A Logic Programming Language based on Binding Algebras

Makoto Hamana

Department of Computer Science, University of Gunma
hamana@cs.gunma-u.ac.jp

Abstract. We give a logic programming language based on Fiore, Plotkin and Turi's binding algebras. In this language, we can use not only first-order terms but also terms involving variable binding. The aim of this language is similar to Nadathur and Miller's λProlog, which can also deal with binding structure by introducing λ-terms in higher-order logic. But the notion of binding used here is finer in a sense than the usual λ-binding. We explicitly manage names used for binding and treat α-conversion with respect to them. Also an important difference is the form of application related to β-conversion, i.e. we only allow the form (M x), where x is a (object) variable, instead of usual application (M N). This notion of binding comes from the semantics of binding by the category of presheaves. We firstly give a type theory which reflects this categorical semantics. Then we proceed along the line of first-order logic programming language, namely, we give a logic of this language, an operational semantics by SLD-resolution and unification algorithm for binding terms.

1 Introduction

The notion of variable binding appears often in many formal systems, programming languages and logics. The three papers on abstract syntax with variable binding by Gabbay-Pitts, Fiore-Plotkin-Turi and Hofmann in LICS'99 gave clear mathematical semantics of binding [GP99,FPT99,Hof99]. In this paper we follow the approach by Fiore-Plotkin-Turi and proceed to go further; we give a type theory based on their semantics and apply it to a logic programming language which can treat terms involving variable binding.

We will briefly illustrate our binding logic programming language. As an example, consider symbolic differentiation defined by the predicate diff:

$$\begin{gathered}\mathsf{diff}([b]\mathsf{var}(a), [b]0)\\ \mathsf{diff}([b]\mathsf{var}(b), [b]1)\\ \forall f, f'.\ \mathsf{diff}([b]\mathsf{sin}(f), [b](\mathsf{cos}(f) \times f')) \Leftarrow \mathsf{diff}([b]f, [b]f')\end{gathered}$$

The predicate $\mathsf{diff}([b]f, [b]g)$ expresses that the differentiation of the function $[b]f$ is $[b]g$, where b is a parameter (bound variable). For this program, we can ask

N. Kobayashi and B.C. Pierce (Eds.): TACS 2001, LNCS 2215, pp. 243–262, 2001.

queries involving (existentially quantified) variables to obtain values for them, e.g.

$$? - \mathsf{diff}([d]\mathsf{var}(c), z)$$

which asks the differentiation of the function $[d]\mathsf{var}(c)$ (a constant function returning the free variable c) for the result of an existentially quantified variable z. Then, the system will return an answer substitution $z \mapsto [d]0$.

This is just a result of an application of the first clause of the program, but notice that α-renaming of bound variables to unify the query and program clause is automatically done. Another example is

$$\begin{array}{l} ? - \mathsf{diff}([b]\mathsf{sin}(x), [b](\mathsf{cos}(x) \times z)) \\ x \mapsto \mathsf{var}(b),\ z \mapsto 1 \quad ; \quad x \mapsto \mathsf{var}(a),\ z \mapsto 0 \end{array}$$

which has two different answers. Note also that by applying the first answer substitution $x \mapsto \mathsf{var}(b)$ to the query, the variable b becomes bound; the system does not do implicit α-renaming for avoiding capture of bound variables by substitution. More involved query is

$$? - \mathsf{diff}([d]([b]\mathsf{sin}(x)) \mathbin{@} c,\ [b](\mathsf{cos}(y) \times z))$$

and then there are three answers: $x \mapsto \mathsf{var}(d), y \mapsto \mathsf{var}(c), z \mapsto 0$; $x \mapsto \mathsf{var}(c), y \mapsto \mathsf{var}(c), z \mapsto 0$; $x \mapsto \mathsf{var}(b), y \mapsto \mathsf{var}(b), z \mapsto 1$. Here β-reduction is automatically performed during unification, while it cannot be used in case of first-order logic programming language. Although our binding logic programming language is a typed language, we omitted the type information in this example. A complete program will be given in Section 9.

This paper is organized into two parts. The first part Section 2-7 deals with a type theory for terms involving variable binding. The second part Section 8 and 9 gives a logic programming language based on the type theory defined in the first part. In Section 10, we give a comparison to related work.

2 Terms

We will give the definition of the language for our binding logic programs, which includes terms, type system and operations on the language. First, we define terms.

Variables and object variables. There are two kinds of variables in our language; usual variables $(x, y, \ldots)$ (as in first-order language) and *object variables* $(a, b, \ldots)$. Object variables are used for the names of bindings. The set of all object variables is denoted by OV. A *world* C is a sequence of object variables having no duplication of the same things.

Types. Types are defined by

$$\tau ::= \iota \mid \delta\tau$$

where ι is a base type and $\delta\tau$ is a higher type abstracted by an object variable (semantically $\delta\tau \cong (\mathsf{OV} \Rightarrow \tau)$ [FPT99]). The set of all types is denoted by Ty.

Signature. A signature Σ is a set of function symbols together with an arity function assigning to each function symbol f, a finite sequence of types as the source type and the type ι as the target type. This is denoted by $f : \tau_1, \ldots, \tau_n \to \iota$. In case of $n = 0$, the function symbol $f : \iota$ is regarded as a constant. Note that there are no higher type constants.

Terms. Terms are defined by

$$t ::= f(t_1, \ldots, t_n) \mid t@a \mid [a]t \mid \xi x \mid \mathsf{var}(a)$$
$$\xi ::= [a_1 := a'_1, \ldots, a_n := a'_n]$$

where $f \in \Sigma$ is a function symbol, a's are object variables. A term $t@a$ denotes an application, $[a]t$ an abstraction, $\mathsf{var}(a)$ an object variable, and ξx a variable where ξ is called an *(syntactic) object variable substitution*. Intuitive meaning of ξx is the following: a variable having suspended substitution for object variables and if x is substituted by some term, its suspension is released and the object variable substitution is applied. We may omit the identity assignment $a_i := a_i$ in this bracket, and the order of the assignment is not important. We often use the shorthand notation $[\boldsymbol{a} := \boldsymbol{a'}]$. In case of $\xi = []$, we just write x. A type judgment is of the form

$$\Gamma \vdash_C t : \sigma : A$$

where
- a *context* Γ is a set of variable declarations $\{x_1 : \tau_1 : A_1, \ldots, x_n : \tau_n : A_n\}$,
- a *world* C is a sequence of object variables,
- a *type* σ,
- a *stage* A is a set of object variables.

Moreover, it must satisfy the condition $\bigcup_i A_i \cup A \cup \mathsf{FOV}(t) \cup \mathsf{GBOV}(t) \subseteq \widetilde{C}$ where FOV and GBOV are defined below, and $\widetilde{C}$ is a set of object variables occurring in C. Namely, the world C is considered as a pool of all possible object variables used (in future) in the judgment. We will see that in the type system the order of bound variables in t respects the order of object variables in C.

Some operations on terms. Let t be a term. Then $\mathsf{FOV}(t)$ denotes a set of all *free object variables*, $\mathsf{GBOV}(t)$ denotes a set of all *globally bound object variables*, i.e. object variables occurring in []-binders, and $\mathsf{OBV}(t)$ denotes a set of all *outermost bound variables*. These are defined as follows:

$$\mathsf{GBOV}([\boldsymbol{a} := \boldsymbol{b}]x) = \{\boldsymbol{a}\},\ \mathsf{GBOV}(t@a) = \mathsf{GBOV}(t),\ \mathsf{GBOV}([a]t) = \mathsf{GBOV}(t) \cup \{a\},$$
$$\mathsf{GBOV}(F(t_1, \ldots, t_n)) = \mathsf{GBOV}(t_1) \cup \ldots \cup \mathsf{GBOV}(t_n);$$
$$\mathsf{FOV}([\boldsymbol{a} := \boldsymbol{b}]x) = \{\boldsymbol{b}\}, \quad \mathsf{FOV}(t@a) = \{a\}, \quad \mathsf{FOV}([a]t) = \mathsf{FOV}(t) - \{a\},$$
$$\mathsf{FOV}(F(t_1, \ldots, t_n)) = \mathsf{FOV}(t_1) \cup \ldots \cup \mathsf{FOV}(t_n);$$
$$\mathsf{OBV}([\boldsymbol{a} := \boldsymbol{b}]x) = \varnothing, \quad \mathsf{OBV}([a]t) = \mathsf{OBV}(t) \cup \{a\},$$
$$\mathsf{OBV}(F(t_1, \ldots, t_n)) = \mathsf{OBV}(t_1) \cup \ldots \cup \mathsf{OBV}(t_n),$$
$$\mathsf{OBV}(((\ldots(([a_1]\ldots[a_n]t)@b_1)@\ldots)@b_n) = \mathsf{OBV}(t).$$

In the definition of $\mathsf{OBV}(([a_1]\dots[a_n]t)@b_1@\dots@b_n)$, the term t may have more binders on the top, and since we do not have constants of higher types in this system, an application can always be expressed as this form. These functions are obviously extended to a function on sets and sequences of terms.

