Encoding Type Systems into HyperLMNtal

Alimujiang Yasen, Kazunori Ueda

Type systems are to prevent the occurrence of execution errors during the running of a program. To examine the properties of both type systems and the expressive power of HyperLMNtal, a modeling language based on hyper-graph rewriting, we encoded polymorphic type systems into HyperLMNtal. Type variables are represented by hyperlinks and typing rules are represented by rewriting rules which allow simple and effective implementation of type checking. This paper describes the encoding with running examples.

1 Introduction

LMNtal (pronounced "elemental") is a language model based on hierarchical graph rewriting that uses point-to-point links to represent connectivity and membranes to represent hierarchy. LMNtal's objectives are to serve as a computational model that encompasses diverse formalisms related to multiset rewriting, concurrency, and mobility, and to provide a practical programming language based on hierarchical graph rewriting and its implementation. In recent years, LMNtal was extended to HyperLMNtal, LMNtal with Hyperlinks, which is better suited for representing multipoint connectivity in situations where one-to-one connectivity (represented by ordinary links in LMNtal) is impossible to apply directly.

In order to demonstrate the expressive power of HyperLMNtal to model formal systems, in the past, various computational models, including the λ-calculus and the π-calculus have been encoded into HyperLMNtal [4] [6]. In those encodings, hyperlinks have proved itself as important component which offers straightforward and more efficient modeling capability. To aim for exploring further in language development, this paper studies the encoding of type systems.

Alimujiang Yasen, Waseda University, Dept. of Information and Computer Science, Waseda University.
Kazunori Ueda, Waseda University, Dept. of Information and Computer Science, Waseda University.

The λ-calculus has been a subject to extensive research among the programming languages community, since many concepts and technologies which were first developed on the λ-calculus were transferred to others. It has been a testing ground for programming languages research [1]. To go down further in the development of our programming language, LMNtal, we want to observe its performance for modeling type systems. We encode the λ-calculus polymorphic type systems into HyperLMNtal. By doing so, we want to observe how suitable HyperLMNtal is for the modeling of formal proof systems.

In next section, we will describe the syntax and other features of HyperLMNtal quickly. Then, there will be a short section about type systems. The encoding will be explained in the next section. The final section is dedicated to conclusion.

2 LMNtal and HyperLMNtal Overview

LMNtal views that computation is manipulation of diagrams, which in our case consists of (i) *atoms*, (ii) *links* for one-to-one connectivity, and (iii) *membranes* that can enclose atoms and other membranes and can be crossed by links. Connectivity and hierarchy are two major structuring mechanisms in LMNtal as well as in the real world. The purpose of LMNtal is to provide a programming and modeling language which enables representing and manipulating directly both real-world and cyberworld applications [5].

The two syntactic categories, *link names* (de-

(Process)	$P ::= 0 \mid p(X_1,...,X_m) \mid P,P \mid \{P\} \mid T:\text{-}T$
(Process template)	$T ::= 0 \mid p(X_1,...,X_m) \mid T,T \mid \{T\} \mid T:\text{-}T \mid @p \mid \$p$

Table 1 Syntax of LMNtal.

noted by X_i) and *atom names* (denoted by p) are presupposed. *Processes* are the principal syntactic category and consist of hierarchical graphs and rewrite rules. Being a model of concurrency, LMNtal uses the terms *(hierarchical) graphs* and *processes* interchangeably. 0 is an inert process, $p(X_1,...,X_m)$ $(m \geq 0)$ is an atom with arity m, P,P is parallel composition, $\{P\}$ is a *cell* formed by enclosing P by a *membrane* { }, and $T:\text{-}T$ is a rewriting rule. The two T's are called the *head* and the *body* of the rule, respectively. A rewrite rule is subject to several syntactic conditions. Most notably, a link name occurring in a rule must occur exactly twice in the rule.

