras via congruences. *Proc. Amer. Math. Soc.* 103, 4 (1988), 1015–1021. https://doi.org/10.2307/2047077

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*. Springer, Berlin, Heidelberg, 408–425. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE, Copenhagen, Denmark, 55–74.

Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (2017), 1–61. https://doi.org/10.23638/LMCS-13(4:28)2017

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Harold Schellinx. 1991. Isomorphisms and nonisomorphisms of graph models. *Journal of Symbolic Logic* 56, 1 (Oct. 1991), 227–249. https://doi.org/10.2307/2274916

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. 2001. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science* 266, 1 (2001), 1–57. https://doi.org/10.1016/S0304-3975(00)00418-7

Dana Scott. 1972. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*. Springer, Berlin, Heidelberg, 97–136. https://doi.org/10.1007/BFb0073967

Dana Scott. 1975a. Data types as lattices. *SIAM J. Comput.* 5, 3 (1975), 522–587. https://doi.org/10.1137/0205037

Dana Scott. 1975b. Some philosophical issues concerning theories of combinators. In *Proceedings of the International Symposium on λ-Calculus and Computer Science Theory*. Springer, Berlin, Heidelberg, 346–366. https://doi.org/10.1007/BFb0029537

Traian Florin Şerbănuţă and Grigore Roşu. 2012. A truly concurrent semantics for the K framework based on graph transformations. In *Proceedings of the 6th International Conference on Graph Transformation (ICGT'12)*. Springer, Bremen, Germany, 294–310.

Mark-Oliver Stehr. 2000. CINNI—a generic calculus of explicit substitutions and its application to λ- ς- and ϕ-calculi. *Electronic Notes in Theoretical Computer Science* 36 (2000), 70–92. https://doi.org/10.1016/S1571-0661(05)80125-2

Christian Urban. 2008. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (01 May 2008), 327–356. https://doi.org/10.1007/s10817-008-9097-2

Jonni Virtema, Jeremy Meyers, and Antti Kuusisto. 2013. Undecidable first-order theories of affine geometries. *Logical Methods in Computer Science* 9, 4 (2013), 1–23. https://doi.org/10.2168/LMCS-9(4:26)2013