3 Object Variable Substitutions

The important idea of our language involving binding is the distinction between object variables and usual variables. This idea actually comes from Fiore-Plotkin-Turi's semantics of abstract syntax with binding in the category $Set^{\mathbb{F}}$, where $\mathbb{F}$ is a category of (abstract) object variables and substitutions (the precise definition below). So we need the notion of object variable substitutions in syntax and semantic levels. In this section, we define these matters. We start with some preliminary definition on categories required in semantics.

The category of presheaves [FPT99]. Let $\mathbb{F}$ be the category which has finite cardinals $n = \{1,\dots,n\}$ (n is possibly 0) as objects, and all functions between them as arrows $m \to n$. We may also call $n \in \mathbb{F}$ a stage. The category $\mathbb{F}$ is also considered as the free cocartesian category on one object, generated from an initial object 1 by an operation $(\_)+1$. From this viewpoint, we assume the following chosen coproduct structure

$$n \xrightarrow{\ \mathsf{old}_n\ } n+1 \xleftarrow{\ \mathsf{new}_n\ } 1$$

with $\mathsf{old}_n(i) = i$ $(1 \le i \le n)$ and $\mathsf{new}_n(*) = n+1$.

Let $\hat{\mathbb{F}}$ be the functor category $Set^{\mathbb{F}}$. The category $\hat{\mathbb{F}}$ is complete, cocomplete and cartesian closed. The functor (for stage extension) $\delta : \hat{\mathbb{F}} \to \hat{\mathbb{F}}$ is defined as follows: for $L \in \hat{\mathbb{F}}, n \in \mathbb{F}, \rho \in \mathrm{arr}\,\mathbb{F}$,

$$(\delta L)(n) = L(n+1); \quad (\delta L)(\rho) = A(\rho + \mathrm{id}_1)$$

and, for $f : L \to M \in \hat{\mathbb{F}}$, the map $\delta f : \delta L \to \delta M$ is given by

$$(\delta f)_n = f_{n+1} : L(n+1) \to M(n+1)\ (n \in \mathbb{F}).$$

Operations on sequences. We fix the notations for sequences: the bracket $\langle\dots\rangle$ is used for a sequence, $+\!\!+$ is the concatenation, ϵ is the empty sequence. Let B be a sequence of object variables. Then $\widetilde{B}$ denotes the set of object variables occurring in B. We write $B \trianglelefteq B'$ when B is a subsequence of B'. More precisely, let $(\widetilde{B}, \le_B)$ and $(\widetilde{B'}, \le_{B'})$ be the posets whose orders respect the orders of elements in B and B'. Then $B \trianglelefteq B'$ if and only if $\forall a_1, a_2 \in \widetilde{B}\ .\ a_1 \le_B a_2 \ \Rightarrow\ a_1 \le_{B'} a_2$. Let A be a set of object variables. A *sequentialization* of A with respect to a world C, denoted by $\overline{A}_C$, or simply $\overline{A}$ if C is clear from the context, is a sequence such that $\overline{A} \trianglelefteq C$ and $\widetilde{\overline{A}} = A$. The notation $|\_|$ is used for the number of object variables occurring in a set or sequence. Note that $|A| = |\overline{A}|$.

Object variable substitutions. Let $A = \langle a_1, \ldots, a_m \rangle$ and $B = \langle b_1, \ldots, b_n \rangle$ be sequences of object variables (which need not to be subsequences of a given world). A *(semantic) object variable substitution* $\rho : A \to B$ is a triple $(|\rho|, A, B)$ where $|\rho| : |A| \to |B|$ is an arrow in $\mathbb{F}$. Then we can define a function $\widetilde{\rho} : \widetilde{A} \to \widetilde{B}$ such that

$$\widetilde{\rho}(a_i) \triangleq b_{|\rho|(i)} \text{ for each } i \in \{1, \ldots, m\}.$$

By abuse of notation, we often write $\rho(a)$ as the value of $\widetilde{\rho}(a)$. The ρ gives rise the natural transformation $\hat{\rho} : \delta^{|A|} \dot{\to} \delta^{|B|} \in \text{arr}\, [\hat{\mathbb{F}}, \hat{\mathbb{F}}]$ whose component $\hat{\rho}_L : \delta^{|A|} L \dot{\to} \delta^{|B|} L \in \text{arr}\, \hat{\mathbb{F}}$ is defined by $\hat{\rho}_{L,k} \triangleq L(\text{id}_k + \rho) : L(k + |A|) \to L(k + |B|) \in \text{arr}\, \mathcal{S}et$ where $L \in \hat{\mathbb{F}}, k \in \mathbb{F}$. For arbitrary object variables a and b, the unique object variable substitution from $\langle a \rangle$ to $\langle b \rangle$ is denoted by $[a \mapsto b]$.

For a variable $\xi x : B$, where $\xi = [\boldsymbol{a} := \boldsymbol{b}], \{\boldsymbol{b}\} \subseteq B,\ A \triangleq \{\boldsymbol{a}\}$, a semantic object variable substitution $\xi^\diamond : \overline{A} \to \overline{B}$ is defined to be $(|\xi^\diamond|, \overline{A}, \overline{B})$, $|\xi^\diamond|(i) = j$ if $((\overline{A})_i := (\overline{B})_j)$ is contained in ξ. Notice that the notation $[\boldsymbol{a} := \boldsymbol{b}]$ is a part of syntax which always occurs with a variable as $[\boldsymbol{a} := \boldsymbol{b}]x$ in the language, while $[a \mapsto b]$ is a meta-level operation for substituting object variables.

Sequences of object variables taken from OV and object variable substitutions form a category **OVS**. A composition $\phi \circ \rho : A \to D$ of two object variable substitutions $\rho : A \to B$ and $\phi : B \to D$ is given by $(|\phi| \circ |\rho|, A, D)$. It is clear that the category $\mathbb{F}$ and **OVS** are equivalent where $|\_| : \mathbf{OVS} \to \mathbb{F}$ is the equivalence, and **OVS** is considered as a named version of $\mathbb{F}$.

4 Type System and its Interpretation

In this section, we give a type system of terms and its interpretation. A binding algebra is used for a model of the type theory (see [FPT99] for more detailed analysis on binding algebras).

Binding algebras. To a signature Σ, we associate the functor $\Sigma : \hat{\mathbb{F}} \to \hat{\mathbb{F}}$ given by $\Sigma(X) \triangleq \coprod_{f:n_1,\ldots,n_k \to \iota \in \Sigma} \prod_{1 \le i \le k} \delta^{n_i}(X)$. A *binding algebra* or simply Σ-*algebra* is a pair (L, α) with L a presheaf and $\alpha : \Sigma L \to L \in \hat{\mathbb{F}}$. We define the category of binding algebras associated to the signature Σ as the category Σ**-Alg**, with objects given by maps $f : (L, \alpha) \to (L', \alpha')$ that are *homomorphic* in the sense that $f \circ \alpha = \alpha' \circ \Sigma(f)$. The presheaf (of abstract object variables) $V \in \hat{\mathbb{F}}$ is defined by

$$V(n) = n\ (n \in \text{obj}\, \mathbb{F});\quad V(\rho) = \rho\ (\rho \in \text{arr}\, \mathbb{F}).$$

A *model* for the language on the signature Σ is a (V+Σ)-algebra $\mathbb{M} = (M, [\gamma, \{\tau^{(f)}\}_{f \in \Sigma}])$ where M is a presheaf, $\gamma : V \to M$ and $\tau^{(f)} : \prod_{1 \le i \le k} \delta^{n_i} M \to M$ for $f : n_1, \ldots, n_k \to \iota$. The interpretation functions $(\mathbb{M}[\![-]\!]_0 : \mathsf{Ty} \to \text{obj}\, \hat{\mathbb{F}}$, $\mathbb{M}[\![-]\!]_1 : \Sigma \to \text{arr}\, \hat{\mathbb{F}})$ are defined by

$$\mathbb{M}[\![\iota]\!]_0 \triangleq M, \quad \mathbb{M}[\![\delta\sigma]\!]_0 \triangleq \delta[\![\sigma]\!], \quad \mathbb{M}[\![f]\!]_1 \triangleq \tau^{(f)}.$$

As usual, given a context $\Gamma = \langle x_1 : \alpha_1, \ldots, x_n : \alpha_n \rangle$, we set $\mathbb{M}[\![\Gamma]\!] \triangleq \mathbb{M}[\![\alpha_1]\!]_0 \times \ldots \times \mathbb{M}[\![\alpha_n]\!]_0$. We omit the subscripts of $\mathbb{M}[\![\_]\!]_0$ and $\mathbb{M}[\![\_]\!]_1$ hereafter, and write just $[\![\_]\!]$ in the case of $\mathbb{M}$.

