Towards a Substrate Framework of Computation

Kazunori Ueda[1]

Department of Computer Science and Engineering, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
ueda@ueda.info.waseda.ac.jp

Abstract. A grand challenge in computing is to establish a substrate computational model that encompasses diverse forms of non-sequential computation. This paper demonstrates how a hypergraph rewriting framework nicely integrates various forms and ingredients of concurrent computation and how simple static analyses help the understanding and optimization of programs. Hypergraph rewriting treats processes and messages in a unified manner, and treats message sending and parameter passing as symmetric reaction between two entities. Specifically, we show how fine-grained strong reduction of the λ-calculus can be concisely encoded into hypergraph rewriting with a small set of primitive operations.

1 Introduction

It is fifty years since Carl Adam Petri formalized Petri Nets in his PhD thesis. Since then, we have seen a lot of proposals to capture and formalize the essence of concurrency. Yet, the world of concurrency and concurrent programming is not like its sequential counterpart where Turing machines and the λ-calculus are the two established formalisms. This indicates two things: one is that the world of concurrency has more aspects to address than the sequential world, and the other is that we don't understand concurrency in sufficient depth yet.

The author's research career started with the design of a concurrent programming model, where he reengineered the nuts and bolts of logic programming (such as first-order terms and unification) to have a simple model of communication and synchronization as an improvement over previous attempts [16]. In the resulting model, Guarded Horn Clauses (GHC), processes work on data structures equipped with logical (single-assignment) variables. Processes communicate by instantiating logical variables by unification and observing their values by matching (one-way unification). The dataflow synchronization mechanism provided by one-way unification was widely recognized as the highlight of concurrent logic programming. However, another key highlight of concurrent logic programming is that first-order terms with logical variables were expressive enough to represent sequences of messages with reply boxes (necessary to encode Concurrent Objects) and channel mobility (exactly in the sense of the π-calculus). This is in sharp contrast with many other concurrency formalisms in which communication is heavily studied but data structures are not treated as primary issues.

It is worth mentioning that the development of the concurrent logic programming paradigm and Guarded Horn Clauses proceeded in parallel with the development of Concurrent Objects in early 1980's, partly stimulated by each other. Both shared the high level goal of establishing a concise model of message-passing concurrency but with somewhat different focuses: Concurrent Objects were designed at a higher level of abstraction, that is, in terms of objects and messages, while concurrent logic programming languages were designed as a substrate model consisting of a smallest possible set of primitive constructs and still allows the *encoding* of Concurrent Objects and operations on them [17].

In spite of the nice properties and the expressive power of Guarded Horn Clauses, it was felt that we could treat processes and data in a more unified manner. Communicating fine-grained processes may form process structures (such as lists or grids of processes) that act as autonomous, concurrent data structures, but process structures and ordinary data structures had to be handled very differently.

With this motivation, the author designed LMNtal a decade ago as his second model of concurrent programming based on a small class of *graph rewriting*, where nodes represent processes or data and edges represent one-to-one channels or links. In fact, nodes are nothing more than atomic logical formulas and edges are nothing more than zero-assignment logical variables (i.e., variables that will never get concrete values). First-order terms, which are trees with constructors and variables, are represented using relations by employing one $(n+1)$-ary relation for each n-ary constructor.

It turned out that LMNtal was almost backward compatible with concurrent logic languages. LMNtal also allowed interpretation as a linear logic programming language [23]. Nevertheless, the real challenge of LMNtal was to have a formalism that could be understood without deep technical knowledge in computer science (such as logic and categories). The adoption of graphs, a widespread mathematical notion, as the basic structure is motivated exactly by this goal. The versatility of the graph data structure became evident when our publicly available LMNtal implementation was tailored into a model checker that used LMNtal as a modeling language for state transition systems in general [25].

The present paper is concerned with our next step of language evolution. We have already incorporated *hyperlinks* (links interconnecting multiple points) in addition to one-to-one links so that it may better cover a broader range of computational models. The resulting language, HyperLMNtal [26], has much in common with Bigraphical Reactive Systems [14], and its versatility and viability seem to be worth exploring in depth to establish it as a substrate model of computation.

The rest of the paper is organized as follows. Section 2 briefly describes LMNtal with some examples. Section 3 describes how LMNtal evolved to HyperLMNtal. Section 4 discusses static analysis techniques that reveal two important properties of programs, capabilities of links and multiplicities of nodes. As one of the most challenging case studies, Section 5 presents a fine-grained encoding

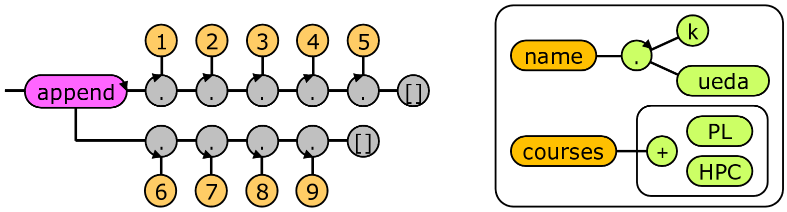

Fig. 1. LMNtal graphs: A tree-shaped graph with one free link (left) and a hierarchical graph with two membranes (right). An arrowhead indicates the first argument and the ordering of links.

of the full reduction of the lambda calculus into HyperLMNtal and how the type system of Section 4 gives further computational structures to the encoding.

2 LMNtal: a Model and Language Based on Graph Rewriting

This section reviews LMNtal briefly. Technical details and relation to other formalisms can be found in [23].

LMNtal embodies the view that computation is manipulation of graphs, which in our setting consist 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 the two major basic structuring mechanisms in both real worlds and cyberworlds including computing, society, biological systems and body of knowledge. The purpose of LMNtal is to provide a concise programming and modeling language that allows us to represent the two mechanisms simultaneously and manipulate them in a direct manner. Figure 1 shows some graphs that can be manipulated by LMNtal.

The choice of links and membranes as structuring mechanisms allows programming with sets and graphs. Although sets and graphs are less common than arrays, records and pointers as programming language constructs, they are more standard and commonly used in the rest of the world and in mathematics in particular. As a formalism of concurrency, an important feature of LMNtal is that it provides a well-defined notion of atomic actions: graph rewriting by a single rule application is always done atomically.

2.1 Basic syntax

The syntax of LMNtal is defined as in Fig. 2, where the syntactic constructs not used in this paper are omitted for simplicity.

The two syntactic categories, *link names* (denoted 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. In the concrete syntax, capitalized names represent links, while other names (e.g., those starting with lowercase letters, numbers, and non-alphanumeric symbols) represent atoms.

A process P must observe the following *link condition*: Each link name in P (excluding link names occurring in rules) may occur *at most twice.* A link whose name occurs exactly once (twice) in P is called a *free link* (*local link*) of P, respectively. The T's are *process templates* that are used in rewrite rules and handle local contexts, namely contexts within particular membranes.

$$\begin{array}{rl} \text{(Process)} & P ::= \mathbf{0} \mid p(X_1,\ldots,X_m) \mid P,P \mid \{P\} \mid T \texttt{:-} T \\ \text{(Process template)} & T ::= \mathbf{0} \mid p(X_1,\ldots,X_m) \mid T,T \mid \{T\} \mid T \texttt{:-} T \mid \texttt{@}p \mid \texttt{\$}p \end{array}$$

Fig. 2. Syntax of LMNtal.

0 stands for an inert process (represented as an empty symbol in the concrete syntax), $p\texttt{(}X_1,\ldots,X_m\texttt{)}$ stands for an atom with m links, and P,P stands for parallel composition called a *molecule.* Note that links of an atom are totally ordered and that multiple links between atoms are allowed as well as links connecting the same atom. $\texttt{\{}P\texttt{\}}$ is called a *cell* and stands for a process enclosed by a *membrane* `{ }`. $T \texttt{:-} T$ stands for a rewrite rule, which is applied to processes located at the same place of the hierarchy formed by membranes. The two T's are called the *head* and the *body* of the rule, respectively. A rewrite rule is subject to several syntactic conditions [23]. 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 $X \texttt{=} Y$ short-circuits the link X and the link Y. A *rule context*, denoted by $\texttt{@}p$, matches the (possibly empty) multiset of all rules within a membrane, while a *process context*, denoted by $\texttt{\$}p$, is to match all processes other than rules within a membrane.

2.2 Operational Semantics

The operational semantics of LMNtal (Fig. 3) consists of structural congruence defined by (E1)–(E10) and the reduction relation defined by (R1)–(R6). (E4) stands for α-conversion. (E9)–(E10) are the interaction rules between atoms/cells and connectors.

Computation proceeds by rewriting processes using rules collocated in the same place of the nested membrane structure. (R1)–(R3) are standard structural rules, while (R4)–(R5) are the mobility rules of `=`. The central rule of LMNtal is (R6), in which θ is to map process contexts into actual processes. For programs that do not use membranes, (R6) degenerates to a simpler form: $T\texttt{,}(T \texttt{:-} U) \longrightarrow U\texttt{,}(T \texttt{:-} U)$.