The reserved atom name, '=', is called a *connector*. The process $X=Y$ connects the other end of link X and the other end of the link Y. A *rule context*, denoted by $@p$, matches a (possibly empty) multiset of all rules within a membrane, while a *process context*, denoted by $\$p$, is to match processes other than rules within a membrane.

An abbreviation called a *term notation* is frequently used during LMNtal programming. It allows an atom b without its final argument to occur as the kth argument of a. For instance, f(a,b) is the same as a(A),f(A,B),b(B). A list with the elements A_i's can be written as $X=[A_1,...,A_n]$, where X is the link to the list. Some atoms such as '+' are written as unary or binary operators.

Example 1 This example demonstrates the use of process contexts:

```
{+X,$p},{+X,$q} :- {$p,$q}.
```

This says that two cells that are connected by a link (X, which connects two unary atoms +) will be fused.

Example 2 In order to apply a new rule set to a process P, one can write

```
{$p,@z}/,{@q} :- {$p,@q}.
```

Here, the '/' notation is the stability constraint: a cell suffixed by a '/' in the head of a rule can match a cell that cannot be reduced further. In this example, when the first cell has been stabilized (first cell rule set $@z$ cannot apply further), the rule set $@q$ from the second cell will be applied to process $p within the first cell.

LMNtal has provided a *module* system which is useful in cases where program flow is complex. A *module* is a set of rules packaged in a membrane with a *module name* definition.

```
{module(module name), RuleSet}
```

The RuleSet is made available only when it is loaded outside the membrane.

Example 3

```
{
    module(seq).
    seq.run( data{$p},rule{@q} )
        :- seq.run( data{$p,@q} ).
}.
    seq.run( data{a(3),b(2)},
    rule{ a(M),b(N):-a(M+N). } ).
```

In this example, the rule set is applied to data with taking advantage of the module. Using modules with process templates together, controlling of complex program flow can be achieved. The result is:

```
seq.run( data{a(5)}).
```

For concise representation of multipoint connectivity, *hyperlinks* have been introduced into LMNtal and extended it to a hierarchical hypergraph rewriting model, HyperLMNtal. Hyperlinks are essential for our encoding. Hyperlinks are local names shared by any number of atoms [2].

.

```
H :- new($x)     | B.
H :- hlink($x)   | B.
H :- .......     | $x >< $y, B.
H :- num($x,$n)  | B.
```

A local fresh name can be created by a *new* guard constructs as shown (first line of the above code). To check if an argument of an atom is a hyperlink, *hlink* (second line) can be used. *Fusion* (third line) may be the most characteristic operation on hyperlinks. Two hyperlinks $x and $y are merged into one. Another operation is to obtain the *cardinality* of a hyperlink by using *num* ($n is bound to the current cardinality of $x). All hyperlinks occur in the *head* of rewriting rule 'H', except the first line.

3 Type Systems

The main goal of our encoding is demonstrating the modeling power of HyperLMNtal. By doing so, we could have new understanding about the benefits of HyperLMNtal as well as ideas of further improvement. We have chosen constraint typing of simply typed λ-calculus as the subject to our encoding. Thereafter, we will extend it to a more complex type system with let polymorphism.

Since those are well-known formalisms, only to remind the reader in case, typing rules are given.

(var $\succ$)	$\Gamma \vdash x : \tau$ if Γ ok, $ftv(\tau) \subseteq \Gamma_{tv}$ and $\sigma \succ \tau$ for some $(x : \tau\ ) \in \Gamma_{ta}$
(abs)	$\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2 \qquad \text{if } x \notin dom(\Gamma_{ta})}{\Gamma \vdash \lambda x(M) : \tau_1 \to \tau_2}$
(app)	$\dfrac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2}$
(let)	$\dfrac{\text{if } A \bigcap \Gamma_{tv} = \emptyset \text{ and } x \notin dom(\Gamma_{ta}) \quad \Gamma, x : \forall A(\tau_1) \vdash M_2 : \tau_2 \qquad A, \Gamma \vdash M_1 : \tau_1}{\Gamma \vdash \mathbf{let} x = M_1 \ \mathbf{in}\ M_2 : \tau_2}$
(nil)	$\Gamma \vdash \mathbf{nil} : \tau\ list$ if Γ ok and $ftv(\tau) \subseteq \Gamma_{tv}$
(list)	$\dfrac{\Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \tau\ list}{\Gamma \vdash M_1 :: M_2 : \tau\ list}$

Table 2 Typing rules.

