IEICE TRANS. INF. & SYST., VOL.E101–D, NO.4 APRIL 2018

PAPER

Name Binding is Easy with Hypergraphs

Alimujiang YASEN[†], ***Nonmember and*** **Kazunori UEDA**[†a)], ***Member***

SUMMARY We develop a technique for representing variable names and name binding which is a mechanism of associating a name with an entity in many formal systems including logic, programming languages and mathematics. The idea is to use a general form of graph links (or edges) called *hyperlinks* to represent variables, graph nodes as constructors of the formal systems, and a graph type called *hlground* to define substitutions. Our technique is based on simple notions of graph theory in which graph types ensure correct substitutions and keep bound variables distinct. We encode strong reduction of the untyped λ-calculus to introduce our technique. Then we encode a more complex formal system called *System* $F_{<:}$, a polymorphic λ-calculus with subtyping that has been one of important theoretical foundations of functional programming languages. The advantage of our technique is that the representation of terms, definition of substitutions, and implementation of formal systems are all straightforward. We formalized the graph type *hlground*, proved that it ensures correct substitutions in the λ-calculus, and implemented *hlground* in HyperLMNtal, a modeling language based on hypergraph rewriting. Experiments were conducted to test this technique. By this technique, one can implement formal systems simply by following the steps of their definitions as described in papers.

key words: *name binding, substitution, hypergraph rewriting, graph type, formal systems*

1. Introduction

1.1 Name Binding

Name binding is present in many fields of computer science. It appears in logic in the form of quantifications such as $\forall x.P$ and $\exists x.Q$ where x is universally quantified in P and existentially quantified in Q. In programming languages, name binding appears in anonymous functions such as Haskell's $\backslash x \rightarrow x + 1$ where $\backslash x$ is a binder that introduces a bound variable x. Besides, proofs in logic and mathematics need to deal with name binding in their formalism.

The best platform for explaining various aspects of name binding is the untyped λ-calculus. In a λ-term $\lambda x.M$, λ is a binding operator and x is a *bound variable* that may occur in the term M which is the *scope* of the binding. Variables that are not bound are called *free variables*. Applying β-reduction to an application $(\lambda x.M)N$ leads to a substitution $M[x := N]$; the term N replaces all the occurrences of the variable x in the term M.

When implementing formal systems involving name binding, complications arise in the definition of substitutions. For instance, applying β-reduction to $(\lambda x.(\lambda y.xy))y$ and naively replacing the occurrences of x by y leads to a wrong result $\lambda y.yy$. The result is wrong because the free variable y in the original term became a bound variable after the substitution, which is called *variable capture*. We could use α-conversion, which regards two λ-terms as equal up to the renaming of bound variables (e.g., $\lambda x.x = \lambda y.y$), to avoid variable capture. By α-conversion, we can rename the bound variable y to another name z to have $(\lambda x.(\lambda z.xz))y$ and carry out the substitution to get a correct result $\lambda z.yz$. It seems that the renaming of bound variables solves the variable capture problem, but the renaming is not necessary at every step of substitution. To judge if renaming is necessary in $(\lambda y.M)[x := N]$, we need to know whether or not y occurs free in N (unless we somehow know that N does not contain y), which is in general a cumbersome process.

Name binding appears in logic, type theory, programming languages and proofs. There has been extensive research on how to handle name binding in practice [6], [8], [18]. Two research communities frequently find themselves in a situation where they have to formalize name binding. Researchers in the programming language community encounter name binding during the development of programming languages both in theory and implementation. Researchers in automated theorem proving have been trying to develop proof systems in which name binding could be easily managed and completely mechanized proofs about the properties of formal systems involving name binding can be correctly generated. Both communities need an intuitive technique which is easy to implement and keeps the formal description close to its informal description [3]. The techniques cited above in this paragraph are mainly oriented towards the mechanized reasoning of metatheories of programming languages. In this paper, we address another direction in which the handling of name binding is important, namely language constructs and techniques for a concrete modeling language that allows us to model, simulate and test formal systems involving name binding in a straightfoward manner.

1.2 Our Solution

Ideally, a technique of name binding should keep terms readable, avoid introducing extra operations in its formalism, and be based on a simple idea. In particular, it should offer an intuitive definition of substitutions as close to

Manuscript received August 9, 2017.
Manuscript revised November 9, 2017.
Manuscript publicized January 12, 2018.

[†]The authors are with the Department of Computer Science and Engineering, Waseda University, Tokyo, 169–8555 Japan.
a) E-mail: ueda@ueda.info.waseda.ac.jp

DOI: 10.1587/transinf.2017EDP7257

theory as possible. By applying such a technique, one should be able to work on the implementation as it is done in the theory. Almost every available technique for name binding fails in one of these aspects. We summarize well-known techniques and compare them to our technique in the final section.

A simple way of representing name binding is using graph links (or edges) as variables. Specifically, when we apply our technique to a formal system, we use *hyperlinks* (links with multiple endpoints) to represent variables, and *atoms* (nodes of graphs) as constructors of the formal system, and graph type *hlground* (described in Sect. 3) to define substitutions. In this technique, we can define more than one kind of hyperlinks to represent different kinds of variables. During the substitution, all bound variables are kept distinct automatically. We implemented the whole idea in HyperLMNtal (pronounced "hyper elemental"), a modeling language based on hypergraph rewriting.

The paper is organized as follows. In Sect. 2, we introduce HyperLMNtal briefly, focusing on its graph elements and graph types. In Sect. 3, we explain our approach of handling name binding by encoding the untyped λ-calculus. Then we describe the graph type *hlground* with generalized semantics and prove that *hlground* ensures correctness of the encoding. In Sect. 4, we use the technique introduced in Sect. 3 to encode a more complex formal system, System $F_{<:}$. In Sect. 5, we give experimental results of the previous two encodings. In Sect. 6, we review related work and conclude the paper.

This work is based on our previous work [24] in which we put forward the basic idea without working implementation. In this paper, we thoroughly examined the basic idea, implemented it in HyperLMNtal, and worked on two nontrivial formal systems involving name binding.

2. Hypergraph Rewriting Model: HyperLMNtal

2.1 Overview of HyperLMNtal

HyperLMNtal [23] is an extension of LMNtal [22], a modeling language based on graph rewriting. Implementation of HyperLMNtal (now integrated into the original LMNtal implementation) is available on the web[†] and features state-space search and LTL model checking. The main idea of HyperLMNtal is that hypergraphs, consisting of *atoms* and *links*, can be used as a platform for various computational models.

The simplified syntax of hypergraphs in HyperLMNtal is given as follows:

$$(\textit{Hypergraphs})\quad P ::= 0 \mid p(X_1, \ldots, X_m) \mid P, P$$

where the two syntactic categories, *links* (denoted by X_i) and *atoms* (denoted by p) are presupposed. A link is either (i) a *regular link* with at most two endpoints or (ii) a *hyperlink*, which may have multiple endpoints, with an attribute which is a natural number. Atoms and links form node-labeled undirected graphs where *the links of an atom (node) are totally ordered*.

Hypergraphs are the principal syntactic category: 0 is an empty hypergraph; $p(X_1, \ldots, X_m)$ is an atom with m (totally ordered) endpoints of (regular or hyper) links; and P, P is parallel composition. Section 2.2 explains why there are two kinds of links in HyperLMNtal.

Computation in HyperLMNtal starts with an initial graph, which we assume without loss of generality is a single nullary atom in this paper. The initial graph is rewritten by rewrite rules repeatedly until none of them become applicable.

A rewrite rule has the form H `:-` $G \mid B$, and will be applied to a hypergraph P if the *head* H matches (i.e., is isomorphic to) a subgraph of P and that subgraph satisfies auxiliary conditions specified in the *guard* G. Then the matched subgraph (together with a subgraph specified by G, if any) is removed and a fresh copy of the hypergraph B (called a *body*) is spliced into the rest of P. The auxiliary conditions include type constraints and (in)equality constraints, and their examples will be given throughout the paper. A rewrite rule may have the form H `:-` B when the guard G is empty. 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)` is the same as `f(A,B),a(A),b(B)`. Also, `C=f(A,B)` is the same as `f(A,B,C)` because `f(A,B,C)` is defined to be equal to `C=D,f(A,B,D)` (by the predefined semantics of the infix atom "=" that interconnects two links) to which the above rule can be applied to obtain `C=f(A,B)`.

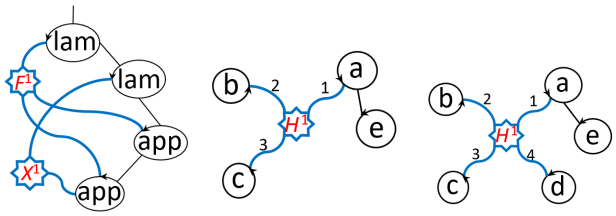
2.2 Links in HyperLMNtal

Both regular links and hyperlinks that occur in the body but not in the head of a rewrite rule are created fresh upon hypergraph rewriting, that is, a created link is distinct from all existing ones. Rewrite rules appearing throughout the paper will explain them. In a rewrite rule, placing `new(L,`*attribute*`)` in the guard imposes a constraint that `L` is a fresh hyperlink with an *attribute* which is a natural number.

A regular link of a hypergraph has at most two endpoints. A regular link with only one endpoint (i.e., occurrence) in a hypergraph is called a *free link*, which is considered to be connected to some atom in the (implicit) context in which the hypergraph is placed. For example, an atom `lam` in Fig. 1 a has a free link as its third argument. In figures, the arrowhead on a circle indicates the first argument of an atom and the ordering of its arguments, and an 8-point star with curved links represents a single hyperlink.

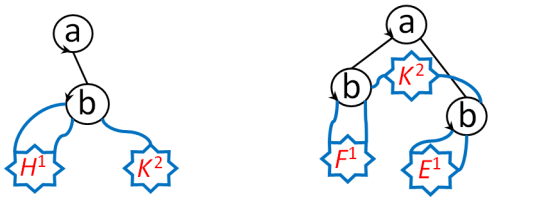
There is a syntactic condition that a regular link name occurring in a rule must occur exactly twice in the rule.