2.3 Extended Syntax and Examples

A rule may be prefixed by a rule name and two `@`'s.

$$\text{(E1)}\quad \mathbf{0}\,,P \equiv P \qquad \text{(E2)}\quad P\,,Q \equiv Q\,,P \qquad \text{(E3)}\quad P\,,(Q\,,R) \equiv (P\,,Q)\,,R$$

$$\text{(E4)}\quad P \equiv P[Y/X] \qquad \text{if } X \text{ is a local link of } P$$

$$\text{(E5)}\quad P \equiv P' \Rightarrow P\,,Q \equiv P'\,,Q \qquad \text{(E6)}\quad P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$$

$$\text{(E7)}\quad X = X \equiv \mathbf{0} \qquad \text{(E8)}\quad X = Y \equiv Y = X$$

$$\text{(E9)}\quad X = Y\,,\ P \equiv P[Y/X] \qquad \text{if } P \text{ is an atom and } X \text{ occurs free in } P$$

$$\text{(E10)}\quad \{X = Y\,,\ P\} \equiv X = Y\,,\ \{P\} \qquad \text{if exactly one of } X \text{ and } Y \text{ occurs free in } P$$

$$\text{(R1)}\quad \frac{P \longrightarrow P'}{P\,,Q \longrightarrow P'\,,Q} \qquad \text{(R2)}\quad \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}} \qquad \text{(R3)}\quad \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)}\quad \{X = Y\,,P\} \longrightarrow X = Y\,,\{P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } \{X = Y\,,P\}$$

$$\text{(R5)}\quad X = Y\,,\{P\} \longrightarrow \{X = Y\,,P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } P$$

$$\text{(R6)}\quad T\theta\,,(T \mathrel{:\!-} U) \longrightarrow U\theta\,,(T \mathrel{:\!-} U)$$

Fig. 3. Structural congruence and reduction relation of LMNtal.

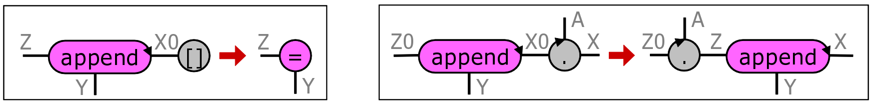

Fig. 4. Reaction rules for `append`.

An abbreviation called a *term notation* allows an atom b without its final argument to occur as the kth argument of a, to mean that the kth argument of a and the final argument of b are interconnected. For instance, `f(a)` is the same as `f(A),a(A)`. This can be written also as `f=a` because `f(A),a(A)` is congruent (i.e., convertible in zero steps) to `f(A),A=A1,a(A1)` (by (E9) of Fig. 3), to which we can apply the abbreviation twice to obtain `f=a`. 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.

Example 1 Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z), '[]'(X0) :- Y=Z.
append(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), append(X,Y,Z).
```

This form makes it explicit that there is no distinction between predicate symbols and constructors, but we can write it also in a more familiar, Prolog-style form:

```
append([],Y,Z) :- Y=Z.
append([A|X],Y,Z0) :- Z0=[A|Z], append(X,Y,Z).
```

Figure 4 shows a diagrammatic representation of the two rules, where the arrowheads indicate the first arguments of atoms and the ordering of arguments. □

Some atoms such as '`+`' are written as unary or binary operators. Parallel composition (e.g., $P_1, P_2, \ldots, P_n$) can be written also in a period-terminated form (e.g., P_1. P_2. ... P_n.).

Example 2 The dining philosopher problem can be represented as a circular graph with philosophers and forks:

```
phi(L1,R1), fork(+R1,+L2),  phi(L2,R2), fork(+R2,+L3),
phi(L3,R3), fork(+R3,+L4),  phi(L4,R4), fork(+R4,+L5),
phi(L5,R5), fork(+R5,+L1).

fork(+X,+L), phi(L,R) :- fork(-X,+L), phi(L,R). % grab left fork

fork(-X,+L), phi(L,R), fork(+R,+Y) :-
fork(-X,+L), phi(L,R), fork(+R,-Y).            % grab right fork

fork(-X,+L), phi(L,R), fork(+R,-Y) :-
fork(+X,+L), phi(L,R), fork(+R,+Y).            % release forks
```

Each link represents a philosopher's access to a fork, and the atoms '`+`' and '`-`' indicate the availability of the fork. The LMNtal model checker constructs and visualizes the state space of the model, where the state space construction algorithm takes advantage of the symmetry of the circular graph to avoid state space explosion. □

Example 3 One of the uses of membranes is to encapsulate rules and delimit their scope of effect. Suppose we have the following rule:

```
{module(m),@m}, {use(m),$p,@p} :- {module(m),@m}, {$p,@p,@m}
```

The first cell stands for a rule set repository with a module name, while the second cell stands for a "test tube" that requires the rules in the module `m`. This rule causes a new copy of `@m`, which is the content of the module `m`, to be loaded to other cells containing `use(m)`. □

The rest of this subsection is about how to specify operations on primitive datatypes.

We note that numbers in LMNtal are unary atoms such as `8(X)`, where `X` is connected to the atom referring to the number. To specify operations on primitive types such as integers, the two constructs, *typed process contexts* and *guards*, are introduced. While the process matched by an ordinary process context is determined by the membrane it belongs to (i.e., hierarchy), the process matched by a typed process context is determined by the graph structure (i.e., connectivity) and the atom name inside the structure. For instance, the guarded rule

```
p(X), $n[X] :- int($n), $n>0 | p(Y), $n[Y], p(Z), $n[Z].
```

means that, when a unary atom `p` is connected to a positive integer, that two-atom molecule will be duplicated. The guard constraint `int($n)` requests that `$n[X]` is a typed process context (with one free link) representing an integer atom, and the constraint `$n>0` requests that the value of the integer is positive.

Available type constraints other than `int` include `unary` (standing for unary atoms) and `ground` (standing for non-hierarchical connected graphs with exactly one free link). Note that `int` is regarded as a subtype of `unary` which in turn is a subtype of `ground`. A rule containing typed process contexts can be viewed as a *rule scheme* that represents a set of rules without guards.

3 Incorporating Hyperlinks

The design decision of LMNtal to feature one-to-one links rather than hyperlinks came from the observation that multi-point connectivity could be encoded using membranes. However, membranes are a general construct that can be used to enclose processes and rules for localized reactions and are somewhat heavyweight for representing just multi-point connectivity or multisets. Also, program variables in most computational models and languages represent possibly shared data, and efficient and succinct encoding of those program variables is of practical importance. This motivated us to incorporate hyperlinks into LMNtal. The main driving force was to build an efficient encoding of Constraint Handling Rules (CHR) [6], a constraint programming language syntactically close to LMNtal.

The first step towards HyperLMNtal was the design of the hyperlink construct. Since the design and implementation of LMNtal was quite stable, it was considered ideal if the extension could be made smoothly without changing the basic framework of the language and its implementation or affecting the performance of existing applications. The two design choices we have made are the following:

- *Distinguish between links and hyperlinks.* Although hyperlinks could be regarded as subsuming links, we maintain the distinction between them. Each link connection has exactly one partner, and this property is not only a fundamental program invariant but also utilized by the implementation of links in many ways, including the access to partners and the garbage-collection of partners.
- *Treat hyperlinks as atoms with local names.* Having decided that hyperlinks are a syntactic category different from links, we must decide whether to incorporate something totally new or something close to an existing category. We already observed that links are local names shared by exactly two atoms [23], and this suggests that hyperlinks should be treated as local names shared by any number of atoms. This can be realized by providing a construct to create a unary atom with a fresh local name, because unlike link names, unary atoms can be copied and discarded in our extended syntax.

Now we describe the constructs provided for hyperlink manipulation. A fresh local name can be created by a `new` guard construct as:

H `:- new($x) |` B

where the hyperlink name `$x` can be used in B. The scope of x (i.e., the set of atoms that can access x) is B initially, but it may extend to other atoms in the course of graph rewriting, as is the case with local names of the π-calculus.

To check if an argument of an atom is a hyperlink, we write:

H `:- hlink($x) |` B

where `$x` occurs in H. Because `hlink` is a subtype of `unary`, the equality and inequality of hyperlinks can be checked using guard constraints '`==`' and '`\==`'.

Motivated by Linear Logic, we allow hyperlinks to be written in the form `!`X (X capital), in which case either of the guard constraints `hlink` and `new` is implicitly provided, depending on whether the hyperlink occurs in the head or not.

The most characteristic operation is the *fusion* of two hyperlinks,

H `:- ... | !X >< !Y,` B`.`

which is a hyperlink version of the connector '`=`' and interconnects two hyperlinks by fusing two hyperlink names. In the abstract syntax, `><` will be denoted as $\bowtie$.

Another characteristic operation is to obtain or check the *cardinality* (i.e., number of endpoints) of a hyperlink:

H `:- num(!X,$n) |` B

where `!X` occurs in H and `$n` is bound to the current cardinality of `!X`.

The shorthand notation illustrated below allows a hyperlink to occur more than once in the head of a rule to represent sharing:

Example 4

