Graduation Thesis

Implementing G-Machine in HyperLMNtal

HyperLMNtal を用いた G-Machine の実装

Submition Date: January 22, 2021

Supervisor: Prof. Dr. Kazunori Ueda

Waseda University
Computer Science and Engennering

Student ID: 1W172146-1

Jin Sano

Abstract

Since language processing systems generally allocate/discard memory with complex reference relationships, including circular and indirect references, their implementation is often not trivial. Here, the allocated memory and the references can be abstracted to the labeled vertices and edges of a graph. And there exists a *graph rewriting language*, a programming language or a calculation model that can handle graph intuitively, safely and efficiently. Therefore, the implementation of a language processing system can be highly expected as an application field of graph rewriting language. To show this, in this research, we implemented *G-machine*, the virtual machine for lazy evaluation, in hypergraph rewriting language, *HyperLMNtal*.

HyperLMNtal is extended from the graph rewriting language/calculus model *LMNtal*. However, it lacked the rigid definition: it was more an extension of the implementation than on the calculus model. The semantics of LMNtal features fine-grained concurrency based on local rewriting. However, since we could not determine the locality of the hyperlink in HyperLMNtal, we couldn't incorporate it into the LMNtal semantics. Thus, we first introduced a scope (*link creation*) and defined the locality of a hyperlink to formalize the syntax/semantics. Now, HyperLMNtal is not just a programming language extended from the basic calculation model, but also a concurrent calculation model based on strict and formal definitions.

G-machine is a virtual machine that performs lazy evaluation, which is the basis of implementation of lazy functional programming languages such as Haskell. The implementation of G-machine requires a heap, which is a more general graph than just a tree so as can share subgraphs. Therefore, HyperLMNtal is ideal to implement this. In our research, we implemented a compiler which translates the source language, the core language, into the execution code for G-machine and G-machine, in HyperLMNtal. We have succeeded to implement the compiler in 404 lines and G-machine in 570 lines and showed that we can implement the language processing system that handles complex data structures in graph rewriting language tersely. In addition, we achieved to visualize G-machine using the HyperLMNtal visualizer.

概要

言語処理系は一般的に，動的なメモリの確保及び破棄を行い，循環参照や間接参照を含む複雑な参照関係にあるデータを扱うため，その実装は自明ではないことが多い．ここで，メモリ領域及びその参照はグラフ理論におけるグラフのラベル（データ）付き頂点と辺に抽象化できる．また，グラフを直感的に安全に効率良く扱うことができるプログラミング言語または計算モデルとして，グラフ書き換え言語がある．従って言語処理系の実装は，グラフ書き換え言語の応用分野として高い期待が持てるのではないかと考えた．そこで，本研究では，ハイパーグラフ書き換え言語；HyperLMNtal を用いて，遅延評価を行う仮想機械；G-machine の実装を行った．

HyperLMNtal はグラフ書き換え言語であり計算モデルである LMNtal の拡張である．ただし，これは実装上の拡張であり，厳密な意味論が存在していなかった．LMNtal の意味論上の特徴は局所的な書き換えに基づく細粒度の並行性だが, HyperLMNtal のハイパーリンクは局所性が判別できないため，そのままでは LMNtal の意味論に自然に組み込むことはできなかった．そこで，まず，スコープ (link creation) を導入し，ハイパーリンクに局所性を定義することで，意味論を厳密に定めた．これによって，HyperLMNtal は，基本計算モデルから拡張された単なるプログラミング言語であるだけではなく，それ自体が，厳密で形式的な定義に基づいた並行計算モデルとなった．

G-machine は Haskell などのプログラミング言語の処理系の実装の基盤となっている遅延評価を実行する仮想機械である．G-machine は，サブグラフの共有が行えるような，木に限らない，より一般のグラフであるヒープを必要とする．そこで， HyperLMNtal を用いて，G-machine，及び関数型言語から G-machine の実行用コードを得るコンパイラを作成した．結果として，G-machine は 404 行，コンパイラは 570 行で実装でき，グラフ書き換え言語で複雑なデータ構造を扱う言語処理系の実装が非常に簡潔にできることがわかった．また，HyperLMNtal のビジュアル化機能を用いて，G-machine の挙動を可視化できた．

Contents

List of Figures

Chapter 1.

Introduction

1.1. Background of the research

1.1.1. Implementation of language processing system

Language processing systems generally allocate/discard memory with complex reference relationships, including circular and indirect references. Thus, their implementation is often not trivial. Here, the allocated memory and the references can be abstracted to the labeled vertices and edges of a graph. That is, language processing systems can be regarded to be dealing a graph throughout their execution steps. To implement a compiler, functional languages are often used. However, functional languages cannot handle general graphs other than trees[1]. And there exists a cost in time and space, to achieve a closure (e.g. we need a garbage collector). Thus, it is not very suitable especially to implement a runtime environment. On the other hand, *graph rewriting language*[2] is a programming language or a calculation model that can handle graph intuitively, safely and efficiently. Therefore, the implementation of a language processing system can be highly expected as an application field of graph rewriting language. To show this, in this research, we implemented the *G-machine*[3], the virtual machine for lazy evaluation, in hypergraph rewriting language, *HyperLMNtal*[4].

1.1.2. Hypergraph rewriting language: HyperLMNtal

HyperLMNtal is a extension of the graph rewriting language/calculus model LMNtal[5]. A graph in LMNtal is consisted of atoms, labeled vertices/data, possibly connected by links, the edges that connects precisely two endpoints. Even though achieving to let graph to be its 1st class citizen, LMNtal is pointer safe; we won't face any null or dangling pointers when we are using LMNtal. In short, we can handle graph safely and easily using LMNtal.

However, since links in LMNtal are only allowed to have exactly two endpoints, it is rather hard to implement a shared data whose number of references changes dynamically. Although it is possible, it was not very suitable from both the viewpoint of programming and the efficiency of the implementation. On the other hand, HyperLMNtal allows exis-

[1] We can *emulate* graph using a tree with keys. However, we must pay at least $\mathcal{O}(\log N)$ to traverse an edge. In LMNtal (a graph rewriting language) on the other hand, we can achieve this in $\mathcal{O}(1)$.

tence of the hyperlink, which is a link but allowed to have arbitrary number of endpoints. Thus, we can easily implement a shared data with HyperLMNtal.

1.2. Previous relevant researches

This section gives brief summary of the previous relevant researches.

1.2.1. Translating HIRAM into GP2

Detlef Plump has shown that the simple imperative language, Hi-Level Random Access Machine (from now on, we abbreviate this as HIRAM), can be translated into the rule-based graph rewriting language, GP2[6] in [7]. The syntax of the HIRAM is shown in Figure 1.1[2].

Where a denotes an address (the index of the register) ($a \in \mathbb{N} \cup \{0\}$), num denotes a integer numeral ($num \in \mathbb{Z}$) and the σ (s in the original paper) denotes the states of the registers as follows:

$$\left\{ \begin{array}{ccc} 0 & \mapsto & list_0 \\ & \vdots & \\ n-1 & \mapsto & list_{n-1} \\ n & \mapsto & list_n \\ n+1 & \mapsto & \texttt{empty} \\ n+2 & \mapsto & \texttt{empty} \\ n+3 & \mapsto & \texttt{empty} \\ & \vdots & \end{array} \right\}$$

where n is a finite number[3]. A bit more precisely, it is a function from the address to the value (list) stored at the address. Notice that a number is also a list of length 1 in HIRAM. Thus $a \subseteq num \subseteq list$ and the registers in HIRAM are also allowed to store addresses, a, to perform pointer manipulations. $\sigma[list/a]$ denotes the updated state defined as follows:

$$\sigma[list/a_1](a_2) \stackrel{def}{=} \left\{ \begin{array}{ll} list & \text{if } a_1 = a_2 \\ \sigma(a_2) & \text{if } a_1 \neq a_2 \end{array} \right.$$

HIRAM has a loop instruction and is Turing complete. Moreover, it features a list, a structured data. And in that sense, it maybe at a higher level. However, it even lacks a function/procedure calls and is certainly not a modern programming language but sort of an assembly language: it is just a list of instructions for a random access machine.

[2]This is mostly based on the [7] but we changed some notations
[3]This is a simplified definition. Please take a look at [7] for more detailed/formal definition

(Program)	*prog*	::= *prog*`;`*prog*	Sequential composition
		| `if` *b* `then` *prog* `else` *prog*	Branching
		| `while` *b* `do` *prog*	Loop
		| *a* `:=` `$`*list*	$\sigma[list/a]$
		| a_1 `:=` a_2	$\sigma[\sigma(a_2)/a_1]$
		| a_1 `:=` `head` a_2	$\sigma[head(\sigma(a_2))/a_1]$ (fails if $\sigma(a_2) =$ `empty`)
		| a_1 `:=` `tail` a_2	$\sigma[tail(\sigma(a_2))/a_1]$ (fails if $\sigma(a_2) =$ `empty`)
		| a_1 `:=` a_2`:`a_3	$\sigma[\sigma(a_2):\sigma(a_3)/a_1]$
		| a_1 `:=` `*`a_2	$\sigma[\sigma(\sigma(a_2))/a_1]$
		| `*`a_1 `:=` a_2	$\sigma[\sigma(a_2)/\sigma(a_1)]$
		| a_1 `:=` `inc` a_2	$\sigma[\sigma(a_2)+1/a_1]$
		| a_1 `:=` `dec` a_2	$\sigma[\sigma(a_2)-1/a_1]$
(Condition)	*b*	::= a_1 `=` a_2	True if $\sigma(a_1) = \sigma(a_2)$; false otherwise
		| a_1 `>` a_2	True if $\sigma(a_1), \sigma(a_2) \in \mathbb{Z}$ and $\sigma(a_1) > \sigma(a_2)$; false otherwise
(list)	*list*	::= `empty`	Empty list
		| *num*	Integers are lists of length 1
		| *list*`:`*list*	Concatenation

Figure 1.1: Syntax of HIRAM programs

(Program)	*prog*	::= def_1 ; ... ; def_n ; *expr*	$n \geq 0$	
(Function Definition)	*def*	::= `def` *var*`(`$var_1, \ldots, var_n$`)` `=` *expr*	$n \geq 0$	
(Expression)	*expr*	::= *num*		Integer numeral
		| *var*		Variable
		| *var*`(`$expr_1, \ldots, expr_n$`)`	$n \geq 0$	Function call
		| `var` *var* `=` *expr* `in` *expr*		Local definition
		| *expr binop expr*		Infix binary application
		| `if` *expr* `then` *expr* `else` *expr*		If expression
(Binary Operators)	*binop*	::= *arithop* | *relop*		
	arithop	::= `+` | `-` | `*` | `/`		Arithmetic
	relop	::= `=` | `<` | `>`		Comparison

Figure 1.2: Syntax of the language implemented in [1]

1.2.2. Implementing a stack machine and a compiler in LMNtal

Kokubo has implemented a stack machine and a compiler in graph rewriting language, LMNtal[1]. The syntax of the source language in this research is given in Figure 1.2[4]. Now, the source language has function definitions/calls: it is certainly a programming language. However, it lacks the structured data (e.g. list). And moreover, it has not accomplished the *higher order function*. Functions in the language is not the 1st class citizen. That is, we cannot apply nor return any function.

In short, the source languages in these previous researches are not functional language.