Here, the $ftv(\tau)$ is a set of free type variables of type schemes. The "Γ ok" indicates that the free type variables of each type scheme are contained in Γ_{tv}, which is a finite set of type variables. Γ_{ta} is a type scheme assignment function, i.e. a finite function mapping some variable, x_i, to type scheme, σ_i. The A in the *let* typing rule is a set of type variables which occur in τ_1.

4 Encoding Type Systems

The encoding went through two phases. In the first phase, we encoded constraint type checking of the simply typed λ-calculus. In second phase, we extended our encoding with let polymorphism. In the first subsection, we will discuss general issues. In the second subsection, we will present the first phase of our encoding. The third subsection will focus on the encoding of let polymorphism.

4.1 Issues

To model the process of type checking in HyperLMNtal, we have to consider following:

1. (**Expression**) Type checking involves terms, types, derivation and so on. Besides, those interact with each other, especially during derivations. We need to come up with a concise way of representing them by using atoms, membranes and links.
2. (**Relation**) During type checking, terms are subject to derivation. Constraints, generated during the derivation, are subject to unification. One type might occur in many terms, and typing context may be shared by multiple terms.
3. (**Sharing**) It can be seen from typing rules that during derivation, type contexts are always delivered or extended to further derivation. There should be resource sharing or copying.
4. (**Sequentiality**) The type checking process is a combination of derivation and unification. Unfortunately, in LMNtal, controlling program flow is not as easy as doing it in imperative programming languages.
5. (**Fresh variables**) The derivation may need to create fresh type variables.

All those are to be considered simultaneously. For the expressions, *atom*s in HyperLMNtal would be a good choice. To represent the relation be-

tween terms and constraints, links are the best solution. To represent complex relation between multiple terms, *hyperlink*s will be used. The shared resources can be stored in *membrane*s with *atom*s, and *template*s could make it easy to copy if needed. To control the programming flow, we will utilize *template*s with *modules*. For reflecting relationship between type variables, hyperlinks is a natural choice. Hyperlinks are always created fresh. Also, equality checking can be performed on hyperlinks, which is very useful for our encoding.

4.2 Encoding of Type Checking

In this section, we will consider the first phase of our encoding. We pick a simple term and follow its type checking process. Our idea will be presented alongside this process. The most simple case might be the type checking of the following term.

$$\Gamma \vdash \lambda x.x : T$$

We want to find out type T. That term is represented by the following code.

$$sequent(tenv\{\}, type(abs(x, x), !T))$$

Here, *sequent* represents a single conclusion judgment of a typing rule. The first argument of *sequent* is $tenv\{\}$, which stands for an empty typing context. The second argument is a *type* atom which has two arguments. In that atom, the first argument is a term subject to type checking and second one is a type which is represented by a hyperlink. That hyperlink always points to computed type of the term. The notation '!' is another way to create a hyperlink, the same with the *new* construct.