[†]http://www.ueda.info.waseda.ac.jp/lmntal/



(a) (b) (c)

Fig. 1 Hyperlinks



(a) A hypergraph (b) Copying with *hlground*

Fig. 2 Copying a subgraph

Otherwise, the link is a hyperlink and must be typechecked in the guard. Consider the following program, where the names preceding "@@" are rule names:

```
init.
ge@@ init :- new(H,1) | a(H,e), b(H), c(H), d(H).
rm@@ d(H) :- hlink(H) | .
```

The initial graph is the atom `init` in the first line. The second line is a rewrite rule `ge` that transforms `init` to a hypergraph `a(H,e),b(H),c(H),d(H)` which is shown in Fig. 1 c. Applying the rule `rm` to the hypergraph shown in Fig. 1 c will result in a hypergraph `a(H,e),b(H),c(H)` shown in Fig. 1 b because the rule `rm` removes one endpoint of the hyperlink `H` and the atom `d`. The `hlink(H)` checks if `H` is an occurrence of a hyperlink. Note that the right-hand side of the rule `rm` stands for an empty hypergraph (0 in the abstract syntax).

A hyperlink can be seen as a data structure consisting of a *core* (represented by an 8-point star in Fig. 1) and one or more regular links called *sublinks* each connecting the core and an atom (which is `a`, `b`, `c`, or `d` in Fig. 1). Numbers are assigned to sublinks in some figures for the purpose of exposition. *Fusion* of two hyperlinks X and Y is written as $X >< Y$. If X and Y belong to different hyperlinks, $X >< Y$ fuses them into a single hyperlink by merging their hyperlink cores.

The above example illustrates two important points about hyperlinks; hyperlinks are a generalization of regular links, and a hyperlink can change the number of its endpoints.

There are good reasons for keeping two kinds of links in HyperLMNtal rather than using hyperlinks only. First, one can write numerous useful programs using regular links alone. Second, a non-free regular link enjoys an invariant that it always has two endpoints during computation in which its endpoints may be redirected or two regular links are interconnected. This invariant ensures key properties about the shapes of graphs that are useful for both implementation and programming. Third, a graph with free (regular) links cannot simply be copied or deleted because they are supposed to be connected to some atoms in the context, while a graph with hyperlinks whose endpoints may exist also in its context can be copied or deleted. This distinction is exploited in the design of *hlground* described in Sect. 3.4.

2.3 Graph Types

HyperLMNtal *graph types* describe classes of graphs with specific shapes. For example, `hlink(L)` ensures that `L` is an occurrence of a hyperlink (i.e., is connected to a hyperlink core), and `unary(K)` specifies that `K` is connected to a unary atom.

Another important graph type provided by HyperLMNtal is *hlground*. A graph type `hlground(L,`$a_1, \ldots, a_n$`)` in a rewrite rule identifies a subgraph of a hypergraph rooted by the link `L`, and $a_1, \ldots, a_n$ ($n \geq 0$) are attributes of hyperlinks that are allowed to occur in the subgraph. The following HyperLMNtal program explains how *hlground* works, and its graphical illustration is given in Fig. 2:

```
init.
ge@@ init :- new(H,1), new(K,2) | a(b(H,H,K)).
cp@@ a(A) :- hlground(A,1) | a(A,A).
```

The rule `ge` rewrites the initial atom `init` into `a(b(H,H,K))` where the hyperlink `H` has attribute 1 and the hyperlink `K` has attribute 2, as shown in Fig. 2 a. The `hlground(A,1)` in the guard of the rule `cp` identifies a hypergraph consisting of an atom `b` and a hyperlink `H` with attribute 1. Applying the rule `cp` to `a(b(H,H,K))` creates fresh copies of the atom `b` and the hyperlink `H`, while sharing the hyperlink `K` with an unmatched attribute between the two copies, as shown in Fig. 2 b.

In the example above, `hlground(L,`$a_1, \ldots, a_n$`)` identifies a subgraph which can be disconnected from the rest of the hypergraph by cutting `L` and hyperlinks with unmatched attributes. This is how *hlground* was originally designed and implemented [14]. The idea of graph type constraints for identifying subgraphs in HyperLMNtal goes back to *ground(L)* of LMNtal, which identifies a subgraph which can be disconnected from the rest of a hypergraph by cutting L. Later, *hlground* was introduced to HyperLMNtal as an extension of *ground* to allow one to specify which hyperlinks should be followed and which should be cut, and was used to encode *bigraphs* [14]. Subsequently, it was used for generating fresh copies of program clauses in higher-order logic programming [25]. Now, from the above example and the historical development of *hlground*, the question then arises: for cases where a subgraph cannot simply be disconnected by cutting the root and hyperlinks with unmatched

attributes, can we still design and implement a reasonable semantics of *hlground*? This will be discussed in Sect. 3.4.

3. Encoding The λ-Calculus

We show how the untyped λ-calculus can be implemented in HyperLMNtal.

3.1 Barendregt's Variable Convention

Barendregt's variable convention is often assumed for formal systems involving name binding [5]. It states that all bound variables are taken distinctly from all free variables, thus bringing convenience to theory by assuming implicit α-conversion. Henceforth, we apply the spirit of variable convention also to bound variables and *keep all bound variables distinct from each other*. We call it *variable convention*.

Figure 3 illustrates the untyped λ-calculus under the variable convention, which has no side conditions or extra operations to guarantee the freshness of variables. In implementations, on the other hand, one usually formalizes freshness conditions or extra operations to ensure correct substitutions. However, if we can keep every two bound variables distinct and keep all free variables different from all bound variables, as we assume, our formalization does not need freshness constraints.

Figure 4 explains how we can ensure the distinctness of bound variables during the substitution in the λ-calculus.

Initially, all the bound variables are given distinct. At step 2, step 6 and step 10, applying substitution to an application causes copying of the underlined term into two places. When a substitution $[x := N]$ is applied to an application $(M_1 M_2)$, there are two cases for the term N. The binders of n and m in step 2 and of n_2 and m_2 in step 6 are within the underlined terms being substituted and these bound variables are renamed. On the other hand, at step 10, the binder of n_1 is located outside of the term being substituted, and therefore there is no need to rename it. It is obvious that such copying of terms in the substitution steps keeps the bound variables distinct and avoids variable capture.

Our renaming technique differs from the classic textbook renaming. In a substitution $(M_1 M_2)[x := N]$, we create copies of N and rename bound variables of the copies of N while copying N, rather than renaming a term to which a substitution is applied. This technique is reasonable in the following sense. First, it shares the advantages of Barendregt's variable convention; variable capture never happens and side conditions are not necessary. Second, the copying is done in the right-hand side of rules as an operation on subgraphs identified by *hlground*. As we will see in later sections, our implementations of formal systems are greatly simplified by this design choice.

In the subsequent sections, we discuss how these ideas are formalized and implemented in HyperLMNtal.

Syntax	$M, N ::= x \mid \lambda x.M \mid M\,N$	
β-reduction	$(\lambda x.M)N \to M[x := N]$	
Substitution	$x[x := N]$	$\equiv N$
	$y[x := N]$	$\equiv y$, if $x \neq y$
	$(\lambda y.M)[x := N]$	$\equiv \lambda y.(M[x := N])$, if $x \neq y$
	$(M_1 M_2)[x := N]$	$\equiv (M_1[x := N])(M_2[x := N])$

Fig. 3 The untyped λ-calculus

$(\lambda x.xx)\lambda m\lambda n.mn$
$(xx)[x := \underline{\lambda m\lambda n.mn}]$ **step 2**
$(x[x := \underline{\lambda m_1\lambda n_1.m_1 n_1}])(x[x := \underline{\lambda m_2\lambda n_2.m_2 n_2}])$
$(\underline{\lambda m_1\lambda n_1.m_1 n_1})(\underline{\lambda m_2\lambda n_2.m_2 n_2})$
$(\lambda n_1.m_1 n_1)[m_1 := \lambda m_2\lambda n_2.m_2 n_2]$
$\lambda n_1.((m_1 n_1)[m_1 := \underline{\lambda m_2\lambda n_2.m_2 n_2}])$ **step 6**
$\lambda n_1.((m_1[m_1 := \underline{\lambda m_3\lambda n_3.m_3 n_3}])(n_1[m_1 := \underline{\lambda m_4\lambda n_4.m_4 n_4}]))$
$\lambda n_1.((\lambda m_3\lambda n_3.m_3 n_3)n_1)$
$\lambda n_1.((\lambda n_3.m_3 n_3)[m_3 := n_1])$
$\lambda n_1.(\lambda n_3.((m_3 n_3)[m_3 := \underline{n_1}]))$ **step 10**
$\lambda n_1.(\lambda n_3.((m_3[m_3 := \underline{n_1}])(n_3[m_3 := \underline{n_1}])))$
$\lambda n_1.(\lambda n_3.(n_1 n_3))$

Fig. 4 Substitution without variable capture

3.2 Hypergraph Terms for λ-Terms

Let $x, y, \ldots$ be variables of the λ-calculus; $X, Y, \ldots$ be hyperlinks; and $L, R, \ldots$ be regular links. $M, N, \ldots$ are used to express both λ-terms and regular links. Their usage should be clear from the context.