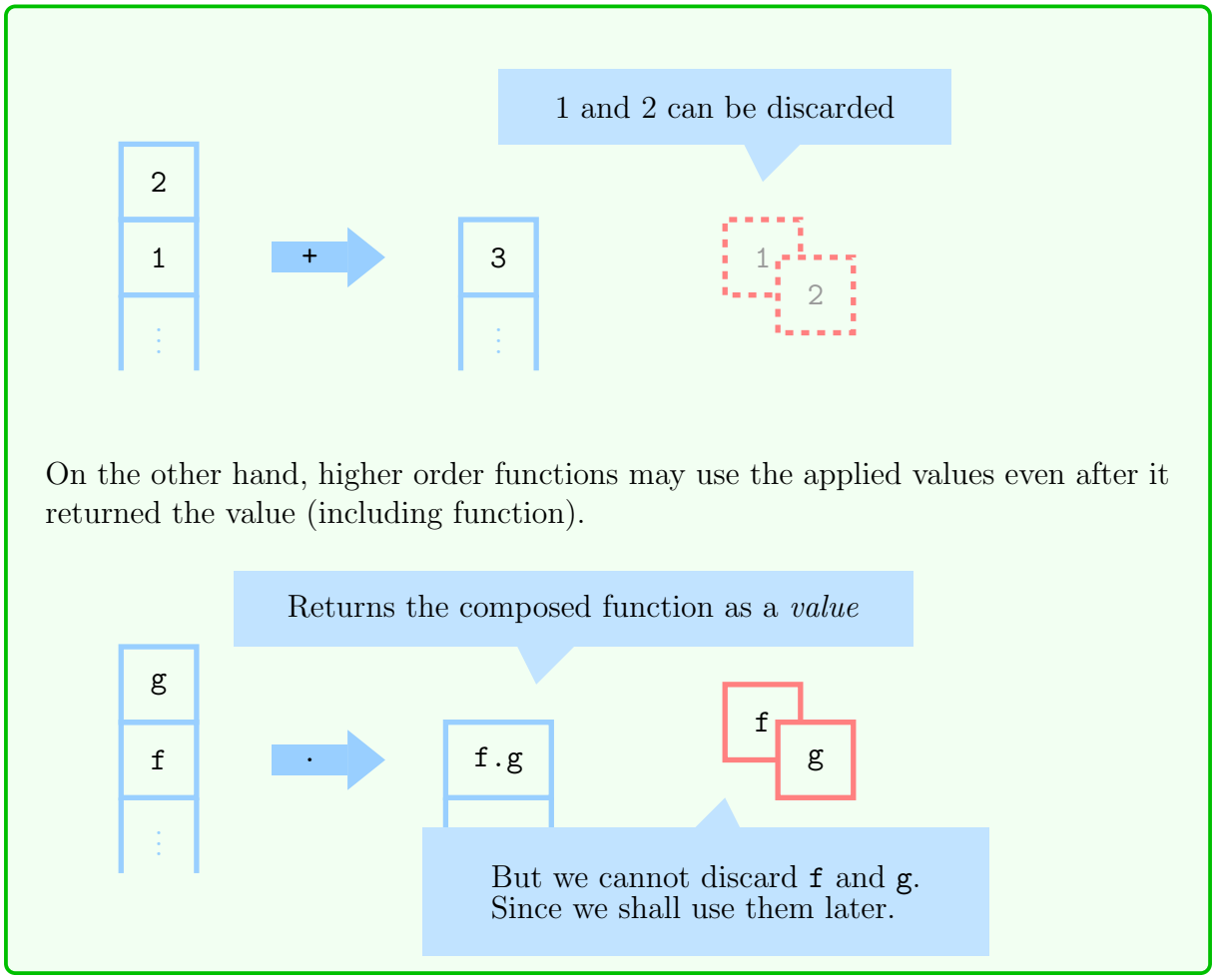
1.2.3. Graph reduction

Functional languages feature a *higher order function* and thus they are certainly powerful than the imperative languages. However, because we have to accomplish this, we cannot just use 1 simple stack to implement them. Rather, we need a *graph*. This problem is known as the *funarg problem*[8].

Example 1.1: Funarg problem

We can discard the argument of the function if the function has returned its value in the stack of the (call by value) imperative language.

[4] Again, we have changed some notations. Since the syntax in the original paper seems to be using some notations incorrectly.



A strict (call by value) functional language can possibly be implemented with just trees, although it would be a very naïve implementation. However, especially for the lazy (call by need) functional language, we need a more general graph more than trees so that the evaluator can share subgraphs (subexpressions). This, *graph reduction/graph rewriting*, is a very popular research area and there is a lot of studies as [9]. However, there was no attempt to implement a compiler and a runtime environment for a (lazy) functional language in a graph rewriting language. Therefore, in this research, we implemented G-machine[3], the virtual machine for lazy evaluation, in hypergraph rewriting language, HyperLMNtal[4].

1.3. Contributions

The contributions of this research are basically the following 2:

- We formalized the syntax and the semantics of HyperLMNtal
- We implemented G-machine and a compiler for it in HyperLMNtal

1.3.1. Formalizing the syntax and the semantics of HyperLMNtal

HyperLMNtal is extended from the graph rewriting language/calculus model LMNtal. However, it lacked the rigid definition: it was more an extension of the implementation than on the calculus model. The semantics of LMNtal features fine-grained concurrency based on local rewriting. However, since we can not determine the locality of the hyperlink in HyperLMNtal, we couldn't incorporate it into the LMNtal semantics. Thus, we first introduced a scope (link creation) adopted from the π-calculus[10] and defined the locality of a hyperlink to formalize the syntax/semantics. Now, HyperLMNtal is not just a programming language extended from the basic calculation model, but also a concurrent calculation model based on strict and formal definitions.

1.3.2. Implementing G-machine in HyperLMNtal

The G-machine is a virtual machine that performs lazy evaluation, which is the basis of implementation of lazy functional programming languages such as Haskell. The implementation of G-machine requires a heap, which is a more general graph than just a tree so as can share subgraphs. Therefore, HyperLMNtal is ideal to implement this.

In our research, we implemented a compiler which translates the source language, the core language, into the execution code for G-machine and G-machine, in HyperLMNtal. We have succeeded to implement the compiler in 404 lines and G-machine in 570 lines and showed that we can implement the language processing system that handles complex data structures in graph rewriting language tersely. In addition, we achieved to visualize G-machine using the HyperLMNtal visualizer[11][12].

1.4. Structure of this paper

The structure of this paper is as the following:

Chapter 2
: We introduce the hypergraph rewriting language HyperLMNtal.

Chapter 3
: We introduce the new formal syntax and semantics of HyperLMNtal.

Chapter 4
: We introduce the design of G-machine.

Chapter 5
: We introduce the G-machine we have implemented in HyperLMNtal and some examples that run on the G-machine.

Chapter 6
: We give some discussions about HyperLMNtal as a programming language.

Chapter 7

We give a conclusion of this paper and discuss the future task.

Chapter 2.

HyperLMNtal: An introduction

HyperLMNtal[4] is a hypergraph rewriting language, which is an extension of the graph rewriting language LMNtal. In this chapter, we first briefly introduce the core syntax and the semantics of the graph rewriting language LMNtal[13, 14] in Section 2.1 and Section 2.2 respectively and its extension run on the current processing system SLIM[15] in Section 2.3. And then explain the current implementation of HyperLMNtal in Section 2.4.

2.1. Syntax of LMNtal

LMNtal is comprised of two kinds of identifiers: *Link names*, denoted by X, which can be identifiers starting with capital letters in the concrete syntax and *atom names*, denoted by p, which can be identifiers that are distinct with link names in the concrete syntax (i.e. identifiers starting with lower letters, numbers, special symbols, etc). The only reserved atom name is "=", which is the name for a *connector* of links.

The syntax of LMNtal is given in Figure 2.1. Intuitively, a *process* is the LMNtal program it self. Surprisingly there is no such thing as *function/procedure calls*: the calculation process proceeds with the *matching* of a sub-graph (process) with the *process templates* on left-hand side (LHS/Head) of a *rule* and pushing the sub-graph based on process templates (and some substitution rules θ described in [13]) on right-hand side (RHS/Body) of the matched rule. The curly braces describe the grouping of the processes, which can compose hierarchies. However, we can think of a *Flat LMNtal* that does not feature this by omitting those with daggers(†). *Process context* and *rule context* represent the rest process/rules of a membrane. *Residual* is the free links of a process context.

Link names outside rules should occur at most twice: the endpoints of a link should be not more than two. If a link name occurs once in a process, then it represents a *free link* of the process. And if a link name occurs twice in a process, then it represents a *local link* of the process. As like the other calculus models, local links can be α-converted: the name of the links convey no *data* at all. As above, we can distinguish local and free links (c.f. local variable and free variable in other calculus models such as λ calculus or π calculus) just by counting the number of the occurrences of link names without some notations like λ or ν.

Link names in left/right-hand sides of a rule must occur twice on left/right-hand side or occur once on each of the hands sides of a rule. That is, the free links of the left and the right-hand sides of a rule must be the same. Thus rewriting of the process won't

(Process)	P	::=	$\mathbf{0}$		Null
		|	$p(X_1, \ldots, X_m)$	$m \geq 0$	Atom
		|	(P, P)		Molecule
		|	$\{P\}$		Cell †
		|	$(T :\!\!- T)$		Rule
(Process Template)	T	::=	$\mathbf{0}$		Null
		|	$p(X_1, \ldots, X_m)$	$m \geq 0$	Atom
		|	(T, T)		Molecule
		|	$\{T\}$		Cell †
		|	$(T :\!\!- T)$		Rule
		|	$\texttt{@}p$		Rule context †
		|	$\texttt{\$}p\,[X_1, \ldots, X_m \| A]$	$m \geq 0$	Process context †
(Residual)	A	::=	$[\,]$		Empty †
		|	$*X$		Bundle †

Figure 2.1: Syntax of LMNtal

delete/yield free links. This is also a notable point; if we had a link that has exactly two endpoints, then the link should always store the property, and there is no chance that a *null/dangling* pointer (link) could occur.

2.1.1. Abbreviations

1. `()` (the parenthesis of the nullary atom) and `[]` can be omitted.
2. $p(s_1, \ldots, s_m), q(t_1, \ldots, t_n)$ can be abbreviated as $q(t_1, \ldots, p(s_1, \ldots, s_{m-1}), \ldots, t_n))$ if the s_m and the t_i has the same link name.

Also, we define the operator precedence as "." < ":-" < "," where period is just an another form of a comma but with lower connectivity. We can omit parentheses if there is no syntactic ambiguities.

2.2. Operational Semantics of LMNtal

We first describe the structural congruence ($\equiv$) rules of a process (i.e. the definition of the equivalence of the process) and the reduction relation ($\longrightarrow$) (i.e. the calculation step).

2.2.1. Structural congruence

The relation $\equiv$ on processes is defined as the minimal equivalence relation which satisfies the rules in Figure 2.2, where $P[Y/X]$ denotes a *link substitution*: substitution of X with Y in P.

(E1)–(E3) let molecules to be multisets. (E4) is a rule for the α-conversion of local link names. Though we have to choose the new link name Y, we don't have to worry about *variable capturing* in a scope with λ or ν. Again, this gives a notable simplicity to LMNtal rather than other calculus models. (E5)–(E6) makes $\equiv$ a congruence; the smaller parts should always be equivalent if the comprised processes are equivalent. (E7)–(E10) are the rules for connectors.

We prove that (E8) is admissible, in some sense, it is redundant, in Theorem 2.1.

Theorem 2.1: Symmetry of $=$

$$X = Y \equiv Y = X$$

Proof 2.1

$$\begin{aligned} & Z = X, Z = Y \\ \equiv_{\text{E9}} & X = Y && \because (Z = Y)[X/Z] = (X = Y) \end{aligned}$$

and

$$Z = X, Z = Y$$

(E1)	$\mathbf{0}, P \equiv P$	
(E2)	$P, Q \equiv Q, P$	
(E3)	$P, (Q, R) \equiv (P, Q), R$	
(E4)	$P \equiv P[Y/X]$	if X if a local link of P
(E5)	$P \equiv P' \Rightarrow P, Q \equiv P', Q$	
(E6)	$P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$	
(E7)	$X = X \equiv \mathbf{0}$	
(E8)	$X = Y \equiv Y = X$	
(E9)	$X = Y, P \equiv P[Y/X]$	if P is an atom and X occurs free in P
(E10)	$\{X = Y, P\} \equiv X = Y, \{P\}$	if exactly one of X and Y occurs free in P

Figure 2.2: Structural congruence on LMNtal processes

$$\equiv_{\mathrm{E2}} Z = Y, Z = X$$
$$\equiv_{\mathrm{E9}} Y = X \qquad \because (Z = X)[Y/Z] = (Y = X)$$

Therefore, $X = Y \equiv Y = X$.

2.2.2. Reduction relation

The reduction relation $\longrightarrow$ (calculation step) on processes are defined as the minimal relation which satisfies the rules given in Figure 2.3.

(R1) states that the reductions can proceed in a local process (besides the other processes together compromise a molecule). This characterize LMNtal as a *concurrent* language. (R2) states that the reductions in a cell (membrane) can proceed by it own, even if there is no help from outer processes. (R3) introduce the structural congruence described before. A matching of a process and a LHS of a rule will be done with changing the process (not LHS of a rule) using the congruence rules. (R4) and (R5) are used when extracting/entering connectors from/in membranes. (R6) is the most important rule. The detailed explanation of the meaning of the θ is given in [13] and We won't explain this fully: in short, it describes the matching of process/rule contexts; the matching of the rest process/rules.

(R1) $$\frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q}$$

(R2) $$\frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}}$$

(R3) $$\frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

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

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

(R6) $T\theta, (T \vdash U) \longrightarrow U\theta, (T \vdash U)$

Figure 2.3: Reduction relation on LMNtal processes

`ground($p)`	checks that a process `$p` constituting a connected sub-graph that has one or more free links
`unary($p)`	checks that a process `$p` is an atom with one link
`int($p)`	checks that a process `$p` is an atom with one link whose name is an integer
`string($p)`	checks that a process `$p` is an atom with one link whose name is an string

Figure 2.4: Primitive types in LMNtal

2.3. Extensions of LMNtal

As like the other programming languages originated from calculus models (e.g. LISP is based on λ-calculus), LMNtal has some extensions for practical programming.