A derivation in a given type system is a tree of judgments, where each judgment is obtained from the ones immediately above it by some typing rule of the system. Each type rule is written a as a number of *premise* judgments above the horizontal line, with a single *conclusion* judgment below the horizontal line. In our encoding, the typing rule *abs* which is applicable to the above term is represented as:

$$abs@@\ sequent(G, type(abs(\$x, M), T)) :-$$
$$unary(\$x) \mid add\_env(G, \$x, !T1, E),$$
$$sequent(E, type(M, !T2)), cs(T, arrow(T1, T2)).$$

Rewriting rules can be prefixed by a *rule name*, such as *abs* by using the '@@' mark. The LHS of a typing rule is a single judgment, which corresponds to the *conclusion* judgment in the typing rule. The *unary* in the guard ensures that $\$x$ is a unary atom. The *add_env* takes a type context G, extends it with a pair, a variable $\$x$ and its supposed type $T1$ (which is newly created). The extended type context can be reached by E. A new *sequent* will be created, which is for type checking of M. And the relationship between types in typing rules is reflected by creating a constraint, which is represented by a *cs* atom. The *cs* reads as "the type T is a function type, $T1 \rightarrow T2$."

To ensure type context sharing and extending in those rules, auxiliary rules for copying and extending type contexts are implemented (one of them is *add_env*). The implementation of other typing rules utilizes similar representation as seen in the above code. The relationship of types is reflected by generating *cs* style atoms. The type context is represented by a membrane which encloses atoms with the name *type*, which illustrates a mapping from a variable to its type. Extending a type context adds a new mapping. In our *var* rule implementation, if we have found two mappings within a type context, those have a common variable and different types, then typing rule *var* creates new *cs* which says those two types are equal.

During the derivation, applying rules creates many *cs* atoms, which are subject to unification after derivation is completed. After derivation is over, redundant data will be removed by using auxiliary rules.

The unification algorithm is implemented to solve a set of constraints (set of *cs*) to find out the type of the given term. We treat them from the viewpoint of a set of equations [3]. In our implementation, data sets and rule sets are separated. A given term for type checking is placed in one membrane, derivation rules and unification rules are placed in the next membrane, occur check rules are in another membrane. This has two major advantages: one is to control sequentiality of type checking (recall Example 3), another is to have clean encoding. The occur check rule set is supplied to data right after the unification. It stops the program from run-

ning further once there is an unsolvable equation detected. Parts of unification rules are as below.

$cs(!T1, !T2)$:- $!T1 == !T2$ |.
$cs(!T1, !T2)$:- $!T1 \backslash = !T2$ | $!T1 >< !T2$.

Thus, we will have a set of equations (we change the atom name *cs* to *sub* after each of them passed occur checking without failing), which represents the type of the given term, as below.

$sub(!H16, arrow(!He, !He))$.

So far, we have presented our work on the first phase of this work. The next subsection will present an encoding of *let polymorphism*. (To manage space, most of the encoding are not displayed.)

4.3 Let Polymorphism

The encoding of let polymorphism is much more challenging, due to several reasons.

1. (**Type scheme**) Until now, our type context only includes a kind of mapping from variables to types. Adding new mapping pairs, from variables to type schemes, will add extra complication and affect our unified structures, such as judgment, type context and so on.
2. (**Sequentiality**) The *let* typing rule asks us to perform type checking twice. This requires more complex sequentiality which leads to difficulty for managing program flow.
3. (**Occurrences**) Keeping track of the occurrences of a let definition inside a let body also adds complexity.
4. (**Instantiation**) To interpret different types for each occurrence of a let definition, we have to come up with a delicate way of instantiation.

It seems we cannot avoid more complex sequentiality, but we can manage it by using *template* and *module* in HyperLMNtal. Another rule set, which controls programmed flow, is added. Each obvious step of type checking is marked with special atoms which are used as *tag*s to detect which rule set should be supplied once the previous process been stabilized. The occurrences of a let definition can be counted by adding an extra rule. In this way we know how many instantiations of a let definition type we should have.

The instantiation is a major problem. The set of equations representing the type of a let definition have to be copied as many times as needed. But the hyperlinks (present in many terms) cannot be copied, because a hyperlink always maintains the same hyperlink name whenever it appears.