In Sect. 3.1, we mentioned that we apply the variable convention to λ-terms. *Hypergraph λ-terms* are straightforward hypergraph representations of λ-terms respecting the variable convention. For example, a λ-term $\lambda x.\lambda x.x$ should be written as $\lambda y.\lambda x.x$, and its corresponding hypergraph λ-term is `R=lam(Y,lam(X,X))`. Another term $\lambda x.\lambda y.(x(yz))$, which respects the variable convention, is encoded as

```
lam(X,L1,R), lam(Y,L2,L1),
app(X,L3,L2), app(Y,Z,L3)
```

which is abbreviated to

```
R=lam(X,lam(Y,app(X,app(Y,Z)))).
```

The hypergraph λ-terms well reflect the structure of original λ-terms. `L1`, `L2` and `L3` are regular links for constructing the abstract syntax trees of hypergraph λ-terms, and one only needs to replace variables of a λ-term by appropriate hyperlinks (`X`, `Y` and `Z`) to have a corresponding hypergraph λ-term. A substitution $M[x := N]$ is encoded as R=`subs`(M, X, N), which means that a hypergraph λ-term N replaces all occurrences of X in a hypergraph λ-term M, and the result will be passed to R. The hypergraph λ-term of the

Church numeral 2, $\lambda f.\lambda x.f(fx)$, is illustrated in Fig. 1 a of Sect. 2.2.

We now define a function φ which translates a λ-term t, which must respect the variable convention, to its hypergraph representation. The function $\varphi(t, R)$ takes two arguments, a λ-term t under variable convention and a regular link R which will be the root of the hypergraph λ-terms representing t, and return a hypergraph λ-term. The auxiliary function φ' takes one more argument, a set Γ of pairs (x, X) of a variable x and a hyperlink X representing x, and returns a hypergraph λ-term and a possibly extended set of pairs. Note that a substitution $M[x := N]$ is explicitly handled in our framework rather than regarding it as a meta-level notation.

$$
\begin{aligned}
\varphi(t, R) &\stackrel{\text{def}}{=} \varphi'(t, R, \emptyset)\\
\varphi'(\lambda x.M, R, \Gamma) &\stackrel{\text{def}}{=} ((\mathtt{lam}(X^1, L, R), G), \Gamma')\\
&\quad \text{where } (G, \Gamma') = \varphi'(M, L, \Gamma \cup \{(x, X^1)\})\\
\varphi'(MN, R, \Gamma) &\stackrel{\text{def}}{=} ((\mathtt{app}(L_1, L_2, R), G_1, G_2), \Gamma'')\\
&\quad \text{where } (G_1, \Gamma') = \varphi'(M, L_1, \Gamma)\\
&\quad \text{and } (G_2, \Gamma'') = \varphi'(N, L_2, \Gamma')\\
\varphi'(x, R, \Gamma) &\stackrel{\text{def}}{=} (R{=}X^i, \Gamma) \quad \text{if}(x, X^i) \in \Gamma\\
\varphi'(x, R, \Gamma) &\stackrel{\text{def}}{=} (R{=}X^2, \Gamma \cup \{(x, X^2)\})\\
&\quad \text{if } \forall Y^i((x, Y^i) \notin \Gamma)\\
\varphi'(M[x := N], R, \Gamma) &\stackrel{\text{def}}{=} ((\mathtt{subs}(L_1,X^1,L_2,R),G_1,G_2),\Gamma'')\\
&\quad \text{where } (G_1, \Gamma') = \varphi'(M, L_1, \Gamma \cup \{(x, X^1)\})\\
&\quad \text{and } (G_2, \Gamma'') = \varphi'(N, L_2, \Gamma')
\end{aligned}
$$

X^i stand for a hyperlink with attribute i, which is created by `new(X,i)`. Hyperlinks with attribute 1 represent bound variables and hyperlinks with attribute 2 represent free variables. The `lam` and `app` atoms represent abstractions and applications, respectively. $R = X^i$ connects the entry point R with X^i, in which case R becomes (a sublink of) a hyperlink.

In hypergraph λ-terms, the tree structure of a λ-term is formed by regular links representing subterm-superterm relationship, whereas variables of the λ-calculus are represented by hyperlinks, giving the additional graph structure to λ-terms. This way, regular links and hyperlinks play distinct and well-motivated roles. Ideas somewhat similar to hypergraph λ-terms can be found in classical *Stoy diagrams* [20], and its conception dates back to Bourbaki [4]. However, we are going to give a concrete encoding of λ-terms in an implemented language, not just their drawings.

3.3 Encoding of the Untyped λ-Calculus

The HyperLMNtal encoding of the untyped λ-calculus is given in Fig. 5 [24]. The rule `beta` encodes the β-reduction, and the next four rules encode the substitution.

Let's see how these rules in Fig. 5 work. The hyperlinks representing bound and free variables in the term represented by N are classified by the second argument of

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

Fig. 5 Encoding of the untyped λ-calculus with capture-free substitution

`hlground(N,1)` in the rules `var2` and `app`. For hyperlinks representing free variables, `hlground(N,1)` causes the sharing of such hyperlinks when the graph rooted by `N` is copied in `app` and removes some endpoints of such hyperlinks when the graph rooted by `N` is removed in `var2`. In `var2`, the constraint `X\=Y` means that `X` and `Y` are different hyperlinks.

To understand how the hyperlinks representing bound variables are correctly copied and shared, recall the reduction example given earlier (Fig. 4). The bound variables m and n at step 2 and the bound variables m_2 and n_2 at step 6 receive new names, but the bound variable n_1 does not receive a new name (step 10) because the binder of n_1 is located outside of the term which is replacing m_3. Of the bound variables of a term $M[x := N]$, those occurring in N could be further classified into two:

- *Truly local variable:* if the binder of a bound variable y is within N, then y is called a truly local variable for N;
- *Partially local variable:* if y occurs in N but its binder is not within N, then y is called a partially local variable for N.

As will be discussed in detail in Sect. 3.4, the graph type *hlground* distinguishes between hyperlinks representing truly local variables those representing partially local variables.

HyperLMNtal allows one to generate small λ-terms and reuse them for constructing larger λ-terms, as shown below:

```
init.
no2@@  N=n(2) :- new(F,1), new(X,1) |
       N=lam(F,lam(X,app(F,app(F,X)))).
init@@ init :- app(n(2),n(2),r).
```

The rule `no2` generates Church numeral 2, shown in Fig. 1 a. The rule `init` generates an application representing a λ-term $(\lambda x.\lambda y.x(xy))(\lambda m.\lambda n.m(mn))$, which will be reduced (by the rules defined in Fig. 5) to a hypergraph λ-term corresponding to the λ-term $\lambda x.\lambda y.x(x(x(xy)))$.

Since we cannot avoid copying of terms in the abstract definition of substitutions, it seems better to treat it as an opportunity to rename bound variables. The α-conversion is realized as the copying and sharing operations of rewrite rules on subgraphs identified by *hlground*. By such copying and sharing, two equal but syntactically separate hypergraph λ-terms are generated, and the idea mentioned in Sect. 3.1

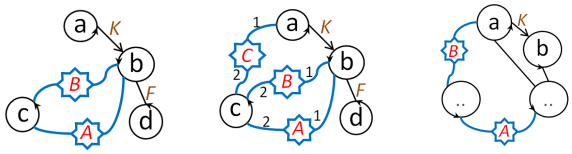


Fig. 6 Hypergraphs

became a reality.

3.4 Graph Type: *hlground*

We say that a HyperLMNtal graph S_1 is a subgraph of S iff there exists a graph S_2 such that $S = S_1, S_2$.

The graph type *hlground* identifies a subgraph of a given hypergraph. In `hlground(`$K, a_1, \ldots, a_n$`)`, the link K is called a *root link*, and $a_1, \ldots, a_n$ are the attributes of hyperlinks allowed to appear in the *hlground*. Links of HyperLMNtal are undirected, but a root link K is always indicated by an arrow in subsequent figures. The direction of such an arrow is from a *source atom* explicitly occurring in the left-hand side of a rule to some *target atom* not explicitly mentioned in the rule. Three hypergraphs are shown in Fig. 6, in which the `a`'s are source atoms and the `b`'s are target atoms of `K`. We assume that all the hyperlinks in Fig. 6 have attributes found in $a_1, \ldots, a_n$.

In Fig. 6 a, cutting the root link `K` splits the hypergraph `a(K),b(B,A,F,K),c(B,A),d(F)` into two subgraphs, `a(K)` and `b(B,A,F,K),c(A,B),d(F)`, where `K` appears as a free link in both graphs. The subgraph identified by `hlground(K,`$a_1, \ldots, a_n$`)` is the latter one with the target atom `b`.

The original *hlground* [14] only handles cases where the cutting of a root link splits a graph into two subgraphs. In Fig. 6 b, in contrast, cutting the root link `K` alone cannot split the hypergraph `a(K,C),b(B,A,F,K),c(B,C,A),d(F)` into two because `K` is on cycles. We introduce some concepts and extend *hlground* to handle such cases.

In the rest of this subsection, we treat the cores of hyperlinks as atoms for the purpose of exposition. When a hyperlink sublink is a root link of *hlground*, we consider the core of the hyperlink as the target atom of the *hlground*. By the definition of source atoms, a hyperlink core never appears as a source atom.

Definition 1 (Maximal attributed path, (Non-)returning path, Pure path)**.** *For a root link K of* `hlground(`$K, a_1, \ldots, a_n$`)` *pointing to a target atom, a* maximal attributed path (or MAP) *from K is a* maximal *sequence of different regular links and different sublinks of hyperlinks, whose attributes are in $a_1, \ldots, a_n$, without cycles. Maximal means that each sequence is taken so that it cannot be extended any further. A MAP which leads to the source atom is called a* returning path*. A MAP which does not lead to the source atom is called a* non-returning path*. A returning path without hyperlinks is called a* pure path. □

In Fig. 6 b, the root link K has returning paths $B_1B_2C_2C_1$ and $A_1A_2C_2C_1$, non-returning paths $B_1B_2A_2$ and $A_1A_2B_2$ and F, and no pure paths.

Definition 2 (Critical sublinks)**.** *Let rp(K) be the set of all returning paths of a root link K. Assume that rp(K) is non-empty and that each returning path contains at least one hyperlink. Then a set of* critical sublinks *is a smallest set of sublinks whose removal cuts all returning paths in rp(K). If there is more than one smallest set of critical sublinks, the ones consisting of sublinks closer to the source atom are chosen.* □

In Fig. 6 b, the set of critical sublinks of the root link K is $\{C_1\}$.