Rules can be denoted with their names as "*rulename* @@ *H* :- *B*". Rules can have *guards* as "*Head* :- *Guard* | *Body*". The features of guard are for (i) type checking (as shown in Figure 2.4) (ii) comparing two processes (as shown in Figure 2.6) and (iii) performing some primitive instructions (as shown in Figure 2.5). Typed process can be copied and removed.

2.4. The implementation of HyperLMNtal

HyperLMNtal[4] is an extension of LMNtal. HyperLMNtal allows the existence of the *hyper links*, which can be connected to arbitrary number of ports (endpoints). For ex-

`$p + $q`	addition
`$p - $q`	subtraction
`$p * $q`	multiplication
`$p / $q`	division

Figure 2.5: Primitive operators in LMNtal

ground	unary	int	meanings
`$p = $q`	`$p == $q`	`$p =:= $q`	The structures of `$p` and `$q` are the same
`$p \= $q`	`$p \== $q`	`$p =\= $q`	The structures of `$p` and `$q` are not the same
-	-	`$p < $q`	`$p` is less than `$q`
-	-	`$p > $q`	`$p` is greater than `$q`
-	-	`$p =< $q`	`$p` is less than or equals to `$q`
-	-	`$p >= $q`	`$p` is greater than or equals to `$q`

Figure 2.6: Comparison operators in LMNtal

ample, they can be used to represent a connection that the number of endpoints changes dynamically through the calculation process.

2.4.1. The syntax and the manipulation of hyperlinks in the current implementation

We briefly explain the newly added syntax and the manipulations for hyperlinks.

Hyperlinks are expressed as "`$x`" (i.e. an extension of the process context) or as "$!X$".

`new` guard construct creates a new hyperlink with a fresh local id. In the following example, the new hyperlink `$x` is created by `new`.

$$H \texttt{ :- new(\$x)} \mid B$$

Using this, all hyperlinks should have distinct names so that they are distinguishable. Current compiler implementation allows users to write a new hyperlink with exclamation mark on RHS without explicitly writing `new`. At that situation, compiler will automatically add it.

`hlink` guard construct checks whether the given link is a hyperlink.

$$H \texttt{ :- hlink(\$x)} \mid B$$

`><` on RHS of a rule is called a *fusion*, which perform the merge of the given two hyperlinks.

$$H \texttt{ :- !X} >< \texttt{!Y}, B$$

`num` guard construct binds the number of the endpoints of the given hyperlink at the first argument to the second argument.

$$H \texttt{ :- num(!X, \$n)} \mid B$$

Hyperlinks on LHS with the same name should match the hyperlinks with the same id. Notice the hyperlinks on LHS with different names can also match the same hyperlinks (i.e. non-injective matching). For example,

```
init.
init :- new($x) | a($x), b($x).
a(!X), b(!Y) :- c.
```

will be reduced to `c` and the remaining rules.

Our current implementation cannot handle hyperlinks in the initial state. Therefore Hyperlinks must not appear outside of rules. Also, our current implementation does not allow to connect hyperlinks with a connector (=).

Chapter 3.

Formalizing HyperLMNtal

We have implemented HyperLMNtal in spite of the fact that the hyperlinks can be (inaccurately) modeled with membranes for (i) the efficiency and (i.e. using membrane costs too much when we just want to represent a multi-connected link) (ii) the convenience for the programmer (i.e. hyperlinks are much easier to read/write rather than membranes). Therefore, in the past, we have not focused on its semantics that much: HyperLMNtal was a more an extension of the implementation rather than a calculus model.

Though the implementation of HyperLMNtal is rather simple and the concept can be easily understood, we found that the semantics cannot be defined easily. We will firstly briefly discuss the difficulty in the semantics in Section 3.1. And then introduce the new semantics which we propose in Section 3.2, Section 3.3 and Section 3.4.

Section 3.2 and Section 3.3 describe the syntax and the semantics of Flat HyperLMNtal. This Flat HyperLMNtal has no normal links but hyperlinks. This is just because we want to discuss the properties of hyperlinks. In Section 3.4, we extend it so that it will be able to deal with membranes, process/rule contexts, etc.

3.1. Introduction to the formalization of HyperLMNtal

3.1.1. The difficulty in HyperLMNtal semantics

The links in LMNtal are only allowed to appear once or twice. This enables us to distinguish the former to be the local link and the latter to be the free link: the locality of the normal link can be determined by just counting the number of the appearance. However, since hyperlinks can appear more than twice, it is difficult to determine whether the given link is a local link or a free link in a process.

If we cannot ensure the locality of the link, we cannot ensure the connector (fusion) has ended the link substitution by looking the local process.

Example 3.1: The difficulty in defining fusion

For example,

$$(a(!X), (!X \bowtie !Y, b(!X, !Y)))$$

should be congruent with

$$(a(!X), b(!X, !X))$$

but

$$(a(!X), b(!Y, !Y))$$

Therefore, we cannot just define the structural congruence rule as

$$!X \bowtie !Y, P \equiv P[!Y/!X] \text{ if } !X \text{ occurs free in } P$$

We must ensure that there is no $!X$ occurs outside of the P. That is, $!X$ should be the local link of "$!X \bowtie !Y, P$".
In (not-hyper) LMNtal, a link appeared twice is a local link, therefore the condition of (E9), "*if X occurs free in P*", requires the X to be a local link in "$X = Y, P$" ($\because$ X occurs in both $X = Y$ and P).
This problem would not happen if we always "left" the $!X \bowtie !Y$. For example, as

$$!X \bowtie !Y, P \equiv !X \bowtie !Y, P[!Y/!X]$$

However, if we cannot ensure the locality of the hyperlink $!X$, then we cannot determine whether we can diminish the $!X \bowtie !Y$ or not.
If we cannot diminish the $!X \bowtie !Y$, then we cannot create it neither (applying the congruence rule reversely). Therefore, the existence of the $!X \bowtie !Y$ would let the process different from that does not have it, which is certainly not the desired behavior based on the current implementation.

This force the semantics to always care about the whole program, which spoils the concurrency based on local reducibility and makes it hard to introduce the congruence rules since the congruence of a process relies on the congruence of its constitutions. That is, we must be able to determine the congruence on smaller processes.

The current implementation does not just look for a local process through the computation process, it perform rewriting of the graphs *globally*, thus this did not draw so much interest before; the semantics was regarded as something obvious according to our implementation. However, strict definition is necessary especially when we want to (i) introduce a new idea (e.g. capability typing) and (ii) alter/extend the calculus model for the desired demands (e.g. program transformation for more efficient implementation, the extension on *hlground*, altering the calculus model to deal with directed hyper-graphs).

3.1.2. Brief overview of the proposed semantics of HyperLMNtal

Since we need to distinguish whether the hyperlink in a given process is a local link (occurs only in the process) or a free link (may occur outside of the process), we introduce a *name restriction* mechanism as "$\lambda X.E$" (*abstraction*) in λ calculus and "$(\nu X)P$" (*channel creation*) in π calculus.

In the proposed Flat HyperLMNtal semantics, $\nu X.P$ ensures that the hyperlink X (in the proposed abstract syntax and semantics of Flat HyperLMNtal, the names of hyperlinks are denoted with X rather than $!X$ and we shall call them just "link" for the simplicity)

(Process)	P	$::=$	$\mathbf{0}$		Null
		$\mid$	$p(X_1, \ldots, X_m)$	$m \geq 0$	Atom
		$\mid$	(P, P)		Molecule
		$\mid$	$\nu X.P$		Link creation
		$\mid$	$(P \vdash P)$		Rule

Figure 3.1: Syntax of Flat HyperLMNtal

is a local link in the process P.

Then the correspondence of the (E9) of the LMNtal,

$$X = Y, P \equiv P[Y/X] \quad \begin{array}{l}\text{if } X \text{ occurs free in } P \text{ and } P \text{ is an atom} \\ \textit{And } X \textit{ is a local link in } X = Y, P\end{array}$$

is

$$\nu X.(X \bowtie Y, P) \equiv \nu X.P[Y/X] \quad \text{where } X \text{ or } Y \text{ occurs free in } P$$

From the next section, we explain this formalized syntax and semantics in more detail.

3.2. Syntax of Flat HyperLMNtal

3.2.1. Processes

As well as LMNtal, HyperLMNtal is comprised of two kinds of identifiers:

- X denotes a link name.
- p denotes an atom name. The only reserved name is $\bowtie$.

The syntax is given in Figure 3.1.

An atom $X \bowtie Y$ is called a (link) *fusion*.

The set of the free link names in a process P is denoted as $fn(P)$ and is defined inductively in Figure 3.2.

3.2.2. Rules

Given a rule $(P \vdash Q)$, P is called the left-hand side and Q is called the right-hand side of the rule. A rule $(P \vdash Q)$ must satisfy the following conditions.

1. Rules must not appear in P.
2. $fn(P) \supseteq fn(Q)$.

Intuitively, the latter condition indicates that we have to denote a *new* hyperlink in a scope of a ν (new) on RHS, which we believe follows a common sense.

$$
\begin{aligned}
fn(\mathbf{0}) &= \emptyset \\
fn(p(X_1, \ldots, X_m)) &= \bigcup_{i=1}^{m} \{X_i\} \\
fn((P, Q)) &= fn(P) \cup fn(Q) \\
fn(\nu X.P) &= fn(P) \setminus \{X\} \\
fn((P \vdash Q)) &= \emptyset
\end{aligned}
$$

Figure 3.2: A set of free link names of a Flat HyperLMNtal process

3.3. Operational Semantics of Flat HyperLMNtal

We first define structural congruence ($\equiv$) and then define the reduction relation ($\longrightarrow$) on processes.

3.3.1. Structural congruence

We define the relation $\equiv$ on processes as the minimal equivalence relation satisfying the rules shown in Figure 3.4. Where $P[Y/X]$ is a link substitution that replaces all free occurrences of X with Y as defined in Figure 3.3. Notice if a free occurrence of X occurs in a location where Y would not be free, α-conversion may be required. Here, we use $=$ to denote the syntactic equivalence of the links and processes.

The proposed congruence rules lack the corresponding rules for (E4) $P \equiv P[Y/X]$ (if X occurs free in P) and (E8) $X = Y \equiv Y = X$ in (not-hyper) LMNtal. This is because that they can be derived from the other rules. In other word, they are admissible. We prove this in Theorem 3.1 and Theorem 3.2.

Also, we prove that the sets of the free link names in congruent processes are the same in Theorem 3.3.

Lemma 3.1: Absorption of a futile link creation

$\nu X.P \equiv P$ where $X \notin fn(P)$

$$\mathbf{0}[Y/X] \stackrel{def}{=} \mathbf{0}$$

$$p(X_1, \ldots, X_m)[Z/Y] \stackrel{def}{=} p(X_1[Z/Y], \ldots, X_m[Z/Y])$$

$$\text{where } X_i[Z/Y] \stackrel{def}{=} \begin{cases} Z & \text{if } X_i = Y \\ X_i & \text{if } X_i \neq Y \end{cases}$$

$$(P, Q)[Y/X] \stackrel{def}{=} (P[Y/X], Q[Y/X])$$

$$(\nu X.P)[Z/Y] \stackrel{def}{=} \begin{cases} \nu X.P & \text{if } X = Y \\ \nu X.P[Z/Y] & \text{if } X \neq Y \wedge X \neq Z \\ \nu W.(P[W/X])[Z/Y] & \text{if } X \neq Y \wedge X = Z \wedge W \notin fn(P) \wedge W \neq Z \end{cases}$$

$$(P \vdash Q)[Y/X] \stackrel{def}{=} (P \vdash Q)$$

Figure 3.3: Link substitution of Flat HyperLMNtal

(E1)	$(\mathbf{0}, P) \equiv P$
(E2)	$(P, Q) \equiv (Q, P)$
(E3)	$(P, (Q, R)) \equiv ((P, Q), R)$
(E4)	$P \equiv P' \Rightarrow (P, Q) \equiv (P', Q)$
(E5)	$P \equiv Q \Rightarrow \nu X.P \equiv \nu X.Q$
(E6)	$\nu X.(X \bowtie Y, P) \equiv \nu X.P[Y/X]$ where $X \in fn(P) \vee Y \in fn(P)$
(E7)	$\nu X.\nu Y.X \bowtie Y \equiv \mathbf{0}$
(E8)	$\nu X.\mathbf{0} \equiv \mathbf{0}$
(E9)	$\nu X.\nu Y.P \equiv \nu Y.\nu X.P$
(E10)	$\nu X.(P, Q) \equiv (\nu X.P, Q)$ where $X \notin fn(Q)$

Figure 3.4: Structural congruence on Flat HyperLMNtal processes

Proof 3.1

$$
\begin{array}{lll}
 & \nu X.P & \\
\equiv_{\mathrm{E5}} & \nu X.(\mathbf{0}, P) & \text{where } P \equiv_{\mathrm{E1}} (\mathbf{0}, P) \\
\equiv_{\mathrm{E10}} & (\nu X.\mathbf{0}, P) & \text{where } X \notin fn(P) \\
\equiv_{\mathrm{E4}} & (\mathbf{0}, P) & \text{where } \nu X.\mathbf{0} \equiv_{\mathrm{E8}} \mathbf{0} \\
\equiv_{\mathrm{E1}} & P &
\end{array}
$$

Lemma 3.2: Absorption of a futile link substitution

$P[X/X]$ is syntactically equivalent with P

Proof 3.2

We prove this by structural induction on processes.