Fortunately, HyperLMNtal has provided some features which will simplify our solution. One is that a hyperlink can be created with an attribute, another is that there is a type constraint which will help us copy a hypergraph connected by hyperlinks with a specific attribute and each copy of a hypergraph will be different from others.

init.
init:- $!X : 1$ | $a(!X)$.
$a(\$hl)$:- $hlground(\$hl, 1)$ | $a(\$hl)$, $a(\$hl)$.

The rule in second line creates a hyperlink named X with attribute '1'. The second rule in third line has a constraint in the guard, *hlground*, which takes a hyperlink and an attribute. The *hlground* ensures that two hypergraphs in the two atoms, $a(\$hl)$, after copying, will have different hyperlink names but the same attribute (maintain '1').

The above features lead to very interesting encoding of let polymorphism. It is observed in *let* typing rule that a set A of variables (those variables appear in type of let definition, but not in the type context, recall *let* typing rule), is generated during type checking of the let definition. Thus, the set A only includes hyperlinks we wish to copy. Other hyperlinks which are created during type checking of let body never appear in the set A.

We have found a novel solution. We assign different attributes to hyperlinks which are created at different stages of let type checking. All hyperlinks created during type checking of the let definition are assigned the same attribute, while hyperlinks created during type checking of let body, are assigned another attribute. This mean we are using a hyperlink attribute to refer to a set of hyperlinks. We can copy hyperlinks with an attribute by using a *hlground* constraint in the guard of a rewriting rule. Therefore, we can remove the set A from our implementation of the *let* typing rule as long as we have the records of the attributes.

Without the presence of the set A, we no longer need the type scheme in *let* rule implementation.

What we only need to do is add a few rules which change an attribute value, which is assigned to each hyperlink when it is created, at the time when type checking of let definition is over.

With the help of hyperlink *attribute*s and the *hlground* constraint, only minor changes are added to merge let type checking to the first phase of our encoding.

5 Discussion and Conclusion

Those encoded type systems are well known classic systems. By encoding those type systems, we want to ascertain that HyperLMNtal, a language model based on hierarchical graph rewriting, is expressive enough to model complex systems.

Another finding is that HyperLMNtal is strong enough to model and help us to study formal proof systems. Besides, this work helped us to realize some potential problems in our language implementation. There are more than forty rules in our encoding, of which only typing rules come from direct translation. Many auxiliary rules for type context related operations, sequentiality control and occur checking are implemented additionally. This reflects the fact that encoding of type systems are not an easy task.

There are several interesting findings that can be learned from this work. First, hyperlink are a very natural way to represent multi-point connectivity. This fact came from observing implementation of type rules and unification, which are conveniently encoded with hyperlinks. Second, hyperlinks with attributes greatly simplified our encoding of let polymorphism. With hyperlinks with attributes, we are able to remove an explicit type scheme from our implementation, since an attribute is used to mark a set A of variables (see let rule). Thus, we avoided bringing too many changes to the first phase of encoding.

In the past, some computational models were encoded into (Hyper)LMNtal. This is the first time that HyperLMNtal is used to model a type system. We plan to encode more type systems to demonstrate the value of HyperLMNtal as a unifying model.

References

[1] Benjamin C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, Massachusetts, 2002.

[2] Kazunori Ueda and Seiji Ogawa, HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model. Künstliche Intelligenz, **26**(1), 2012, pp. 27-36.

[3] Alberto Martelli and Ugo Montanari, An Efficient Unification Algorithm. ACM Transactions on Programming Languages and Systems, **4**(2), 1982, pp. 258-282.

[4] Kazunori Ueda, Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting. In Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008), LNCS 5117, Springer, 2008, pp. 392-408 .

[5] Kazunori Ueda, LMNtal as a Hierarchical Logic Programming Language. Theoretical Computer Science, **410**(46), 2009, pp. 4784-4800.

[6] Kazunori Ueda, Towards a Substrate Framework of Computation. To appear in Concurrent Objects and Beyond (COB 2012), LNCS, Springer, 2013 (26 pages).