Unification of Hypergraph λ-Terms

Alimujiang Yasen, Kazunori Ueda

Unification of Hypergraph λ-Terms

Alimujiang Yasen and Kazunori Ueda

Dept. of Computer Science and Engineering, Waseda University

Abstract. We developed a technique for modeling formal systems involving name binding in a modeling language based on hypergraph rewriting. A hypergraph consists of graph nodes, edges with two endpoints and edges with multiple endpoints. The idea is that hypergraphs allow us to represent terms containing bindings and that our notion of a graph type keeps bound variables distinct throughout rewriting steps. We previously encoded the untyped λ-calculus and the evaluation and type checking of System $F_{<:}$, but the encoding of System $F_{<:}$ type inference requires a unification algorithm. We studied and successfully implemented a unification algorithm modulo α-equivalence for hypergraphs representing untyped λ-terms. The unification algorithm turned out to be similar to nominal unification despite the fact that our approach and nominal approach to name binding are very different. However, some basic properties of our framework are easier to establish compared to the ones in nominal unification. We believe this indicates that hypergraphs provide a nice framework for encoding formal systems involving binders and unification modulo α-equivalence.

1 Introduction

Unification solves equations over terms. For a unification problem $M = N$, a unification algorithm finds a substitution $\delta = [X := P, Y := Q, \dots]$ for unknown variables X and Y occurring in terms M and N so that applying δ to the original problem make $\delta(M)$ and $\delta(N)$ equal. Depending on the terms occurring in the unification problem, a unification algorithm is classified as (standard) *first-order* unification and *higher-order* unification, where higher-order unification solves equations over higher-order terms such as λ-terms. First-order unification is simple in theory and efficient in implementation [7,11], whereas higher-order unification is more complex both in theory and implementation [5].

The reason why higher-order unification is complex is that they solve equations of terms modulo α-, β- and possibly η-equivalence, denoted as $=_{\alpha\beta\eta}$. Alpha-equivalence equates two λ-terms M and N up to the renaming of their bound variables, denoted as $M =_\alpha N$; β-equivalence equates two terms under $(\lambda a.M)N =_\beta M[a := N]$; and η-equivalence states that $(\lambda a.Ma) =_\eta M$ where a does not occur free in M. Although higher-order unification is required in logic programming languages and proof assistants based on higher-order approach [9], full higher-order unification is undecidable and may not generate most general unifiers. Higher-order pattern unification is a simple version of higher-order unification which solves terms modulo $\alpha\beta_0\eta$-equivalence [8], where β_0-equivalence

is a form of β-equivalence $(\lambda x.M)N =_{\beta_0} M[x := N]$ where N must be a variable not occurring free in $\lambda x.M$. Most importantly, it is an efficient process with linear-time decidability [8,18]. Higher-order pattern unification is popular in practice because of that. For instance, the latest implementation of $\lambda Prolog$ is actually an implementation of a sublanguage of $\lambda Prolog$ called L_λ, which only uses higher-order pattern unification [10]. However, the infrastructure for implementing a variant of the λ-calculus is not lightweight, and a restriction to β_0-equivalence asks users for good programming practice to avoid cases which do not respect the restriction. A first-order style unification algorithm for terms involving name binding is preferred in these respects.

One such unification algorithm is nominal unification [14], which solves equations of nominal terms. In nominal terms, *names* are equipped with the *swapping* operation and the *freshness* condition [4]. The work in [6,2] shows the connection between nominal unification and higher-order pattern unification; if two nominal terms are unifiable, then their translated higher-order pattern counterparts are also unifiable. Alpha-equivalence is assumed for higher-order terms in theory. Yet, in the higher-order approach, implementing a meta-language (a variant of the typed λ-calculus) means that one must also consider $=_{\beta_0\eta}$. In nominal unification, only $=_\alpha$ is needed, and variable capture is allowed during the unification in the sense that a unifier may bring a name a into the scope of a as in $(\lambda a.X)[X := a]$. Nominal unification solves problems in two phases; solving equations of terms and solving freshness constraints.

Using graphs to represent λ-terms has a long history [19,20]. In our earlier work, we studied a hypergraph-based technique for representing terms involving name binding [16], using HyperLMNtal [13] as a representation and implementation language. The idea was that hypergraphs could naturally express terms containing bindings; atoms (nodes of graphs) represent constructors such as abstraction and application; hyperlinks (edges with multiple endpoints) represent variables; and regular links (edges with two endpoints) connect constructors with each other. In this technique, two isomorphic (but not identical) hypergraphs representing α-equivalent terms containing bindings have two syntactically different textual representations in HyperLMNtal. For example, two instances of the λ-term $\lambda a.aa$ are represented by α-equivalent but syntactically different hypergraphs such as `abs(A,(app(A,A)),L)` and `abs(B,(app(B,B)),R)` as shown in Fig. 1.

(a) (b)

Fig. 1: Two α-equivalent terms represented as hypergraphs

In Fig. 1, circles are atoms, straight lines are regular links and eight-point stars with curved lines are hyperlinks. The arrowheads on circles indicate the first arguments of atoms and the ordering of their arguments. These two hypergraphs, rooted at `L` and `R`, are isomorphic, i.e., have the same shape, but are syntactically not identical. (Later, we explain why regular links between `abs` and `app` atoms are implicit in the above two terms.)

Our idea was first proposed in [16], where we developed the theory with the encoding of the untyped λ-calculus. Our formalism separates bound and free variables by Barendregt's variable convention [1] and also requires bound variables to be distinct from each other. A graph type called *hlground* (meaning ground graphs made up of hyperlinks) keeps bound variables distinct during the substitution. For example, $\lambda a.M$ and $\lambda a.N$ do not exist at the same time, and if $\lambda a.M$ exists, a may occur in M only. Such conventions may look too strict, but our experiences show that it brings great convenience in practice. For example, in our recent work [17], we encoded System $F_{<:}$ easily in HyperLMNtal; implementing the type checking of System $F_{<:}$ required the equality checking of types containing type variable binders, which was handled by directly applying α-equality rules in theory. As the next step, we want to implement the type inference of System $F_{<:}$, which means that we should study the unification of terms containing name binding within our formalism.

Hypergraphs representing λ-terms are called *hypergraph λ-terms*. This paper considers unification problems for equations over hypergraph λ-terms modulo $=_\alpha$. Hypergraph λ-terms have nice properties; for two abstractions `L=abs(A,M)` and `R=abs(B,N)`, `A` does not occur in `N` and `B` does not occur in `M`, and `A` and `B` are always different hyperlinks. These properties greatly simplified the reasoning in our previous work, and we expect such simplicity in this work as well.

The outline of the paper is as follows. In Section 2, we briefly describe *hypergraph λ-terms* and the definition of substitutions. In Section 3, we present the unification algorithm and related proofs. In Section 4, we give some examples. In Section 5, we briefly describe the implementation of the unification algorithm. In Section 6, we review related work and conclude the paper.

2 Hypergraph λ-Terms

HyperLMNtal is a modeling language based on hypergraph rewriting [13] that is intended to be a substrate language of diverse computational models, especially those addressing concurrency, mobility and multiset rewriting. Moreover, we have successfully encoded the λ-calculus with strong reduction in HyperLMNtal in two different ways, one in the fine-grained approach [12] and the other in the coarse-grained approach [16]. This paper takes the latter approach that uses hyperlinks to represent binders, where the representation of λ-terms is called *hypergraph λ-terms*. We briefly describe HyperLMNtal and hypergraph λ-terms.