```
a(!X), b(!X), c(!X) :- .
```

is the same as

```
a($x), b($x0), c($x1) :-
   hlink($x), hlink($x0), hlink($x1),
   $x==$x0, $x==$x1 | .
```

This rule removes three unary atoms `a`, `b`, `c` if they share the same hyperlink, in which event the cardinality of that hyperlink is reduced by three. □

4 Analyzing HyperLMNtal Programs

Since graphs are highly general data structures, programming with graphs will greatly benefit from tools and techniques for analyzing and understanding the properties and the behavior of programs.

With this motivation, we have developed a model checker for LMNtal and its integrated development environment (IDE)[25] and found them extremely useful for analyzing the state space of nondeterministic concurrent systems. The model checker can presently handle systems with a half billion states with various optimization techniques and shared-memory parallel processing. The LMNtal model checker was later extended to handle hypergraphs.

HyperLMNtal as a programming language requires no declarations of any kind (variables, types, procedures, etc.), and could be positioned as *a scripting language for model checking* in the sense that it allows concise description and quick development of small- to medium-scale models. To analyze hypergraph rewriting, however, static analysis will also play an important rôle because we are far less familiar with computing with hypergraphs than computing with lists and trees. It will also identify important subclasses of programs that are more legitimate or likely to be correct than others.

With this in mind, we describe two static analysis techniques that address aspects of program properties in a clear way. Our goal here is not to analyze programs as precisely as possible; rather, we are concerned with extracting simple properties by abstracting others. Chemistry suggests that the fundamental properties of graphs are those about (chemical) atoms and those about bonds, and we follow this metaphor. For the sake of simplicity, henceforth we focus on Flat HyperLMNtal, a fragment of HyperLMNtal without membranes, and call Flat HyperLMNtal simply as HyperLMNtal.

4.1 Assigning Polarities and Capabilities to Links and Hyperlinks

Let a *port*, denoted by $\langle a, i\rangle$, stand for the ith argument of an atom a, which is an endpoint of a link or a hyperlink. Firstly, we are interested in which port of an atom may be connected to which port of the same or another atom. This information will lead to a type system that deals with graph structures based on a local view. Although the connectivity information alone may not capture the global shape of data (e.g., whether a grid forms a square or a rectangle; whether a list is terminated by a nil or not), it addresses local properties of structures with sharing (i.e., more than one path leading to a single atom) and cycles.

Secondly, we may be interested in the rôle of each port. Informally, by a rôle we mean the polarity or capability of a port, where a polarity stands for the direction of access (i.e., whether the port is used for sending data or receiving data), information flow, or ownership (i.e., whether the port of an atom is used to access data it owns or is used to be accessed by its owner), while a capability additionally stands for whether the port of a hyperlink has exclusive access to the partner(s) or shares them with others. We focus on this second aspect of ports, namely their polarities and capabilities.

Technically, we choose to represent the capability of a port using a real number between -1 and $+1$ inclusive. The capability value $+1$ means that the port stands for an exclusive, full ownership of (or exclusive reference to) the partner atom(s), while a value $0 < c < 1$ stands for a non-exclusive, partial ownership (or shared reference). The value -1 stands for a sole source or access point of data available to the partner(s). A hyperlink with a -1 port and several positive fractional ports is a *directed hyperarc* of a specific kind called a *backward hyperarc* [7] (Fig. 5), and can be represented by a family of pointers pointing to the atom with the -1 port. A value $-1 < c < 0$ represents a partial source and appears when a partial ownership is returned through that port. The value 0 stands for an inactive port not connected to anywhere else.

This notion of capability inherits the author's capability type system designed for a class of Flat GHC programs [21], except that the system described in the present paper simplifies the original one thanks to the unified treatment of

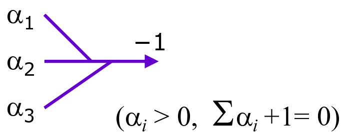

Fig. 5. A backward hyperarc with three owners.

processes and data and by focusing on individual (hyper)links. The use of fractions in type systems later appeared in [3] and subsequent papers with different settings. Our type system is unique in that it uses negative as well as positive fractions, enjoying symmetry around zero. Historical account of fractional type systems can be found in [19].

The capability type system for (Flat) HyperLMNtal consists of the constraints given in Fig. 6, where a capability type c is formulated as a function from the set of ports to the closed interval $[-1, +1]$. To be useful, our type system is necessarily polymorphic because capabilities are very often split and passed to atoms with the same name, that is, they keep decreasing in the course of recursion. The polymorphism is realized by suffixing each atom in a rule l :- $r_1, ..., r_n$ as l_s :- $r_{1,s.1}, ..., r_{n,s.n}$, where a suffix s is a sequence of indexes, and $s.i$ means appending an index i to s. Atoms with the same name and different suffixes are subject to the same constraints but may adopt different solutions.

The first constraint, (Conn), represents the relationship between the ports of a connector and a fuser.

The key advantage of our formulation is that the central type constraint on a (hyper)link is exactly *Kirchhoff's current law* (KCL), i.e., the capabilities of the endpoints of a (hyper)link sum up to 0.

Constraint (Coop) states that

- if a left-hand side occurrence of L has a positive capability, all the left-hand-side occurrences must have positive capabilities and jointly act as the source of data in the rule, and
- otherwise exactly one of L's occurrences in the right-hand side must have a negative capability and act as a single source of data.

In other words, it states that that a hyperlink L represents a backward hyperarc (if $k = 0$) or transforms a backward hyperarc into another backward hyperarc (if $k \geq 1$) by using the rule. Note that the capability of a hyperlink occurrence on the left-hand side of a rule must be negated because the left-hand side acts as a template of rewriting. This constraint states also that the capability of a hyperlink port should be nonzero. The non-zero condition is to disallow the "silent" participation to and withdrawal from a hyperlink.

Constraint (Link) states that the capability of a link should be either 1 or -1 (i.e., non-fractional). We could allow a singleton hyperlink with a zero capability as was done in [21], but will not discuss it here.

Although we have installed the suffix system to allow polymorphism, Occam's razor tells us that type inference as an explanation of program properties should

(Conn) If $(X_1 \mathtt{=}_s X_2) \in Q$ then $c(\langle \mathtt{=}_s, 1\rangle) + c(\langle \mathtt{=}_s, 2\rangle) = 0$;
If $(X_1 \bowtie_s X_2) \in Q$ then $c(\langle \bowtie_s, 1\rangle) + c(\langle \bowtie_s, 2\rangle) = 0$

Let a link or a hyperlink L occur $n\,(\geq 1)$ times in P and Q at $p_1, \ldots, p_n$, of which the occurrences in P are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

(KCL) $-c(p_1) - \cdots - c(p_k) + c(p_{k+1}) + \cdots + c(p_n) = 0$ (Kirchhoff's Current Law)

(Coop) If $k = 0$ then $\mathcal{R}\big(\{c(p_1), \ldots, c(p_n)\}\big)$;
If $k \geq 1$ then $\mathcal{R}\big(\{-c(p_1), c(p_{k+1}), \ldots, c(p_n)\}\big)$;
where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \exists s \in S\big(s < 0 \land \forall s' \in S \setminus \{s\}\ (s' > 0)\big)$$

(Link) If L is a link then $c(p_k) \in \{-1, 1\}$ for $1 \leq k \leq n\,(= 2)$

Fig. 6. Capability constraints imposed by a rule P :- Q.

– prefer maximally general solutions to those with unnecessary constraints and
– prefer least polymorphic (i.e., most uniform) solutions to those that give different types to each instance of atoms with the same name.

As an example, we first consider polarizing `append` (Sect. 2.3) in a monomorphic setting.

Example 5 The constraints imposed by (KCL) and (Conn) on `append` in a monomorphic setting are as follows:

$c(\langle \mathtt{append}, 1\rangle) + c(\langle \mathtt{[]}, 1\rangle) = 0$	by `X0`
$c(\langle \mathtt{append}, 3\rangle) + c(\langle \mathtt{append}, 2\rangle) = 0$	by `=`
$c(\langle \mathtt{append}, 1\rangle) + c(\langle \mathtt{.}, 3\rangle) = 0$	by `X0`
$c(\langle \mathtt{append}, 3\rangle) = c(\langle \mathtt{.}, 3\rangle)$	by `Z0`
$c(\langle \mathtt{.}, 2\rangle) = c(\langle \mathtt{append}, 1\rangle)$	by `X`
$c(\langle \mathtt{.}, 2\rangle) + c(\langle \mathtt{append}, 3\rangle) = 0$	by `Z`

This is satisfiable with the following solution satisfying (Coop) and (Link) also:

$c(\langle \mathtt{append}, 1\rangle) = 1$, $c(\langle \mathtt{append}, 2\rangle) = 1$, $c(\langle \mathtt{append}, 3\rangle) = -1$,
$c(\langle \mathtt{[]}, 1\rangle) = -1$,
$c(\langle \mathtt{.}, 2\rangle) = 1$, and $c(\langle \mathtt{.}, 3\rangle) = -1$, (no constraints on $c(\langle \mathtt{.}, 1\rangle)$)

which intuitively means an `append` reads its first and the second arguments and writes to the third argument. However, this is not the only satisfying assignment, and another solution is:

$c(\langle \mathtt{append}, 1\rangle) = -1$, $c(\langle \mathtt{append}, 2\rangle) = -1$, $c(\langle \mathtt{append}, 3\rangle) = 1$,
$c(\langle \mathtt{[]}, 1\rangle) = 1$,
$c(\langle \mathtt{.}, 2\rangle) = -1$, and $c(\langle \mathtt{.}, 3\rangle) = 1$, (no constraints on $c(\langle \mathtt{.}, 1\rangle)$).

The latter solution makes practical sense. A '`.`' here can be thought of an *active message* that activates `append` which generates another '`.`' that may act on the subsequent procedure connected to the third argument of `append`. This message-oriented scheduling policy was studied and implemented in the logic programming context [20], but our framework which does not distinguish between predicate symbols and constructors provides more uniform treatment of different reduction strategies. □

A set D of atoms is said to *dominate* the left-hand side of a rule if all the atoms in the left-hand side can be reached (by following directed (hyper)links) from the atoms in D. In the case of the above example, the atom `append` dominates the left-hand sides in the first solution, while '`.`' and `[]` dominate the left-hand sides in the second solution. One may wish to interpret dominators as procedures and non-dominators as data.

An example involving hyperlinks will be described in Section 5.6.

The capability type system turns hypergraphs into directed hypergraphs and allows them to be represented using one-way pointers. The type system thus provides key information for compiler optimizations even when undirected links are a more natural tool for modeling purposes (e.g., when directed links break symmetry). It will also help deeper understanding of graph structures and debugging.

We have confirmed that most simple LMNtal programs allow uniform polarization that gives the same polarity vector to atoms with the same name. We have found exceptions as well:

Example 6 A program constructing a fullerene (C_{60}) structure does allow polarization but needs two different polarity vectors for `c`:

```
dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
   p(T0,T1,T2,T3,T4), p(L0,L1,H0,T0,H4), p(L2,L3,H1,T1,H0),
   p(L4,L5,H2,T2,H1), p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).

dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9), /* top half */
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1). /* bottom half */

/* icosahedron -> fullerene */
p(L0,L1,L2,L3,L4) :- X=c(L0,c(L1,c(L2,c(L3,c(L4,X))))).
```

It is easy to see that there is no uniform (monomorphic) solution because exactly half of the 180 ports provided by the 60 ternary carbon atoms must be positive, namely 1.5 ports per atom. □

Thanks to the algebraic formulation, it is rather straightforward to prove the subject reduction property:

Theorem (subject reduction). If a program $P : c$ and $P \longrightarrow Q$ then $Q : c$.

4.2 Composition Analysis

Many programs that handle data structures enjoy beautiful invariants with respect to the size of data. The `append` program is a typical example, where the

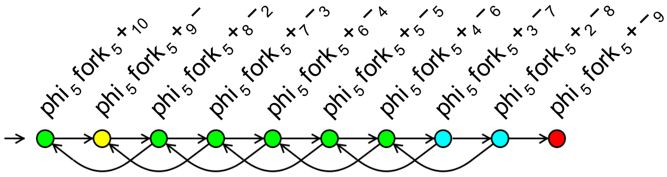

Fig. 7. Size space of the dining philosopher program.

recursive rule just changes connectivity and preserves all the atoms, while the base case rule loses two atoms, `append` and `[]`. The possibly non-terminating dining philosopher program preserves the total number of atoms, while the composition of the '`+`'s and the '`-`'s keeps changing. This suggests that composition analysis (exactly in the sense of chemistry) could be a useful tool to analyze properties about the number of atoms.

Remarkably, properties about the increase and decrease of atoms in state transition systems have been studied in depth in the field of Petri Nets [1]. Boundedness of the number of tokens at each place is a fundamental property of Petri Nets and is analyzed by forming a reachability graph of possible markings. The possible markings of unbounded Petri Nets can be represented using coverability graphs, which over-approximates possible markings of unbounded Petri Nets using the ordinal number ω, and much work has been done on the algorithms for constructing coverability graphs [5][15].

Now notice that place/transition nets, the most basic form of Petri Nets, are exactly multiset rewriting systems; they are different representations of the same thing. Note also that graph rewriting degenerates to multiset rewriting simply by forgetting about links. Thus, it is almost trivial to have a multiset rewriting system corresponding to a given LMNtal program and analyze its state space that captures just the composition of atoms.

Example 7 Figure 7 shows the state space of the dining philosopher program with respect to the number of atoms, which was visualized by our LMNtal IDE. The multiplicities of atoms are indicated by suffixes as in chemical formulae. The shape clearly indicates that the number of available forks is reduced one by one, possibly leading to deadlock, while it may be increased by two at a time. □

Example 8 Let us consider `append` again. The polarization of links establishes an interpretation of the initial graph such as

```
X = append([1,3,5,7,9],[11,13,15])
```

as a binary tree. Composition analysis tells that, upon termination, there will be a single list consisting of the initial elements. Now notice that the preorder traversal of the above tree, modulo `append` and `[]`, visits the list elements in increasing order, and the second rule of `append` is exactly a tree rotation operation that does not affect preorder traversal (modulo `append` and `[]`).

The same line of argument applies to establish the associativity of append:

```
X = append(append([1,3,5],[7,9]),[11,13,15])
X = append([1,3,5],append([7,9],[11,13,15]))
```

Each of the above graphs is obtained by rotating the two `append` atoms at the top of the other graph, which preserves preorder traversal modulo `append` and `[]`. □

5 Encoding the Pure Lambda Calculus into HyperLMNtal

Substitutions is the éminence grise of the λ-calculus.
—Abadi et al. (1991) [2]

The usual implementation of functional programming languages based on a weak evaluation paradigm (no reduction inside a lambda), betray the very spirit, i.e., the higher-order nature, of lambda-calculus.
—Asperti (1998)

One of the most significant challenges in the (Hyper)LMNtal project has been to have a concise encoding of the λ-calculus. This may sound surprising, but it was a real challenge because HyperLMNtal's connection to the λ-calculus was far less obvious than to concurrency calculi such as the π-calculus and the ambient calculus [27]. The encoding of the λ-calculus is significant because the λ-calculus and λ-terms play fundamental rôles not only in functional languages but in the treatment of variable binding, scoping, and substitutions that appear in various formalisms.

The core of the λ-calculus is β-reduction, $(\lambda x.M)N \rightarrow M[x \mapsto N]$, but the definition of substitutions used here is far from simple and provoked various alternative formulations. In particular, "to replace all the free occurrences of x by copies of N" does not necessarily reflect actual implementation, which may share the representation of N whenever possible but must sometimes make copies of N (e.g., when applying another λ-term to N).

One of the formalisms aiming at the precise representation of the λ-calculus is the $\lambda\sigma$-calculus [2], which provides two syntactic categories, λ-terms and explicit substitutions, and gives rewrite rules to both.

Another approach to formalizing the λ-calculus is to adopt graph representation of λ-terms; a bound variable can most naturally be represented as an edge (or a hyperedge) that connects the defining and applied occurrences of the same variable. Most previous work in this approach adopted Interaction Nets [8] to represent and manipulate graphs ([9][11][12], to name a few). Many of the encodings of the λ-calculus into Interaction Nets pursued optimal sharing or efficiency, and resulted in more or less involved representation of λ-terms to achieve the objective. One notable exception is the encoding by Sinot [18], which addressed the simplicity of the encoding, but it focused on the weak λ-calculus that did not evaluate the body of λ-abstractions. Indeed, as KCLE [12] suggests, encoding of the pure calculus can be much less concise (in terms of the number of rules involved) than the encoding of the weak calculus. Weak λ-calculi may be appropriate for the foundations of functional languages, but the applications of the λ-calculus as a whole call for *strong* (or *pure*) λ-calculi as well.

This raises one question: Is there any *concise* graph-based encoding of the *pure* λ-calculus? With Interaction Nets, YALE [11] proposes a relatively simple solution but still needed to simulate "boxes" for scope management. So the next question is: To obtain a more concise encoding (appropriate, say, for an undergraduate text), what additional constructs should be included to the graph rewriting framework?

A concrete answer to these questions was given using LMNtal by presenting a fine-grained and highly nondeterministic encoding of the pure λ-calculus (with open terms) and discussing its properties [24]. Although the membrane construct of LMNtal provides powerful functionalities such as the copying of the graph enclosed by a membrane, the encoding used membranes only to represent and manipulate fresh local names, called *colors*, so that each rewrite step could be executed in (almost) constant amortized time. Thus the encoding was essentially not specific to LMNtal, and the evolution of LMNtal to HyperLMNtal gives us another chance of bringing insights on what constructs are most basic for concise encoding. The purpose of this section is to describe our encoding of the pure λ-calculus into typed HyperLMNtal whose hyperlink manipulation is significantly more restricted than that of untyped HyperLMNtal. Since each of the proposed rewrite rules is simple and well-motivated, the proposed method is expected to serve not only as an encoding but as a fine-grained reformulation of the pure λ-calculus.

5.1 Representing $\boldsymbol{\lambda}$-terms in HyperLMNtal

Now we describe our encoding of the λ-calculus into (Flat) HyperLMNtal. Our starting point was the encoding into Interaction Nets. Interaction Nets is a non-hierarchical graph rewriting formalism with strong syntactic conditions, and HyperLMNtal can be considered as a model and a language that extends Interaction Nets by alleviating their syntactic conditions and introducing hyperlinks. Of various encodings into Interaction Nets, Sinot's encoding [18] is one of the simplest in the sense that it dispenses with the explicit management of free variables in each λ-abstraction. However, the method is to compute weak head normal forms (terms of the form $xM_0 \ldots M_n$ $(n \geq 0)$ or $\lambda x.M$, where M and M_i are not necessarily in normal form) and the computation is serialized using a control token navigating over the λ-graph. Our goal, in contrast, is to encode the basic reduction semantics of the pure λ-calculus, preserving and manifesting nondeterminism inherent in the formalism.

5.2 Representing $\boldsymbol{\lambda}$-terms

First of all, we define the encoding from a λ-term L into an LMNtal process. The result must have exactly one free link (say R), which is connected to the atom referring to L. So the translation function $\mathcal{T}$ receives as arguments the λ-term L and the free link name R.

- When L is a variable x, it is represented as a unary atom with the name x connected to R via a binary atom `fv` indicating a free variable:

$$\mathcal{T}(x,R) \stackrel{\text{def}}{=} \mathtt{fv}(x,R) \quad (=\ \ R\,\texttt{=}\,\mathtt{fv}(x)).$$

- When L is a λ-abstraction $\lambda x.M$, let k (≥ 0) be the number of free occurrences of x in M, and $\mathcal{T}_x(M,[R_1,\ldots,R_k],R)$ be a process obtained from $\mathcal{T}(M,R)$ by removing all unary atoms x and their tags `fv` and changing them into free links $R_1,\ldots,R_k$. (For example, $\mathcal{T}_x(x,[R_1],R) =\ \ R\,\texttt{=}\,R_1$.) Then

$$\begin{aligned}\mathcal{T}(\lambda x.M,R) \stackrel{\text{def}}{=} {} & \mathtt{lambda}(R_0,R',R),\ \mathcal{T}_x(M,[R_1,\ldots,R_k],R'),\\ & \mathit{connect}[R_0,R_1,\ldots,R_k]\,,\end{aligned}$$

  where $\mathit{connect}[R_0,R_1,\ldots,R_k]$ is a process with free links R_0, R_1, $\ldots$, R_k defined as follows:

$$\begin{aligned}\mathit{connect}[R_0] &\stackrel{\text{def}}{=} \mathtt{rm}(R_0)\\ \mathit{connect}[R_0,R_1] &\stackrel{\text{def}}{=} R_0\,\texttt{=}\,R_1\\ \mathit{connect}[R_0,R_1,\ldots,R_n] &\stackrel{\text{def}}{=} \mathtt{cp}(R_1,R_0',R_0),\ \mathit{connect}[R_0',R_2,\ldots,R_n] \quad (n\geq 2).\end{aligned}$$

- When L is an application MN:

$$\mathcal{T}(MN,R) \stackrel{\text{def}}{=} \mathtt{apply}(R_1,R_2,R),\ \mathcal{T}(M,R_1),\ \mathcal{T}(N,R_2)\,.$$

Bound variables are encoded into LMNtal links, but because of the Link Condition of LMNtal, bound variables not occurring exactly twice requires the branching or termination of links. We employ a unary atom `rm` (remove) to terminate unused bound variables and a ternary atom `cp` (copy) to bifurcate links. The encoding of a bound variable with more than two occurrences forms a tree of `cp`'s, but the form of the tree does not count for our encoding and its properties. For example, a combinator $\mathbf{I} = \lambda x.x$ is represented as

```
lambda(X,X,Result)     (= Result=lambda(X,X))
```

where `Result` is the free link name representing the result. The Church encodings of natural numbers, $\lambda fx.f^n x$ ($n \geq 0$), can be represented as

```
0: lambda(rm,lambda(X,X),Result)
1: lambda(F,lambda(X,apply(F,X)),Result)
2: lambda(cp(F0,F1),lambda(X,apply(F0,apply(F1,X))),Result)
```

and so on.

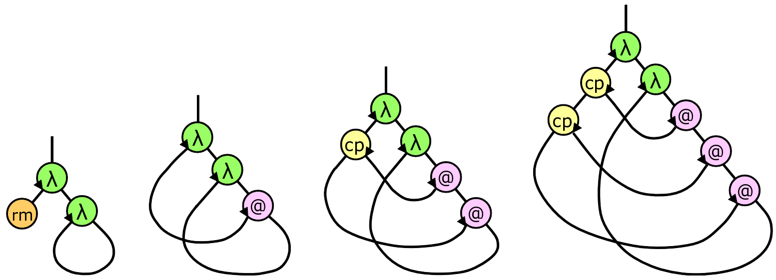

Fig. 8. Graph representation of the Church numerals $0, 1, 2, 3$, where '`@`' stands for `apply`.

5.3 Reaction Rules with Color Management

Figure 9 shows a complete set of rules that encodes the pure λ-calculus using our λ-term representation, in which `cpc` stands for a complement of `cp` and hereafter denoted as $\overline{\texttt{cp}}$. Intuitively, an atom a is said to be a complement of b if they may cancel each other. Likewise, `topc` and `subc` are the complements of `top` and `sub` and will be denoted as $\overline{\texttt{top}}$ and $\overline{\texttt{sub}}$.

The first rule, `beta`, performs "bare" β-reduction, that is, performs parameter passing without copying the argument even when it is referenced more than once. Rule `beta` alone is sufficient if all formal parameters are used exactly once; otherwise we need reaction rules for the atoms `cp`, $\overline{\texttt{cp}}$, `rm`, and $\overline{\texttt{rm}}$ (11 rules following `beta`) to destroy or copy graph structures incrementally. The final four rules are for the color management described next.

The ternary `cp`'s in λ-terms are first converted to quinternary `cp`'s by Rule `c2c`. The additional third and fourth arguments form a pair of complementary circuits for distinguishing between `cp`'s with different origins (i.e., `cp`'s copied from the ones belonging to different λ-abstractions) when copying nested λ-abstractions. The additional information is called a *color* after the Petri Net terminology. Let us focus on the circuit formed by the third arguments and come back to the other circuit formed by the fourth arguments later. Each color is represented using a hyperlink that interconnects all atoms sharing that color.

Colors form tree-shaped partial order. Two colors in the supercolor-subcolor relationship are interconnected by an atom `sub`. The topmost color is connected to the atom `top`. Figure 10 shows a graph structure consisting of one `cp` with a top color, one `cp` and one $\overline{\texttt{cp}}$ with the complementary pair of a subcolor, and one `cp` and two $\overline{\texttt{cp}}$'s with the complementary pair of another subcolor.

Each quinternary `cp` is given a top color initially. Rule `c2c` creates an independent top color cell for each `cp`, but whether to create independent top color cells or share a single top color cell does not affect the correctness of our encoding.

Graph copying starts when beta reduction takes place and a `cp` on the formal parameter side meets `apply`, `lambda`, or `fv` in the argument term. Figure 11 depicts important rewrite rules related to `cp`'s.