Type system and interpretation. We give the typing rules and simultaneously the interpretation of terms by an arbitrary model $\mathbb{M}$ in the category $\hat{\mathbb{F}}$ in Fig. 1. This definition is read as if the upper terms of a typing rule are interpreted as the arrows in the right-hand side of the definition, the interpretation of the lower term is given by the composition of these in $\mathbb{M}$ described in the lower part of the right-hand side.

Let the set St_C be $\wp\widetilde{C}$, which is the set of stages under a given world C. We use a $\mathsf{Ty}, \mathsf{St}_C$-indexed set for variable and term sets. A $\mathsf{Ty}, \mathsf{St}_C$-indexed set L is a disjoint union of its components: $L = \Sigma_{\tau \in \mathsf{Ty}, A \in \mathsf{St}_C} L_{\tau,A}$. We write $l : \tau : A \in L$ for $l \in L_{\tau,A}$. We often use the letter X or Γ (typing context) for denoting $\mathsf{Ty}, \mathsf{St}_C$-indexed set of *variables*. A $\mathsf{Ty}, \mathsf{St}_C$-indexed set $T(C, \Gamma)$ denotes terms under a signature Σ and a world C and a typing context Γ, which is defined as $T(C, \Gamma) = \Sigma_{\tau \in \mathsf{Ty}, A \in \mathsf{St}_C} T(\Gamma)_{\tau,A}$ where $T(\Gamma)_{\tau,A} = \{t \mid \Gamma \vdash_C t : \tau : A\}$.

Example 1.
We will show an example of typing derivation and interpretation. Assume a signature Σ for untyped lambda terms: $\mathsf{lam} : \delta\iota \to \iota$ and $\mathsf{app} : \iota, \iota \to \iota$. An example of typing for a term $\mathsf{lam}([a]\mathsf{var}(b))$ (intended meaning is a λ-term $\lambda a.b$) is

$$\dfrac{\dfrac{\vdash_{a,b} \mathsf{var}(b) : \iota : a, b}{\vdash_{a,b} [a]\mathsf{var}(b) : \delta\iota : b}}{\vdash_{a,b} \mathsf{lam}([a]\mathsf{var}(b)) : \iota : b.}$$

Let us interpret this term in a *syntactic algebra* [FPT99]: let Λ be the presheaf defined by $\Lambda(n) = \{t \mid n \vdash t\}$ $(n \in \mathbb{F})$ where

$$\dfrac{1 \le i \le n}{n \vdash \mathsf{VAR}(i)} \quad \dfrac{n+1 \vdash t}{n \vdash \mathsf{LAM}((n+1).t)} \quad \dfrac{n \vdash t_1 \quad n \vdash t_2}{n \vdash \mathsf{APP}(t_1, t_2).}$$

Define $[\![\iota]\!] = \Lambda$ and $(V + \Sigma)$-algebra Λ with an algebra structure

$$[\gamma, [\![\mathsf{lam}]\!], [\![\mathsf{app}]\!]] : V + \delta\Lambda + \Lambda \times \Lambda \to \Lambda$$

defined as follows: for $n \in \mathbb{F}$,

$$\begin{aligned} \gamma_n : n \to \Lambda(n), &\qquad \gamma_n(i) = \mathsf{VAR}(i), \\ [\![\mathsf{lam}]\!]_n : \Lambda(n+1) \to \Lambda(n), &\qquad [\![\mathsf{lam}]\!]_n(t) = \mathsf{LAM}((n+1).t), \\ [\![\mathsf{app}]\!]_n : \Lambda(n) \times \Lambda(n) \to \Lambda(n), &\qquad [\![\mathsf{app}]\!]_n(t_1, t_2) = \mathsf{APP}(t_1, t_2). \end{aligned}$$

The interpretation of a term $\vdash_{a,b} \mathsf{lam}([a]\mathsf{var}(b)) : \iota : b$ is the arrow

$$\mathbf{1} \xrightarrow{\gamma^*} \delta\Lambda \xrightarrow{\widehat{2}_\Lambda} \delta^2\Lambda \xrightarrow{\widehat{\rho}_\Lambda} \delta^2\Lambda \xrightarrow{\delta[\![\mathsf{lam}]\!]} \delta\Lambda \in \hat{\mathbb{F}}$$

whose component for $n \in \mathbb{F}$ is the function

$$\mathbf{1} \xrightarrow{\gamma_n^*} \Lambda(n+1) \xrightarrow{\widehat{2}_{\Lambda,n}} \Lambda(n+2) \xrightarrow{\widehat{\rho}_{\Lambda,n}} \Lambda(n+2) \xrightarrow{[\![\mathsf{lam}]\!]_{n+1}} \Lambda(n+1) \in Set$$

$$* \longmapsto \mathsf{VAR}(n+1) \longmapsto \mathsf{VAR}(n+2) \longmapsto \mathsf{VAR}(n+1) \longmapsto \mathsf{LAM}((n+2).\mathsf{VAR}(n+1))$$

Variables

$$\frac{}{\Gamma_1, x : \sigma : A, \Gamma_2 \vdash_C [\boldsymbol{a} := \boldsymbol{b}]x : \sigma : B} \qquad \frac{}{\widehat{\pi; [\boldsymbol{a} := \boldsymbol{b}]}^{\diamond}_{[\![\sigma]\!]}}$$

where $A = \{\boldsymbol{a}\}$ and $B = \{\boldsymbol{b}\}$, $\pi : [\![\Gamma_1]\!] \times \delta^{|A|}[\![\sigma]\!] \times [\![\Gamma_2]\!] \to \delta^{|A|}[\![\sigma]\!]$ is a projection in $\hat{\mathbb{F}}$ (the operation "$\diamond$" is defined in Section 3).

Object variables

$$\frac{}{\Gamma \vdash_C \mathsf{var}(a) : \iota : A} \quad a \in A \qquad \frac{}{!_{[\![\Gamma]\!]}; \gamma^*; \widehat{\mathsf{k}}_{[\![\iota]\!]} : [\![\Gamma]\!] \to \delta^{|A|}[\![\iota]\!]}$$

where k is the position of the object a occurring in $\overline{A}$, a map $|\mathsf{k}| : 1 \to |A| \in \mathbb{F}$ is defined by $|\mathsf{k}|(1) = k$, and $\gamma^* : \mathbf{1} \to \delta[\![\iota]\!]$ is the adjoint mate of $\gamma : V \to [\![\iota]\!]$ by the natural bijection $\hat{\mathbb{F}}(\mathbf{1}, \delta[\![\iota]\!]) \cong \hat{\mathbb{F}}(V, [\![\iota]\!])$ [FPT99].

Constants

$$\frac{}{\Gamma \vdash_C F : \sigma : A} \qquad \frac{}{!_{[\![\Gamma]\!]}; [\![F]\!]; \widehat{!}_{[\![\sigma]\!]} : [\![\Gamma]\!] \to \delta^{|A|}[\![\sigma]\!]}$$

where $F : \sigma \in \Sigma$, $!_{[\![\Gamma]\!]} : [\![\Gamma]\!] \to \mathbf{1} \in \hat{\mathbb{F}}$ and $! : 0 \to |A| \in \mathbb{F}$.

Function terms

$$\frac{\Gamma \vdash_C t_1 : \sigma_1 : A, \ldots, \Gamma \vdash_C t_n : \sigma_n : A}{\Gamma \vdash_C F(t_1, \ldots, t_n) : \sigma : A} \qquad \frac{f_1 \quad \ldots \quad f_n : [\![\Gamma]\!] \to \delta^{|A|}[\![\sigma_n]\!]}{\langle f_1, \ldots, f_n \rangle; \cong; \delta^{|A|}[\![F]\!]}$$

where $F : \sigma_1, \ldots, \sigma_n \to \sigma \in \Sigma$ and $\cong : \delta^{|A|}[\![\sigma_1]\!] \times \ldots \times \delta^{|A|}[\![\sigma_n]\!] \to \delta^{|A|}([\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!])$.

Abstractions

$$\frac{\Gamma \vdash_C t : \sigma : A}{\Gamma \vdash_C [a]t : \delta\sigma : A - \{a\}} \quad a \in A \text{ and } \langle a \rangle +\!\!+ \overline{\mathsf{OBV}(t)} \trianglelefteq C \qquad \frac{f : [\![\Gamma]\!] \to \delta^{|A|}[\![\sigma]\!]}{f; \widehat{\rho}_{[\![\sigma]\!]} : [\![\Gamma]\!] \to \delta^{|A|}[\![\sigma]\!]}$$

where $\rho : \overline{A} \to \overline{A - \{a\}} +\!\!+ \langle a \rangle$ is a permutation map defined by $|\rho|(i) = j$ where $(\overline{A})_i = (\overline{A - \{a\}} +\!\!+ \langle a \rangle)_j$.