2.1 HyperLMNtal

In HyperLMNtal, *hypergraphs* consist of graph nodes called *atoms*, undirected edges with two endpoints called *regular links* and edges with multiple endpoints called *hyperlinks*. The simplified syntax of hypergraphs in HyperLMNtal is as follows,

$$(\textit{Hypergraphs})\ \ P ::= 0 \mid p(A_1, \ldots, A_m) \mid P, P$$

where *link names* (denoted by A_i) and *atom names* (denoted by p) are presupposed. *Hypergraphs* are the principal syntactic category: 0 is an empty hypergraph; $p(A_1, \ldots, A_m)$ is an atom with arity m; and P, P is parallel composition. A hypergraph P is transformed by a rewrite rule of the form H `:-` G`|` B when a subgraph of P matches (i.e., is isomorphic to) H and auxiliary conditions specified in G are satisfied, in which case the subgraph of P is rewritten into another hypergraph B. The auxiliary conditions include type constraints and equality constraints. In HyperLMNtal programs, names starting with lowercase letters denote atoms and names starting with uppercase letters denote links. An abbreviation called *term notation* is frequently used in HyperLMNtal programs. It allows an atom b without its final argument to occur as an argument of a when these two arguments are interconnected by regular links. For instance, `f(a,b)` represents the graph `f(A,B),a(A),b(B)`, and `C=app(A,B)` represents the graph `app(A,B,C)`. The latter example shows that an n-ary constructor can be represented by an $(n+1)$-ary HyperLMNtal atom whose final argument stands for the root link of the constructor.

In a rewrite rule, placing a constraint `new(`A`,`a`)` in the guard means that `A` is created as a hyperlink with an *attribute* a given as a natural number. A type constraint specified in the guard describes a class of graphs with specific shapes. For example, a graph type `hlink(`A`)` ensures that A is a hyperlink occurrence. A graph type `hlground(`$A, a_1, \ldots, a_n$`)` identifies a subgraph rooted at the link A, where $a_1, \ldots, a_n$ are the attributes of hyperlinks which are allowed to occur in the subgraph. The identified subgraph may be copied or removed according to rewrite rules. Details appear in Section 2.2.

2.2 Hypergraph λ-Terms

We write *hypergraph λ-terms* by the following syntax.

$$\begin{array}{lll} (\textit{Terms})\ \ M ::= A & & \textit{variables} \\ \quad\quad\quad\quad\ \ \texttt{abs}(A, M) & & \textit{abstractions} \\ \quad\quad\quad\quad\ \ \texttt{app}(M, M) & & \textit{applications} \end{array}$$

Here, the A are hyperlinks whose attributes are determined as follows: hyperlinks representing variables bound inside M or in a larger term containing M are given attribute 1 (denoted A^1), while those not bound anywhere are given attribute 2 (denoted A^2). Hypergraph λ-terms are straightforwardly obtained from λ-terms. For example, the Church numeral 2

$$\lambda x.\lambda y.x(xy)$$

is written as

```
R=abs(A,abs(B,app(A,app(A,B)))).
```

Note that both `abs` and `app` are ternary atoms, where their third arguments, made implicit by the term notation, are links connected to their parent atoms or represented by the leftmost `R`.

The following rewrite rules shows how to work with hypergraph λ-terms in HyperLMNtal.

```
N=n(2) :- new(A,1), new(B,1) | N=abs(A,abs(B,app(A,app(A,B)))).
init :- r=app(n(2),n(2)).
init.
```

The first rule creates a hypergraph representing the Church numeral 2. The second rule creates an application of two Church numerals.

The idea behind the hypergraph-based approach is that it applies the principle of *Barendregt's variable convention* (bound variables should be separated from free variables to allow easy reasoning) also to bound variables; all bound variables should be distinct from each other upon creation and should be kept distinct from each other during substitution. Besides keeping bound variables distinct, one should avoid variable capture during substitution.

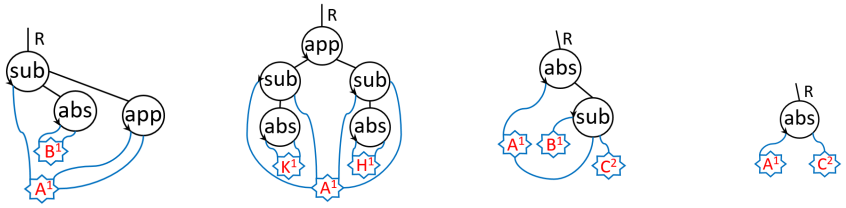
In a substitution $(\lambda y.M)[x := N]$, replacing x with N in M will not lead to variable capture if y is kept distinct from the variables of N. The idea is to ensure that variables appear distinctly in M_1 and M_2 in an application M_1M_2. Concretely, in a substitution $(M_1M_2)[x := N]$, we generate two α-equivalent but syntactically different copies of N, say N_1 and N_2, to have $(M_1[x := N_1])(M_2[x := N_2])$. For a hypergraph λ-term with distinct variables, applying such strategy in the substitution ensures that $y \notin fv(N)$ for $(\lambda y.M)[x := N]$. To summarize, we use distinct hyperlinks with appropriate attributes to represent distinct variables of λ-terms and don't allow multiple binders of the same variable.

We use `sub` atoms to represent substitutions; `R=sub(`X`,`N`,`M`)` represents $M[x := N]$. The definition of substitutions for hypergraph λ-terms is given in Fig. 2, where each rule is prefixed by a rule name. The rule `beta` implements β-reduction, and the other four rules implement substitutions. When the rule `var2` is applied, a subgraph matched with `hlground(N,1)` is removed. When the rule `app` is applied, two α-equivalent but syntactically different copies of a subgraph matched by `hlground(N,1)` are created. The `hlink(X)` checks if `X` is a hyperlink.

The graph type `hlground(N,1)` identifies a subgraph rooted at `N`, then rewriting may copy or remove the subgraph. When copying a subgraph identified by `hlground(N,1)` in a rule, it creates fresh copies of hyperlinks which have the attribute `1` and have no occurrences outside of the subgraph, while it shares hyperlinks which have the attribute `1` but have occurrences outside of the subgraph between the copies of the subgraph. It always shares hyperlinks which have an attribute different from `1` between the copies of the subgraph. When removing a

```
beta@@ R=app(abs(X,M),N)     :- R=sub(X,N,M).
var1@@ R=sub(X,N,X)          :- hlink(X) | R=N.
var2@@ R=sub(X,N,Y)          :- X\=Y, hlground(N,1) | R=Y.
abs@@  R=sub(X,N,abs(Y,M))   :- R=abs(Y,sub(X,N,M)).
app@@  R=sub(X,N,app(M1,M2)):- hlink(X), hlground(N,1) |
                                R=app(sub(X,N,M1), sub(X,N,M2)).
```

Fig. 2: Definition of substitutions on hypergraph λ-terms



(a) (b) (c) (d)

Fig. 3: Applying a substitution on an application

subgraph identified by `hlground(N,1)` in a rule, it removes the subgraph along with all hyperlink endpoints in the subgraph.

For example, the rule `app` rewrites `R=sub(A,abs(B,B),app(A,A))` in Fig. 3a to `R=app(sub(A,abs(K,K),A),sub(A,abs(H,H),A))` in Fig. 3b, where the constraint `hlground(N,1)` identifies a subgraph `N=abs(B,B)` which is copied into `abs(K,K)` and `abs(H,H)`. The rule `var2` rewrites `R=abs(A,sub(B,A,C))` in Fig. 3c to `R=abs(A,C)` in Fig. 3d, where `hlground(N,1)` identifies a subgraph `N=A` and then the subgraph containing one endpoint of `A` is removed. For more details of `hlground`, readers are referred to our previous work [16].