Note that the smallest set of critical sublinks "closest to the source atom" is uniquely determined for the following reason. Instead of considering a graph cut as a set of links, we consider a cut as a subgraph containing the source atom and obtained by cutting those links. Then, from the property of minimum cut in graph theory, the intersection of all subgraphs each corresponding to a minimum cut is a subgraph corresponding to a minimum cut. Clearly, such a subgraph is uniquely determined.

Definition 3 (Global hyperlinks)**.** Global hyperlinks *are the hyperlinks containing critical sublinks.* □

In Fig. 6 b, the set of global hyperlinks for the root link K is $\{C\}$.

Definition 4 (Local path)**.** *A local path is the prefix of a returning path starting from the target atom and ending at the global hyperlink, excluding that global hyperlink.* □

In Fig. 6 b, the local paths of the root link K are B_1B_2 and A_1A_2.

Definition 5 (*hlground*)**.** *For a root link K in a hypergraph S that does not contain pure paths for K,* `hlground(`$K, a_1, \ldots, a_n$`)` *is a subgraph $G_K^{a_1,\ldots,a_n}$ consisting of non-returning paths and local paths of K and atoms on them.* □

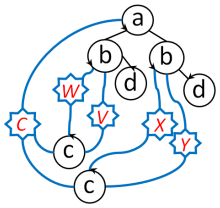
We give an example of *hlground* after Definition 6.

Copying and the removal of a link K of type *hlground* in a rewrite rule causes the copying and the removal of $G_K^{a_1,\ldots,a_n}$, respectively, in the following manner:

Definition 6 (Operations on *hlground* graphs)**.** *When K is of type* `hlground(`$K, a_1, \ldots, a_n$`)` *and is copied (or removed) in the right-hand side of a rewrite rule,*

- *links in $G_K^{a_1,\ldots,a_n}$ will be freshly copied (or removed),*
- *global hyperlinks and hyperlinks whose attribute is not in $a_1, \ldots, a_n$ will be shared (or their sublinks in $G_K^{a_1,\ldots,a_n}$ will be removed), and*
- *atoms within $G_K^{a_1,\ldots,a_n}$ will be copied (or removed),*

respectively. □



(a) Copying a subgraph

(b) Removing a subgraph

Fig. 7 Rewriting a subgraph

Consider the following HyperLMNtal program,

```
cp@@ a(R,T) :- hlground(R,1) | a(R,R,T).
rm@@ a(R,T) :- hlground(R,1) | a(T).
```

and let us apply these two rewrite rules to the hypergraph shown in Fig. 6 b. We assume that all hyperlinks in Fig. 6 b have attribute 1. The rule `cp` copies a subgraph identified by *hlground* as shown in Fig. 7 a, where hyperlinks `A` and `B` are copied into `X,Y` and `W,V`, respectively, and `C` is shared between the two copies and the rest of the hypergraph. The rule `rm` deletes a subgraph identified by *hlground* as shown in Fig. 7 b, where an endpoint of a hyperlink `C` is removed together.

From the definition of *hlground*, we know that a subgraph `hlground`$(K, a_1, \ldots, a_n)$ exists if and only if there are no pure paths for K. For example, there is no `hlground`$(K, a_1, \ldots, a_n)$ for the hypergraph shown in Fig. 6 c.

3.5 Correctness of The Encoding

As shown in Fig. 5, our encoding of the untyped λ-calculus has only five rules (one β-rule and four substitution rules). In this section, we show that the encoding handles hypergraph λ-terms correctly.

Lemma 1. *In a hypergraph λ-term* `R=lam`(X, S)*, the hyperlink X has possible occurrences in the hypergraph λ-term rooted at S only.*

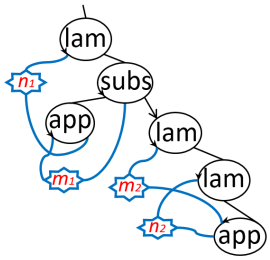
Proof. Follows from the definition of hypergraph λ-terms in Sect. 3.2. □

We first prove that, for any root link in any hypergraph λ-term, the *hlground* in our encoding of the untyped λ-calculus in Fig. 5 never fails.

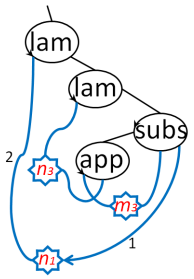
Theorem 1 (Non-existence of pure paths)**.** *There are no pure paths for an arbitrarily chosen root link in any hypergraph λ-term.*

Proof. Suppose that each subterm of a given λ-term is given a level number in the standard way, i.e., the whole λ-term is of level 0, and subterms M and N in a term $\lambda x.M$, MN and $M[x := N]$ of level n are of level $n+1$. Similarly, the toplevel atom of a hypergraph lambda-term of level n is given the level number n.

Suppose the source atom of a root link L is of level n and the target atom of L is of level $n + 1$. If there is a pure path for L, we must keep following different regular links until we reach the source atom of level n. However, from the construction (defined by the function φ) of a hypergraph λ-term, following the next regular link always takes us to an atom at a deeper level. Hence, assuming the target atom is of a deeper level than the source atom, there must be no pure paths, i.e., returning paths consisting only of regular links. If we exchange the source and the target atoms of L, the claim still holds because the notion of a pure path does not depend on which endpoint of L is a source atom. □



(a) $\lambda n_1.((m_1 n_1)[m_1 := \underline{\lambda m_2 \lambda n_2.m_2 n_2}])$



(b) $\lambda n_1.(\lambda n_3.((m_3 n_3)[m_3 := \underline{n_1}]))$

Fig. 8 Hypergraph λ-terms during substitution

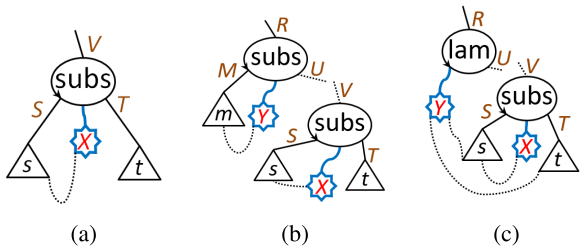
When *hlground* identifies a set of global hyperlinks $\{H_1, \ldots, H_n\}$ of a root K in a hypergraph λ-term, $H_1, \ldots, H_n$ are actually the hyperlinks representing the partially local variables of the corresponding λ-term, as illustrated in the following examples.

The two hypergraphs given in Fig. 8 are hypergraph representations of two λ-terms at the second renaming and the third renaming steps of the reduction in Fig. 4. The links with arrowheads in the figure indicate the root links.

The rule `app` of Fig. 5 will be applied to these two terms. In Fig. 8 a, there are no partially local variables in $\lambda m_2 \lambda n_2.m_2 n_2$ to be substituted for m_1. For the second λ-abstraction in Fig. 8 b, the variable n_1 is a partially local variable. As a hypergraph λ-term, n_1 is a global hyperlink because of the returning paths $\{n_{1_2}, n_{1_2} n_3, n_{1_2} n_3 m_3\}$[†] (here and henceforth, we may use hyperlinks rather than sublinks to denote some of the path elements when the order of sublinks of a hyperlink is unambiguous).

The rewrite rules in Fig. 5 correspond to β-reduction

[†]For uniformity with the original λ-terms, hyperlinks are denoted with lowercase letters in this example.



(a) (b) (c)

Fig. 9 Hypergraph λ-term substitution

and substitution in Fig. 3. The rules `beta`, `var1` and `abs` are straightforward. On the other hand, the rules `var2` and `app` have *hlground* type constraints. The copying of terms during substitution appears only in the rule `app`; it creates two α-equivalent but non-identical copies of subgraphs rooted at N in $(M_1M_2)[x := N]$. Therefore, the renaming technique discussed in Sect. 3.1 avoids variable capture in the rule `abs`. Again, *hlground* identifies a subgraph in the rule `var2` which is then removed. In the next lemma, we show that *hlground* correctly identifies *truly local variables* and *partially local variables* of hypergraph λ-terms during the substitution.

Lemma 2. *For the substitution* $\texttt{subs}(S,X,T,V)$ *obtained by applying the rule* `beta` *in Fig. 5 to* $\texttt{app(lam(X,S),T,V)}$ *where links S and T are roots of some hypergraph λ-terms, the truly local variables and the partially local variables appearing in the substituting term rooted at T will be identified as local hyperlinks (hyperlinks whose endpoints are all in* $\texttt{hlground}(T)$*) and global hyperlinks of* $\texttt{hlground}(T)$*, respectively.*

Proof. First we recall that truly local variables are bound inside the term rooted at T and partially local variables are bound outside of the term rooted at T. We use $s[S]$ and $t[T]$ to represent some hypergraph λ-terms rooted at S and T, respectively. Note that S and T could be sublinks of hyperlinks (which is the case when $s[S]$ and $t[T]$ represent variables) although they are shown as regular links in Fig. 9.

First case: The root link T has no returning paths. Consider V=`subs(`S`,`X`,`T`)`$,s[S],t[T]$ in Fig. 9 a. By Lemma 1, hyperlinks bound inside $t[T]$, which represent truly local variables of $t[T]$, will be identified as local hyperlinks of $\texttt{hlground}(T)$.

Second case: The root link T has returning paths. Consider R=`lam(`Y`,`U`)`$,$`subs(`S`,`X`,`T`,`V`)`$,s[S],t[T]$ in Fig. 9 c, where the `lam` represents a direct or an indirect superterm of (a term represented by) the `subs`. Y represents a partially local variable of $t[T]$ and may appear also in $s[S]$. If Y does not occur in $s[S]$, Y is a global hyperlink of $\texttt{hlground}(T)$. If Y occurs in $s[S]$, there are returning paths Y and YX, therefore Y is again a global hyperlink of $\texttt{hlground}(T)$. When $t[T]$ has more returning paths representing partially local variables $Y_2,\dots Y_n$, each of them is a global hyperlink for the same reason. As shown in the first case, hyperlinks bound inside $t[T]$ will be identified as local hyperlinks of $\texttt{hlground}(T)$. □

Note that a bound variable may be introduced by `subs` occurring as a superterm of $\texttt{subs}(S,X,T,V)$, for example R=`subs(`M`,`Y`,`U`)`$,$`subs(`S`,`X`,`T`,`V`)`$, m[M], s[S], t[T]$ in Fig. 9 b. However, by Lemma 1 and the `beta` rule in Fig. 5, Y may only occur in $m[M]$ and therefore not in $t[T]$; therefore it will not form a returning path of the root link T as covered by the first case of the proof.