Applications

$$\frac{\Gamma \vdash_C t : \delta\sigma : A}{\Gamma \vdash_C t@a : \sigma : A \cup \{a\}} \quad a \trianglelefteq C \qquad \frac{g : [\![\Gamma]\!] \to \delta^{|A|+1}[\![\sigma]\!]}{g; \widehat{\rho}_{[\![\sigma]\!]} : [\![\Gamma]\!] \to \delta^{|A \cup \{a\}|}[\![\sigma]\!]}$$

where $\rho : \overline{A} +\!\!+ \langle a \rangle \to \overline{A \cup \{a\}}$ is a map defined by $|\rho|(i) = j$ where $(\overline{A} +\!\!+ \langle a \rangle)_i = (\overline{A \cup \{a\}})_j$. Notice that A may contain a, i.e. applying the same object variable more than twice is possible.

Fig. 1. Type system and interpretation

where $|\rho| : 2 \to 2$ is a swapping map. The presheaf of untyped lambda terms Λ can be shown to be an initial V+Σ-algebra [FPT99].

5 Free Object Variable Substitutions

In this section, we define how object variable substitutions defined in Section 3 are applied to terms. We need a notion of *free* object variable substitutions. The reason we consider free one is motivated by the following examples: Consider an object variable substitution $\rho : \langle a, b\rangle \to \langle a\rangle$ defined by $\widetilde{\rho}(a) = a$ and $\widetilde{\rho}(b) = a$ and apply it to a term $[a][b]F(\mathsf{var}(a), \mathsf{var}(b))$, we have an ill-typed term

$$[a][a]F(\mathsf{var}(a), \mathsf{var}(a)).$$

This violates the condition on a world consisting of the sequence of object variables having *no* duplication because in this case the world must contain the sequence $\langle a, a\rangle$. The order of binding is reflected from the order of object variables in a world (see the typing rule of abstractions). We must also avoid capturing of bound object variables. So we need the following definition.

A *free object variable substitution* $\rho : C \to C'$ *for a term* $t : \iota : A$ is an object variable substitution which satisfies

$$\widetilde{\rho}_{\restriction \mathsf{GBOV}(t)} \text{ is 1-1, and } \rho(a) \notin \mathsf{GBOV}(t) \text{ for } a \in A.$$

The substitution ρ is also obviously extended to functions on sequences, sets of object variables (written as ρA), and on typing contexts:

$$\rho(x_1 : \sigma_1 : A_1, \ldots, x_n : \sigma_n : A_n) = x_1 : \sigma_1 : \rho A_1, \ldots, x_n : \sigma_n : \rho A_n.$$

Also this ρ is extended to functions on terms in two ways $\rho^\sharp$ and $\rho^\natural$ from $T(C, \Gamma)$ to $T(C', \Gamma)$ defined as follows:

$$\rho^\sharp([\boldsymbol{a} := \boldsymbol{b}]x) = [\boldsymbol{a} := \boldsymbol{\rho(b)}]x, \quad \rho^\sharp(F(t_1, \ldots, t_n)) = F(\rho^\sharp(t_1), \ldots, \rho^\sharp(t_n)),$$
$$\rho^\sharp([a]t) = [a]\rho^\sharp_a(t), \quad \rho^\sharp(t@a) = \rho^\sharp(t)@\rho(a), \quad \rho^\sharp(\mathsf{var}(a)) = \mathsf{var}(\rho(a));$$
$$\rho^\natural([\boldsymbol{a} := \boldsymbol{b}]x) = [\boldsymbol{\rho(a)} := \boldsymbol{\rho(b)}]x, \quad \rho^\natural([a]t) = [\rho(a)]\rho^\natural(t),$$

and other cases for $\rho^\natural$ are the same as $\rho^\sharp$. Here ρ_a is the map defined by $\rho_a(a) = a, \rho_a(b) = \rho(b)$ if $b \neq a$.

Lemma 1.
Let $\rho : C \to C'$ be a free object variable substitution for a term t. Then the term t substituted by ρ is well-typed:

$$\frac{\boldsymbol{x} : \boldsymbol{\sigma} : \boldsymbol{A} \vdash_C t : \sigma : A}{\boldsymbol{x} : \boldsymbol{\sigma} : \boldsymbol{A} \vdash_{C'} \rho^\sharp t : \sigma : \rho A} \qquad \frac{\boldsymbol{x} : \boldsymbol{\sigma} : \boldsymbol{A} \vdash_C t : \sigma : A}{\boldsymbol{x} : \boldsymbol{\sigma} : \boldsymbol{\rho A} \vdash_{C'} \rho^\natural t : \sigma : \rho A} \quad .$$

and the following diagrams commute in $\hat{\mathbb{F}}$*:*

$$\begin{array}{ccc} \Pi_i \delta^{|A_i|}[\![\sigma_i]\!] & \xrightarrow{[\![t]\!]} & \delta^{|A|}[\![\sigma]\!] \\ {\scriptstyle \Pi_i id}\downarrow & & \downarrow{\scriptstyle \hat{\rho}_{0\ [\![\sigma]\!]}} \\ \Pi_i \delta^{|A_i|}[\![\sigma_i]\!] & \xrightarrow{[\![\rho^\sharp t]\!]} & \delta^{|\rho A|}[\![\sigma]\!] \end{array} \qquad \begin{array}{ccc} \Pi_i \delta^{|A_i|}[\![\sigma_i]\!] & \xrightarrow{[\![t]\!]} & \delta^{|A|}[\![\sigma]\!] \\ \downarrow{\scriptstyle \Pi_i \hat{\rho}_{i\ [\![\sigma_i]\!]}} & & \downarrow{\scriptstyle \hat{\rho}_{0\ [\![\sigma]\!]}} \\ \Pi_i \delta^{|\rho A_i|}[\![\sigma_i]\!] & \xrightarrow{[\![\rho^\flat t]\!]} & \delta^{|\rho A|}[\![\sigma]\!] \end{array}$$

where $\rho_i : \overline{A_i} \to \overline{\rho A_i}$ *and* $\rho_0 : \overline{A} \to \overline{\rho A}$ *are defined by the restricted functions* $(\widetilde{\rho})_{\restriction A_i} : A_i \to \rho A_i$ *and* $(\widetilde{\rho})_{\restriction A} : A \to \rho A$ *respectively.*

Proof. By induction on the derivation of the judgment $\Gamma \vdash_C t : \sigma : A$. □

Note that this lemma also gives well-typedness of terms by weakening and strengthening of a world using suitable ρ. From these typing, the mnemonic for these two ways of extension is that $\rho^\flat$ is "naturally applied" because the stages in typing context and term equally are applied, and $\rho^\sharp$ is "half applied" because only the term part is applied.

6 Substitutions on Terms

In this section, we define substitution on terms. A *substitution* (not an object variable substitution) θ is a $\mathsf{Ty}, \mathsf{St}_C$-indexed function from Γ to $T(C, \Gamma')$, which gives mappings:

$$x : \tau : A \mapsto \Gamma' \vdash_C t : \tau : A$$

for each $x \in \Gamma$, and we use the shorthand notation $[\boldsymbol{x} \mapsto \boldsymbol{t}]$ for this. The identity substitution $[\boldsymbol{x} \mapsto \boldsymbol{x}]$ is also written as ϵ. The substitution θ is extended to a $\mathsf{Ty}, \mathsf{St}_C$-indexed function θ^* from $T(C, \Gamma)$ to $T(C, \Gamma')$ defined by

$$\theta^*(\xi x) = \xi^{\diamond\sharp} \circ \theta(x) \ \ (x{:}\tau{:}A \in \Gamma), \quad \theta^*(F(t_1, \ldots, t_n)) = F(\theta^*(t_1), \ldots, \theta^*(t_n))$$
$$\theta^*([a]t) = [a]\theta^*_a(t), \quad \theta^*(t@a) = \theta^*(t)@a, \quad \theta^*(\mathsf{var}(a)) = \mathsf{var}(a)$$

where for $x : \sigma : A \in \Gamma$

$$\theta_a(x) = \theta(x) \text{ if } a \in A, \quad \theta_a(x) = \textit{undefined} \text{ if } a \notin A.$$

A composition $\theta_2 \circ \theta_1$ of two substitutions θ_2 and θ_1 is given by $\theta_2^* \circ \theta_1$. We omit the superscript $*$ from substitutions hereafter.

We say a term t_1 is an *instance* of t_2, written as $t_1 \succcurlyeq t_2$, if there exist a 1-1 free object variable substitution ρ and a substitution θ such that $t_1 = \theta\rho^\sharp t_2$. We say a substitution $\theta_2 : X \to \mathrm{T}(C, Y)$ is more *general* than $\theta_1 : X \to T(C, Y)$, written as $\theta_1 \succcurlyeq \theta_2$, if for all $x \in X$, $\theta_1(x) \succcurlyeq \theta_2(x)$. The relation $\preccurlyeq$ is the inverse of $\succcurlyeq$. Namely the notion of generality of substitution is defined modulo renaming of object variables, which is the different point from the usual notion of generality in first-order languages.