3 Unification

We extend hypergraph λ-terms with *unknown variables* of unification problems, denoted by $X, Y, \ldots$, in a standard manner. Let A, B, C, D be hyperlinks, M, N, P be some hypergraph λ-terms, and L, R be regular links occurring as the last arguments of the atoms representing λ-term constructors.

The assumed equality between hypergraph λ-terms in our unification is α-equivalence with freshness constraints. When no confusion may arise, we write $=$ instead of $=_\alpha$ for the sake of simplicity. For a unification problem $M = N$ of two hypergraphs M and N containing unknown variables $X, Y, \ldots$, the goal is to find hypergraph λ-terms which replace $X, Y, \ldots$ and ensure the α-equivalence of M and N. To reason about the equality of non-ground hypergraph λ-terms (hypergraphs containing unknown variables), we use the concepts of *swapping* $\leftrightarrow$ and *freshness* $\#$ from the nominal approach [4].

Lemma 1. *In hypergraph λ-terms, for an abstraction* `L=abs(A,M)`*, the hyperlink A occurs in M only.*

Proof. Follows from the construction of hypergraph λ-terms. □

Henceforth, note that the last arguments of atoms representing λ-term constructors are implicit in terms related by = and #.

Lemma 2. *For two α-equivalent hypergraph λ-terms*

$$\mathtt{abs}(A, M) = \mathtt{abs}(B, N) \ ,$$

the following holds,

- $A\#N$ *and* $B\#M$*,*
- $M = [A \leftrightarrow B]N$ *and* $[A \leftrightarrow B]M = N$*,*

where $A\#N$ *denotes that* A *is fresh for* N *(or* A *is not in* N*) and* $[A \leftrightarrow B]N$ *denotes the swapping of* A *and* B *in* N*.*

Proof. Follows from Lemma 1 and the fact that hyperlinks representing bound variables are distinct in hypergraph λ-terms. □

In Lemma 2, we could use renaming $M = [A/B]N$ and $[B/A]M = N$ instead of swapping, where $[A/B]N$ means replacing B by A in N. Moving $[A/B]$ to the left-hand side of = requires the switching of A and B. Using swapping saves us from such switching operation in the implementation. Another point is that it is clear from their definitions that swapping subsumes renaming. In $[A \leftrightarrow B]N$, swapping $[A \leftrightarrow B]$ applies to every hyperlink in N until it reaches an unknown variable X occurring in N. We *suspend* swapping when it encounters an unknown variable X until X is instantiated to a non-variable term in the future.

Definition 1. *Let* π *be a list of swappings* $[A_1 \leftrightarrow B_1, \ldots, A_n \leftrightarrow B_n]$*,* $var(\pi) = \{A_1, B_1, \ldots, A_n, B_n\}$*, and* $\pi^{-1} = [A_n \leftrightarrow B_n, \ldots, A_1 \leftrightarrow B_1]$*. Applying* π *to a term* M *is written as* $\pi \cdot M$*. When* M *is an unknown variable* X*, we call* $\pi \cdot M$ *a suspension. The inductive definition of applying swappings to hypergraph λ-terms is defined as follows, where* $\pi @ \pi'$ *is a concatenation of* π *and* π'*.*

$$\begin{aligned}
\pi@[A \leftrightarrow C] \cdot B &\stackrel{def}{=} \pi \cdot B \qquad (A \neq B, B \neq C)\\
\pi@[A \leftrightarrow C] \cdot A &\stackrel{def}{=} \pi \cdot C\\
\pi@[C \leftrightarrow A] \cdot A &\stackrel{def}{=} \pi \cdot C\\
\pi \cdot \mathtt{abs}(A, M) &\stackrel{def}{=} \mathtt{abs}(A, \pi \cdot M)\\
\pi \cdot \mathtt{app}(M, N) &\stackrel{def}{=} \mathtt{app}(\pi \cdot M, \pi \cdot N)\\
\pi \cdot (\pi' \cdot M) &\stackrel{def}{=} \pi@\pi' \cdot M\\
[\,] \cdot M &\stackrel{def}{=} M
\end{aligned}$$

We don't apply swapping to hyperlinks representing the bound variables of an `abs` (the fourth rule in Definition 1) because all bound variables are distinct in hypergraph λ-terms, and a swapping is only created from two abstractions using the rule `=abs` in Fig. 4. We use a *freshness constraint* $\#$ in the equality judgment of non-ground hypergraph λ-terms, and write $\theta \vdash M = N$ to denote that M and N are α-equal terms under a set θ of freshness constraints called a *freshness environment*. For example,

$$\{\mathtt{A}\#X, \mathtt{B}\#X\} \vdash \mathtt{abs(A,}X\mathtt{)} = \mathtt{abs(B,}X\mathtt{)}$$

is a valid judgment. Likewise, we write $\theta \vdash A\#M$ to say that $A\#M$ holds under θ. For example, $\mathtt{A}\#X \vdash \mathtt{A}\#\mathtt{app(}X\mathtt{,B)}$ is a valid judgment. With swapping and freshness constraints, judging the equality of two non-ground hypergraph λ-terms is simple, as shown in Fig. 4.

$$\frac{}{\theta \vdash A = A} \quad \texttt{=hlink}$$

$$\frac{\theta \vdash M = [A \leftrightarrow B] \cdot N \quad \theta \vdash A\#N \quad \theta \vdash B\#M}{\theta \vdash \mathtt{abs}(A, M) \;=\; \mathtt{abs}(B, N)} \quad \texttt{=abs}$$

$$\frac{\theta \vdash M_1 = M_2 \quad \theta \vdash N_1 = N_2}{\theta \vdash \mathtt{app}(M_1, N_1) \;=\; \mathtt{app}(M_2, N_2)} \quad \texttt{=app}$$

$$\frac{(A\#X) \in \theta \text{ for all } A \in var(\pi @ \pi')}{\theta \vdash \pi \cdot X = \pi' \cdot X} \quad \texttt{=susp}$$

$$\frac{A \neq B}{\theta \vdash A\#B} \quad \texttt{\#hlink}$$

$$\frac{\theta \vdash A\#N}{\theta \vdash A\#\mathtt{abs}(B, N)} \quad \texttt{\#abs}$$

$$\frac{\theta \vdash A\#M \quad \theta \vdash A\#N}{\theta \vdash A\#\mathtt{app}(M, N)} \quad \texttt{\#app}$$

$$\frac{(\pi^{-1} \cdot A\#X) \in \theta}{\theta \vdash A\#\pi \cdot X} \quad \texttt{\#susp}$$

Fig. 4: The equality and freshness judgments for non-ground hypergraph λ-terms

The soundness of most of the rules in Fig. 4 should be self-evident. Below we give some lemmas to justify `=susp` and `#susp`. It is important to note that the rules in Fig. 4 are assumed to be used in a goal-directed manner starting from hypergraph λ-terms M and N. In the following lemmas, *"obtained by applying rules in Fig. 4 and Definition 1"* means that we use the rules in Fig. 4 in goal-directed, backward manner and the rules in Definition 1 in the left-to-right direction. By doing so, we come up with a set of unification rules which works on two unifiable terms and fails for two non-unifiable terms.

When judging the equality of two non-ground hypergraph λ-terms using the rules in Fig. 4, swappings are only generated by the rule `=abs`, and these swappings are applied to terms by the rules in Definition 1. During such process, we may have terms such as $\theta \vdash \pi \cdot M = \pi' \cdot N$ and $\theta \vdash A \# \pi \cdot M$. As mentioned before, a swapping is always created from two abstractions which have distinct bound hyperlinks. Therefore, in a judgment, swappings enjoy the following properties: Each swapping always has two distinct hyperlinks, and two swappings generated by the rule `=abs` have no hyperlinks in common. For example, in a judgment, there are no swappings such as $[A \leftrightarrow A]$ and $[A \leftrightarrow B, B \leftrightarrow C]$.