```
beta@@ H=apply(lambda(A, B), C) :- H=B, A=C.

l_c@@  lambda(A,B)=cp(C,D,!L,!M) :-
         C=lambda(E,F), D=lambda(G,H),
         A=cpc(E,G,!L1,!M1), B=cp(F,H,!L2,!M),
         sub(!L1,!L2,!L), subc(!M1), .
a_c@@  apply(A,B)=cp(C,D,!L,!M) :-
         C= apply(E,F), D= apply(G,H),
         A=cp(E,G,!L,!M1), B=cp(F,H,!L,!M2), !M=jn(!M1,!M2).

c_c1@@ cpc(A,B,!L1,!M1)=cp(C,D,!L2,!M2), sub(!L1,!L2,!L) :-
         A=C, B=D, sub(!L1,!L2,!L), !L1 >< !M1, !L2 >< !M2.
c_c2@@ cpc(A,B,!L1,!M1)=cp(C,D,!L2,!M2), top(!L2) :-
         C=cpc(E,F,!L1,!M11), D=cpc(G,H,!L1,!M12), !M1=jn(!M11,!M12),
         A=cp(E,G,!L2,!M21), B=cp(F,H,!L2,!M22), !M2=jn(!M21,!M22),
         top(!L2).
f_c@@  fv($u)=cp(A,B,!L,!M) :- unary($u) |
         A=fv($u), B=fv($u), !L >< !M.

l_r@@  lambda(A,B)=rm :- A=rmc, B=rm.
a_r@@  apply(A,B)=rm :- A=rm, B=rm.
c_r1@@ cp(A,B,!L,!M)=rmc :- A=rmc, B=rmc, !L >< !M.
c_r2@@ cpc(A,B,!L,!M)=rm :- A=rm, B=rm, !L >< !M.
r_r@@  rmc=rm :- .
f_r@@  fv($u)=rm :- unary($u) | .

promote@@ subc(!L1), sub(!L1,!L2,!L3) :- !L2 >< !L3.
join@@ !Y=jn(!X,!X) :- !X >< !Y.
c2c@@  A=cp(B,C) :- A=cp(B,C,!L,!M), top(!L), topc(!M).
gc@@   top(!L), topc(!L) :- .
```

Fig. 9. HyperLMNtal encoding of the pure λ-calculus.

When a `cp` meets an `apply`, it copies the partner, splits itself, and proceeds to the copying of the `apply`'s two arguments. In this case, the color of the split `cp`'s remains unchanged (Rule `a_c`).

When a `cp` meets a `lambda`, it copies the partner and splits itself in the same manner, but in this case it turns into a complementary pair of `cp` and $\overline{\texttt{cp}}$. Furthermore, the complementary pair is made to have different colors as described below (Rule `l_c`).

The hyperlink `!L` on the left-hand side of `l_c` stands for the current color. The right-hand side creates a subcolor `!L2` and its complement `!L1`. A $\overline{\texttt{cp}}$ moving anticlockwise (in the representation of Fig. 8) from the x side of a λ-term $\lambda x.M$ and a `cp` moving clockwise from the M side are given the same color held by the first and the second arguments of `sub`, respectively.

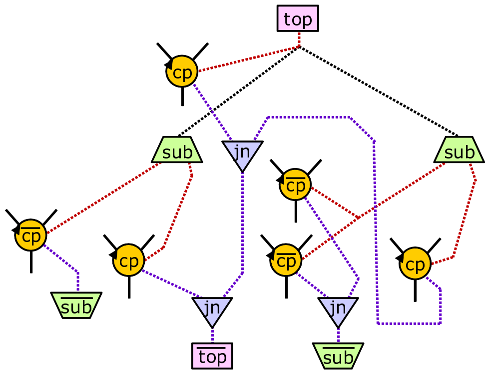

Fig. 10. Coloring `cp` atoms. Non-circular atoms and dotted edges form a circuit for color management.

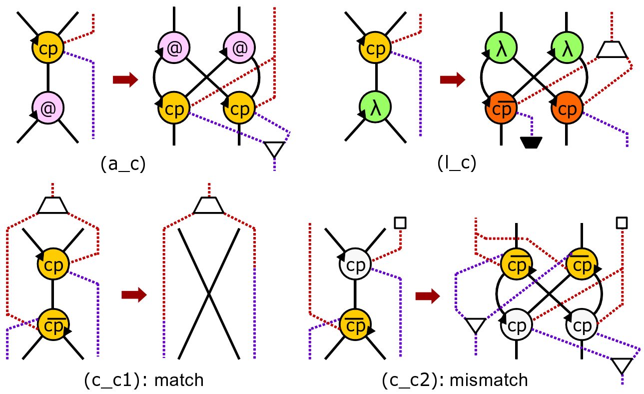

Fig. 11. Reaction rules for `cp` atoms. A white triangle stands for `jn`; a white trapezoid stands for `sub`; a black trapezoid stands for $\overline{\texttt{sub}}$; and a white square stands for `top`.

As can be seen from the Church numerals example, the link representing the bound variable of a λ-abstraction is either terminated by `rm` or is split using zero or more `cp`'s and connected to some places in the body. Accordingly, each $\overline{\texttt{cp}}$ will eventually meet, and is annihilated by, either an `rm` or a `cp` with the complementary color (possibly after crossing and copying `cp`'s with the top color). When a $\overline{\texttt{cp}}$ meets a `cp` with the top color, it copies the partner using `c_c2`, splits itself, and proceeds.

In contrast, a `cp` may not meet a $\overline{\texttt{cp}}$ with the complementary color, because it may *escape* the scope of the λ-abstraction through a link representing nonlocal variables. The color of a `cp` that has escaped must be changed back to the original color. This is done using `promote`, which fuses a subcolor with its supercolor by removing the atom `sub` when all the $\overline{\texttt{cp}}$'s of the subcolor disappear (Fig. 12). This promotion mechanism was realized using membranes in our first encoding [24], where the emptiness checking of membranes was used in an essential way.

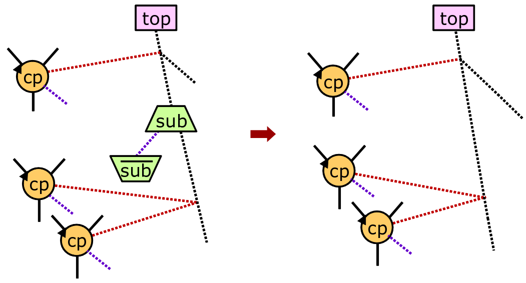

Fig. 12. Color promotion triggered by the short-circuiting of `sub` and $\overline{\texttt{sub}}$.

Based on this experience, a cardinality operator of some sort was considered an indispensable construct for hyperlinks, and was incorporated into HyperLMNtal.

The concurrency research community has been deeply concerned with the choice of primitives which may affect the expressive power of the formalism. Thus, exactly what primitives are needed to encode the lambda calculus is a central rather than marginal issue. The major contribution of the encoding presented in Fig. 9 is that a symmetric hyperlink circuit works nicely for color management and is amenable to capability typing.

Recall that, when a `cp` is annihilated, it is not allowed to discard the color capability carried by its third argument in a typed setting; the `cp` instead returns it through the fourth argument. When a `cp` is copied into two, the capabilities distributed to the two copies through their third arguments are returned through the fourth arguments and are joined by an atom `jn` defined in the rule `join`. The initial `cp`'s carry the top color `top`, and their fourth arguments are connected to $\overline{\texttt{top}}$, the complement of `top`.