Also substitution can be given by a derived rule "cut".

Lemma 2.
Let $\theta : \Gamma \to T(C, \Gamma')$ be a substitution and $x \in \Gamma$. Define the restricted substitution $\theta' : \{x : \sigma : A\} \to T(C, \Gamma')$ defined by $\theta'(x) = \theta(x)$. Then the "cut" rule is derivable.

$$\frac{\Gamma' \vdash_C t : \sigma : A \quad x : \sigma : A, \Gamma'' \vdash_C s : \tau : B}{\Gamma', \Gamma'' \vdash_C \theta'(s) : \tau : B}$$

Example 2.
The substitution defined here also allows *captured* substitution which does not occur in case of ordinary systems having binding, e.g.

$$\frac{\vdash_a \mathsf{var}(a) : \iota : a \quad x : \iota : a \vdash_a [a]x : \iota : \varnothing}{\vdash_a [a]\mathsf{var}(a) : \iota : \varnothing}$$

where the object variable a is captured by the binder $[a]$. So, this $[\_]$-binder can be considered as a *hole of context* appeared in several context calculi [Oho96], [SSK01] rather than an ordinary binder. However this is because the stages of $\mathsf{var}(a)$ and x match in the substitution process and if these does not match, "capture avoiding" like substitution can also be simulated, e.g.

$$\frac{\vdash_a \mathsf{var}(a) : \iota : a \quad x : \iota : \underline{\varnothing} \vdash_a [a]x : \iota : \varnothing}{\textit{not substituted}}.$$

Generally, when all variables occurring in a binding term $[a]t$ have a stage not containing the bound variable a in the typing context, the binder $[a]$ can be considered as a usual (λ-like) binder.

7 Equational Logic

The final section of the type theory of our language is on equational logic. In a binding logic programming language, terms equated by the equational logic are implicitly identified. So, later we consider unification of terms, which is necessary in operational semantics, *modulo this equality*. This is similar in case of λProlog, where higher-order unification that is a unification modulo $\beta\eta$-equality is used.

The equational logic given in Fig. 2 is basically similar to the equational logic for the λ-calculus, but an important difference is the β-axiom which is restricted only for application of an object variable, not an arbitrary term.

Note that in these axioms, both sides of equations must be well-typed terms. So they satisfy some conditions to be well-typed, e.g. $a' \in A$ and $a' \in \mathsf{OBV}(t)$ in the axiom (α).

The reason why we use $\sharp$ for the extension of object variable substitution here is the both sides of terms of equations are in the same context. We call a term which matches the left-hand side of $\beta_0(\eta)$-axiom a $\beta_0(\eta)$-redex.

Example 3.
An example of an equation is

$$x : \iota : a \vdash_{a,b} ([a]x)@b = [a := b]x : \iota : b$$

Axioms

$$(\alpha)\ \ \Gamma \vdash_C [a]t = [a'][a \mapsto a']^\sharp t : \delta\sigma : A$$
$$(\beta_0)\ \ \Gamma \vdash_C ([a]t)@b = [a \mapsto b]^\sharp t : \sigma : A$$
$$(\eta)\ \ \Gamma \vdash_C [a](t@a) = t : \delta\sigma : A$$

Inference rules

$$(\text{Congr-}F)\frac{\Gamma \vdash_C s_1 = t_1 : \sigma_1 : A, \ldots, \Gamma \vdash_C s_n = t_n : \sigma_n : A}{\Gamma \vdash_C F(s_1, \ldots, s_n) = F(t_1, \ldots, t_n) : \sigma : A}\ F : \sigma_1, \ldots, \sigma_n \to \sigma$$

$$(\text{Congr-}[])\frac{\Gamma \vdash_C s = t : \sigma : A}{\Gamma \vdash_C [a]s = [a]t : \delta\sigma : A - \{a\}}\ a \in A,\ \langle a \rangle{+\!\!+}\overline{\mathsf{OBV}(s)} \trianglelefteq C,\ \langle a \rangle{+\!\!+}\overline{\mathsf{OBV}(t)} \trianglelefteq C$$

$$(\text{Congr-}@)\frac{\Gamma \vdash_C s = t : \delta\sigma : A}{\Gamma \vdash_C s@a = t@a : \sigma : A \cup \{a\}}\ a \trianglelefteq C \quad (\text{Ref})\frac{\Gamma \vdash_C t : \sigma : A}{\Gamma \vdash_C t = t : \sigma : A}$$

$$(\text{Subst})\frac{\Gamma, x : \sigma : A \vdash_C t = t' : \sigma : A}{\Gamma \vdash_C [x \mapsto s]t = [x \mapsto s]t' : \sigma : A} \quad (\text{Sym})\frac{\Gamma \vdash_C s = t : \sigma : A}{\Gamma \vdash_C t = s : \sigma : A}$$

$$(\text{Tr})\frac{\Gamma \vdash_C s = t : \sigma : A \quad \Gamma \vdash_C t = u : \sigma : A}{\Gamma \vdash_C s = u : \sigma : A}$$

Fig. 2. Equational logic

where the first x in the equation is regarded as $[a := a]x$. This example shows why the syntactic object variable substitution part in variables is necessary, namely, it is needed to state (α) and (β_0)-axioms for terms involving usual variables. Application of a semantic object variable substitution (e.g. $[a \mapsto b]$) to a term is stopped at a (usual) variable, and a corresponding syntactic object variable substitution (e.g. $[a := b]$) is remained there.

Theorem 1.
The equational logic is sound and complete for the categorical semantics described in Section 4, i.e.

$$\Gamma \vdash_C s = t : \sigma : A \ \Leftrightarrow\ \forall \mathbb{M}\ .\ \mathbb{M}[\![\Gamma \vdash_C s : \sigma : A]\!] = \mathbb{M}[\![\Gamma \vdash_C t : \sigma : A]\!].$$

Proof. Soundness is proved by checking each rules are sound and completeness is by ordinary way of constructing a term model. □

8 Unification

Unification is a fundamental operation in the operational semantics of binding logic programming language by SLD-resolution. In this section, we give an unification algorithm for binding terms.

A *unification problem* is a pair $(\Gamma \vdash_C t : \tau : A, \Gamma' \vdash_{C'} t' : \tau : A')$ of terms. A *unifier* to the unification problem is a pair (θ, ρ) satisfying

$$\Gamma'' \vdash_{C'} \theta\rho^\natural t = \theta t' : A'$$

where

- $\theta : \rho\Gamma \cup \Gamma' \to T(C', \Gamma'')$ is a substitution,
- $\rho : C \to C'$ is an object variable substitution such that $\widetilde{\rho}_{\restriction A} : A \to A'$ is a 1-1 free object variable substitution (so $|A| \leq |A'|$. If not, swap the order of the pair).

This means that ρ is a renaming to standardize the world C of the left-hand side of the unification problem to C', and then θ is a unifier (in usual first-order sense) of two terms.

A *standardized unification problem* is a unification problem satisfying the following properties:

(i) C and C' are mutually disjoint.
(ii) Γ and Γ' are mutually disjoint.
(iii) $|A| = |A'|$.
(iv) t and t' has no β_0, η-redexes.
(v) The bound variables in t and t' are numbered by the method of de Bruijn level. So C and C' contain some segments of natural numbers $\mathbb{N}$.
(vi) The types of all variables are ι.

Proposition 1.
If a unification problem $(\Gamma \vdash_C t : \tau : A, \Gamma' \vdash_{C'} t' : \tau : A')$ *has a unifier* $(\theta, \rho{:}C \to C')$*, then it can be translated to the standardized unification problem*

$$(\Gamma_0 \vdash_{C_0} \phi t : \tau : A_0, \Gamma'_0 \vdash_{C'_0} \phi' t' : \tau : A'_0)$$

having the "essentially same" unifier $(\theta', \rho'{:}C_0 \to C'_0)$ *by the pair* $(\phi{:}C \to C_0, \phi'{:}C' \to C'_0)$ *of 1-1 object variable substitutions such that* $\phi' \circ \rho = \rho_0 \circ \phi$*. This "essentially same" means that each assignment*

$$x : \delta^n \iota \mapsto [k] \ldots [k+n] s \ \in \theta$$

1-1 corresponds to an assignment

$$x_0 : \iota \mapsto s_0 \ \in \theta'$$

and $\phi' s \equiv s_0$*.*

Proof. (sketch) The conditions for standardized unification problem can be satisfied by the following way:

(i) By suitably renaming object variables.
(ii) By suitably renaming variables.
(iii) By suitably weakening and strengthing of stages by using derived rules.
(iv) By β_0, η_0-rewriting using the equational axioms from left to right.
(v) By standard way.
(vi) For each variable $\xi f : \delta^n \iota : A$ occurring in t and t', replace it with

$$[k] \ldots [k+n] \xi f_0 \text{ where } f_0 : \iota : A \cup \{k, \ldots, k+n\}$$

where k is determined from the context which fits with the method of de Bruijn level [dB72,FPT99]. Since we do not have constants of higher-order types ($\delta^m \iota$) all variables are represented by this form.