Lemma 3. *If the judgment*

$$\theta \vdash \pi \cdot M = \pi' \cdot N$$

is obtained by applying rules in Fig. 4 and Definition 1, then $var(\pi) \cap var(\pi') = \emptyset$ *holds.*

Proof. Follows from the fact that hyperlinks of a swapping are distinct. □

Note that the rules in Fig. 4 and Definition 1 generate non-empty swappings only to the right-hand side of equations, so the π above is actually empty. Nevertheless, we have non-empty swappings in the left-hand side in this and the following lemmas because the claims generalize to equations generated by the unification algorithm described later in Fig. 5.

Lemma 4. *If the judgment*

$$\theta \vdash \pi \cdot \texttt{abs(A,M)} = \pi' \cdot \texttt{abs(B,N)},$$

is obtained by applying rules in Fig. 4 and Definition 1, then $A \notin var(\pi @ \pi')$ *and* $B \notin var(\pi @ \pi')$ *hold.*

Proof. The same as the proof of Lemma 3. □

The next lemma states how swappings move between two sides of $=$ in a judgment.

Lemma 5. $\theta \vdash M = \pi \cdot N$ *obtained by applying rules in Fig. 4 and Definition 1 holds if and only if* $\theta \vdash \pi^{-1} \cdot M = N$ *holds.*

Proof. ($\Rightarrow$) Let $\pi = [A_1 \leftrightarrow B_1, \ldots, A_n \leftrightarrow B_n]$. Because freshness constraints are generated only from the rule `=abs`, we can assume that $A_1, \ldots, A_n$ occur only in N, that $B_1, \ldots, B_n$ occur only in M, and that θ contains $\{A_1 \# M, \ldots, A_n \# M, B_1 \# N, \ldots, B_n \# N\}$. If $N = A_i$ for some i, then $M = B_i$ by assumption and the rule `=hlink`, in which case $\pi^{-1} \cdot M = A_i$ and the lemma holds. If N is a hyperlink not in $var(\pi)$, then M and N are the same hyperlink not in $var(\pi)$ and the lemma holds obviously. If N is an unknown variable, the lemma is again obvious from the rule `=susp`. The other cases are straightforward by structural induction.
($\Leftarrow$) The proof of the other direction is similar. □

The next lemma justifies the rule `#susp` in Fig. 4.

Lemma 6. *$\theta \vdash A \# \pi \cdot M$ obtained by applying rules in Fig. 4 and Definition 1 holds if and only if $\theta \vdash \pi^{-1} \cdot A \# M$ holds.*

Proof. ($\Rightarrow$) By Lemma 4 and the fact that freshness constraints are created by the rule `=abs`, we know that $A \notin var(\pi)$. Therefore, if $\theta \vdash A \# \pi \cdot M$, $\theta \vdash \pi^{-1} \cdot A \# M$ holds.
($\Leftarrow$) For the same reason, $A \notin var(\pi^{-1})$. Therefore, if $\theta \vdash \pi^{-1} \cdot A \# M$ holds, $\theta \vdash A \# \pi \cdot M$ holds. □

The next lemma justifies the rule `=susp` in Fig. 4.

Lemma 7. *$\theta \vdash \pi \cdot M = \pi' \cdot M$ obtained by applying rules in Fig. 4 and Definition 1 holds for π and π' if and only if $A \# M \in \theta$ for all $A \in var(\pi @ \pi')$.*

Proof. ($\Rightarrow$) By lemma 3, we know that $var(\pi) \cap var(\pi') = \emptyset$. Therefore, in order for $\theta \vdash \pi \cdot M = \pi' \cdot M$ to hold, π and π' should have no effects on M, which means $var(\pi @ \pi') \cap var(M) = \emptyset$, which is the same as $A \# M \in \theta$ for all $A \in var(\pi @ \pi')$.
($\Leftarrow$) If $A \# M \in \theta$ for all $A \in var(\pi @ \pi')$, obviously, $\theta \vdash \pi \cdot M = \pi' \cdot M$ holds. □

Theorem 1. *The relation $=$ defined in Fig. 4 is an equivalence relation, i.e.,*

(a) $\theta \vdash M = M$,
(b) $\theta \vdash M = N$ implies $\theta \vdash N = M$,
(c) $\theta \vdash M = N$ and $\theta \vdash N = P$ implies $\theta \vdash M = P$.

Proof.
(a) When M is a hyperlink A, then $A = A$ follows from the rule `=hlink`. When M is an abstraction, note that M stands for an α-equivalence class. For example, M stands for either M=`abs(A,A)` or M=`abs(B,B)`. Assume $P = P$ (as induction hypothesis), $A \# P$, and that B occurs in P, then $P = [A \leftrightarrow B] @ [B \leftrightarrow A] \cdot P$ holds. Let $N = [B \leftrightarrow A] \cdot P$, then it is clear that $B \# N$. Clearly, `abs(B,P)` = `abs(A,N)` holds, therefore $M = M$ holds for abstractions. When M is an application, the proof is again by structural induction. The equivalence of terms containing suspension follows from the rule `=susp` and Lemma 7.
(b) When M and N are hyperlinks, $\vdash M = N$ by the rule `=hlink` simply implies $\vdash N = M$. When M and N are M=`abs(`A, N_1`)` and N=`abs(`B, N_2`)` respectively, $\vdash M = N$ leads to $\vdash N_1 = [A \leftrightarrow B] \cdot N_2$, $\vdash A \# N_2$ and $\vdash B \# N_1$ by the rule `=abs`. By Lemma 5 and the induction hypothesis, we have $\vdash N_2 = [A \leftrightarrow B] \cdot N_1$, $\vdash A \# N_2$ and $\vdash B \# N_1$ which leads to `abs(`B, N_2`)` = `abs(`A, N_1`)`. When M and N are applications, the proof is by the rule `=app` and using the induction hypothesis twice. The equivalence of terms containing suspension follows from the rule `=susp` and Lemma 7.
(c) When M, N and P are hyperlinks, it holds. When M, N and P are M = `abs(`A, M_1`)`, N = `abs(`B, M_2`)` and P = `abs(`C, M_3`)`, we have $\vdash M_1 = [A \leftrightarrow B] \cdot M_2$, $\vdash A \# M_2$, $\vdash B \# M_1$ and $\vdash M_2 = [B \leftrightarrow C] \cdot M_3$, $\vdash B \# M_3$, $\vdash C \# M_2$ by `=abs`. By Lemma 1, we know that $A \# M_3$ and $C \# M_1$. By Lemma 5 and the