This lemma and Definition 6 ensure that the last four rules in Fig. 5 correctly perform substitution and keep bound variables distinct. The next two theorems show the completeness of the encoding in Fig. 5.

Theorem 2. *If $M \to M'$ in Fig. 3, then $\varphi(M,R) \to^* \varphi(M',R)$ by the rewrite rules in Fig. 5.*

Proof. The proof is by structural induction. We note that the definition of φ uses φ', where $\varphi'(N,R,\Gamma)$ produces a hypergraph λ-term of N rooted at R. The Γ is a list to keep track of already translated variables, and such Γ's are not essential here.

In the following, we consider the cases where $M \to M'$ reduces the β-redex at the root of M, i.e., M is of the form $(\lambda x.P)N$. The other cases where $M \to M'$ reduces a redex inside M easily follow from the root cases.

First case: let $M = (\lambda x.x)N$, then $M' = N$. The rewriting of $\varphi(M,R)$ by `beta` and `var1` is as follows.

$$\begin{aligned}&\texttt{app(lam}(X^1,X^1),L,R),\ \varphi'(N,L,\Gamma)\\ &\to \texttt{subs}(X^1,X^1,L,R),\ \varphi'(N,L,\Gamma)\\ &\to \varphi'(N,R,\Gamma)\end{aligned}$$

The resulted hypergraph λ-term corresponds to N.

Second case: let $M = (\lambda x.y)N$, then $M' = y$. The rewriting of $\varphi(M,R)$ by `beta` and `var2` is as follows.

$$\begin{aligned}&\texttt{app(lam}(X^1,L_1),L_2,R),\ \varphi'(y,L_1,\Gamma'),\ \varphi'(N,L_2,\Gamma'')\\ &\to \texttt{subs}(L_1,X^1,L_2,R),\ \varphi'(y,L_1,\Gamma'),\ \varphi'(N,L_2,\Gamma'')\\ &\to \varphi'(y,R,\Gamma')\end{aligned}$$

In `var2`, $\texttt{hlground}(L,1)$ identifies the scope of $\varphi'(N,L_2,\Gamma'')$, which is then deleted upon reduction. The resulted hypergraph λ-term corresponds to y.

Third case: let $M = (\lambda x.\lambda y.M_1)N$, then $M' = \lambda y.(M_1[x := N])$. The rewriting of $\varphi(M,R)$ by `beta` and `abs` is as follows.

$$\begin{aligned}&\texttt{app(lam}(X^1,\texttt{lam}(Y^1,L_1)),L_2,R),\ \varphi'(M_1,L_1,\Gamma'),\\ &\varphi'(N,L_2,\Gamma'')\\ &\to \texttt{subs(lam}(Y^1,L_1),X^1,L_2,R),\ \varphi'(M_1,L_1,\Gamma'),\\ &\quad\varphi'(N,L_2,\Gamma'')\\ &\to \texttt{lam}(Y^1,\texttt{subs}(L_1,X^1,L_2),R),\ \varphi'(M_1,L_1,\Gamma'),\\ &\quad\varphi'(N,L_2,\Gamma'')\end{aligned}$$

The resulted hypergraph λ-term corresponds to $\lambda y.(M_1[x := N])$.

Fourth case: let $M = (\lambda x.M_1M_2)N$, then $M' = (M_1[x := N])(M_2[x := N])$. The rewriting of $\varphi(M,R)$ by

beta and app is as follows.

$$\begin{aligned}
&\texttt{app(lam(}X^1\texttt{,app(}L_1, L_2\texttt{))}, L, R), \varphi'(M_1, L_1, \Gamma'),\\
&\varphi'(M_2, L_2, \Gamma''),\ \varphi'(N, L, \Gamma)\\
&\to \texttt{subs(app(}L_1, L_2\texttt{)}, X^1, L, R),\ \varphi'(M_1, L_1, \Gamma'),\\
&\quad \varphi'(M_2, L_2, \Gamma''),\ \varphi'(N, L, \Gamma)\\
&\to \texttt{app(subs(}L_1, X^1, L_3\texttt{)}\texttt{,subs(}L_2, X^1, L_4\texttt{)}, R),\\
&\quad \varphi'(M_1, L_1, \Gamma'),\ \varphi(M_2, L_2, \Gamma''),\\
&\quad \varphi'(N, L_3, \Gamma_1),\ \varphi'(N, L_4, \Gamma_2).
\end{aligned}$$

Note that $\varphi'(N, L, \Gamma)$, $\varphi'(N, L_3, \Gamma_1)$ and $\varphi'(N, L_4, \Gamma_2)$ are α-equivalent but non-identical hypergraph λ-terms by the definition of φ. In app, the subgraph $\varphi'(N, L, \Gamma)$ identified by hlground(L,1) is copied into $\varphi'(N, L_3, \Gamma_1)$ and $\varphi'(N, L_4, \Gamma_2)$. The resulted hypergraph λ-term corresponds to $(M_1[x := N])(M_2[x := N])$. □

Theorem 3. *If $M \to^* M'$ in Fig.3, then $\varphi(M, R) \to^* \varphi(M', R)$ by the rewrite rules in Fig. 5.*

Proof. By repeated application of Theorem 2. □

Next, in order to show the soundness of the encoding in Fig. 5, we first introduce a lemma about the application of substitutions.

Lemma 3. *Applying the rules except the rule* beta *in Fig. 5 to a hypergraph λ-term always terminates with a unique hypergraph λ-term.*

Proof. The substitution rules in Fig. 5 will be applied to G which is of the form subs(M, X, N, R). Assume M has no subs atoms. When M is a hyperlink, either the rule var1 or var2 will be applied to G to have a unique G', where the rewriting terminates. When M is either lam or app, then the rule abs or app will be applied to G, gradually moving the substitution into the subterms of M. Applying the substitution rules to G in any order will result in a hypergraph λ-term in which its every subterm has a copy of the substitution. We know that regular links represent subterm-superterm relations in hypergraph λ-terms, and there are no pure paths in M by Theorem 1. This means the rules abs and app will be applied only finitely many times, and the copies of the substitutions in the subterms will finally have hyperlinks at their first arguments, which means these substitutions will terminate with unique results.

Now assume M has one or more subs atoms. There is no rule in Fig. 5 for rewriting two subs's simultaneously. Different subs's may move down concurrently, but the one above the other cannot overtake the one below it. Therefore the result is the same as rewriting subs's sequentially from the lowest-level one. □

Now we are ready to prove the soundness of the encoding. Note that soundness is less obvious than completeness because our encoding handles substitutions explicitly but does not specify particular order in which they are processed.

Theorem 4. *Let G and G' be hypergraph λ-terms such that $G \to^* G'$, where $\to^*$ starts with the application of the rule* beta *in Fig. 5, followed by zero or more applications of other rules. Then there exist λ-terms H and H' such that*
(i) $G \to^* \varphi(H, R)$ without using beta *in Fig. 5,*
(ii) $G' \to^* \varphi(H', R)$ without using beta *in Fig. 5, and*
(iii) $H \to^* H'$ in Fig. 3.

Proof. When G does not contain subs atoms, apply the rule beta to G once to have $G \to G''$, then apply other rules to have $G'' \to^* \varphi(H', R)$, where G' is somewhere between G'' and $\varphi(H', R)$ inclusive. The latter rewriting terminates with correct substitution by Lemma 2 and Lemma 3. Since G has no subs atoms, $G \to^* \varphi(H, R)$ holds with zero step, and we have $H \to H'$ (in one step) by reducing the β-redex corresponding to the one in G.

The interesting case is where G contains possibly nested, not yet expanded substitutions (i.e., subs atoms) in which the β-redex occurs. Let G be a hypergraph λ-term subs(M_1, X_1, N_1, R), subs(M_2, X_2, N_2, N_1), ..., subs(M_n, X_n, N_n, N_{n-1}), where each M_i has m_i (≥ 0) occurrences of X_i. Assume N_i contains the β-redex under consideration. After applying beta to G, we can apply the other rules to have $G \to G'' \to^* \varphi(H', R)$. We can also apply the rules except beta to G to have $G \to^* \varphi(H, R)$. Lemmas 2 and 3 guarantee the correctness and the termination of both reduction sequences. Note that the latter reduction copies the β-redex in G to $\prod_{k=1}^{i} m_k$ places in H, but it is easy to see that performing β-reduction to all those β-redexes in H leads to H'. □

Theorem 5. *Let M be a λ-term without substitutions. If $\varphi(M, R) \to^* G$ in Fig. 5, then there exists M' such that $M \to^* M'$ and $G = \varphi(M', R)$ in Fig. 3.*

Proof. By repeated application of Theorem 4. □

Section 5 shows working examples of the untyped λ-calculus.

3.6 Implementation of *hlground*

Implementation of *hlground* defined in Sect. 3.4 can be done by using well-known graph algorithms including *depth-first search* and *max-flow min-cut*.

There are two cases of *hlground* subgraphs: the first case in which there are no returning paths for the given root link, as shown in Fig. 6 a, and the second case in which there is at least one returning path for the given root link, as shown in Fig. 6 b. Algorithm 1 checks if an *hlground* subgraph exists in the following way. In the first case, the function *cycle_exist_dfs* returns false and then the function *get_local_atoms* computes the local atoms of the *hlground*, which are atoms within the *hlground*. In the second case, the function *cycle_exist_dfs* returns true and the function *purepath_exist_dfs* returns false, and then the function *mincut* finds global hyperlinks. Then, again, the function *get_local_atoms* computes the local atoms of the *hlground*.