A similar mechanism is implemented for the $\overline{\texttt{cp}}$'s sharing the same subcolor; the third arguments of the $\overline{\texttt{cp}}$'s are connected to the first argument of `sub`, while their fourth arguments are connected, possibly via `jn`'s, to the complementary atom $\overline{\texttt{sub}}$. From Fig. 10, one can observe that a hyperlink circuit between a `sub` and a corresponding $\overline{\texttt{sub}}$ form a circuit involving $\overline{\texttt{cp}}$'s and containing `jn`'s on the $\overline{\texttt{sub}}$ side. When the $\overline{\texttt{cp}}$'s are short-circuited, the `jn`'s will detect the identity of two hyperlinks and join their capabilities. The `sub` and the $\overline{\texttt{sub}}$ will establish one-to-one connection eventually, when they annihilate each other and triggers promotion. Similarly, observe from Fig. 10 that a `top` and a corresponding $\overline{\texttt{top}}$ form a circuit involving `cp`'s, which also contains `sub`'s on the `top` side and `jn`'s on the $\overline{\texttt{top}}$ side, which is symmetric if the `sub`'s and the `jn`'s are ignored.

Rule `promote` is applied asynchronously with other rules; it is not necessarily applied as soon as all the $\overline{\texttt{cp}}$'s of some color disappear. The delay of `promote` simply delays the reaction between `cp`'s and $\overline{\texttt{cp}}$'s (using `c_c1` and `c_c2`) and does not cause wrong reactions by affecting the applicability of other rules.

Of the remaining rules, `f_c` copies global free variables. This rule contains a side condition, `unary($u)`, that specifies that the first argument of `fv` is connected to some unary atom, which will be copied in the right-hand side because

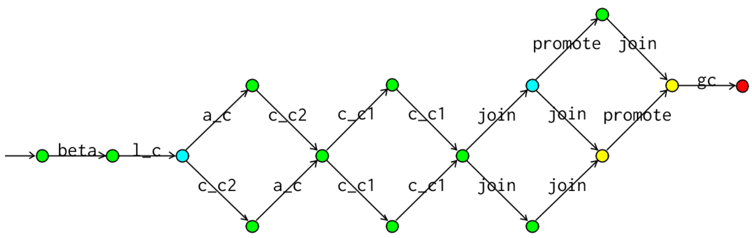

Fig. 13. State space of the omega combinator.

`$u` occurs twice there. Rules `l_r`, `a_r`, `c_r1`, `c_r2`, `r_r`, and `f_r` are to delete any partner that an `rm` or an $\overline{\texttt{rm}}$ may encounter. Rule `gc` is to delete a topmost color not referenced any more.

5.4 Examples

From numerous examples we have run using our LMNtal system, we pick two well-known λ-terms to illustrate our encoding.

Example 9 The omega combinator $\Omega = (\lambda x.xx)(\lambda x.xx)$ is a beautiful λ-term that involves copying. Figure 13 shows the state space of Ω encoded into HyperLMNtal, where `beta` is allowed to be used only once for the purpose of our analysis. Figure 14 shows some of those states. Graph (G1) is the initial state decorated with colors by `c2c`. The only rule applicable to (G1) is `beta`, which yields (G2). Now λ reacts with `cp` and is split into two ((G3)). (G4) is obtained from (G3) by `a_c` and `c_c2` (in either order). Now the two complementary pairs of `cp` and $\overline{\texttt{cp}}$ are canceled by two applications of `c_c1`, and the resulting (disconnected) graph, (G5), consists of Ω (left) and a graph of used colors (right), where the Ω graph came with a color representation different from (G1). The color graph will be erased by `join`, `promote` and `gc`.

When we do not restrict the number of β-reductions, the encoding turns out to have infinitely many states (unlike Ω in the original λ-calculus which has only one state) because the erasure of the garbage graph may be delayed arbitrarily long. □

Example 10 The exponentiation of Church numerals seems to be an important test of λ-calculus encodings because the extremely simple encoding of m^n, $\lambda mn.nm$, involves exponential amount of graph copying. It is important also because it requires the evaluation of the bodies of λ-abstractions.

The program in Fig. 15 reduces to

$$\texttt{R = }\underbrace{\texttt{apply(fv(s),apply(fv(s),...,apply(fv(s)}}_{\text{81 times}}\texttt{,fv(0))...))}$$

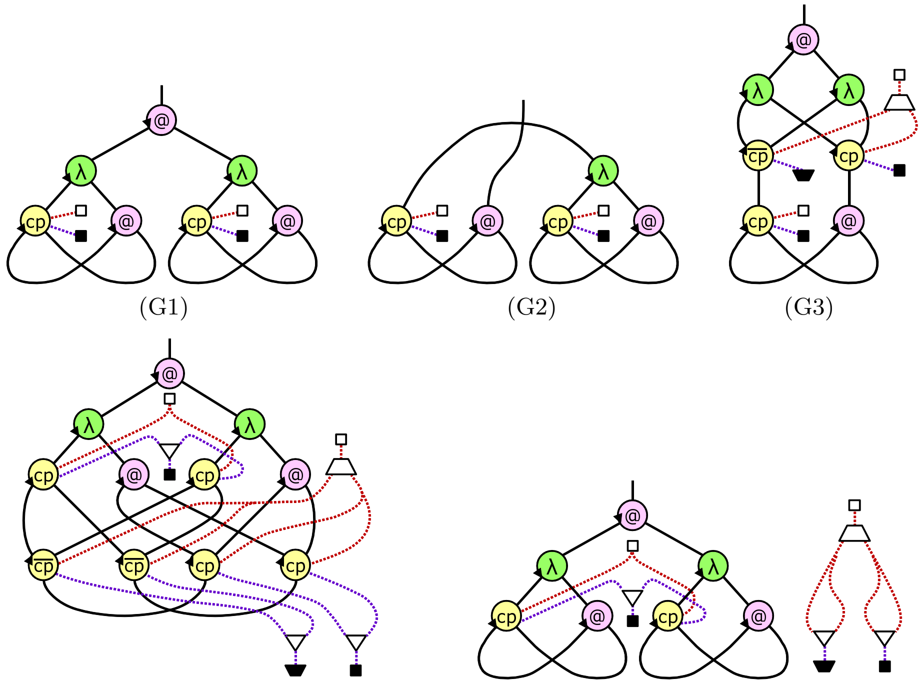

Fig. 14. Reduction of the omega combinator. a white square stands for `top`; a black square stands for $\overline{\texttt{top}}$; a white trapezoid stands for `sub`; a black trapezoid stands for $\overline{\texttt{sub}}$; and a white triangle stands for `jn`.

that stands for $\mathtt{s}^{3^{2^2}}(\mathtt{0})$. The encoding is highly nondeterministic, reflecting the fine-grainedness of the encoding. Even the computation of `R = apply(n(2),n(2))` (2^2) has 2874 possible states. □

5.5 Properties of the Encoding

The encoding described above decomposes β-reduction into many small microsteps that allow asynchronous, out-of-order execution. The adequacy of the encoding is therefore not obvious; recall that the confluence and termination of the $\lambda\sigma$-calculus was not obvious, either [4][13]. Furthermore, because of the asynchrony, the "meaning" of an intermediate state of graph reduction broken into microsteps is far from obvious.

To address the above problems, in [24] we proposed to interpret graphs using λ-terms with additional binder constructs corresponding to `rm` and `cp` atoms, and established several important properties of the (original) encoding through well-known properties of the λ-calculus. Exactly the same technique can be used to establish the properties of the encoding described in this paper because the two encodings differ only in the *representation* of colors that is abstracted in the