`=hln`	$\{A = A\} \cup P, \delta$	$\Longrightarrow P, \delta$
`=abs`	$\{\mathtt{abs}(A, M) = \mathtt{abs}(B, N)\} \cup P, \delta$	$\Longrightarrow \{M = [B \leftrightarrow A]N, A\#N, B\#M\} \cup P, \delta$
`=app`	$\{\mathtt{app}(M_1, N_1) = \mathtt{app}(M_2, N_2)\} \cup P, \delta$	$\Longrightarrow \{M_1 = M_2\,,\ N_1 = N_2\} \cup P, \delta$
`=rm`	$\{\pi \cdot X = \pi' \cdot X\} \cup P, \delta$	$\Longrightarrow P, \delta$
`=var`	$\left.\begin{array}{r}\{M = \pi \cdot X\}\\ \{\pi \cdot X = M\}\end{array}\right\} \cup P, \delta$	$\Longrightarrow \begin{array}{l}\delta'(P), \delta' \circ \delta, \text{ where } \delta' = [X := \pi^{-1} \cdot M]\\ \qquad\qquad \text{provided } X \text{ does not occur in } M\end{array}$
`#hln`	$\{A\#B\} \cup P, \delta,$	$\Longrightarrow P, \delta$
`#abs`	$\{A\#\mathtt{abs}(B, N)\} \cup P, \delta$	$\Longrightarrow \{A\#N\} \cup P, \delta$
`#app`	$\{A\#\mathtt{app}(M, N)\} \cup P, \delta$	$\Longrightarrow \{A\#M, A\#N\} \cup P, \delta$
`#sus`	$\{A\#\pi \cdot X\} \cup P, \delta$	$\Longrightarrow \{\pi^{-1} \cdot A\#X\} \cup P, \delta$

Fig. 5: Unification of hypergraph λ-terms

induction hypothesis, we have $\{A\#M_3, C\#M_1\} \vdash M_1 = [A \leftrightarrow B]@[B \leftrightarrow C]\cdot M_3$, which is the same as $\{A\#M_3, C\#M_1\} \vdash M_1 = [A \leftrightarrow C] \cdot M_3$, which leads to $\vdash \mathtt{abs}(A, M_1) = \mathtt{abs}(C, M_3)$ by `=abs`. The proof of applications is trivial. The equivalence of terms containing suspension follows from the rule `=susp` and Lemma 7. □

A substitution δ is a finite set of mappings from unknown variables to terms, written as $[X := M_1, Y := M_2, \dots]$ where its domain, $dom(\delta)$, is a set of distinct unknown variables $\{X, Y, \dots\}$. Applying δ to a term M is written as $\delta(M)$ and is defined in a standard manner. A composition of substitutions is written as $\delta \circ \delta'$ and defined as $(\delta \circ \delta')(M) = \delta(\delta'(M))$. The ε denotes an identity substitution. Substitution commutes with *swapping*; i.e., $\delta(\pi \cdot M) = \pi \cdot (\delta(M))$. For example, applying $[X := A]$ to $[A \leftrightarrow B]\cdot\mathtt{app}(N, X)$ will result in $\mathtt{app}(N, B)$. For two sets of freshness constraints θ and θ', and substitutions δ and δ', writing $\theta' \vdash \delta(\theta)$ means that $\theta' \vdash A\#\delta(X)$ holds for all $(A\#X) \in \theta$, and $\theta \vdash \delta = \delta'$ means that $\theta \vdash \delta(X) = \delta'(X)$ for all $X \in dom(\delta) \cup dom(\delta')$.

The definitions of unification, most general unifiers and idempotent unifiers are similar to the ones in nominal unification [14]. A unification problem P is a finite set of equations over hypergraph λ-terms and freshness constraints. Each equation $M = N$ may contain unknown variables $X, Y, \dots$. A solution of P is a unifier denoted as (θ, δ), consisting of a set θ of freshness constraints and a substitution δ. A unifier (θ, δ) of a problem P equates every equation in P, i.e., establishes $\theta \vdash \delta(M) = \delta(N)$. $\mathcal{U}(P)$ denotes the set of unifiers of a problem P. For P, a unifier $(\theta, \delta) \in \mathcal{U}(P)$ is a *most general unifier* if for any unifier $(\theta', \delta') \in \mathcal{U}(P)$, there is a substitution δ'' such that $\theta' \vdash \delta''(\theta)$ and $\theta' \vdash \delta'' \circ \delta = \delta'$. A unifier $(\theta, \delta) \in \mathcal{U}(P)$ is idempotent if $\theta \vdash \delta \circ \delta = \delta$.

The unification algorithm is described in Fig. 5, where P is a given unification problem and δ is a substitution which is usually initialized to ε. Each rule

arbitrarily selects an equation or a freshness constraint from P and transforms it accordingly. The rule `=abs` transforms an equation and creates two freshness constraints, where all freshness constraints we need are obtained. That is why the rule `=rm` simply deletes an equation without creating any freshness constraints. The rule `=var` creates a substitution δ' from an equation (if $X \notin M$), applies δ' to P and adds δ' to δ. The rules in Fig. 5 essentially correspond to the rules in Fig. 4 except for the rule `=var`. The next lemma justifies the rule `=var`.

Lemma 8. *Substitution generated by the rule* `=var` *in Fig. 5 preserves* $=$ *and* $\#$ *obtained by applying rules in Fig. 4. That is,*

(a) If $\theta' \vdash \delta(\theta)$ *and* $\theta \vdash M = N$ *hold, then* $\theta' \vdash \delta(M) = \delta(N)$ *holds.*
(b) If $\theta' \vdash \delta(\theta)$ *and* $\theta \vdash A \# M$ *hold, then* $\theta' \vdash A \# \delta(M)$ *holds.*

Proof. The proof of both is by structural induction. *(a)* We only show the case of abstraction. Assume $M = \texttt{abs(A,X)}$, $N = \texttt{abs(B,Y)}$, $\delta = [\texttt{X} := P_1, \texttt{Y} := P_2]$. Then we have $\theta = \{\texttt{A}\#\texttt{Y}, \texttt{B}\#\texttt{X}\}$, $\theta \subseteq \theta'$, $\texttt{A}\#P_2$, and $\texttt{B}\#P_1$. From $\theta \vdash M = N$, we have $\texttt{X} = [\texttt{B} \leftrightarrow \texttt{A}]Y$. Using $\texttt{A}\#P_2$ and $\texttt{B}\#P_1$, and by the induction hypothesis, $P_1 = [\texttt{B} \leftrightarrow \texttt{A}]P_2$ holds. Therefore, $\theta' \vdash \delta(\texttt{abs(A,X)}) = \delta(\texttt{abs(B,Y)})$ holds. *(b)* The proof is by structural induction. □

Terms in the hypergraph approach and the nominal approach are first-order terms without built-in β-reduction. To represent bound variables, the nominal approach uses concrete names and the hypergraph approach uses hyperlinks which are identified by names when writing hypergraph terms as text. Our unification and nominal unification both assume α-equality for terms. Therefore, it is not surprising that our unification algorithm happens to be similar to the nominal unification algorithm. Nevertheless, there are differences. Our algorithm does not have a rule for handling two abstractions with the same bound variable. Also, the rule `=rm` is different from the $\approx$?`-suspension` rule in nominal unification [14]. This is because Lemma 7 is different from its counterpart in nominal unification: the former states the freshness of every variable of $\pi @ \pi'$ and the latter states the freshness of the variables in the *disagreement set* of π and π'.

Theorem 2. *For a given unification problem* P*, the unification algorithm in Fig. 5 either fails if* P *has no unifier or successfully produces an idempotent most general unifier.*

Proof. Given in Appendix with related lemmas. The structure of the proof in [14] applies to our case basically, though our formalization allows the interleaving of the `=` and `#` rules of the algorithm. □

4 Examples of The Unification

We apply the unification algorithm in Fig. 5 to three unification problems.

Example 1. A unification problem

$$\texttt{abs(A,abs(B,}X\texttt{))} = \texttt{abs(C,abs(D,}X\texttt{))}$$

has a solution.