♢ *Case* $\boldsymbol{0}$:

$$\mathbf{0}[X/X] \stackrel{def}{=} \mathbf{0}$$

♢ *Case* $p(X_1, \ldots, X_m)$:

$$p(X_1, \ldots, X_m)[X/X] \stackrel{def}{=} p(X_1[X/X], \ldots, X_m[X/X]) = p(X_1, \ldots, X_m)$$

$$\text{since } X_i[X/X] = X_i \text{ where } X_i[X/X] \stackrel{def}{=} \begin{cases} X & \text{if } X_i = X \\ X_i & \text{if } X_i \neq X \end{cases}$$

♢ *Case* (P, Q) :

We have $P[X/X] = P$ and $Q[X/X] = Q$ by induction hypothesis.

Therefore, $(P, Q)[X/X] \stackrel{def}{=} (P[X/X], Q[X/X]) = (P, Q)$

♢ *Case* $\nu Y.P$:

$$(\nu Y.P)[X/X] \stackrel{def}{=} \begin{cases} \nu Y.P & \text{if } Y = X \\ \nu Y.P[X/X] & \text{if } Y \neq X \\ = \nu Y.P & \because\ P[X/X] = P \text{ by induction hypothesis} \end{cases}$$

Since $Y \neq X \wedge Y = X$ could never happen, there is no chance for "variable capturing" and α-conversion for its avoidance (the third option of the link substitution scheme for the link creation), which possibly makes the process not syntactically equivalent (α-equivalent though), won't happen.

◊ *Case* $(P \vdash Q)$:

$$(P \vdash Q)[X/X] \stackrel{def}{=} (P \vdash Q)$$

Theorem 3.1: α-equivalence

$$\nu X.P \equiv \nu Y.P[Y/X] \text{ where } Y \notin fn(P)$$

Proof 3.3

We prove this using Lemma 3.1 and Lemma 3.2.

◊ *Case* $X \in fn(P)$:

$$\begin{aligned}
&\nu X.\nu Y.(Y \bowtie X, (X \bowtie Y, P)) \\
&\equiv_{\text{E5, E6}} \nu X.\nu Y.(X \bowtie X, P) && \because P[X/Y] = P \text{ since } Y \notin fn(P) \\
&\equiv_{\text{E5, Lemma 3.1}} \nu X.(X \bowtie X, P) && \because Y \notin fn((X \bowtie X, P)) \\
&\equiv_{\text{E6}} \nu X.P && \because P[X/X] = P \text{ by Lemma 3.2}
\end{aligned}$$

and

$$\begin{aligned}
&\nu X.\nu Y.(Y \bowtie X, (X \bowtie Y, P)) \\
&\equiv_{\text{E2, E3, E5, E9}} \nu Y.\nu X.(X \bowtie Y, (Y \bowtie X, P)) \\
&\equiv_{\text{E5, E6}} \nu Y.\nu X.(Y \bowtie Y, P[Y/X]) \\
&\equiv_{\text{E5, Lemma 3.1}} \nu Y.(Y \bowtie Y, P[Y/X]) && \because X \notin fn((Y \bowtie Y, P[Y/X])) \\
&\equiv_{\text{E6}} \nu Y.P[Y/X] && \because (P[Y/X])[Y/Y] = P[Y/X] \\
& && \text{by Lemma 3.2}
\end{aligned}$$

◊ *Case* $X \notin fn(P)$:

In this case, we first forcibly include free link X in order to exploit the former proof.

$$
\begin{aligned}
& \nu X.P \\
& \equiv_{\text{Lemma 3.1}} P \\
& \equiv_{\text{E1}} (\mathbf{0}, P) \\
& \equiv_{\text{E4, E7}} (\nu X.\nu X.X \bowtie X, P) \\
& \equiv_{\text{E4, Lemma 3.1}} (\nu X.X \bowtie X, P) && \because X \notin fn(\nu X.X \bowtie X) \\
& \equiv_{\text{E10}} \nu X.(X \bowtie X, P) && \because X \notin fn(P) \\
& \equiv_{\text{The former proof}} \nu Y.(X \bowtie X, P)[Y/X] && \because X \in fn((X \bowtie X, P)) \\
& = \nu Y.(Y \bowtie Y, P[Y/X]) \\
& \equiv_{\text{E6}} \nu Y.(P[Y/X])[Y/Y] && \because Y \in fn((Y \bowtie Y, P[Y/X])) \\
& \equiv_{\text{Lemma 3.2}} \nu Y.P[Y/X]
\end{aligned}
$$

Therefore, $\nu X.P \equiv \nu Y.P[Y/X]$ if $Y \notin fn(P)$.

Theorem 3.2: Symmetry of $\bowtie$

$$X \bowtie Y \equiv Y \bowtie X$$

Proof 3.4

$$
\begin{aligned}
& \nu Z.(Z \bowtie X, Z \bowtie Y) \\
& \equiv_{\text{E6}} \nu Z.(X \bowtie Y) && \because (Z \bowtie Y)[X/Z] = X \bowtie Y \\
& \equiv_{\text{Lemma 3.1}} X \bowtie Y
\end{aligned}
$$

and

$$
\begin{aligned}
& \nu Z.(Z \bowtie X, Z \bowtie Y) \\
& \equiv_{\text{E2, E5}} \nu Z.(Z \bowtie Y, Z \bowtie X) \\
& \equiv_{\text{E6}} \nu Z.(Y \bowtie X) && \because (Z \bowtie X)[Y/Z] = Y \bowtie X \\
& \equiv_{\text{Lemma 3.1}} Y \bowtie X
\end{aligned}
$$

Therefore, $X \bowtie Y \equiv Y \bowtie X$.

Theorem 3.3: The sets of the free links of congruent processes

$$fn(P) = fn(Q) \text{ if } P \equiv Q$$

Proof 3.5

We prove this by structural induction on processes (and structural congruent rules).

◊ *Case* $(\boldsymbol{0}, P) \equiv P$:

$fn((\mathbf{0}, P)) = fn(\mathbf{0}) \cup fn(P) = fn(P)$

◊ *Case* $(P, Q) \equiv (Q, P)$:

$fn((P, Q)) = fn(P) \cup fn(Q) = fn(Q) \cup fn(P) = fn((Q, P))$

◊ *Case* $(P, (Q, R)) \equiv ((P, Q), R)$:

$fn((P, (Q, R))) = fn(P) \cup fn(Q) \cup fn(R) = fn(((P, Q), R))$

◊ *Case* $P \equiv P' \Rightarrow (P, Q) \equiv (P', Q)$:

We have $fn(P) = fn(P')$ if $P \equiv P'$ by induction hypothesis.

Therefore, $fn((P, Q)) = fn(P) \cup fn(Q) = fn(P') \cup fn(Q) = fn((P', Q))$

◊ *Case* $P \equiv P' \Rightarrow \nu X.P \equiv \nu X.P'$:

We have $fn(P) = fn(P')$ if $P \equiv P'$ by induction hypothesis.

Therefore, $fn(\nu X.P) = fn(P) \setminus \{X\} = fn(P') \setminus \{X\} = fn(\nu X.P')$

◊ *Case* $\nu X.(X \bowtie Y, P) \equiv \nu X.P[Y/X]$ *where* $X \in fn(P) \vee Y \in fn(P)$:

Since $X \in fn(P) \vee Y \in fn(P)$, $fn(P[Y/X])$ should be equivalent with $(fn(P) \setminus \{X\}) \cup \{Y\}$.

Therefore,

$fn(\nu X.(X \bowtie Y, P)) = (fn(P) \cup \{Y\}) \setminus \{X\}$

$= ((fn(P) \setminus \{X\}) \cup \{Y\}) \setminus \{X\} = fn(P[Y/X]) \setminus \{X\} = fn(\nu X.P[Y/X])$

◊ *Case* $\nu X.\nu Y.X \bowtie Y \equiv \boldsymbol{0}$:

$fn(\nu X.\nu Y.X \bowtie Y) = fn(X \bowtie Y) \setminus \{X, Y\} = \{X, Y\} \setminus \{X, Y\} = \emptyset = fn(\mathbf{0})$

◊ *Case* $\nu X.\boldsymbol{0} \equiv \boldsymbol{0}$:

$fn(\nu X.\mathbf{0}) = \emptyset \setminus \{X\} = \emptyset = fn(\mathbf{0})$

◊ *Case* $\nu X.\nu Y.P \equiv \nu Y.\nu X.P$:

$fn(\nu X.\nu Y.P) = (fn(P) \setminus \{Y\}) \setminus \{X\} = (fn(P) \setminus \{X\}) \setminus \{Y\} = fn(\nu Y.\nu X.P)$

$$\text{(R1)}\quad \frac{P \longrightarrow P'}{(P, Q) \longrightarrow (P', Q)}$$

$$\text{(R2)}\quad \frac{P \longrightarrow P'}{\nu X.P \longrightarrow \nu X.P'}$$

$$\text{(R3)}\quad \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)}\quad (P, (P \vdash Q)) \longrightarrow (Q, (P \vdash Q))$$

Figure 3.5: Reduction relation on Flat HyperLMNtal processes

$\Diamond$ *Case* $\nu X.(P, Q) \equiv (\nu X.P, Q)$ *where* $X \notin fn(Q)$:

$$\begin{aligned} & fn(\nu X.(P, Q)) \\ &= (fn(P) \cup fn(Q)) \setminus \{X\} \\ &= (fn(P) \setminus \{X\}) \cup fn(Q) \quad \because X \notin fn(Q) \\ &= fn((\nu X.P, Q)) \end{aligned}$$

3.3.2. Reduction relation

We define the reduction relation $\longrightarrow$ on processes as the minimal relation satisfying the rules in Figure 3.5. We proved that the free links of a process, decrease monotonously through a reduction in Theorem 3.4. That is, we won't face a new free link after the reduction, which is a very natural property of a calculus model.

Example 3.2: Non-injective matching

Can the rule

$$(p(X, Y) \vdash q(X, Y))$$

rewrite an atom $p(X, X)$?
More precisely, can the process

$$(p(X, X), (p(X, Y) \vdash q(X, Y)))$$

reduces to something?
The rule cannot be α-converted to the form

$$(p(X, X) \vdash \ldots)$$

However, the atom $p(X, X)$ can be converted to $\nu Y.(Y \bowtie X, p(X, Y))$ using (E7) and Lemma 3.1.

Therefore, it can be rewritten as

$$
\begin{aligned}
&(p(X,X),(p(X,Y)\vdash q(X,Y)))\\
&\equiv_{\text{Lemma 3.1}} \nu Y.(p(X,X),(p(X,Y)\vdash q(X,Y)))\\
&\equiv_{\text{E7}} \nu Y.(Y\bowtie X,(p(X,Y),(p(X,Y)\vdash q(X,Y))))\\
&\longrightarrow \nu Y.(Y\bowtie X,(q(X,Y),(p(X,Y)\vdash q(X,Y))))\\
&\equiv_{\text{E7}} \nu Y.(q(X,X),(p(X,Y)\vdash q(X,Y)))\\
&\equiv_{\text{Lemma 3.1}} (q(X,X),(p(X,Y)\vdash q(X,Y)))
\end{aligned}
$$

As the above, we can match non-injective free links using congruence rule on the link fusion.

Theorem 3.4: The set of the free links of a process through reduction

$$fn(P)\supseteq fn(Q) \text{ if } P\longrightarrow Q$$

Proof 3.6

We prove this by structural induction on processes (and reduction relations).

◊ *Case* $\dfrac{P\longrightarrow P'}{(P,Q)\longrightarrow(P',Q)}$:

We have $fn(P)\supseteq fn(P')$ if $P\longrightarrow P'$ by induction hypothesis.