Algorithm 1 Find hlground

```
1:  procedure FIND_HLGROUND(G, L)                    ▷ Graph G and root link L
2:      global_hlinks ← ∅
3:      result ← true                                ▷ true if hlground exists
4:      if cycle_exist_dfs(G, L) then
5:          if ¬ purepath_exist_dfs(G, L) then
6:              global_hlinks ← mincut(G, L)
7:          else
8:              result ← false
9:          end if
10:     end if
11:     if result then
12:         local_atoms ← get_local_atoms(G, L, global_hlinks)
13:     end if
14:     return result
15: end procedure
```

In the both cases, Algorithm 1 returns `true`, and the global variables *global_hlinks* and *local_atoms* hold the calculated results. If there are pure paths, Algorithm 1 returns `false`.

Our implementation of *mincut* uses *Dinic's blocking flow algorithm* [10] to find global hyperlinks for the root link. Dinic's blocking flow algorithm has a time complexity of $O(V^2E)$, where V is the number of vertices and E is the number of edges. In our implementation, $V = N_a + N_h$ where N_a is the number of atoms and N_h is the number of hyperlinks, and $E = E_r + \sum_{i=1}^{N_h} sublinks(i)$ where E_r is the number of regular links and $\sum_{i=1}^{N_h} sublinks(i)$ is the total number of all hyperlink sublinks.

4. Encoding System $F_{<:}$

System $F_{<:}$ [17] is an extension of System F, which is *polymorphic λ-calculus* with *subtyping*. System F allows binding of type variables as well as binding of term variables. By having bound type variables, System F provides powerful polymorphism since abstract types are instantiated later by concrete types through type applications. System F has been used as a basis for research on polymorphism. System $F_{<:}$ also supports *subtyping* found in many programming languages.

We encode System $F_{<:}$ and test it with the inputs in Challenge 3 of the PoplMark challenge, which is a set of benchmarks designed to evaluate techniques of name binding for both theorem proving systems and programming languages [3]. Although our encoding running on a HyperLMNtal system is not a certified implementation of System $F_{<:}$, it is an executable prototype specification that can be used by itself or testing other implementations.

4.1 Terms and Types

We follow the idea of representing variables by hyperlinks to have a readable hypergraph representation of System $F_{<:}$ terms and types. Hyperlinks with attribute 1, denoted as X^1, represent term variables, and hyperlinks with attribute 2, denoted as X^2, represent type variables. In addition, we use lists to represent records, patterns and their types. Hyperlinks $L_1, \ldots, L_n$ (which we assume to have attribute 0) represent labels in records, patterns and types. Tr and Ty are regular links connected to hypergraphs representing System $F_{<:}$ terms and types, respectively. R is a regular link appearing as the last argument of atoms. The atoms `type` and `stype` represent *typing* relation and *subtyping* relation, respectively. The hypergraph representation of System $F_{<:}$ types and terms is straightforward. System $F_{<:}$ types are represented as follows:

(*Types*) Ty ::=

X^2	*type variables*
$\mid$ `top`(R)	*maximum type*
$\mid$ `arr`(Ty, Ty, R)	*type of functions*
$\mid$ `all`$(\text{stype}(X^2, Ty), Ty, R)$	*universal type*
$\mid$ `rcd_ty`$([\text{type}(L_i, Ty), \ldots], R)$	*type for records*

Similarly, System $F_{<:}$ terms are represented as follows:

(*Terms*) Tr ::=

X^1	*term variable*
$\mid$ `abs`$(\text{type}(X^1, Ty), Tr, R)$	*term abstraction*
$\mid$ `app`(Tr, Tr, R)	*term application*
$\mid$ `abs`$(\text{stype}(X^2, Ty), Tr, R)$	*type abstraction*
$\mid$ `tapp`(Tr, Ty, R)	*type application*
$\mid$ `rcd`$([\text{p}(L_i, Tr), \ldots], R)$	*record*
$\mid$ `proj`(Tr, L_i, R)	*projection*
$\mid$ `let`(P, Tr, Tr, R)	*pattern binding*

where the atom `p` represents a field in records, and P is a regular link connected to patterns shown below:

(*Patterns*) P ::=

`type`(X^1, Ty, R)	*variable pattern*
$\mid$ `pat`$([\text{p}(L_i, P), \ldots], R)$	*record pattern*

We use HyperLMNtal lists to represent type contexts:

(*Type Context*) Γ ::=

`ctx`$([], R)$
$\mid$ `ctx`$([\text{type}(X^1, Ty), \ldots], R)$
$\mid$ `ctx`$([\text{stype}(X^2, Ty), \ldots], R)$

For example, Church numeral one in System $F_{<:}$, namely

$$\lambda X <: top \,.\, \lambda S <: X \,.\, \lambda Z <: X \,.\, \lambda s : X \to S \,.\, \lambda z : Z \,.\, s\, z$$

(as in [17]), is written as

```
sone =def abs(stype(X,top),
          abs(stype(S,X),
            abs(stype(Z,X),
```

```
        abs(type(St,arr(X,S)),
          abs(type(Zt,Z),app(St,Zt))))))
```

where the hyperlinks are created by `new(X,2)`, `new(S,2)`, `new(Z,2)`, `new(St,1)`, `new(Zt,1)`.

Next, we show how System $F_{<:}$ evaluates its terms and types and how it is implemented in HyperLMNtal.

4.2 Evaluation

System $F_{<:}$ uses the *call-by-value* strategy in its evaluation of terms and types, as shown below. Note that x is a term variable, X, T, T_1, T_2 are type variables and t is a term.

$$(\lambda x : T.t)\, v \longrightarrow t[x := v] \qquad \textit{term evaluation}$$
$$(\lambda X <: T_1.t)\, T_2 \longrightarrow t[X := T_2] \qquad \textit{type evaluation}$$

Here, v is a *value* which is either a term abstraction or a type abstraction whose hypergraph representation was shown in Sect. 4.1:

$$\text{Value} \quad v ::= \lambda x : T.t \qquad \textit{term abstraction}$$
$$\mid \lambda X <: T.t \qquad \textit{type abstraction}$$

The evaluation rules of terms and types are implemented as follows:

```
appabs  @@ R= app(abs(type(A,B),C),value(D)) :-
              hlground(B,1,2) |
              R=eval(subst(A,D,C)).
tapptabs@@ R=eval(tapp(abs(stype(X,T11),T),T12)) :-
              hlground(T11,1,2) |
              R=eval(subst_tp_tm(X,T12,T)).
```

We implemented *call by value* with HyperLMNtal rewrite rules. The idea here is to use *tokens*, represented by the atoms `eval` and `value`, to control the order of evaluation. The original idea of implementing *call by value* using graphs can be found in [19]. Our implementation has the following rules:

```
R=eval(abs(A,B))      :- R=value(abs(A,B)).
R=eval(app(A,B))      :- R=app(eval(A),B).
R=app(value(A),B)     :- R=app(A,eval(B)).
```

The atom `eval` has two arguments; the first is a regular link pointing to the term being evaluated, and the second is another regular link to pass the evaluation result. The atom `value` wraps a term if it is an abstraction. When an abstraction is applied to a value, we generate a substitution. The first rule says that an abstraction is a value. The second rule evaluates the first argument of an application. The third rule evaluates the second argument of an application if the first argument is already a value. The rule `appabs` will be applied when an abstraction is applied to a value. *Call by value* also applies to records and patterns. The details are omitted.

Since both term abstractions and type abstractions are evaluated, we need to define three kinds of substitutions.

```
R=subst(X,V,X) :- hlink(X) | R=V.
R=subst(X,V,Y) :- X\=Y, hlground(V,1,2) | R=Y.
R=subst(X,V,app(M,N)) :- hlink(X), hlground(V,1,2) |
  R=app(subst(X,V,M),subst(X,V,N)).
R=subst(X,V,abs(stype(Y,T),M)) :- X\=Y |
  R=abs(stype(Y,T),subst(X,V,M)).
R=subst(X,V,tapp(M,N)) :- R=tapp(subst(X,V,M),N).
```

Fig. 10 The term to term substitution

```
R=subst_tp_tp(X,V,X) :- hlink(X) | R=V.
R=subst_tp_tp(X,V,Y) :- X\=Y, hlground(V,1,2) | R=Y.
R=subst_tp_tp(X,V,top) :- hlink(X), hlground(V,1,2) |
  R=top.
R=subst_tp_tp(X,V,arr(M,N)) :-
  hlink(X), hlground(V,1,2) |
  R=arr(subst_tp_tp(X,V,M),subst_tp_tp(X,V,N)).
R=subst_tp_tp(X,V,all(stype(Y,T),Z)) :-
  X\=Y, hlground(V,1,2) |
  R=all(stype(Y,subst_tp_tp(X,V,T)), subst_tp_tp(X,V,Z)).
```

Fig. 11 The type to type substitution

```
R=subst_tp_tm(X,V,Y) :- X\=Y, hlground(V,1,2) | R=Y.
R=subst_tp_tm(X,V,abs(type(Y,T),Z)) :-
  X\=Y, hlground(V,1,2) |
  R=abs(type(Y,subst_tp_tp(X,V,T)), subst_tp_tm(X,V,Z)).
R=subst_tp_tm(X,V,app(M,N)) :-hlground(V,1,2)  |
  R=app(subst_tp_tm(X,V,M), subst_tp_tm(X,V,N)).
R=subst_tp_tm(X,V,abs(type(Y,T),Z)) :-
  X\=Y, hlground(V,1,2) |
  R=abs(stype(Y,subst_tp_tp(X,V,T)), subst_tp_tm(X,V,Z)).
R=subst_tp_tm(X,V,tapp(M,N)) :- hlground(V,1,2)  |
  R=tapp(subst_tp_tm(X,V,M), subst_tp_tp(X,V,N)).
```

Fig. 12 The type to term substitution

A hypergraph term `R=subst(X,V,Y)` is a substitution in which occurrences of a *term variable* `X` in a *term* `Y` will be replaced by a term `V`, as shown in Fig. 10. A hypergraph term `R=subst_tp_tp(X,V,Y)` is a substitution in which occurrences of a *type variable* `X` in a *type* `Y` will be replaced by a type `V`, as shown in Fig. 11. A hypergraph term `R=subst_tp_tm(X,V,Y)` is a substitution in which occurrences of a *type variable* `X` in a *term* `Y` will be replaced by a type `V`, as shown in Fig. 12. The link `R` returns the result of a substitution.