$\{\texttt{abs(A,abs(B,}X\texttt{))} = \texttt{abs(C,abs(D,}X\texttt{))}\}$, ε
$\{\texttt{abs(B,}X\texttt{)} = [\texttt{C} \leftrightarrow \texttt{A}] \cdot \texttt{abs(D,}X\texttt{)}$, $\texttt{A\#abs(D,}X\texttt{)}$, $\texttt{C\#abs(B,}X\texttt{)}\}$, ε (=abs)
$\{X = [\texttt{D} \leftrightarrow \texttt{B}, \texttt{C} \leftrightarrow \texttt{A}] \cdot X$, $\texttt{A\#}X$, $\texttt{C\#}X$, $\texttt{B\#}[\texttt{C} \leftrightarrow \texttt{A}] \cdot X$, $\texttt{D\#}X\}$, ε (=abs,#abs,#hln)
$\{\texttt{A\#}X$, $\texttt{C\#}X$, $\texttt{B\#}X$, $\texttt{D\#}X\}$, ε (=rm,#sus)
Success

The problem has the most general unifier $(\{\texttt{A\#}X, \texttt{C\#}X, \texttt{B\#}X, \texttt{D\#}X\}, \varepsilon)$, which says that X can be any term not containing `A`, `B`, `C` or `D`.

Example 2. A unification problem

$$\texttt{abs(A,abs(B,app(}X\texttt{,B)))} = \texttt{abs(C,abs(D,app(D,}X\texttt{)))}$$

has no solution.

$\{\texttt{abs(A,abs(B,app(}X\texttt{,B)))} = \texttt{abs(C,abs(D,app(D,}X\texttt{)))}\}$, ε
$\{\texttt{abs(B,app(}X\texttt{,B))} = [\texttt{C} \leftrightarrow \texttt{A}] \cdot \texttt{abs(D,app(D,}X\texttt{))}$, (=abs)
$\texttt{A\#abs(D,app(D,}X\texttt{))}$, $\texttt{C\#abs(B,app(}X\texttt{,B))}\}$, ε
$\{\texttt{app(}X\texttt{,B)} = [\texttt{D} \leftrightarrow \texttt{B}] \cdot \texttt{app(D,}[\texttt{C} \leftrightarrow \texttt{A}] \cdot X\texttt{)}$, (=abs,#abs,#app,#hln)
$\texttt{A\#X}$, $\texttt{C\#X}$, $\texttt{B\#}$ $\texttt{app(D,}[\texttt{C} \leftrightarrow \texttt{A}] \cdot X\texttt{)}$, $\texttt{D\#app(}X\texttt{,B)}\}$, ε
$\{X = \texttt{B}$, $\texttt{B} = [\texttt{D} \leftrightarrow \texttt{B}$, $\texttt{C} \leftrightarrow \texttt{A}] \cdot X$, $\texttt{A\#}X$, $\texttt{C\#}X$, $\texttt{D\#}X$, $\texttt{B\#}X\}$, ε (=app,#app,#hln,#sus)
$\{\texttt{B} = \texttt{D}$, $\texttt{B\#B}\}$, $[\texttt{X} := \texttt{B}]$ (=var,#hln)
Failure

The problem is unsolvable; it fails due to both $\texttt{B} = \texttt{D}$ and $\texttt{B\#B}$.

Example 3. A unification problem

$$\texttt{abs(A,app(}X\texttt{,}Y\texttt{))} = \texttt{abs(B,app(app(B,}Y\texttt{),}X\texttt{))}$$

has no solution.

$\{\texttt{abs(A,app(}X\texttt{,}Y\texttt{))} = \texttt{abs(B,app(app(B,}Y\texttt{),}X\texttt{))}\}$, ε
$\{\texttt{app(}X\texttt{,}Y\texttt{)} = [\texttt{B} \leftrightarrow \texttt{A}] \cdot \texttt{app(app(B,}Y\texttt{),}X\texttt{)}$, (=abs)
$\texttt{A\#app(app(B,}Y\texttt{),}X\texttt{)}$, $\texttt{B\#app(}X\texttt{,}Y\texttt{)}\}$, ε
$\{X = \texttt{app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}$, $Y = [\texttt{B} \leftrightarrow \texttt{A}] \cdot X$, (=app,#app,#hln)
$\texttt{A\#}X$, $\texttt{A\#}Y$, $\texttt{B\#}X$, $\texttt{B\#}Y\}$, ε
$\{Y = [\texttt{B} \leftrightarrow \texttt{A}] \cdot \texttt{app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}, \texttt{A\#app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}$, (=var)
$\texttt{A\#}Y$, $\texttt{B\#app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}$, $\texttt{B\#}Y\}$, $[X := \texttt{app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}]$
$\{Y = \texttt{app(B,}[\texttt{B} \leftrightarrow \texttt{A}, \texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}$, $\texttt{A\#}Y$, $\texttt{B\#}Y$, $\texttt{A\#A}$, $\texttt{A\#A}$, $\texttt{B\#}Y\}$, (#app,#hln,#sus)
$[X := \texttt{app(A,}[\texttt{B} \leftrightarrow \texttt{A}] \cdot Y\texttt{)}]$
Failure

The problem is unsolvable; it fails due to $\texttt{A\#A}$.

5 Implementation

We implemented the unification of hypergraph λ-terms in HyperLMNtal in a straightforward manner[1]. There are a total of 52 rewrite rules in the implementation; 12 rewrite rules corresponding to the 9 rules in Fig. 5 (4 rules for the `=var` rule), 14 rules for the occur-check, 7 rules for implementing applying swapping to terms, 7 rewrite rules for substitution, and several auxiliary rules for list management. Interestingly, the implementation of substitution $M[X := N]$ turned out to be essentially the same as that for the λ-calculus, i.e., $\texttt{sub}(X, N, M)$ in Fig. 2. The implementation solved a number of unification problems, including the examples in this paper. HyperLMNtal brought simplicity in the sense that the rewrite rules of the implementation are extremely close to the unification rules discussed in this paper.

6 Related Work and Conclusion

Complexity of formalizing unification over terms containing name binding is largely determined by the approach taken for representing such terms. There are two prominent unification algorithms: higher-order pattern unification [8] and nominal unification [14].

A higher-order approach implements a variant of the λ-calculus as a meta-language, which is used to encode formal systems involving name binding [9]. The meta-language implicitly handles substitution and implicitly restricts bound variables to be distinct. Users reason about formal systems indirectly through the meta-language, in which terms are higher-order terms. Higher-order pattern unification unifies equations of terms modulo $=_{\alpha\beta_0\eta}$. It finds functions to substitute unknown variables, which means that variable capture never happens. The characteristics of higher-order pattern unification are the result of letting the meta-language handle everything implicitly. In the nominal approach, boundable *names* are equipped with *swapping* and *freshness* to ensure correct substitutions [4]. Users reason on formal systems through nominal terms which are first-order terms. As the result, nominal unification solves equations of terms modulo $=_\alpha$, because $=_{\beta\eta}$ is not needed for first-order terms, and allows for variable capture in the unification while preserving α-equivalence. We believe that having no restrictions on bound variables is the cause of somewhat complex proofs in the nominal unification. One observation is that using a higher-order meta-language implicitly ensures the distinctness of bound variables in the higher-order approach. In the nominal approach, such restriction on bound variables does not exist.

Our approach uses hyperlinks to represent variables, hypergraphs to represent terms and `hlground` followed by hypergraph copying to avoid variable capture. Unlike the nominal approach, we use fresh hyperlinks whenever needed and `hlground` manages hyperlinks. In our approach, it is natural to restrict a hyperlink to be bound only once and every abstraction is syntactically unique. Just

[1] Implementation is available at https://gitlab.com/alimjanyasin .

like nominal unification, our unification only considers α-equivalence and allows variable capture in the unification. The key idea of our technique is that implementing α-renaming (as the copying of hypergraphs identified by `hlground`) leads to the simplification of overall reasoning. Urban pointed out that the proofs of nominal unification in [14] are clunky and presented simpler proofs in [15]. Proofs in this paper are even somewhat simpler than the proofs in [15]. In our unification algorithm, the basic properties are easy to establish; Lemmas 4, 5, 6 and 7 are intuitive and simple. In particular, we proved equivalence relation (Theorem 1) without much efforts.