Therefore, $fn((P,Q))=fn(P)\cup fn(Q)\supseteq fn(P')\cup fn(Q)=fn((P',Q))$

◊ *Case* $\dfrac{P\longrightarrow P'}{\nu X.P\longrightarrow \nu X.P'}$:

We have $fn(P)\supseteq fn(P')$ where $P\longrightarrow P'$ by induction hypothesis.

Therefore, $fn((P,Q))=fn(P)\setminus\{X\}\supseteq fn(P')\setminus\{X\}=fn((P',Q))$

◊ *Case* $\dfrac{P\longrightarrow P'}{\nu X.P\longrightarrow \nu X.P'}$:

We have $fn(P)\supseteq fn(P')$ where $P\longrightarrow P'$ by induction hypothesis.

Therefore, $fn(\nu X.P)=fn(P)\setminus\{X\}\supseteq fn(P')\setminus\{X\}=fn(\nu X.P')$

◊ *Case* $\dfrac{Q\equiv P \quad P\longrightarrow P' \quad P'\equiv Q'}{Q\longrightarrow Q'}$:

We have $fn(P)\supseteq fn(P')$ where $P\longrightarrow P'$ by induction hypothesis and $fn(Q)=fn(P)$ where $Q\equiv P$ and $fn(P')=fn(Q')$ where $P'\equiv Q'$ by Theorem 3.3.

Therefore, $fn(Q)=fn(P)\supseteq fn(P')=fn(Q')$

◊ *Case* $(P, (P \vdash Q)) \longrightarrow (Q, (P \vdash Q))$:

We have $fn(P) \supseteq fn(Q)$ by the second syntactical condition of a rule.

Therefore, $fn((P, (P \vdash Q))) = fn(P) \supseteq fn(Q) = fn((Q, (P \vdash Q)))$

3.4. HyperLMNtal semantics

In this section, we will briefly look at the extended syntax and semantics of Flat HyperLMNtal to suit our current implementation, in which introduce a hierarchy of processes and localization of rules; membranes.

3.4.1. Abstract syntax and semantics of HyperLMNtal

Figure 3.6 gives the syntax of HyperLMNtal based on the current implementation. The newly added syntax is denoted with a dagger (†). The reserved names are = and `><`. Notice the hyperlink is defined in a very similar way to an atom. This is because that the hyperlinks are implemented as an atom with a name `!` and a hidden pointer to an id (this corresponds with the name of the hyperlink in the proposed syntax and semantics) in our current system. Therefore, hyperlinks can be connected to normal link and a port of an atom that is connected to a hyperlink can be matched with a normal link. For example,

```
init.
init :- a(!H).
a(X) :- b(X).
```

will be reduced to `b(!H)` and the remaining rules in our implementation. We allow applying the same abbreviation rules that has used in LMNtal. Also, we can apply the abbreviation schemes for atoms to hyperlinks. For example, $!H_1(X), X >< Y, !H_2(Y)$ can be abbreviated as $!H_1 >< !H_2$.

Figure 3.7 gives the set of the free hyperlink names of a HyperLMNtal process. Notice that the namespaces of normal links and hyperlinks are completely different. When we say "X occurs free in P", then it is the normal link that occurs free (once) in a process P and we have not mentioned about the hyperlinks in that process at all.

The syntactic conditions of both LMNtal and the proposed Flat HyperLMNtal must be satisfied: the number of the end points of a normal link should be kept in at most two, a rule $P :- Q$ must satisfy $fn(P) \supseteq fn(Q)$, etc.

Figure 3.9 shows the structural congruence on HyperLMNtal processes. The newly added congruence relation is denoted with a dagger (†). Here, $P[Y/X]$ is a normal link substitution which replaces a normal link X with a normal link Y and $P\langle H_2/H_2\rangle$ is a hyperlink substitution, which is defined in the Figure 3.8. Notice that α-conversion might be required in hyperlink substitutions as we have already discussed in the Flat

(Process)	P	::=	$\mathbf{0}$		Null
		|	$p(X_1, \ldots, X_m)$	$m \geq 0$	Atom
		|	(P, P)		Molecule
		|	$\{P\}$		Cell
		|	$\nu H.P$		Hyperlink creation †
		|	$!H(X)$		Hyperlink †
		|	$(T :\!\!- T)$		Rule
(Process Template)	T	::=	$\mathbf{0}$		Null
		|	$p(X_1, \ldots, X_m)$	$m \geq 0$	Atom
		|	(T, T)		Molecule
		|	$\{T\}$		Cell
		|	$\nu H.T$		Hyperlink creation †
		|	$!H(X)$		Hyperlink †
		|	$(T :\!\!- T)$		Rule
		|	$@p$		Rule context
		|	$\$p\,[X_1, \ldots, X_m \mid A]$	$m \geq 0$	Process context
(Residual)	A	::=	$[\,]$		Empty
		|	$*X$		Bundle

Figure 3.6: Syntax of HyperLMNtal based on the current implementation

HyperLMNtal semantics.

Figure 3.10 shows the reduction relation on HyperLMNtal process. The newly added reduction relation is denoted with a dagger (†). The process contexts can also match a hyperlink but a link creation. That is, the copying of a process won't divide a hyperlink (in other words, create a new hyperlink).

3.4.2. Translation to the concrete syntax

Any rule $(P \vdash Q)$ can be rewritten as $(\nu X_1.\cdots.\nu X_n.P' :\!\!- \nu Y_1.\cdots.\nu Y_m.Q')$ $(n \geq 0, m \geq 0)$, a structurally congruent rule, where no link creation appears in P' and Q' and $Y_i \notin fn(P')$.

Then, rewrite it as

$$
\begin{aligned}
fn(\mathbf{0}) &= \emptyset \\
fn(p(X_1, \ldots, X_m)) &= \emptyset \\
fn((P, Q)) &= fn(P) \cup fn(Q) \\
fn(\{P\}) &= fn(P) \\
fn(\nu H.P) &= fn(P) \setminus \{!H\} \\
fn(!H(X)) &= \{!H\} \\
fn((P \vdash Q)) &= \emptyset
\end{aligned}
$$

Figure 3.7: Free hyperlink names of a HyperLMNtal process based on the current implementation

$$\mathbf{0}\langle H_2/H_1\rangle \stackrel{def}{=} \mathbf{0}$$

$$p(X_1, \ldots, X_m)\langle H_2/H_1\rangle \stackrel{def}{=} p(X_1, \ldots, X_m)$$

$$(P, Q)\langle H_1/H_2\rangle \stackrel{def}{=} (P\langle H_1/H_2\rangle, Q\langle H_1/H_2\rangle)$$

$$
(\nu H.P)\langle H_1/H_2\rangle \stackrel{def}{=} \begin{cases} \nu H.P & \text{if } H = H_1 \\ \nu H.P\langle H_1/H_2\rangle & \text{if } H \neq H_1 \wedge H \neq H_2 \\ \nu H_3.(P\langle H_3/H\rangle)\langle H_1/H_2\rangle & \text{if } H \neq H_1 \wedge H = H_2 \\ & \quad \wedge\, !H_3 \notin fn(P) \wedge H_3 \neq H_2 \end{cases}
$$

$$
!H(X)\langle H_1/H_2\rangle \stackrel{def}{=} \begin{cases} !H_2(X) & \text{if } H = H_1 \\ !H(X) & \text{if } H \neq H_1 \end{cases}
$$

$$(P \vdash Q)\langle H_1/H_2\rangle \stackrel{def}{=} (P \vdash Q)$$

Figure 3.8: Hyperlink substitution of HyperLMNtal based on the current implementation

(E1)	$\mathbf{0}, P \equiv P$	
(E2)	$P, Q \equiv Q, P$	
(E3)	$P, (Q, R) \equiv (P, Q), R$	
(E4)	$P \equiv P[Y/X]$	if X if a local link of P
(E5)	$P \equiv P' \Rightarrow P, Q \equiv P', Q$	
(E6)	$P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$	
(E7) †	$P \equiv Q \Rightarrow \nu H.P \equiv \nu H.Q$	
(E8)	$X = X \equiv \mathbf{0}$	
(E9)	$X = Y, P \equiv P[Y/X]$	if P is an atom and X occurs free in P
(E10)	$\{X = Y, P\} \equiv X = Y, \{P\}$	if exactly one of X and Y occurs free in P
(E11) †	$\nu H_1.(!H_1 >< !H_2, P) \equiv \nu H_1.P\langle H_1/H_2\rangle$	where $!H_1 \in fn(P) \vee !H_2 \in fn(P)$
(E12) †	$\nu H_1.\nu H_2.(!H_1 >< !H_2) \equiv \mathbf{0}$	
(E13) †	$\nu H.\mathbf{0} \equiv \mathbf{0}$	
(E14) †	$\nu H_1.\nu H_2.P \equiv \nu H_2.\nu H_1.P$	
(E15) †	$\nu H.\{P\} \equiv \{\nu H.P\}$	
(E16) †	$\nu H.(P, Q) \equiv (\nu H.P, Q)$	where $!H \notin fn(Q)$

Figure 3.9: Structural congruence on HyperLMNtal processes based on the current implementation

(R1)	$\dfrac{P \longrightarrow P'}{P, Q \longrightarrow P', Q}$	
(R2)	$\dfrac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}}$	
(R3) †	$\dfrac{P \longrightarrow P'}{\nu H.P \longrightarrow \nu H.P'}$	
(R4)	$\dfrac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$	
(R5)	$\{X = Y, P\} \longrightarrow X = Y, \{P\}$	if X and Y occur free in $\{X = Y, P\}$
(R6)	$X = Y, \{P\} \longrightarrow \{X = Y, P\}$	if X and Y occur free in P
(R7)	$T\theta, (T \vdash U) \longrightarrow U\theta, (T \vdash U)$	

Figure 3.10: Reduction relation on HyperLMNtal processes based on the current implementation

$$(P' \texttt{ :- num}(!X_1) \texttt{ =:= } N_1, \cdots, \texttt{num}(!X_n) \texttt{ =:= } N_n \texttt{ | } Q'')$$

where N_i is a number of the occurrence of the hyperlink X_i in P (except those have normal links connected to the fusion atom) and Q'' is a inductively translated concrete syntax form of Q'. The current implementation does not support to write as `!H(X)`. We have to write all the hyperlinks inside of the ports of the atoms that are connected to. For example, `a(!H)` or `!H = X`.

3.4.3. Further observation

Basically, implementing a hyperlink as a special atom was a great decision. It allows both implementation and semantics relatively easy. However, it results with a little annoying observation when we use hyperlinks with membranes. That is, hyperlinks can be separated with a membrane. For example,the following program

```
init.
init :- {a(!H)}.
{a(X), $p[X]} :- a(X), {$p[X]}.
a(!H) :- b.
```

will be reduced to `a(X), {!H(X)}` and the forth rule `a(!H) :- b` would not be applied since the hyperlink `!H(X)` is not at the same hierarchy with the rule.

Both our current implementation and the proposed semantics support this idea. Thus we may do not need to worry about this. However, if we think of a hyperlink as an extension of a normal link, this is certainly a strange behavior. Well, for the case above, we can apply the rule `a(X), {!H = X} :- b` to the program and if we do so, then the program will be reduced to `b`. But still, we do not think this is a natural behavior based on our intuition.

Therefore, we may like to modify our implementation to let all the hyperlinks to be in the same hierarchy with the atoms connected to them at every transition state. Then we should modify our semantics to allow this. This can be done in a very simple way; just adding the following congruence rule, which allows a hyperlink to move anywhere to go along with the connected atom.

$$(E18) \quad \{!H(X), P\} \equiv !H(X), \{P\}$$

Chapter 4.

Syntax/Semantics of G-Machine

G-Machine is a virtual machine for non-strict functional languages which uses *graph-reduction*. Since LMNtal is a language that has focused on graph rewriting, it is important to check how easily we can achieve this (or if not, why we cannot) . In this chapter, we will introduce the functional language we implemented, core language, and the implemented G-Machine based on [16].

4.1. General introduction to lazy evaluation and graph-reduction

Strict evaluation strategy requires all the arguments to be evaluated before performing beta-reduction (application). Where non-strict doesn't require this. Thus, the easiest way to accomplish beta reduction in a non-strict evaluation is just replacing the variable with the corresponding applied term: Call By Name. For example, $(\lambda x.x + x)(1 + 2)$ will be reduced to $(1 + 2) + (1 + 2)$. However, this is not efficient since we should calculate $1 + 2$ twice. To improve efficiency, we can consider the reducing expressions as "graph" not just tree which allows sharing of expressions. For example, the former expression can be represented as the graph in Figure 5.2. Notice that the expression $1 + 2$ (blue-colored subgraph) is shared. By sharing expressions and evaluating once for a each, we can reduce the steps of calculation process.