Also the object variables in the stages of the judgment were accordingly replaced by the above replacement including numbering by de Bruijn level. □

From this proposition, we see that it is sufficient to consider a unification algorithm for standardized unification problem, and any unification problem can be standardized. We can easily recover a unifier of original problem from the unifier of standardized one.

The unification problem is extended to a pair of sequences of terms and a unifier is similarly defined. For a unification problem, a *system* is of the form

$$t_1{:}A_1 =^? t'_1{:}A'_1, \ldots, t_n{:}A_n =^? t'_n{:}A'_n.$$

The substitution θ and object variable substitution ρ are extended to functions on systems defined by $\theta(t_i{:}A_i =^? t'_i{:}A'_i)_{i=1,\ldots,n} \stackrel{def}{=} (\theta t_i{:}A_i =^? \theta t'_i{:}A'_i)_{i=1,\ldots,n}$ and $\rho(t_i{:}A_i =^? t'_i{:}A'_i)_{i=1,\ldots,n} \stackrel{def}{=} (\rho^\sharp t_i{:}A_i =^? \rho^\sharp t'_i{:}A'_i)_{i=1,\ldots,n}$.

Unification algorithm BUA. The unification algorithm BUA for binding terms is given by the set of transformation rules on systems given in Fig. 3. This is based on Martelli and Montanari's unification algorithm for first-order terms by the set of transformations [MM82].

$$(F\text{-elim})\frac{G_1, F(s_1,\ldots,s_n) : A =^? F(t_1,\ldots,t_n) : B, G_2}{G_1, s_1 : A =^? t_1 : B, \ldots, s_n : A =^? t_n : B, G_2}$$

$$(\text{triv})\frac{G_1, \xi x : A =^? \xi x : A, G_2}{G_1, G_2} \qquad ([]\text{-elim})\frac{G_1, [k]s : A =^? [k]t : B, G_2}{G_1, s : A \cup \{k\} =^? t : B \cup \{k\}, G_2}$$

$$(\text{ov-elim})\frac{G_1, \mathsf{var}(a) : A =^? \mathsf{var}(b) : B, G_2}{\rho(G_1, s : A =^? t : B, G_2)} \qquad (@\text{-elim})\frac{G_1, s@a : A =^? t@b : B, G_2}{\rho(G_1, s : A =^? t : B, G_2)}$$

$$(\text{v-elim1})\frac{G_1, \xi x : B =^? t : A, G_2}{\theta\rho(G_1, G_2)} \qquad (\text{v-elim2})\frac{G_1, t : A =^? \xi x : B, G_2}{\theta\rho(G_1, G_2)}$$

In (v-elim1) and (v-elim2), $x \notin \mathsf{VAR}(t)$.

Fig. 3. Unification algorithm BUA

Side conditions in BUA: in (ov-elim) and (@-elim), $a, b \notin \mathbb{N}, \rho = [a \mapsto b], |\rho A| = |\rho B|$ (a is possibly b). In (v-elim1) (and (v-elim2)), the following data is used:

- $x : \tau : D \in \Gamma$ ($\in \Gamma'$ in (v-elim2)),
- an isomorphic object variable substitution $\rho : \overline{B} \to \overline{A}$, where $\overline{A} = \langle a_1, \ldots, a_n \rangle$, $\overline{B} = \langle b_1, \ldots, b_n \rangle$, is defined by $\overline{\rho}(i) \triangleq i$.
- an object variable substitution $\xi^\diamond : \overline{D} \to \overline{B}$,
- a substitution θ is arbitrarily taken from the set $\Theta = \{[x{:}\tau{:}D \mapsto s{:}\tau{:}D] \mid \langle s, E \rangle \in S,\ E \subseteq D\}$ (i.e. there are $\#\Theta$-different θ's) where

$$S \triangleq \{\langle t[p_1 := \xi'_1 \circ \rho^{-1}(t_{\restriction p_1}), \ldots, p_n := \xi'_n \circ \rho^{-1}(t_{\restriction p_n})],\ \bigcup_i \xi'_i B\rangle$$
$$\mid \{p_1, \ldots, p_n\} = \mathsf{OVPOS}(t) \text{ and } p_1 \neq \ldots \neq p_n,$$
$$\xi'_1, \ldots, \xi'_n \in \{\xi' : \overline{B} \to \overline{D} \mid \xi^{\diamond} \circ \xi' = \mathsf{id}_B\} \cup \{\epsilon\}\}.$$

Note that $\mathsf{OVPOS}(t)$ denotes the set of all positions of free object variables occurring in t, $t[p := s]$ is a replacement of a term t at the position p with the term s, and $t_{\restriction p}$ denotes a subterm of t at the position p. Also (v-elim1) and (v-elim2) produce $\#\Theta$-different branches of derivation.

Let G and G' be systems. We write $G \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta,\rho}} G'$ for a one-step transformation with a computed substitution θ and ρ (in (@-elim), (v-elim1) and (v-elim2), these are obtained, otherwise they are ϵ). We write $G \overset{\mathsf{BUA}}{\rightsquigarrow^*_{\theta,\rho}} G'$ if there exists a sequence of transformation

$$G \equiv G_1 \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta_1,\rho_1}} G_2 \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta_2,\rho_2}} \cdots \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta_n,\rho_n}} G_n \equiv G'$$

such that $\theta = \theta_{n-1} \circ \cdots \circ \theta_2 \circ \theta_1$ and $\rho = \rho_{n-1} \circ \cdots \circ \rho_2 \circ \rho_1$, where we suitably extend codomains and domains of object variable and usual substitutions to make them composable by adding identity assignments. If $n = 1$ then $\theta = \epsilon$ and $\rho = \epsilon$. We use $\square$ for the empty system. When $G \overset{\mathsf{BUA}}{\rightsquigarrow^*_{\theta,\rho}} \square$, we say that θ is a *computed unifier*.

Properties of BUA. As in the case of higher-order unification, a unification for binding terms also may not have most general unifiers. For example, a unification problem

$$(x{:}\iota{:}a, b \vdash_{a,b} ([a]x)@b\,,\ \vdash_{a,b} F(\mathsf{var}(b), \mathsf{var}(b)))$$

has the four unifiers

$$x \mapsto F(\mathsf{var}(a), \mathsf{var}(a)), \qquad x \mapsto F(\mathsf{var}(a), \mathsf{var}(b))$$
$$x \mapsto F(\mathsf{var}(b), \mathsf{var}(a)), \qquad x \mapsto F(\mathsf{var}(b), \mathsf{var}(b)).$$

The last two unifiers can be obtained from the first two. And there is no unifier more general than these first two unifiers (see the definition of generality in Section 6, where only a 1-1 free object variables substitution is allowed to get an instance). So instead of most general unifier, we will just compute unifiers that are more general than unifiers of a given unification problem (completeness).

The computed unifiers has the following desired properties: let $(\Gamma \vdash_C t{:}\tau{:}A, \Gamma' \vdash_{C'} t'{:}\tau{:}A')$ be a *standardized* unification problem.

- Soundness: if $t{:}A =^? t'{:}A' \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta,\rho}} \square$, then (θ, ρ) is a unifier of the unification problem.
- Completeness: if (θ, ρ) is a unifier of the unification problem, then $\exists \theta' \preccurlyeq \theta\ .\ t{:}A =^? t'{:}A' \overset{\mathsf{BUA}}{\rightsquigarrow_{\theta',\rho}} \square$.
- Decidability: there are no infinite transformation sequence by BUA.

- Normalizedness: computed answer unifiers are β_0, η-normalized.
- An answer object variables substitution $\rho : A_1 \to A_1'$ in a computed unifier is free and 1-1 where $A_1 \trianglelefteq A$ and $A_1' \trianglelefteq A'$. This can be extended to a 1-1 object variable substitution $\rho_1 : C \to C'$ required in the definition of unifier.

These can be shown by a similar way to the first-order case. Note that in higher-order unification, a set of sound and complete idempotent unifiers is called a *complete set of unifiers* and is not decidable in general, but our BUA is decidable.

Example 4.
We will give an example of unification by BUA. Consider a unification problem

$$(\vdash_{a,b,c} [a][b]F(c,a,b) : c,\ x{:}\iota{:}a, b, c \vdash_{a,b,c} [b][c]x : a).$$

This problem is translated to a standardized unification problem

$$(\vdash_{1,2,c} [1][2]F(c,1,2) : c,\ x{:}\iota{:}a, 1, 2 \vdash_{a,1,2} [1][2]x : a).$$

The computed unifiers is $(x \mapsto F(a,1,2),\ c \mapsto a)$ and the corresponding unifier to the original problem is $(x \mapsto F(a',a,b),\ c \mapsto a')$ which is obtained by recovering names from de Bruijn numbers and putting dash to object variables in the right-hand side to make the worlds in both sides of the original unification problem disjoint.