To conclude, we worked on the unification of hypergraph λ-terms and the result shows that our approach has taken the promising strategy as indicated by simple proofs of fundamental properties needed for the unification algorithm. We successfully implemented the unification algorithm in HyperLMNtal. This work suggests that our hypergraph rewriting framework provides a convenient platform to work with formal systems involving name bindings and unification of their terms. In the future, we plan to use this unification algorithm to encode type inferences of formal systems involving name binding. Besides, it should be interesting to reformalize logic programming languages such as αProlog [3] using our hypergraph-based approach and implement them in HyperLMNtal to see how much simplicity our approach can provide in practice.

Acknowledgement

The authors are indebted to anonymous referees for their useful comments and pointers to the literature. This work is partially supported by Grant-In-Aid for Scientific Research ((B)26280024), JSPS, Japan, and Waseda University Grant for Special Research Projects.

References

1. H. Barendregt: The Lambda Calculus: its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, North-Holland, **103** (1984).
2. James Cheney: Relating Nominal and Higher-Order Pattern Unification. In Proceedings of the 19th International Workshop on Unification, LNCS 3132, Springer-Verlag, 104–119 (2005).
3. James Cheney, Christian Urban: αProlog: A Logic Programming Language with Names, Bindings and α-Equivalence. In Proceedings of the 20th International Conference on Logic Programming, 269–283 (2004).
4. M. J. Gabbay, A. M. Pitts: A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing, **13**, 341–363 (2002).
5. G. J. Huet: A Unification Algorithm for Typed λ-Calculus. Theoretical Computer Science,**1**(1), 27–57 (1975).
6. Jordi Levy, Mateu Villaret: Nominal Unification from a Higher-Order Perspective. In Proceedings of Rewriting Techniques and Applications, LNCS 5117, Springer-Verlag, 246–260 (2008).

7. Alberito Martelli, Ugo Montanari: An Efficient Unification Algorithm. ACM Transactions on Programming Languages and Systems, **4**(2), 258–282 (1982).
8. Dale Miller: A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. J. Logic and Comput, **1**, 497–536 (1991).
9. Frank Pfenning, Conal Elliott: Higher-Order Abstract Syntax. In Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, 199–208 (1988).
10. Xiaochu Qi: An Implementation of the Language Lambda Prolog Organized around Higher-Order Pattern Unification. Ph. D. thesis, University of Minnesota (2009)
11. J. A. Robinson: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM, **12**(1), 23–41 (1965).
12. Kazunori Ueda: Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting. In Proceedings of Rewriting Techniques and Applications, LNCS 5117, Springer-Verlag, 392–408 (2008).
13. Kazunori Ueda, Seiji Ogawa: HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model. Küstliche Intelligenz, **26**(1), 27–36 (2012).
14. C. Urban, A.M. Pitts, M.J. Gabbay: Nominal unification. J. Theoretical Computer Science, **323**(1–3), 473-497 (2004).
15. C. Urban: Nominal Unification Revisited. In Proceedings of UNIF 2010, 1–11 (2010).
16. Alimujiang Yasen, Kazunori Ueda: Hypergraph Representation of Lambda-Terms. In Proceegins of 10th International Symposium on Theoretical Aspects of Software Engineering, 113–116 (2016).
17. Alimujiang Yasen, Kazunori Ueda: Name Binding is Easy with Hypergraphs. submitted.
18. Zhenyu Qian: Linear unification of higher-order patterns. In Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development, LNCS 668, Springer-Verlag, 391–405 (1993).
19. N. Bourbaki: Théorie des ensembless, Hermann (1970).
20. Ian Mackie: Efficient λ-Evaluation with Interaction Nets. In Proceedings of Rewriting Techniques and Applications, LNCS 3091, Springer-Verlag, 155–169 (2004).

A Appendix

A.1 Adequacy of Equivalence

The relation $=$ defined in Fig. 4 and the standard α-equivalence $=_\alpha$ (based on graph isomorphism) for ground hypergraph λ-terms are the same.

Proposition 1 (adequacy). *For ground hypergraph λ-terms M and N, the relation $M =_\alpha N$ holds if and only if $\emptyset \vdash M = N$ holds in Fig. 4, and $\emptyset \vdash A\#M$ holds if and only if A in not in the set $fv(M)$, defined by*

$$\begin{aligned} fv(A) &\stackrel{def}{=} \{A\} \quad \textit{(A is a hyperlink)}, \\ fv(\texttt{abs}(A, M)) &\stackrel{def}{=} fv(M)\backslash\{A\}, \\ fv(\texttt{app}(M, N)) &\stackrel{def}{=} fv(M) \cup fv(N). \end{aligned}$$

Proof. Let M and N be hyperlinks. If $M =_\alpha N$ holds, then $\emptyset \vdash M = N$ holds by the rule `=hlink`. The other direction is similar. Let M and N be $\texttt{abs}(A, M_1)$ and $\texttt{abs}(B, N_1)$, respectively. If $M =_\alpha N$, this means $M_1 =_\alpha N_1[B := A]$, A is not in N_1 and B is not in M_1. Therefore, $\emptyset \vdash M = N$ holds by the rule `=abs`. If $\emptyset \vdash M = N$, $M =_\alpha N$ is clear from the premise of `=abs`. Let M and N be M_1M_2 and N_1N_2. If $M =_\alpha N$, then we have $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$. Clearly, $\emptyset \vdash M = N$ from the rule `=app`. The other direction is similar.

It is easy to see that $\emptyset \vdash A\#M$ in Fig. 4 and A not being in $fv(M)$ are the same for ground hypergraph λ-terms. If one of them holds, so does the other. □

A.2 Correctness of Unification

Here, we give the details of the correctness proof of the unification algorithm in Fig. 5.

Lemma 9. *The unification algorithm always terminates.*

Proof. To show that the algorithm terminates, we need to define the size of terms $|M|$ as follows.

$$|A| \stackrel{\text{def}}{=} 1$$
$$|\texttt{abs}(A,M)| \stackrel{\text{def}}{=} 1 + |M|$$
$$|\texttt{app}(M,N)| \stackrel{\text{def}}{=} 1 + |M| + |N|$$
$$|\pi \cdot X| \stackrel{\text{def}}{=} 1$$

For a unification problem P, a measure of the size of P is a lexicographically ordered pair of natural numbers (n, m), where n is the number of different unknown variables in P and m is the size of all equations in P, defined as

$$m \stackrel{\text{def}}{=} \sum_{(M=N)\in P} |M| + |N|.$$

The `=` rules in Fig. 5 decrease (n, m). The rule `=var` eliminates one unknown variable, so n decreases. The rule `=rm` decreases m and may decrease n. Other `=` rules decrease m and do not change n.

The `#` rules decrease the size of freshness constraints, which is $\sum_{(A\#M)\in P} |M|$. Eventually, all remaining freshness constraints in a solvable problem P will have the form $A\#X$, for which there are no applicable rules.