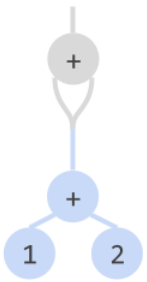


Figure 4.1: Graph representation of an expression

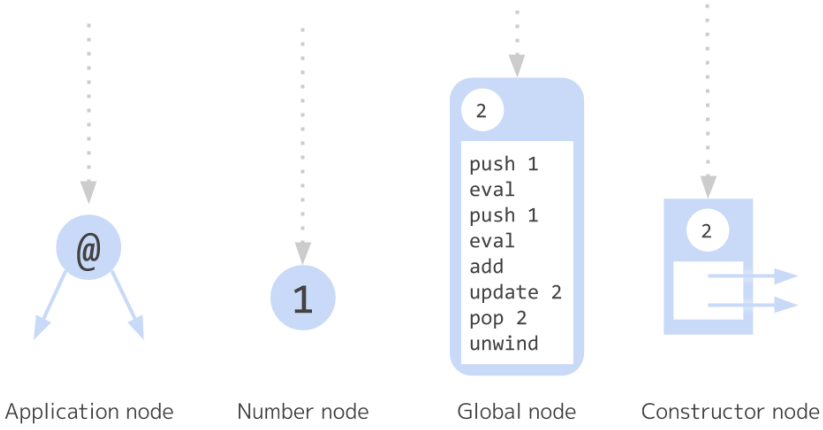


Figure 4.2: Example of nodes in G-Machine

Though not-hyper Flat LMNtal can handle graph, the end points of links in LMNtal are restricted to be just 2. Therefore, it is not easy to represent sharing. Using membrane is one solution to this problem but it is rather hard to write these kinds of programs with membranes. Instead, HyperLMNtal is a very suitable language to deal with this. Since hyperlinks are allowed to connect one or more ports.

4.2. An introduction to G-Machine

The components of G-Machine is a code, stack, dump heap and a global environment

Heap is a multiset of nodes. Where nodes are basically one of "application node", "global (i.e. super combinator/function) node" and "a node for a primitive value (e.g. integer)". Notice that the global node is completely different from the global environment. These nodes form graph since the application node has addresses (i.e. directed edges/-pointers) to its argument nodes. Figure. 4.2 shows the example of nodes in G-Machine. Here, the label for the application node is written as `@`. Each nodes might have or not have edges whose head is the node (the gray-colored arrow).

Stack is a stack of an addresses to nodes of a chained applications. The process of evaluating application is called *unwinding*, which is basically done by pushing the address of the right child of application node on top of the stack.

Dump is a stack of the stack. This is used to evaluate an expression pointed by the top of the stack until it reaches to Weak Head Normal Form. To accomplish this, we have to store the current stack and the code to dump. This is mainly for evaluating an expression with a primitive operator. For example, we can not add expressions but numbers: we have to evaluate the both arguments until they are to be numbers.

Code is a list of instructions. The evaluation is done by following these instructions. Codes are obtained by compiling (top-level) functions and kept in global nodes.

Global environment is a function from the name of the top-level functions to the addresses of the corresponding global nodes.

4.3. Syntax of the core language

The core language is an lazy functional language. The syntax of the core language is given in Figure 4.3. This is an abstract syntax. Where *var* is a variable (identifier) and *num* is an integer. A more concrete syntax can be obtained from [16].

A core language program is consisted of a list of functions. We call them super-combinators. Since these functions are allowed to have a free variable, a variable which is not a locally defined variable, in their body expressions (i.e. the right-hand side of the "=") these are certainly not combinators and thus not super-combinators technically. However, in the reference [16], these are noted as super-combinators, therefore we shall call them so, like wisely.

It is not designed to be written by programmers directly, but is designed to be suitable for an input of the compiler. For example, this language lacks lambda abstraction (anonymous function), which can be achieved by "program transformation" technique, lambda lifting. And user defined data types must be written as `Pack{2, 2} 1 Pack{1, 0}` instead of like `Cons 1 Nil` (with a predefinition `data List a = Cons List a | Nil`). The first argument of `Pack` is the *tag* of the data type. For example, we may assign 1 for `Nil` and 2 for `Cons`. The second argument is an arity of the data type (the number of the arguments). In the former example, we need 0 to be the second argument of the `Pack{1, 0}` since `Nil` should have no argument and 2 to be the second argument of the `Pack{2, 2}` since the `Cons` should have 2 arguments. The decomposition of these kinds of data structures in the case expression are done with the tags we assigned for each types. For example, `case list of Nil -> ... | Cons h t -> ...` can be done with `case list of <1> -> ... ; <2> h t -> ...` in the core language.

4.4. Syntax of G-Machine

A set of instructions of G-Machine in given in Figure 4.4. where i is an integer *instr* is an instruction *name* is a name of top-level super-combinator and *code* is a code (a list of instructions). The operation semantics of these instructions will be given in Section 4.6.

4.5. Compilation scheme of G-Machine

The compilation to the codes are done with the following rules. Where $l_1 +\!\!+ l_2$ describes a concatenation of lists l_1 and l_2.

$\mathcal{SC}$ compiles a super-combinator $f\ x_1 \ldots x_n = e$. The output is a G-Machine code for a global node.

(Program)	*program*	::=	$sc_1; \ldots; sc_n$	$n \geq 1$
(Super Combinator)	*sc*	::=	*var* $var_1 \ldots var_n$ `=` *expr*	$n \geq 0$
(Expression)	*expr*	::=	*var*	Variable
		|	*expr expr*	Application
		|	*expr binop expr*	Infix binary application
		|	`let` *defns* `in` *expr*	Local definition
		|	`let rec` *defns* `in` *expr*	Local recursive definition
		|	`case` *expr* `of` *alts*	Case expression
		|	`if` *expr* `then` *expr* `else` *expr*	If expression
		|	*num*	Number
		|	`Pack{`*num*`,`*num*`}`	Constructor
(Definitions)	*defns*	::=	$defn_1; \ldots; def_n$	$n \geq 1$
	defn	::=	*var* `=` *expr*	
(Alternatives)	*alts*	::=	$alt_1; \ldots; alt_n$	$n \geq 1$
	alt	::=	`<`*num*`>` $var_1 \ldots var_n$ `->` *expr*	$n \geq 0$
(Binary Operators)	*binop*	::=	*arithop* | *relop*	
	arithop	::=	`+` | `-` | `*` | `/`	Arithmetic
	relop	::=	`<` | `/=`	Comparison

Figure 4.3: Syntax of the core language

(Code)	*code*	::=	$[instr_1, \ldots, instr_n]$	$n \geq 0$
(Instruction)	*instr*	::=	`Slide` *num*	
		|	`Alloc` *num*	
		|	`Update` *num*	
		|	`Pop` *num*	
		|	`Unwind`	
		|	`PushGlobal` *var*	
		|	`PushInt` *num*	
		|	`Push` *num*	
		|	`Mkap`	
		|	`Eval`	
		|	`Cond` *code code*	
		|	`Add` | `Sub` | `Mul` | `Div`	
		|	`Lt` | `Neq`	
		|	`Pack` *num num*	
		|	`Casejump` $\begin{bmatrix} num_1 \texttt{ -> } code_1 \\ \vdots \\ num_n \texttt{ -> } code_n \end{bmatrix}$	$n \geq 1$
		|	`Split` *num*	

Figure 4.4: Instructions of G-Machine

(Node)	*node*	::=	`NNum` *num*		Number
		|	`NGlobal` *num code*		Global
		|	`NApp` *a a*		Application
		|	`NConstr` *num* $[a_1, \ldots, a_n]$	$n \geq 0$	Constructor
		|	`NInd` *a*		Indirection

Figure 4.5: Nodes of G-Machine

(Stack)	*stack*	::=	$[a_1, \ldots, a_n]$	$n \geq 0$
(Dump)	*dump*	::=	$[\langle stack_1, code_1 \rangle, \ldots, \langle stack_n, code_n \rangle]$	$n \geq 0$
(Global Environment)	*env*	::=	$\{var_1 \mapsto a_1, \ldots, var_n \mapsto a_n\}$	$n \geq 1$
(Heap)	*heap*	::=	$\{a_1 \mapsto node_1, \ldots, a_n \mapsto node_n\}$	$n \geq 0$

Figure 4.6: The Stack, dump, global environment and the heap of G-Machine

$$\mathcal{SC}[\![f\ x_1 \ldots x_n = e]\!] = \mathcal{R}[\![e]\!]\ [x_1 \mapsto 0, \ldots, x_n \mapsto n-1]\ n$$

$\mathcal{R}$ is for a compilation of the body expression of super-combinators e (i.e. the right-hand side of $=$). Where ρ is an environment (the function from the names of variables to their indices), d is an arity of the supercombinator. After evaluating the e following the code obtained from $\mathcal{E}[\![e]\!]\ \rho\ d$, we need to (i) `update` the pointer which previously pointed the supercombinator to point the address of the node newly obtained by evaluating the body expression of the supercombinator, (ii) `pop` out the addresses which correspond the local variables (the right-hand side of the supercombinator definition) and (iii) then perform evaluation, which is basically `unwinding`, as before.

$$\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Update}\ d, \texttt{Pop}\ d, \texttt{Unwind}]$$

$\mathcal{E}$ compiles an expression in a strict-context for a better efficiency. This is designed not to offend the general laziness of the core language. Whether the given expression is in the strict context or not can be determined by the derivations shown in p.126 of [16].

If the given expression is an integer, we need a code to generate the number node with the integer (i.e. `push integer` to the heap)

$$\mathcal{E}[\![i]\!]\ \rho = [\texttt{PushInt}\ i]$$

If the given expression is a local definition, we need to evaluate all of those expressions we have defined locally in a non-strict context since we may not require these and we can evaluate the body expression in a strict context because we must evaluate this invariably. Where ρ^{+i} is defined to meet $\rho^{+n}x = (\rho x) + n$. This is because the distances of the variables from the top of the stack increases by pushing the each local definitions. Also, we need to clear the n local definitions and `slide` the top of the

stack at the very last process.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{let } x_1 = e_1; \ldots; x_n = e_n \texttt{ in } e]\!]\ \rho \\
= \quad \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
\quad\ \ \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
\quad\ \ \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide } n] \\
\qquad\quad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_0 \mapsto 0]
\end{array}
$$

The evaluation/execution and compilation of a local recursive definition is basically the same as the local (not-recursive) definition. The difference is that we need the environment with all the locally defined variables not just when we are compiling the body expression but also when we are compiling the right-hand side of definitions and that we firstly need to `allocate` dummy pointers on the top n of the stack when we evaluate. After the evaluation of a local definition, the allocated dummy pointers are `updated` to point the node which is a result of the evaluation so as not to traverse these dummy pointers, which would cause "segmentation fault" or "null pointer exception" in such languages as "c" or "Java".