The definitions of all the substitutions straightforwardly follow the structure of the terms or types `Y` to which the substitution is applied. We use `hlground(V,1,2)` whenever a term represented by `V` is copied or removed according to the structure of the terms and types `Y`. The reason of `hlground` having two attributes in its arguments is that two kinds of variables are abstracted in System $F_{<:}$, term variables (represented by hyperlinks with attribute 1) and type variables (represented by hyperlinks with attribute 2); therefore both attributes should be specified to ensure correct substitutions. Besides those two points, no complications are involved in the definition of substitutions.

4.3 Type Checking with Equality Assumptions

We can easily turn the typing rules of System $F_{<:}$ into the rewrite rules of HyperLMNtal. Type checking is a process of decomposing type judgments and generating a proof tree. Since the hypergraph representation of System $F_{<:}$ terms and types is almost the same as the description of System $F_{<:}$ terms and types in theory, let us first review a pen-and-paper style type checking so that we know how to model such a process in HyperLMNtal.

Among System $F_{<:}$ typing rules, the rule `TA-TABS` for type abstractions is shown below:

$$\text{TA-TABS}\,\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1.t_2 \ : \ \forall X <: T_1.T_2}$$

The typing rule `TA-TABS` states that the two type variables X's bound by λ and $\forall$ in the conclusion have the same supertype T_1. When a type judgment is given for type checking, the two X's come with different names, which are different hyperlinks in our case. In the premise, the type context is extended by X`<:`T_1 where X refers to the both bound variables of the conclusion. The both abstractions are opened up in the premise, which means we need to rename one type variable to another so that X`<:`T_1 can be provided for typechecking free occurrences (in t_2 and T_2) of the both bound type variables later. Meanwhile, we need to ensure that the both bound type variables have the same supertype T_1. We use hyperlink fusion to merge the two bound type variables in the premise.

An example of a typechecking step with TA-TABS (for *sone* in Sect. 4.1) is given below:

$$\frac{\begin{array}{l} X <: top \vdash \lambda S <: X.\lambda Z <: X.\lambda m : X \to S.\lambda n : Z.mn \\ \quad : \forall S' <: X.\forall Z' <: X.(X \to S') \to Z' \to S' \end{array}}{\begin{array}{l} \vdash \lambda X <: top.\lambda S <: X.\lambda Z <: X.\lambda m : X \to S.\lambda n : Z.mn \\ \quad : \forall X' <: top.\forall S' <: X'.\forall Z' <: X'.(X' \to S') \to Z' \to S' \end{array}}$$

First, we fuse X and X' into one hyperlink so that all the occurrences of them will be identified by the same hyperlink X. Meanwhile, an equality assumption `eq(top,top)` is created. Because `eq(top,top)` obviously holds, it is removed. If we continue the type checking, fusions will be performed several times and many `eq`s will be generated, and all the `eq`s will be removed eventually because the original judgment is correct.

Solving an `eq` is essentially the same as judging if two λ-terms are α-equal because types are abstracted in System $F_{<:}$. An intuitive example is checking $\lambda x.x =_? \lambda y.y$. First we open up abstractions to have $x =_? y[y := x]$, then we have $x =_? x$, which is `true`. The fusion operation acts like an instant variable-variable substitution. Some of the rewrite rules for checking the equality of two types are shown in Fig. 13.

Now we are ready to implement the typing rules. The implementation is straightforward. In the following, we show the implementation of TA-TABS. The hypergraph

```
eq(top,top) :- .
eq(A,A)     :- hlink(A) | .
eq(all(stype(A,B),C), all(stype(A1,B1),C1)) :-
                    A><A1, eq(B,B1), eq(C,C1).
eq(arr(A,B),arr(A1,B1)) :- eq(A,A1), eq(B,B1).
```

Fig. 13 Equality checking of types

`typeof(G,type(Tr,Ty))` represents a type judgment `G` $\vdash$ `Tr` : `Ty` which says that a term `Tr` has the type `Ty` under the type context `G`:

```
typeof(G,type(abs(stype(X1,T11),A),
              all(stype(X2,T12),B))) :-
  hlground(T11) |
  add_env(stype(X1,T11),G,G1),
  typeof(G1,type(A,B)), eq(T11,T12), X1><X2.
```

Here, `add_env(stype(X1,T11),G,G1)` extends the type context `G` with a pair `X1<:T11` and returns the extended type context `G1`. Hyperlinks `X1` and `X2` are fused, and `eq(T11,T12)` is generated for checking the equality of `T11` and `T12`. Type checking is a process of decomposing type judgments and eliminating them eventually. A same type variable may occur in different branches of the derivation tree of the type checking, and such variable occurrences should have the same name to reflect that they are of the same type. Therefore, we use `hlground(T11)` to share the type variables occurring in `T11` between the copies of `T11`.

When type checking is completed successfully, all the `eq`s will be eliminated eventually. When type checking fails for a given type judgment, some `eq` will remain. For example, consider the type checking of Church numeral `typeof([],type(`*sone*`,SZero))`, where *sone* is as defined in Sect. 4.1 and *SZero* is the encoding of the type of Church numeral zero

$$\forall X <: top\,.\,\forall S <: X\,.\,\forall Z <: X\,.\,(X \to S) \to Z \to Z$$

encoded as

```
SZero ≝ abs(stype(X,top),
          abs(stype(S,X),
            abs(stype(Z,X),
              arr(arr(X,S),arr(Z,Z))))).
```

The type checking will end up with a single `eq(X,Y)`, where `X` and `Y` are not the same, thus indicating that the type judgment is wrong. It is wrong because `SZero` is the type of Church number zero in System $F_{<:}$.

Some of the original typing rules of System $F_{<:}$ are not syntax-directed. To implement them, they should be converted to syntax-directed rules called *algorithmic typing rules* [17]. Indeed, we implemented the algorithmic typing rules of System $F_{<:}$.

5. Experiments

In the previous sections, we showed the implementation of

Table 1 Computing Church numerals (1)

Term	Steps(beta)	Execution Time (in ms)
$2\,2\,I\,I$	64(11)	< 1
$2\,2\,2\,I\,I$	358(48)	31
$3\,3\,I\,I$	356(45)	46
$3\,2\,2\,I\,I$	7344(676)	23046
$2\,2\,3\,I\,I$	1308(148)	359
$4\,4\,I\,I$	3522(347)	7983

The I is $\lambda x.x$ and $2, 3, 4$ are Church numerals. $2\,2\,I\,I$ is the same as $((2\,2)\,I)\,I$.

Table 2 Computing Church numerals (2)

Term	Steps(beta)	Execution Time (in ms)
$2\,2$	41(5)	< 1
$2\,2\,2$	287(30)	31
$3\,3$	241(16)	46
$3\,2\,2$	6313(435)	22812
$2\,2\,3$	977(65)	390
$4\,4$	2491(89)	8140

the untyped λ-calculus and System $F_{<:}$[†]. The evaluator of System $F_{<:}$ consists of 30 rewrite rules, the typechecker of System $F_{<:}$ consists of 54 rules, and the definition of substitution consists of 22 rules. We conducted experiments of

1. computing Church numerals in the untyped λ-calculus,
2. computing factorials of Church numerals in System $F_{<:}$,
3. typechecking in System $F_{<:}$.

These experiments were made using a Windows PC with an Intel Core 7 Quad, 3.40 GHz, and 16.0 GB of RAM, running under Windows 10 Pro.

Table 1 shows the execution time and how many times rules are applied until an input is fully evaluated. These inputs are used in benchmarking several λ-evaluators [12], [13]. Table 2 shows examples which evaluate to Church numerals only with strong (full) reduction, i.e., a strategy that reduces terms inside abstraction. Note that our implementation performs strong reduction, while most graph-based evaluators are focused on some form of weak reduction. Note also that the main point of our implementation is that we keep the encoding close to its abstract form as in theory, and efficient reduction is not our primary concern.

The above experimental results are obtained from our encoding of the untyped λ-calculus in which rules are given in a particular order as shown in Fig. 5. The results will be different if we change the order of rules because (i) *hlground* copies and removes a subgraph in these rules, and (ii) the present HyperLMNtal implementation tries rules in a given order, so rules at the top are most likely to be applied.

As can be expected, the number of steps in Table 2 are less than the number of steps in Table 1, while execution time in both tables is very similar.

The number of steps for the input shown in Table 1 is greater than the number of steps for the same input in [12], [13]. This is because terms and substitutions in our approach correspond to the standard definitions, while term and substitutions in [12], [13] are designed for efficient reduction by sharing terms whenever possible.

[†]available at https://gitlab.com/alimjanyasin

Table 3 PoplMark: factorial of Church numerals

Input (n!)	Atoms	Steps	Execution Time (in ms)
2!	185	397	31
3!	213	730	109
4!	229	1350	343
5!	245	2254	1155
6!	261	3494	3156
7!	277	5122	7750
8!	293	7190	17780
9!	309	9750	40561

2! is the factorial of Church numeral 2 in System $F_{<:}$.

Table 4 PoplMark: type checking

Type Checking Input	Steps	Execution Time (in ms)
one	274	15
succ	692	31
pluspp	1134	78
multpp	804	62
two plus three	4629	359
three times (two plus two)	3660	296
two times one	4704	390
one hundred	22725	8297

We tested our implementation of System $F_{<:}$ with the benchmark of the PoplMark Challenge[††]. The results of evaluating System $F_{<:}$ terms are shown in Table 3, and the results of type checking in System $F_{<:}$ are shown in Table 4.