Predicate terms:

$$\frac{\Gamma \vdash_C t_1 : \sigma_1 : A \quad \ldots \quad \Gamma \vdash_C t_n : \sigma_n : A}{\Gamma \vdash_C p(t_1, \ldots, t_n) : A} \quad p : \sigma_1, \ldots \sigma_n$$

Note that no types are given in predicate terms.

Definite Horn clauses:

$$\frac{\Gamma \vdash_C p(\boldsymbol{t}) : A \quad \Gamma \vdash_C p_1(\boldsymbol{t_1}) : A_1 \quad \ldots \quad \Gamma \vdash_C p_n(\boldsymbol{t_n}) : A_n}{\vdash_C \forall \Gamma\ .\ p(\boldsymbol{t}) : A \Leftarrow p_1(\boldsymbol{t_1}) : A_1, \ldots, p_n(\boldsymbol{t_n}) : A_n}$$

If n is 0, the symbol "$\Leftarrow$" is omitted.

Queries:

$$\frac{\Gamma \vdash_C p(\boldsymbol{t}) : A \quad \Gamma \vdash_C p_1(\boldsymbol{t_1}) : A_1 \quad \ldots \quad \Gamma \vdash_C p_n(\boldsymbol{t_n}) : A_n}{\vdash_C \exists \Gamma\ .\ p_1(t_1) : A_1, \ldots, p_n(t_n) : A_n}$$

Fig. 4. Typing rules for formulas

---

9 Binding Logic Programming

In this section, we describe a binding logic programming language.

$$(\text{Axiom})\frac{P \ \triangleright_C \forall\Gamma' \,.\, \theta(p_1(\boldsymbol{t_1})) : A_1 \quad \ldots \quad P \ \triangleright_C \forall\Gamma' \,.\, \theta(p_n(\boldsymbol{t_n})) : A_n}{P \ \triangleright_C \forall\Gamma' \,.\, \theta(p(\boldsymbol{t})) : A}$$

where
- $\vdash_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A \Leftarrow p_1(\boldsymbol{t_1}) : A_1, \ldots, p_n(\boldsymbol{t_n}) : A_n \ \in P$
- a substitution $\theta : \Gamma \to T(C, \Gamma')$

$$(\exists\text{−intro})\frac{P \ \triangleright_C \forall\Gamma' \,.\, p_1(\theta\boldsymbol{t_1}) : A_1 \quad \ldots \quad P \ \triangleright_C \forall\Gamma' \,.\, p_n(\theta\boldsymbol{t_n}) : A_n}{P \ \triangleright_C \exists\Gamma \,.\, p_1(\boldsymbol{t_1}) : A_1, \ldots, p_n(\boldsymbol{t_n}) : A_n} \ \theta : \Gamma \to T(C, \Gamma')$$

$$(\text{Inst})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A}{P \ \triangleright_C \forall\Gamma' \,.\, \theta(p(\boldsymbol{t})) : A} \ \theta : \Gamma \to T(C, \Gamma') \quad (\text{Wld-St})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A}{P \ \triangleright_{C'} \forall\Gamma \,.\, p(\boldsymbol{t}) : A}$$

$$(\text{Repl})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A \quad \Gamma \vdash_C t_1 = t_1' : A \ldots \Gamma \vdash_C t_n = t_n' : A}{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t'}) : A}$$

$$(\text{St-w})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A}{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A \cup \{a\}} \ a \notin \mathsf{OBV}(t) \quad (\text{St-s})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A \cup \{a\}}{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A} \ a \notin \mathsf{FOV}(t)$$

$$(\text{Rename})\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A}{P \ \triangleright_{C'} \forall\rho\Gamma \,.\, p(\rho^\sharp \boldsymbol{t}) : A} \quad (\text{Open } \alpha)\frac{P \ \triangleright_C \forall\Gamma \,.\, p(\boldsymbol{t}) : A}{P \ \triangleright_C \forall\Gamma \,.\, p(\rho^\sharp \boldsymbol{t}) : A}$$

In (Rename), $\rho : C \to C'$ is 1-1. In (Open α), $\rho : C \to C$ is 1-1. In (World-st), C' is obtained from C by deleting object variables not occurring in $\forall\Gamma \,.\, p(\boldsymbol{t}) : A$.

Fig. 5. Inference rules of the logic

9.1 Logic

We need formulas for logic programming. We assume the signature Σ is the disjoint union of Σ_F for function symbols and $\Sigma_P = \{p : \sigma_1, \ldots \sigma_n, \ \ldots\}$ for predicate symbols. The typing rules of formulas are given in Fig. 4.

Next we describe the logic of our logic programming language. A *program* P is a set of definite Horn clauses. Note that here a definite Horn clause includes the part $\vdash_C$ on the top, so in a program, each clause may have different world C. A *sequent* within this logic is of the form

$$P \ \triangleright_C Q$$

where Q is a well-typed formula by $\vdash_C Q$. Note that we use the symbol $\vdash$ for the well-typed terms and formulas, and $\triangleright$ for the sequent.

The inference rules of this logic is given in Fig. 5. The substitution θ in ($\exists$-intro) is called *witness* of this existentially quantified formula. Weakening of a world can be performed by (Rename) rule using suitable ρ.

The reason why we only let ρ be 1-1 for substituting object variables in the above rules comes from Hofmann's observation on the denotation on decidable equality in the presheaf category $\hat{\mathbb{F}}$ [Hof99]. Let us rephrase his observation in our system: consider a program for inequality of object variables by the predicate ineq and the following sequent:

$$\{\vdash_{a,b} \mathsf{ineq}(\mathsf{var}(a), \mathsf{var}(b)) : a, b\} \ \triangleright_{a,b} \mathsf{ineq}(\mathsf{var}(a), \mathsf{var}(b)) : a, b.$$

Then, if we apply a surjective object variable substitution $\rho : \langle a, b\rangle \to \langle a\rangle$, we can derive the following

$$\{\vdash_{a,b} \mathsf{ineq}(\mathsf{var}(a), \mathsf{var}(b)) : a, b\} \rhd_a \mathsf{ineq}(\mathsf{var}(a), \mathsf{var}(a)) : a.$$

This says $\mathsf{var}(a)$ and $\mathsf{var}(a)$ are inequal, but clearly this cannot be considered as correct use of the predicate ineq. Same things will happen in any predicate having more than two object variables, so we restrict ρ to 1-1.

Example 5.
We describe a program for syntactic differentiation given in introduction more precisely in our typed language:

$$P = \left\{ \begin{array}{l} \vdash_{a,b} \mathsf{diff}([b]\mathsf{var}(a), [b]0) : a, \\ \vdash_{a,b} \mathsf{diff}([b]\mathsf{var}(b), [b]1) : \varnothing, \\ \vdash_{a,b} \forall f{:}\iota{:}b, f'{:}\iota{:}b \,.\, \mathsf{diff}([b]\mathsf{sin}(f), [b](\mathsf{cos}(f) \times f')) : \varnothing \Leftarrow \mathsf{diff}([b]f, [b]f') : \varnothing. \end{array} \right\}$$

A query under P is stated as the sequent

$$P \rhd_{d,c} \exists z{:}\delta\iota{:}\varnothing \,.\, \mathsf{diff}([d]\mathsf{var}(c), z) : c,$$

and an answer substitution for this is $\theta : \{z{:}\delta\iota{:}\varnothing\} \to T(\langle d\rangle, \varnothing)$ given by $z \mapsto [d]0$.

Example 6.
We also give a program for capture-avoiding substitution of λ-terms defined in Example 1. The predicate $\mathsf{sub}([a]s, t, u)$ means that u is a term obtained from the term s by replacing all a with t.

$$P = \left\{ \begin{array}{ll} \vdash_a & \forall y{:}\varnothing \,.\, \mathsf{sub}([a]\mathsf{var}(a), y, y) : \varnothing \\ \vdash_{a,b} & \forall y{:}\varnothing \,.\, \mathsf{sub}([a]\mathsf{var}(b), y, \mathsf{var}(b)) : b \\ \vdash_a & \forall e_1, e_2{:}a, z_1, z_2{:}\varnothing, y{:}\varnothing \,.\, \mathsf{sub}([a]\mathsf{app}(e_1, e_2), y, \mathsf{app}(z_1, z_2)) : \varnothing \\ & \quad \Leftarrow \mathsf{sub}([a]e_1, y, z_1) : \varnothing,\ \mathsf{sub}([a]e_2, y, z_2) : \varnothing \\ \vdash_{a,b} & \forall x{:}a, b, y{:}\varnothing, z{:}b \,.\, \mathsf{sub}([a]\mathsf{lam}([b]x), y, \mathsf{lam}([b]z)){:}\varnothing \\ & \quad \Leftarrow \mathsf{sub}([a]x, y, z) : b \end{array} \right\}$$

9.2 Operational Semantics

As in ordinary logic programming language, we consider an existentially quantified formula as a *query* and witnesses as answer substitutions for the query. We will give a version of SLD-resolution which treats binding terms for a proof method of existentially quantified formulas with witnesses. Let us prove the following sequent

$$P \rhd_D \exists \Delta \,.\, p_1(\boldsymbol{s_1}) : A_1, \ldots, p_n(\boldsymbol{s_n}) : A_n$$

by getting witnesses. For this sequent, a *goal* is of the form

$$D, \Delta \,;\, p_1(\boldsymbol{s_1}){:}A_1, \ldots, p_n(\boldsymbol{s_n}){:}A_n.$$

We define SLD-resolution as a transformation rule on goals. Let G_1, G_2, G_3 be sequences of predicate terms.