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{let rec } x_1 = e_1; \ldots; x_n = e_n \texttt{ in } e]\!]\ \rho \\
= \quad [\texttt{alloc } n] +\!\!+ \\
\quad\ \ \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update } (n-1)] +\!\!+ \ldots +\!\!+ \\
\quad\ \ \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update } 0] +\!\!+ \\
\quad\ \ \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{Slide } n] \\
\qquad\quad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_0 \mapsto 0]
\end{array}
$$

If the given expression is a binary application with a primitive operator, we will evaluate the both arguments since we definitely need them. Therefore, the compilation of the arguments can be done with the same scheme $mathcal{E}$, in a strict context. However, the environment of the first arguments (actually, either one of both) must be converted as ρ^{+1} since the location of the variables change by pushing the second argument to the top of the stack as we described in the compiling scheme of local definition. We need to perform the primitive operation, addition for this time, after evaluating the arguments, therefore we put the operator at the very last position of the generated code. The expressions with the other primitive operators (such as `-` , `*` , `<`) are also dealt with the same manner.

$$\mathcal{E}[\![e_0 + e_1]\!]\ \rho = \mathcal{E}[\![e_1]\!]\ \rho +\!\!+ \mathcal{E}[\![e_0]\!]\ \rho^{+1} +\!\!+ [\texttt{Add}]$$

An expression with an unary primitive operator is compiled as the following. Notice that the argument can be compiled in a strict context as we have already discussed in the binary operator.

$$\mathcal{E}[\![\texttt{negate}\ e]\!]\ \rho = \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Neg}]$$

If the given expression is a case expression `case` e `of` *alts*, we can compile the e in a strict context since we will evaluate this definitely. After the evaluation of the e, we will be executing the `Casejump` instruction, which has one or more codes and simply looks at the tag of the structured data, the evaluated expression, and determines which code to execute in the execution phase. The compilation scheme of the *alts* will be described later.

$$\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho = \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{Casejump}\ \ \mathcal{D}[\![alts]\!]\ \ \rho]$$

If the given expression is a constructor and sufficient applications can be compiled with the following rule. If the application is in-sufficient then we should use program conversion technique to make it sufficient as described in [16]. Notice that we should compile all the arguments in a non-strict context since we don't want to always evaluate the arguments of the structured data. For example, an infinite list can achieved by not evaluating the second argument of a `Cons` somewhere.

$$\mathcal{E}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1 \ldots e_a]\!]\ \rho = \mathcal{C}[\![e_a]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)} +\!\!+ [\texttt{Pack}\ \ t\ a]$$

If the given expression is an if expression `if` e_0 `then` e_1 `else` e_2, we will evaluate e_0 but we won't evaluate one of e_1 or e_2. However, this will be accomplished by the `Cond` instruction, which has two codes and will look at the top of the stack and determine which code to execute in the evaluating process. Therefore, we can compile the every three argument in a strict context. Notice that we haven't convert the environment as ρ^{+1} since we will be discarding the result of the e_0 at the top of the stack and we put the address to the node which consists he expression e_1 or e_2 at the top of the stack, the exact same location where e_0 was before.

$$\mathcal{E}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\ \rho = \mathcal{E}[\![e_0]\!]\ \rho +\!\!+ [\texttt{Cond}\ (\mathcal{E}[\![e_1]\!]\ \rho)(\mathcal{E}[\![e_2]\!]\ \rho)]$$

The default cases including an application, a variable and a constructor are compiled in a non-strict context. Since the evaluation of an application should be done

in a lazy manner so does the application to a constructor too. The evaluation of a variable and a unary constructor can be done in a step and the generated code should have no difference regardless of whether we compiled it in a strict context or non-strict context. Therefore, we would like to compile them in a non-strict context so we would not have to repeat the same compilation rule both in compilation of strict and non-strict context. We also need to `eval`uate the expression until it reaches to the weak head normal form.

$$\mathcal{E}[\![e]\!]\ \rho = \mathcal{C}[\![e]\!]\ \rho +\!\!+ [\texttt{Eval}]$$

$\mathcal{C}$ compiles expressions in a non-strict context to accomplish the laziness of the core language. The compilation scheme is basically the same as the $\mathcal{E}$ so we won't explain them in detail.

$$\mathcal{C}[\![i]\!]\ \rho = [\texttt{PushInt}\ i]$$

$$\begin{array}{l}\mathcal{C}[\![\texttt{let}\ x_1 = e_1; \ldots; x_n = e_n\ \texttt{in}\ e]\!]\ \rho \\ = \quad \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\ \quad\ \ \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\ \quad\ \ \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\![\texttt{Slide}\ n]\!] \\ \qquad\quad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_0 \mapsto 0]\end{array}$$

$$\begin{array}{l}\mathcal{C}[\![\texttt{let rec}\ x_1 = e_1; \ldots; x_n = e_n\ \texttt{in}\ e]\!]\ \rho \\ = \quad [\texttt{Alloc}\ n] +\!\!+ \\ \quad\ \ \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{Update}\ (n-1)] +\!\!+ \ldots +\!\!+ \\ \quad\ \ \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{Update}\ 0] +\!\!+ \\ \quad\ \ \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\![\texttt{Slide}\ n]\!] \\ \qquad\quad \text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_0 \mapsto 0]\end{array}$$

$$\mathcal{E}[\![\texttt{Pack}\{t,a\}\ e_1 \ldots e_a]\!]\ \rho = \mathcal{C}[\![e_a]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)} +\!\!+ [\texttt{Pack}\ t\ a]$$

$$\begin{array}{rcl}\mathcal{C}[\![e_0\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho +\!\!+ \mathcal{C}[\![e_0]\!]\ \rho^{+1} +\!\!+ [\texttt{Mkap}] \\ \\ \mathcal{C}[\![f]\!]\ \rho & = & [\texttt{PushGlobal}\ f] \\ & & \text{where } f \text{ is a name of a supercombinator} \\ \\ \mathcal{C}[\![x]\!]\ \rho & = & [\texttt{Push}\ (\rho\ x)] \quad \text{where } x \text{ is a local variable}\end{array}$$

Notice that the compiling with the primitive operator $\odot$ is simply generating the code [`PushGlobal` $\odot$]. For example, `if` e_1 `then` e_2 `else` e_3 is firstly transpiled into `if` e_1 e_2 e_3 (notice that the if expression is now an expression just applying all the three arguments e_1, e_2, e_3 to the `if` primitive) and then compiled as $[\ldots, \texttt{PushGlobal if}]$.

The alternatives in a `case` expression are compiled as the following.

$$\mathcal{D}[\![alt_1, \ldots, alt_n]\!]\ \rho = [\mathcal{A}[\![alt_1]\!]\ \rho, \ldots, \mathcal{A}[\![alt_n]\!]\ \rho]$$

An alternative in a `case` expression is compiled as the following. The generated code is basically a tuple of tag and a code. The tag is used to match the constructor. If the tag has matched, then the combined code will be executed. We firstly need to `split` our structured data and push all of them on the stack. By doing so, we can access our variable with the indices obtained from our new environment ρ'. At the very last phase of the evaluation, we should `slide` the addresses on the stack which were assigned for the newly defined variables since we would no longer need them anymore.

$$\mathcal{A}[\![\texttt{<}t\texttt{>}\ x_1 \ldots x_n\ \texttt{->}\ body]\!]\ \rho = t\ \texttt{->}\ [\texttt{Split}\ n] +\!\!+ \mathcal{E}[\![body]\!]\ \rho' +\!\!+ [\texttt{Slide}\ n]$$

$$\text{where } \rho' = \rho^{+n}[x_1 \mapsto 0 \ldots x_n \mapsto n-1]$$

4.6. Operational Semantics of G-Machine

As we have seen in Section 4.2, G-Machine has code, stack, dump, heap and global environment. Here in this section, we will look at the state transition rules for G-Machine. The transition rules are described as the following.

	old code	old stack	old dump	old heap	old global environment
$\Longrightarrow$	new code	new stack	new dump	new heap	new global environment

The list of the instructions, code, can be represented as $instr_1 : \ldots : instr_n$ as well as $[instr, \ldots, instr]$ as like in Haskell language.

The existence and a creation of a node *node* at the address a in a heap h is represented as $h[a : node]$.

Also, existence of the relation of the name of a supercombinator *name* and an address of the corresponding global node a in a global environment m is represented as $m[name : a]$.

$$\begin{array}{lllllll} & \texttt{PushGlobal}\ f : i & & s & d & h & m[f : a] \\ \Longrightarrow & & i & a : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{PushInt}\ n : i & & s & d & h & m \\ \Longrightarrow & & i & a : s & d & h[a : \texttt{NNum}\ n] & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Mkap} : i & a_1 : a_2 : s & d & h & & m \\ \Longrightarrow & & i & a : s & d & h[a : \texttt{NApp}\ a_1\ a_2] & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Push}\ n : i & & a_0 : \ldots : a_{n+1} : s & d & h[a_{n+1} : \texttt{NApp}\ a_n\ a'_n] & m \\ \Longrightarrow & & c & a'_n : a_0 : \ldots : a_{n+1} : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Slide}\ n : i & a_0 : \ldots : a_n : s & d & h & m & \\ \Longrightarrow & & c & a_0 : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Update}\ n : i & a : a_0 : \ldots : a_n : s & d & h & & m \\ \Longrightarrow & & i & a_0 : \ldots : a_n : s & d & h[a_n : \texttt{NInd}\ a] & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Pop}\ n : i & a_0 : \ldots : a_n : s & d & h & m & \\ \Longrightarrow & & i & s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & \texttt{Alloc}\ n : i & & s & d & h & m \\ \Longrightarrow & & i & a_1 : \ldots : a_n : s & d & h \begin{bmatrix} a_1 : \texttt{NInd hNull} \\ \ldots \\ a_n : \texttt{NInd hNull} \end{bmatrix} & m \end{array}$$

$$\begin{array}{lllllll} & \odot : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0, a_1 : \texttt{NNum}\ n_1] & m & \\ \Longrightarrow & & i & a : s & d & h[a : \texttt{NNum}\ (n_0 \odot n_1)] & m \end{array}$$

Where $\odot$ is a binary operator such as `Add` or `Neq`.

$$\begin{array}{lccccc} & \texttt{Neg} : i & a : s & d & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & i & a' : s & d & h[a' : \texttt{NNum}\ (-n)] & m \end{array}$$

$$\begin{array}{lccccc} & \texttt{Cond}\ i_1\ i_2 : i & a : s & d & h[a : \texttt{NNum}\ 1] & m \\ \Longrightarrow & i_1 +\!\!+ i & s & d & h & m \end{array}$$

$$\begin{array}{lccccc} & \texttt{Cond}\ i_1\ i_2 : i & a : s & d & h[a : \texttt{NNum}\ 0] & m \\ \Longrightarrow & i_2 +\!\!+ i & s & d & h & m \end{array}$$

$$\begin{array}{lcccc} & \texttt{Eval} : i & a : s & d & h & m \\ \Longrightarrow & [\texttt{Unwind}] & [a] & \langle i, s\rangle : d & h & m \end{array}$$

$$\begin{array}{lcccc} & \texttt{Pack}\ t\ n : i & a_1 : \ldots : a_n : s & d & h & m \\ \Longrightarrow & i & s & d & h[a : \texttt{NConstr}\ t\ [a_1 : \ldots : a_n]] & m \end{array}$$

$$\begin{array}{lcccc} & \texttt{Casejump}\ [\ldots, t\ \texttt{->}\ i', \ldots] : i & a : s & d & h[a : \texttt{NConstr}\ t\ ss] & m \\ \Longrightarrow & i' +\!\!+ i & a : s & d & h & m \end{array}$$

$$\begin{array}{lcccc} & [\texttt{Split}] : i & a : s & d & h[a : \texttt{NConstr}\ t\ [a_1, \ldots, a_n]] & m \\ \Longrightarrow & i & a_1 : \ldots : a_n : s & d & h & m \end{array}$$

$$\begin{array}{lcccc} & [\texttt{Unwind}] & a : s & \langle i', s'\rangle : d & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & i' & a : s' & d & h & m \end{array}$$

$$\begin{array}{lcccc} & [\texttt{Unwind}] & a : s & \langle i', s'\rangle : d & h[a : \texttt{NConstr}\ n\ as] & m \\ \Longrightarrow & i' & a : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & [\texttt{Unwind}] & a_0 : s & d & h[a_0 : \texttt{NInd}\ a] & m \\ \Longrightarrow & [\texttt{Unwind}] & a : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & [\texttt{Unwind}] & a_0 : \ldots : a_n : s & d & h[a_0 : \texttt{NGlobal}\ n\ c] & m \\ \Longrightarrow & c & a_0 : \ldots : a_n : s & d & h & m \end{array}$$

$$\begin{array}{lllllll} & [\texttt{Unwind}] & a : s & d & h[a : \texttt{NApp}\ a_1\ a_2] & m \\ \Longrightarrow & [\texttt{Unwind}] & a_1 : a : s & d & h & m \end{array}$$

When we have a code [Unwind] but have an empty dump, then we are now at a terminal state. The stack should now contain exactly 1 address pointing to the desired result.

Chapter 5.

The implementation of G-Machine in HyperLMNtal

In this chapter, we will explain briefly about the G-machine and the compiler we implemented. The source code for each are in Appendix A. Then we will see some examples to show we have implemented G-machine properly.