In Table 3, inputs are factorials of Church numerals in System $F_{<:}$. Each of $2!, 3!, \ldots, 9!$ is given in the form of nested `let`, `pattern` terms and basic arithmetic operations defined in System $F_{<:}$, so the benchmark tests the correctness of the implementation with broad coverage. The size of an input is reflected by the number of atoms it contains.

In Table 4, inputs are type judgments in System $F_{<:}$. We observed that our implementation correctly judged all the inputs.

On the PoplMark Challenge website, there are several implementations of System $F_{<:}$. They use nominal technique in αProlog, locally nameless technique in Coq, a combination of higher-order abstract syntax and de Bruijn indices in ATS/LF, and de Bruijn indices in Isabelle/HOL. Here in HyperLMNtal, we used hypergraph based technique, tested our implementation with the inputs of Challenge 3 of the PoplMark Challenge and obtained results in reasonable execution time. This means our hypergraph based technique passed its first test and gave us confidence to pursue further research in future.

6. Related Work and Conclusion

There are a number of proposals in which graphs are used to represent terms with name binding for various motivations. Of these, term graphs are mainly concerned with the sharing of subterms, the aspect which we do not pursue in the present work. Of the proposals addressing binding and substitution, graphs are used to achieve an efficient closed

[††]https://www.seas.upenn.edu/~plclub/poplmark/

reduction strategy for untyped λ-terms [12], [13], for optimal reduction [2], for a small set of rules for strong reduction [21], and so on. Nonetheless, they express multiple occurrences of variables using sharing constructors such as cp (copy) atoms. Terms in these formalizations do not look like standard λ-terms. Furthermore, these techniques take a fine-grained approach, that is, one or two graph elements are rewritten in one step of reduction. As a result, their definition of substitutions does not exactly correspond to the standard definition. In our approach, we have hyperlinks to express variables naturally, and terms represented in our technique look like the terms of the theory. In our technique, the definition of substitution is kept close to that in theory, which seems suitable for quick modeling of formal systems involving name binding.

In addition to the historical account in Sect. 3.2, graph representation of name binding can be found for instance in [11], where variable identity and binders are represented using links different from links representing the tree structure of terms. Another paper [1] describes a technique of handling recursion in the λ-calculus using cyclic graphs. Our emphasis, in contrast, is to stay close to the abstract description of formal systems involving name binding. Although our hypergraph λ-terms do not have cyclic structures formed by regular links, we can handle recursion by using fixpoint combinators, but this is beyond the scope of the present paper.

There are several well-known techniques for representing name binding that are not graph based. A classic technique, the *de Bruijn representation* [6], uses natural numbers to represent variable names. As the result, it needs to define *shifting* operation to keep the indices correct during the substitution and a context to keep track of indices of free variables. Although the *de Bruijn representation* is always criticized for its poor readability, it has been used to complete all the tasks in the PoplMark challenge [7]. The *higher-order abstract syntax* uses the higher-order features of the λ-calculus to encode object systems. However, the cost of implementing high-order features is high [15], [16]. The locally nameless representation [8] is a simple solution, but it has to define variable opening and variable closing operations to define substitutions. Besides, terms in this technique look different from the standard λ-terms. The λ-calculus can be easily encoded in the αProlog language [9], which is based on the *nominal logic* [18], However, the formalization of name binding with swapping and freshness constraints, which are the fundamental part of the nominal logic, seems somewhat difficult to understand for non-experts. Most of these techniques are oriented towards the mechanized reasoning of metatheories of programming languages, while our objective has been to find an appropriate language construct for describing runnable models.

Each of these techniques comes with its own advantages and disadvantages. In designing our technique, we mainly considered the following perspectives:

- readability of terms,
- keeping the framework close to the theory from the user's viewpoint,
- natural support for free variables,
- simplicity of the theory and the cost of the implementation.

To conclude, our contributions are the following. We developed the idea of using hyperlinks as variables and hypergraphs as terms involving name binding in a hypergraph rewriting framework. We designed and implemented graph type *hlground* that is based on graph-theoretic notions and yet enables us to define substitutions. The untyped λ-calculus was encoded and its correctness was proved. We have tested our theory by encoding $F_{<:}$. Implementing System $F_{<:}$ is not a trivial task because System $F_{<:}$ has two kinds of bound variables. Consequently, guaranteeing correct substitutions requires careful reasoning. In our case, substitutions are defined simply by following the structures of terms and using *hlground* whenever a term is copied or removed during the substitution. We implemented type checking of System $F_{<:}$ in HyperLMNtal, a language without built-in unification.

Our technique is currently available only on *HyperLMNtal*, and users should be moderately familiar with the language and know how to use the graph type *hlground*. However, once a user gets the idea, he/she who wants to make a formal system runnable should be able to do so in *HyperLMNtal* by straightforward encoding. Users do not need to 'reinvent the wheel' as in the de Bruijn representation.

We consider the hypergraph based approach a promising one, and hope that HyperLMNtal will find a new application in the prototyping of formal systems involving name binding such as programming languages, type systems and logic. One concern we have currently is that the current implementation of the graph type *hlground* is not very efficient. However, the hypergraph terms obtained from our technique follow the same structure, and it may be possible to exploit it. Our extension to *hlground* currently works under ordinary, i.e., don't-care nondeterministic, execution mode of HyperLMNtal. It is our future work to incorporate our *hlground* into the model checking framework of HyperLMNtal.

Acknowledgments

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

References

[1] Z.M. Ariola and S. Blom, "Cyclic Lambda Calculi," Proc. TACS'97, LNCS 1281, pp.77–106, Springer, 1997.
[2] A. Asperti and S. Guerrini, The Optimal Implementation of Functional Programming Languages, Cambridge University Press, 1998.
[3] B.E. Aydemir, A. Bohannon, M. Fairbairn, J.N. Foster, B.C. Pierce,

P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, "Mechanized Metatheory for the Masses: The POPLMark challenge," Proc. 18th International Conference on Theorem Proving in Higher Order Logics, LNCS 3603, pp.50–65, Springer, 2005.

[4] N. Bourbaki, Théorie des Ensembles, Hermann, 1970.

[5] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, Volume 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1981.

[6] N.G. de Bruijn, "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," Indagationes Mathematicae, vol.75, no.5, pp.381–392, 1972.

[7] S. Berghofer, "A Solution to the POPLMARK Challenge Using de Bruijn indices in Isabelle/HOL," J. Automated Reasoning, vol.49, no.3, pp.303–326, 2012.

[8] A. Charguéraud, "The Locally Nameless Representation," J. Automated Reasoning, vol.49, no.3, pp.363–408, 2012.

[9] J. Cheney and C. Urban, "αProlog: A Logic Programming Language with Names, Binding and α-Equivalence," Proc. ICLP 2004, LNCS 3132, pp.269–283, Springer, 2004.

[10] Y. Dinitz, "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation," Soviet Math Doklady, vol.11, pp.1277–1280, 1970.

[11] W. Kahl, "Relational Treatment of Term Graphs with Bound Variables," Logic Jounal of the IGPL, vol.6, no.2, pp.259–303, 1998.

[12] I. Mackie, "An Interaction Net Implementation of Closed Reduction," Proc. IFL 2008, LNCS 5836, pp.43–59, Springer, 2008.

[13] I. Mackie, "YALE: Yet Another Lambda Evaluator Based on Interaction Nets," Proc. ICFP 98, pp.117–128, ACM, 1998.

[14] M. Meguro, Y. Naoki, and K. Ueda, "Model Checker for Multiple Models of Computation," Conference of Japan Society for Software Science and Technology, 2012, http://www.ueda.info.waseda.ac.jp/lmntal/.

[15] D.A. Miller and G. Nadathur, "Higher-Order Logic Programming," Proc. ICLP'86, LNCS 225, pp.448–462, Springer, 1986.

[16] F. Pfenning and C. Elliott, "Higher-Order Abstract Syntax," Proc. PLDI'88, pp.199–208, ACM, 1988.

[17] B.C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[18] A.M. Pitts, "A First Order Theory of Names and Bindings," Information and Computation, vol.186, pp.165–193, 2003.

[19] F.-R. Sinot, "Call-by-Name and Call-by-Value as Token-Passing Interaction Nets," TLCA 2005, LNCS 3461, pp.386–400, Springer, 2005.

[20] F. Turbak and D. Gifford, Design Concepts in Programming Languages, The MIT Press, 2008.

[21] K. Ueda, "Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting," Proc. RTA 2008, LNCS 5117, pp.392–408, Springer, 2008.

[22] K. Ueda, "LMNtal as a Hierarchical Logic Programming Language," Theoretical Computer Science, vol.410, no.46, pp.4784–4800, 2009.

[23] K. Ueda and S. Ogawa, "HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model," Künstliche Intelligenz, vol.26, no.1, pp.27–36, 2012.

[24] A. Yasen and K. Ueda, "Hypergraph Representation of Lambda-Terms," Proc. TASE 2016, pp.113–116, IEEE Computer Society, 2016.

[25] A. Yasen and K. Ueda, "Implementing a Subset of Lambda Prolog in HyperLMNtal," Conference of Japan Society for Software Science and Technology, 2014, http://jssst.or.jp/files/user/taikai/2014/toc.html.

Alimujiang Yasen received his B.S and M.Eng degrees in Computer Science and Technology from XinJiang University in 2007 and 2010, respectively. He is now a PhD candidate in the Department of Computer Science and Engineering, Waseda University.

Kazunori Ueda received his M. Eng. and Dr. Eng. degrees from the University of Tokyo in 1980 and 1986, respectively. He joined NEC in 1983, and from 1985 to 1992, he was with the Institute for New Generation Computer Technology (ICOT) on loan. He joined Waseda University in 1993 and has been Professor since 1997. He is also Visiting Professor of Egypt-Japan University of Science and Technology sinde 2010. His research interests include design and implementation of programming languages, concurrency and parallelism, high-performance verification, and hybrid systems.