For an unsolvable problem P, the algorithm terminates with P containing terms of equations which cannot be made α-equivalent and invalid freshness constraints: (i) $A = B$ where A and B are different hyperlinks; (ii) $M = N$ where M and N start with different constructors such as `abs` and `app`; (iii) one of M and N is a hyperlink and another is a constructor; (iv) $\pi \cdot X = M$ where M is either $\texttt{abs}(A,M_1)$ or $\texttt{app}(M_2,N)$ with X occurring in M_1, M_2 and N; (v) having a freshness constraint such as $A\#A$.

By these facts, we can conclude that the algorithm terminates in both success and failure cases. □

Lemma 10. *if* $\theta \vdash \delta(\pi \cdot X) = \delta(M)$ *then* $\theta \vdash \delta \circ [X := \pi^{-1} \cdot M] = \delta$.

Proof. By commuting δ and π and by Theorem 1 (b), we have $\theta \vdash \delta(M) = \pi \cdot \delta(X)$. By Lemma 5 and commuting again, we have $\theta \vdash \delta(\pi^{-1} \cdot M) = \delta(X)$, which implies $\theta \vdash \delta \circ [X := \pi^{-1} \cdot M] = \delta$. □

Lemma 11. *For a problem* P, $(\theta, \delta) \in \mathcal{U}(\delta_1(P))$ *iff* $(\theta, \delta \circ \delta_1) \in \mathcal{U}(P)$.

Proof. Follows from the definition of substitution composition. □

In Fig. 5, the only rule that creates substitution is the rule `=var`. It is easy to see that `=var` creates a substitution $[X := \pi^{-1} \cdot M]$ with $X \notin dom(\delta)$.

When applying the unification rules, the `=hln`, `=app`, `=rm` and all `#` rules just simplifies some of equations and freshness constraints or removes some of them, without creating anything really new. Interesting ones are the rule `=abs` which creates new freshness constraints and the rule `=var` which creates a new mapping. Therefore, in the following Lemmas, we focus on these two rules.

Lemma 12.

(a) If $(\theta, \delta) \in \mathcal{U}(P)$ *and* $P, \delta \Longrightarrow P', \delta'' \circ \delta$ *using the rule* `=var` *creating* $\delta'' = [X := \pi^{-1} \cdot M]$, *then* $(\theta, \delta) \in \mathcal{U}(P')$ *and* $\theta \vdash \delta \circ \delta'' = \delta$.
(b) If $(\theta, \delta) \in \mathcal{U}(P)$ *and* $P, \delta \Longrightarrow P', \delta$ *using the rule* `=abs` *creating* $\theta'' = \{A\#N, B\#M\}$, *then* $(\theta, \delta) \in \mathcal{U}(P')$ *and* $\theta \vdash \delta(\theta'')$.

Proof.
(a) We can write $P, \delta \Longrightarrow P', \delta'' \circ \delta$ as $P, \delta \Longrightarrow \delta''(P), \delta'' \circ \delta$. By $(\theta, \delta) \in \mathcal{U}(P)$ and $(\pi \cdot X = M)$ or $(M = \pi \cdot X)$ is in P, $\theta \vdash \delta(\pi \cdot X) = \delta(M)$ holds, which leads to $\theta \vdash \delta \circ \delta'' = \delta$ by Lemma 10. By Lemma 11, we have $(\theta, \delta) \in \mathcal{U}(P')$ which is the same as $(\theta, \delta \circ \delta'') \in \mathcal{U}(P)$.
(b) By the assumption, we have $\theta \vdash \delta(\mathtt{abs}(A, M)) = \delta(\mathtt{abs}(B, N))$ and $\theta'' = \{A\#N, B\#M\}$. In order to derive the above, Fig.4 tells that we must have $\theta \vdash A\#\delta(N)$, $\theta \vdash B\#\delta(M)$ and $\theta \vdash \delta(M) = [A \leftrightarrow B] \cdot \delta(N)$, from which the conclusions follow. □

Lemma 13.

(a) If $(\theta, \delta) \in \mathcal{U}(P')$ *and* $P, \delta \Longrightarrow P', \delta'' \circ \delta$ *using the rule* `=var` *creating* $\delta'' = [X := \pi^{-1} \cdot M]$, *then* $(\theta, \delta \circ \delta'') \in \mathcal{U}(P)$.
(b) If $(\theta, \delta) \in \mathcal{U}(P')$ *and* $P, \delta \Longrightarrow P', \delta$ *using the rule* `=abs` *creating* $\theta'' = \{A\#N, B\#M\}$, *then* $(\theta, \delta) \in \mathcal{U}(P)$.

Proof.
(a) $P, \delta \Longrightarrow P', \delta'' \circ \delta$ can be written as $P, \delta \Longrightarrow \delta''(P), \delta'' \circ \delta$. Clearly, $(\theta, \delta \circ \delta'') \in \mathcal{U}(P)$ follows from Lemma 11 and the assumption $(\theta, \delta) \in \mathcal{U}(\delta''(P))$.
(b) The proof is similar to the proof of second part of Lemma 12, but in the opposite direction. □

Theorem 2. *For a given unification problem P, the unification algorithm in Fig. 5 either fails if P has no unifier or successfully produces an idempotent most general unifier.*

Proof. For a unification problem which has no unifiers, the algorithm fails as explained in Lemma 9. For a solvable unification problem P_0, the proof proceeds in three steps: (i) a unifier is generated, (ii) it is most general, and (iii) it is idempotent.

First, the algorithm transforms P_0 as

$$P_0, \delta_0 \Longrightarrow P_1, \delta_1 \Longrightarrow \cdots \Longrightarrow P_n, \delta_n \not\Longrightarrow$$

by substitutions $\delta_1^{'}, \ldots, \delta_n^{'}$ and freshness constraints $\theta_1^{'}, \ldots, \theta_m^{'}$ where $\delta_0 = \varepsilon$, $\delta_1 = \delta_1^{'} \circ \delta_0$, $\ldots$, $\delta_n = \delta_n^{'} \circ \delta_{n-1}$, and the $\theta_i^{'}$ stands for freshness constraints created by the ith application of the rule `=abs`. By the `#` rules in Fig. 5, we know that P_n consists only of freshness constraints of the form $A\#X$. Let us denote P_n as θ. By Lemma 13 and $(\theta, \varepsilon) \in \mathcal{U}(P_n)$, we have $(\theta, \delta) \in \mathcal{U}(P_0)$ where $\delta = \delta_n^{'} \circ \cdots \circ \delta_1^{'}$.

Second, for any other unifier $(\theta', \delta') \in \mathcal{U}(P_0)$, by Lemma 12 we have $\theta' \vdash \delta' \circ \delta_1^{'} = \delta', \ldots, \theta' \vdash \delta' \circ \delta_n^{'} = \delta'$ and $\theta' \vdash \delta'(\theta_1^{'}), \ldots, \theta' \vdash \delta'(\theta_m^{'})$. From the former, we have $\theta' \vdash \delta' \circ \delta_n^{'} \circ \cdots \circ \delta_1^{'} = \delta'$, which is the same as $\theta' \vdash \delta' \circ \delta = \delta'$. From the latter, we have $\theta' \vdash \delta'(\theta'')$ where $\theta'' = \theta_1^{'} \cup \cdots \cup \theta_m^{'}$. From $\theta' \vdash \delta' \circ \delta = \delta'$ and $\theta' \vdash \delta'(\theta'')$, we have $\theta' \vdash (\delta' \circ \delta)(\theta'')$. Since we know that $\delta(\theta'')$ is transformed into θ, we have $\theta' \vdash \delta'(\theta)$. Therefore (θ, δ) is the most general unifier.

Third, since δ' is any unifier, we have $\theta \vdash \delta \circ \delta = \delta$. Therefore (θ, δ) is the idempotent most general unifier. □