```
N=n(2) :- N=lambda(cp(F0,F1),lambda(X,apply(F0,apply(F1,X)))).
N=n(3) :- N=lambda(cp(F0,cp(F1,F2)),
                   lambda(X,apply(F0,apply(F1,apply(F2,X))))).
R = apply(apply(apply(apply(n(2),n(2)),n(3)),fv(s)),fv(0)).
```

Fig. 15. Church numerals and their exponentiation.

extended λ-term representation of graphs. We do not repeat the technical details but mention that the encoding enjoys the following properties, whose proofs can be found in [24]:

1. *Preservation of strong normalization*: If a λ-term M is strongly normalizing, the HyperLMNtal encoding of M is strongly normalizing.
2. *Soundness*: If an HyperLMNtal encoding G of M reduces in 0 or more steps to G' which is an encoding of some term M', then $M \longrightarrow^* M'$.
3. *Completeness*: If G is a HyperLMNtal encoding of M and $M \longrightarrow^* M'$, then there is a HyperLMNtal encoding G' of M' such that G can be reduced to G' in 0 or more steps.

Most previous encodings into Interaction Nets used two kinds (i.e., two colors) of copying tokens. Two colors sufficed in [18] because it did not evaluate bodies of λ-abstractions. YALE and KCLE computed normal forms, but did so by explicitly managing nonlocal variables, which added certain complexity. Although not for computing normal forms, Lang's encoding [10] employed many colors, where colors were represented as sequences of fresh names. Color comparison was based on whether one color was a prefix of the other, whose practical cost is yet to be studied. Lamping's optimal sharing [9] also employed many colors (called *levels*), and further employed tokens called croissants and brackets (both coming with many colors as well) to achieve sharing and complicated level management. Our encoding pursues a different direction: the size of the rewrite system. Color hierarchy implemented using hyperlinks lead to a rewrite system that added only a few rules to the rules for handling all possible pairs of atoms that may meet.

Our rewrite system could be slimmed down further. Rule `c2c` can be dispensed with by starting with colored `cp` atoms. Rules `l_r`, `a_r`, `c_r1`, `r_r`, `f_r` (i.e., all rules involving `rm` and $\overline{\texttt{rm}}$ except `c_r2`), plus `gc` are just for garbage collection and tidying up the tree of `cp`'s, and could be dispensed with. (Rule `c_r2` cannot be removed because it kills $\overline{\texttt{cp}}$'s whose cardinality is counted.) This leaves us only nine essential rules, `beta`, `l_c`, `a_c`, `c_c1`, `c_c2`, `f_c`, `c_r2`, `promote`, and `join`, which suffice for the full evaluation of colored λ-term representation.

5.6 Typing the encoding

The encoding described in this section is designed to allow capability typing. We omit the typing constraints but show a well-typing of atom ports:

$$
\begin{array}{l}
c(\langle \texttt{@}, 1\rangle) = c(\langle \texttt{@}, 2\rangle) = 1, \quad c(\langle \texttt{@}, 3\rangle) = -1,\\
c(\langle \lambda, 1\rangle) = -1, \quad c(\langle \lambda, 2\rangle) = 1, \quad c(\langle \lambda, 3\rangle) = -1,\\
c(\langle \texttt{cp}, 1\rangle) = c(\langle \texttt{cp}, 2\rangle) = -1,\ c(\langle \texttt{cp}, 5\rangle) = 1,\\
c(\langle \texttt{cp}, 3\rangle) > 0, \quad c(\langle \texttt{cp}, 4\rangle) < 0, \quad c(\langle \texttt{cp}, 3\rangle) + c(\langle \texttt{cp}, 4\rangle) = 0,\\
c(\langle \overline{\texttt{cp}}, 1\rangle) = c(\langle \overline{\texttt{cp}}, 2\rangle) = 1, \quad c(\langle \overline{\texttt{cp}}, 5\rangle) = -1,\\
c(\langle \overline{\texttt{cp}}, 3\rangle) > 0, \quad c(\langle \overline{\texttt{cp}}, 4\rangle) > 0, \quad c(\langle \overline{\texttt{cp}}, 3\rangle) + c(\langle \overline{\texttt{cp}}, 4\rangle) = 0,\\
c(\langle \texttt{rm}, 1\rangle) = 1, \quad c(\langle \overline{\texttt{rm}}, 1\rangle) = -1,\\
c(\langle \texttt{jn}, 1\rangle) > 0, \quad c(\langle \texttt{jn}, 2\rangle) > 0, \quad c(\langle \texttt{jn}, 3\rangle) < 0,\\
c(\langle \texttt{jn}, 1\rangle) + c(\langle \texttt{jn}, 2\rangle) + c(\langle \texttt{jn}, 3\rangle) = 0,\\
c(\langle \texttt{sub}, 1\rangle) = -1, \quad c(\langle \texttt{sub}, 2\rangle) < 0, \quad c(\langle \texttt{sub}, 3\rangle) > 0,\\
c(\langle \texttt{sub}, 2\rangle) + c(\langle \texttt{sub}, 3\rangle) = 0,\\
c(\langle \overline{\texttt{sub}}, 1\rangle) = 1, \quad c(\langle \overline{\texttt{top}}, 1\rangle) = 1, \quad c(\langle \texttt{top}, 1\rangle) = -1
\end{array}
$$

Ports constrained by inequalities and zero-sum constraints are polymorphic ports for hyperlinks. The above constraints give us an interpretation that

- `@` works on λ in Rule `beta`,
- `cp` and `rm` work on `@`, λ, $\overline{\texttt{cp}}$, $\overline{\texttt{rm}}$, and `fv` in Rules `a_c`, `l_c`, `c_c1`, `c_c2`, `c_r1`, `f_c`, `a_r`, `l_r`, `c_r2`, `r_r`, and `f_r`,
- $\overline{\texttt{top}}$ works on `top` in Rule `gc`,
- $\overline{\texttt{sub}}$ works on `sub` in Rule `promote`, and
- `jn` works by itself.

Thus, the capability typing provides all atoms (except `jn`) with active or passive rôles in reaction, and this directionality information should be useful in an optimized implementation of our encoding.

6 Conclusion

We have shown that programming with controlled use of links and hyperlinks provides us with a uniform framework of concurrent and non-deterministic computation that allows (among other things) concise and fine-grained encoding of the strong λ-calculus. The encoding shows that the carefully chosen set of hyperlink operations (equality checking and fusing) are powerful enough to express multiset operations necessary to encode the scope management of the λ-calculus.

The simple capability type system with a $[-1, +1]$ real-valued type domain gives interpretation of hyperlinks as backward (directed) hyperlinks. The type constraints are formulated around Kirchhoff's current law, and could be solved rather easily using SAT (for links) or SMT (for hyperlinks) solvers. Although we advocate distinguishing between links and hyperlinks syntactically, the distinction is for practical reasons, since the capability type system is powerful enough to automatically infer which ports and hyperlinks are used as (and can be implemented as) links. The capability type system seems to advocate a symmetric program structure with respect to capability management; that is, hyperlink capabilities split into fractions in a tree-like manner should eventually be joined in the tree-like manner.

Our ongoing work includes the encoding of Bigraphical Reactive Systems into Flat HyperLMNtal, in which hyperlinks are used in a much more sophisticated way. Two major directions of future work are (i) to apply the proposed type system to aggressive compiler optimization and (ii) to develop a verification framework for programs based on hyperlink rewriting.

Acknowledgments

The author is indebted to the present and past members of the LMNtal group for fruitful discussions and building the (Hyper)LMNtal system on which the present work was successfully based. He would like to thank anonymous referees for their careful reviewing and useful comments. This work is partially supported by Grant-In-Aid for Scientific Research ((B) 23300011), JSPS, Japan.

References

1. van der Aalst, W. and Stahl, C.: Modeling Business Processes: A Petri Net-Oriented Approach. The MIT Press, Cambridge, MA (2011)
2. Abadi, M., Cardelli, L, Curien P.-L. and Lévy, J.-J.: Explicit Substitutions. *Journal of Functional Programming*, Vol. 1, No. 4, 375–416 (1991)
3. Boyland, J.: Checking Interference with Fractional Permissions. In: *Proc. Tenth International Symposium on Static Analysis (SAS 2003)*, LNCS 2694, Springer-Verlag, 55–72 (2003)
4. Curien, P.-L., Hardin, T. and Lévy, J.-J.: Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *J. ACM*, Vol. 43, No. 2, 362–397 (1996)
5. Finkel, A: The Minimal Coverability Graph for Petri Nets. In: *Papers from the 12th International Conference on Applications and Theory of Petri Nets (APN'91)*, LNCS 674, Springer-Verlag, 210–243 (1993)
6. Frühwirth, T.: *Constraint Handling Rules.* Cambridge University Press, Cambridge, UK (2009)
7. Gallo, G., Longo, G., Pallottino, S. and Nguyen, S.: Directed Hypergraphs and Applications. *Discrete Applied Mathematics*, Vol. 42, No. 2–3, 177 - 201 (1993)
8. Lafont, Y.: Interaction Nets. In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM, 95–108 (1990)
9. Lamping, J.: An Algorithm for Optimal Lambda-Calculus Reductions. In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM, 16–30 (1990)
10. Lang, F.: *Modèles de la β-réduction pour les implantations.* Ph.D. Thesis, Ècole Normale Supérieure de Lyon (1998)
11. Mackie, I.: YALE: Yet Another Lambda Evaluator Based on Interaction Nets. In: *Proc. Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM, 117–128 (1998)
12. Mackie, I.: Efficient λ-Evaluation with Interaction Nets, In: *Proc. Proc. 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, LNCS 3091, Springer-Verlag, 155–169 (2004)
13. Melliès, P.-A.: Typed λ-Calculi with Explicit Substitutions May Not Terminate. In *Proc. TCLA'95*, LNCS 902, Springer-Verlag, 1995, 328–334

14. Milner, R.: *The Space and Motion of Communicating Agents*. Cambridge University Press, Cambridge, UK (2009)
15. Reynier, P.-A. and Servais, F.: Minimal Coverability Set for Petri Nets: Karp and Miller Algorithm with Pruning. In: *Proc. 32nd International Conference on Application and Theory of Petri Nets and Concurrency (ICATPN'11)*, LNCS 6709, Springer-Verlag, 69–88 (2011)
16. Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E.: The Fifth Generation Project: Personal Perspectives. Comm. ACM, Vol. 36, No. 3, 46–103 (1993). (This is actually a collection of single-authored articles, and my article (pp. 65–76) was originally titled "Kernel Language in the Fifth Generation Computer Project.")
17. Shapiro, E. and Takeuchi, A.: Object oriented programming in Concurrent Prolog. New Generation Computing, Vol. 1, No. 1, 25-48 (1983).
18. Sinot, F.-R.: Call-by-Name and Call-by-Value as Token-Passing Interaction Nets. In: *Proc. Seventh International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, LNCS 3461, Springer-Verlag, 386–400 (2005)
19. Suenaga, K. and Kobayashi, N.: Fractional Ownerships for Safe Memory Deallocation. In: *Proc. 7th ASIAN Symposium on Programming Languages and Systems (APLAS 2009)*, LNCS 5904, 128–143 (2009)
20. Ueda, K. and Morita, M.: Moded Flat GHC and Its Message-Oriented Implementation Technique. New Generation Computing, Vol. 13, No. 1, 3–43 (1994)
21. Ueda, K.: Resource-Passing Concurrent Programming. In: *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software* (TACS'01), LNCS 2215, Springer-Verlag, 95–126 (2001)
22. Ueda, K. and Kato, N.: LMNtal: A Language Model with Links and Membranes. In: *Proc. Fifth International Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer-Verlag, 110–125 (2005)
23. Ueda, K.: LMNtal as a Hierarchical Logic Programming Language. *Theoretical Computer Science*, Vol. 410, No. 46, 4784–4800 (2009)
24. Ueda, K.: Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting. In: *Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, LNCS 5117, Springer-Verlag, 392–408 (2008)
25. Ueda, K., Ayano, T., Hori, T., Iwasawa H. and Ogawa, S.: Hierarchical Graph Rewriting as a Unifying Tool for Analyzing and Understanding Nondeterministic Systems. In: *Proc. Sixth International Colloquium on Theoretical Aspects of Computing (ICTAC 2009)*, LNCS 5684, Springer-Verlag, 349–355 (2009)
26. Ueda, K. and Ogawa, S.: HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model. Künstliche Intelligenz, Vol. 26, No. 1, 27–36 (2012). DOI: 10.1007/s13218-011-0162-3
27. Ueda, K., Encoding Distributed Process Calculi into LMNtal. *Electronic Notes in Theoretical Computer Science*, Vol. 209 (2008), 187–200.