SLD-resolution. Define a one-step SLD-resolution

$$D, \Delta\,;\, G_1,\, p(\boldsymbol{s}){:}B,\, G_2 \;\rightsquigarrow_{\underline{\theta},\rho}\; C_0, \Delta'; \theta_0(\rho G_1, \phi' G_3, \rho G_2)$$

where $\vdash_C \forall \Gamma\,.\, p(\boldsymbol{t}) : A \Leftarrow G_3 \;\in P$,

$$\theta = \underline{\theta_0} \circ \phi_1 : \Delta \to T(C_0, \Delta'), \quad \rho = \rho_0 \circ \phi : D \to C_0,$$

and $\phi_1 : \Delta \to \rho_0\Delta_0 \cup \Gamma_0$ is defined by $x : \sigma : A \mapsto x : \sigma : \rho_0\phi A$. Here let (θ, ρ) be a computed unifier by BUA for the unification problem

$$(\Delta \vdash_D p(\boldsymbol{s}) : B,\; \Gamma \vdash_C p(\boldsymbol{t}) : A),$$

between the goal and the head of the Horn clause and (θ_0, ρ_0) a "essentially same" unifier for the standardized problem. Then $\underline{\theta_0}$ is obtained by recovering original variables from the substitution θ_0.

We write $H \rightsquigarrow^*_{\theta,\rho} H'$ if there exits a sequence of transformation

$$H \equiv H_1 \rightsquigarrow_{\theta_1,\rho_1} H_2 \rightsquigarrow_{\theta_2,\rho_2} \cdots \rightsquigarrow_{\theta_n,\rho_n} H_n \equiv H'$$

such that $\theta = \theta_{n-1} \circ \cdots \circ \theta_2 \circ \theta_1$ and $\rho = \rho_{n-1} \circ \cdots \circ \rho_2 \circ \rho_1$. If $n = 1$ then $\theta = \epsilon$ and $\rho = \epsilon$. We use $\square$ for the empty goal. When $H \rightsquigarrow^*_{\theta,\rho} \square$, this is a *successful* SLD-resolution, otherwise, it is fail.

Theorem 2.
The SLD-resolution is sound for the logic given in Section 9.1, i.e.

$$D, \Delta; G \rightsquigarrow_{\theta,\rho} \square \;\;\Rightarrow\;\; P \rhd_{C_0} \forall \Delta_1.\theta G$$

where $\theta : \Delta \to T(C_0, \Delta_1)$ *and* $\rho : D \to C_0$.

Also, the SLD-resolution is complete, i.e. if $P \rhd_D \exists \Delta.G$ *is provable and its previous sequent in the proof is* $P \rhd_D \forall \Delta_1.\theta G$, *then*

$$D, \Delta; G \rightsquigarrow_{\theta',\rho} \square$$

such that $\theta' \preccurlyeq \rho_1 \circ \theta$, *where* $\theta : \Delta \to T(D, \Delta_1)$, $\theta' : \Delta \to T(C_0, \Delta_2)$, $\rho : D \to C_0$, *and* $\rho_1 : \Delta_1 \to \Delta_2$ *is defined by* $x : \sigma : A \mapsto x : \sigma : \rho A$.

10 Related Work

Nadathur and Miller's λProlog [NM88] is one of the most famous language treating variable binding in logic programming. This language deals with higher-order hereditary Harrop formulas instead of first-order Horn clauses in case of Prolog. The unification is extended to that for simply typed λ-terms modulo α,β,η-conversion, called higher-order unification. A problem is that higher-order unification is undecidable in general and quite complex. This point is different from ours, our binding logic programming language uses simple decidable unification.

But the class of formulas of λProlog is wider than ours, higher-order hereditary Harrop formulas can contain $\forall, \Rightarrow$ in the body of clauses.

Later, Miller proposed succeeding language L_λ [Mil91], which is a subset of λProlog where β-equality is restricted to β_0-equality defined by $(\lambda x.M)x = M$. Miller observed that "when examining typical λProlog programs, it is clear that most instances of β-conversion performed by the interpreter are, in fact, instances of β_0-conversion" [Mil00]. As a result, the unification modulo α,β_0,η becomes very simple and decidable, and also resemble first-order unification.

Our β_0-axiom is (essentially) same as Miller's β_0-axiom, but the motivation of this axiom comes from a different reason, i.e. initial algebra semantics of binding by presheaves [FPT99]. The recent consideration of semantics of abstract syntax with variable binding including Fiore-Plotkin-Turi's has a common principle, where they prohibit general function types in signature, such as $\mathsf{lam} : (term \Rightarrow term) \to term$, and instead, use restricted function types only from (object) variables, such as $\mathsf{lam} : (var \Rightarrow term) \to term$. The reason for this restriction is to get structural recursion (or induction) on data defined by this kind of signature [FPT99,Hof99,GP99,DFH95]. So, now Miller's practical observation on the restriction of β-equality in λProlog and the restriction of function types in semantical studies on variable binding coincide. Hence our type theory involving the β_0-axiom can be considered as reasonable.

So L_λ is similar to our language but again the class of formulas in L_λ (hereditary Harrop formulas) is wider than ours (Horn clauses). It is not currently clear whether all L_λ programs can be (easily) translated to our binding logic programs. Due to β_0's restriction, unification in L_λ becomes *pattern unification*, which is a higher-order unification only for particular class of λ-terms called higher-order patterns. A λ-term is a higher-order pattern if every free variable occurrence is applied to at most distinct bound variables. So, for example, $\lambda x.Fxx =^? \lambda y.gy$ (F is a free variable) is *not* a pattern unification problem, hence L_λ cannot solve it. But our BUA algorithm can solve it, this point is an advantage of our language on L_λ.

Pitts and Gabbay's FreshML [PG00] is a functional programming language based on their semantics of binding – FM-set theory [GP99]. FreshML is quite similar to ours, they use the notion of stage too (called support), but they attached object variables *not contained* by its stage as a type information of terms. They in particular payed attention to freshness of object variables in terms, which is not considered in this paper. In FM-set theory, a permutation group is used to formalize renaming and binding of object variables, where only injective variable substitutions are used. This affects to syntax, for example, an application of the same object variables twice, such as a term $t@a@a$, is impossible in their type system. But in our case, since the category $\mathbb{F}$ has all functions between (abstract) object variables, our type system allow such terms. Another difference is that FreshML does not use η-axiom.

Acknowledgments. I am grateful to Gordon Plotkin for motivating this work and designing the basis of the type theory in this paper. I also thank Daniele Turi for discussions on this work, Martin Hofmann, Hirofumi Yokouchi and the

anonymous referees for helpful comments. This work was done while I was visiting LFCS, University of Edinburgh where I have been supported by JSPS Research Fellowships for Young Scientists.

References

[dB72] N. de Bruijn. Lambda clculus notation with nameless dummies, a tool for automatic formula manipulation, whith application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–391, 1972.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani and G. Plotkin, editors, *Typed Lambda Clculi and Apllications, LNCS 902*, pages 124–138, 1995.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstrat syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202, Washington, 1999. IEEE Computer Society Press.

[GP99] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, 1999. IEEE Computer Society Press.

[Hof99] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213, Washington, 1999. IEEE Computer Society Press.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification,. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil00] D. Miller. Abstract syntax for variable binders: An overview. In John Lloyd, et. al., editor, *Proceedings of Computation Logic 2000, LNAI 1861*, 2000.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions of Programming Languages*, 4(2):258–282, 1982.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[Oho96] A. Ohori. A typed context calculus. In *Preprint RIMS-1098*. Research Institute for Mathematical Sciences, Kyoto University, 1996.

[PG00] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[SSK01] M. Sato, T. Sakurai, and Y. Kameyama. A simply typed context calculu with first-class environments. In *5th International Symposium on Functional and Logic Programming*, volume LNCS 2024, pages 359–374, 2001.