5.1. Project overview

The program is separated into three parts; the testing module (test15.lmn in Listing 5.1), the compiler (compiler15.lmn in Appendix A.1) and G-machine (gm15.lmn in Appendix A.2)[1] The program is executed with the following command:

```
slim --hl -t test15.lmn gm15.lmn compiler15.lmn
```

```
% Testing module

{ test.

    % Testing code
    Program = [
        ["id", "x"] = "x",
        ["main"   ] = ("id", 3)
    ].

    compileSC(Program). % To starts compiling

    % The initial code and the stack.
    % (The initial heap (global nodes) and global environment
    %  will be produced by compiling the program (super combinators)).
    code = [pushGlobal("main"), unwind].
    stack = [].
    dump = [].
```

[1]The number, 15, just denotes the version and is not important in this paper.

```
    % Predifined primitive operators.
    primitives @@
    :- uniq |
      m("+", !Add),
        !Add =
          nGlobal(2, [push(1), eval, push(1), eval, add, update(2), pop(2),
  unwind]),
      m("-", !Sub),
      !Sub =
          nGlobal(2, [push(1), eval, push(1), eval, sub, update(2), pop(2),
  unwind]),
      m("*", !Mul),
      !Mul =
          nGlobal(2, [push(1), eval, push(1), eval, mul, update(2), pop(2),
  unwind]),
      m("/", !Div),
      !Div =
          nGlobal(2, [push(1), eval, push(1), eval, div, update(2), pop(2),
  unwind]),
      m("negate", !Neg),
      !Neg = nGlobal(1, [push(0), eval, neg, update(1), pop(1), unwind]),
      m("<", !LT),
      !LT =
          nGlobal(2, [push(1), eval, push(1), eval, lt, update(2), pop(2),
 unwind]),
      m("\=", !Neq),
      !Neq =
          nGlobal(2, [push(1), eval, push(1), eval, neq, update(2), pop(2),
  unwind]),
      m("if", !IF),
      !IF =
          nGlobal(3, [ push(0), eval, cond([push(1)], [push(2)]),
               update(3), pop(3), unwind]
          ).
}.


compile @@
{compiler. @rCompiler}, {test. $p. @rTest}
  :- {compiling. @rCompiler. $p. @rTest}.


exec @@
{gm. @rGM}, {compiling. @rCompiler. $p}/
  :- {@rGM. $p}.


% Terminates if the dump is empty.
terminateWithNum @@
```

```
{code = [unwind], stack = [!A], !A = nNum(N), dump = [], $p[], @r}
  :- int(N)
     | result = N.

% Terminates if the dump is empty.
terminateWithConstr @@
{code = [unwind], stack = [!A], !A = nConstr(Tag, Args), dump = [], $p[],
  @r}
  :- ground(Args), int(Tag)
     | result = nConstr(Tag, Args).
```

Listing 5.1: The testing module

Here, in the testing module `test15.lmn`, the testing code is

```
Program = [
  ["id", "x"] = "x",
  ["main"   ] = ("id", 3)
].
```

. This testing code should satisfy the LMNtal Shape Type[17] defined as Figure 5.1. We will introduce more examples later in Section 5.2.

By executing this with a trace option `-t`, we firstly see this to be compiled as the following:[2]

```
m("id",!H37).
nGlobal(1,[push(0),eval,update(1),pop(1),unwind],!H37).
m("main",!Hf4).
nGlobal(0,[pushInt(3),pushGlobal("id"),mkap,eval,update(0),pop(0),unwind
],!Hf4).
```

Where the `m("id", !H37)` and `m("main", !Hf4)` are the global environment, that maps the name of the supercombinators, "id" and "main", to the address of the whose global nodes exist. Then we are to execute our virtual machine.

5.2. Examples

This section gives some core language examples run on the G-machine we have implemented.

[2]This is a simplified output.

```
defshape scs(Prog) {
  Prog = scs :- Prog = [sc|scs].
  Prog = scs :- Prog = [sc].
  SC = sc :- string(Name) | SC = ([Name|vars] = exp).
  Vars = vars :- Vars = [].
  Vars = vars :- string(Var) | Vars = [Var|vars]
  E = exp :- int(N) | E = N.
  E = exp :- E = (exp, exp).
  E = exp :- string(X) | E = X.
  E = exp :- E = if(E, E, E).
  E = exp :- E = case(E, alts).
  E = exp :- E = exp + exp.
  E = exp :- E = exp - exp.
  E = exp :- E = exp * exp.
  E = exp :- E = exp / exp.
  E = exp :- E = exp < exp.
  E = exp :- E = exp \= exp.
  E = exp :- E = negate(exp).
  E = exp :- int(Tag), int(Arity) | E = eConstr(Tag, Arity).
  Alts = alts :- [altNT|alts].
  Alts = alts :- [altNT].
  Alt = altNT :- int(Tag) | Alt = alt(Tag, vars, exp).
} nonterminal {
  exp(E), sc(SC), scs(SCs), vars(Vars), alts(Alts), altNT(Alt)
}
```

Figure 5.1: The shape type of the program

5.2.1. Recursive functions

Here is a program that calculates the factorial number and a program that calculates the Fibonacci number.

```
Program = [
  ["fac", "n"] = if(1 < "n", ("n" * ("fac", "n" - 1)), 1),
  ["main"    ] = ("fac", 5)
].
```

Listing 5.2: Factorial

This example above will result with the `result(120)` which is $1 \times 2 \times 3 \times 4 \times 5$ in 166 steps.

```
Program = [
  ["fib", "n"] = if(2 < "n",
                    ("fib", "n" - 1) + ("fib", "n" - 2),
                    1),
  ["main"    ] = ("fib", 7)
].
```

Listing 5.3: nFib

The example above should result `result(13)` in 793 steps.

Here is a *nFib* program. This function is famous since the returned value denotes the number of the function called. And therefore it is a popular program in benchmark tests.

```
Program = [
  ["nfib", "n"] = if(0 < "n",
                     1 + ("nfib", "n" - 1) + ("nfib", "n" - 2),
                     1),
  ["main"     ] = ("nfib", 10)
].
```

Listing 5.4: nFib

We have observed that the number of the steps for each `nfib` function calls are about 34.5. That is, we need about 34.5 steps to calculate the `if 0 < n then 1 + nfib (n - 1) + nfib (n - 2) else 1` and the function call for `nfib`.

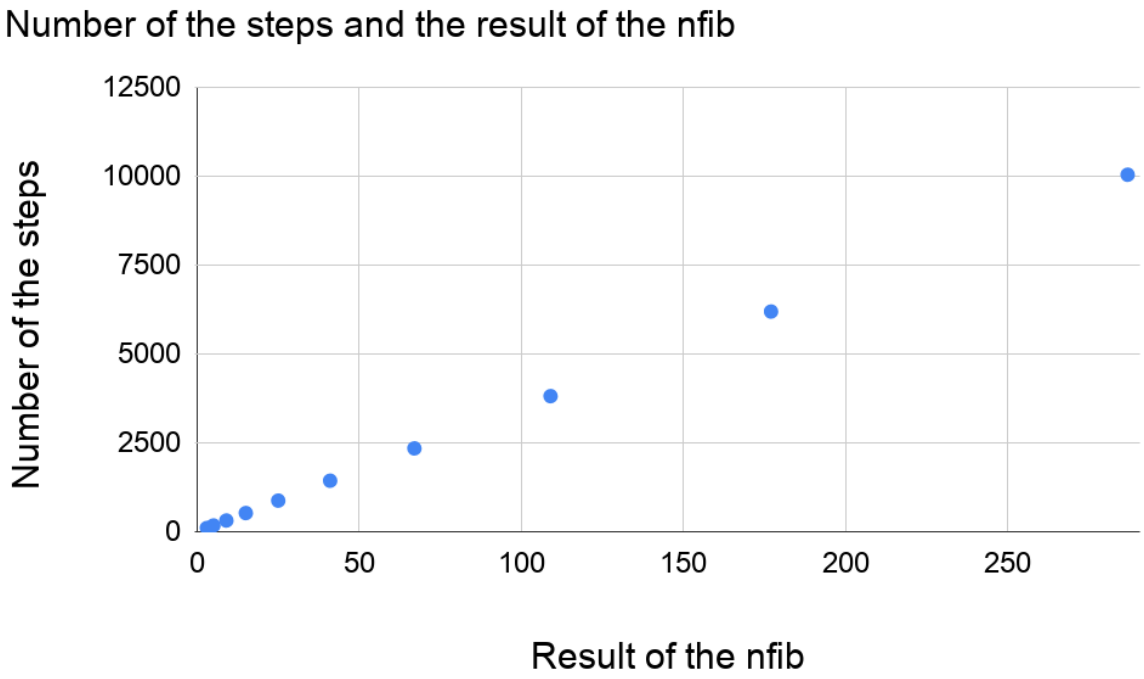


Figure 5.2: Number of steps and the result of the `nfib` in G-machine

5.2.2. Higher order functions

Then we are to check that our G-machine have achieved the *higher order function*. Here is a program that exploit SKI combinators:

```
Program = [
  ["k", "x", "y"     ] = "x",
  ["s", "x", "y", "z"] = (("x", "z"), ("y", "z")),
  ["id"              ] = (("s", "k"), "k"),
  ["main"            ] = ("id", 3)
].
```

Listing 5.5: SKI combinators

After the execution of the program, we will get the result `result(3)` since the we have applied `3` to the ideal function `id`. Notice that we actually did not need an I combinator: we didn't define the function `id` as `id x = x` (though it is certainly possible). Instead, we defined the `id` as `id = s k k` since the I combinator is equivalent with the `SKK` (or `SKS`). Furthermore we can observe that the `id` function has no argument and is composed of only `s` and `k`s. The `id` is certainly a function. Thus, we achieved to *return a function*. Also, notice the `s` function is taking two `k` in the body of the `id` function. Therefore we achieved to *take a function as an argument*. Finally, at the same place, we can observe that the `s` is not taking 3 arguments, there are only two `k`s, thus there we performed a *partial application*. This could be only achieved if we *curried* the function. Thus, we can see that our G-machine (and a compiler) have successfully implemented the currying.

Here is a more realistic example with higher order functions:

```
Program = [
  ["incr"] = ("+", "x"),
  ["main"] = ("incr", ("incr", 3))
].
```

Listing 5.6: Partial application to the addition function

Running this program, we shall obtain `result(5)` as the result. Notice that we defined the function `incr` not as `incr x = x + 1` but `incr = (+) 1`. This returns the function that takes one argument and increase it by 1. This shows the power of the functional programming. We can define a general function, the addition in this case, and apply some specific value, which is 1 in this case, and get a function for a specific purpose.

The example above applies the `incr` twice but how about applying it more times ? We definitely do not want to write `incr` more. Then we may use the `twice` function as the following example:

```
Program = [
  ["twice", "f", "x"] = ("f", ("f", "x")),
  ["incr"           ] = ("+", "x"),
  ["main"           ] = ((("twice", "twice"), "incr"), 3)
].
```

Listing 5.7: Partial application to the addition

This program yield `7` as the result. Since we have double the effect of the function, that doubles the effect of the function (`incr`). That is, the function `twice twice incr` is equivalent with `(+) 4`.

5.2.3. Non-strict evaluation

The examples we have seen so far can be all achieved with a strict languages. However, we have implemented a lazy evaluator. To prove this, we can do the following experiment.

Firstly, we shall consider the example, results with a runtime failure.

```
Program = [
  ["main"] = 1/0
].
```

Listing 5.8: Division by zero

This program can be compiled (Actually, whole this program is compiled in a *strict* context as `[pushInt(0),pushInt(1),div,update(0),pop(0),unwind]` (this is a code in a main global node). Notice that the division is in-lined as `div` instead of calling the function as `pushGlobal("/")`.). and executed but since we will eventually perform the division, we will never succeed to get the final result. The program terminates in the following state[3]:

```
code([div,update(0),pop(0),unwind]).
stack([!A1,!A0,!Main]).
dump([]).
nNum(0,!A0).
nNum(1,!A1).
nGlobal(0,[pushInt(0),pushInt(1),div,update(0),pop(0),unwind],!Main).
```

Listing 5.9: Runtime Error

[3]This is based on the output of the implemented G-machine but is simplified

We now have an empty dump, therefore we have finished to evaluate the arguments for a primitive operators, they have already reached to the Weak Head Normal Form, this time 1 and 0. Thus, we are to perform the division, only to find that it is impossible since we cannot divide the `1`, pointed by the top of the stack `!A1`, by `0`, pointed by the second element of the stack.

Then, how about the following program? Does this program reaches the final state and returns some value or stops in the execution phase ?

```
Program = [
  ["k", "x", "y"] = "x",
  ["main"       ] = (("k", 2), 1/0)
].
```

Listing 5.10: Applying the division-by-zero expression

If we evaluate this in a strict strategy, call by value, then we should perform the devision by 0 since the expression `1/0` is applied. However, if we evaluate this in a non-strict strategy, call by name or call by need, then we do not have to execute the division. Since the `K` combinator discards the second argument. Thus, we can obtain the final result `2` executing this program on the G-machine we have implemented. We can check this running the machine.

Here is a list of rules used in the transition of G-machine in the former program.

```
---->pushGlobal
---->unwindGlobalWithEmptyDump
---->pushInt
---->pushInt
---->pushGlobal
---->mkap
---->mkap
---->pushInt
---->pushGlobal
---->mkap
---->mkap
---->eval
---->unwindApp
---->unwindApp
---->unwindGlobalWithNonEmptyDump
---->push
---->eval
---->unwindInt
---->update
---->pop0
---->unwindInt
---->update
---->pop0
```

```
---->terminateWithNum
```

Notice that there is no `div` seen in the transition step. We certainly did not perform the division. This shows an advantage of the non-strict evaluation.

5.2.4. Call By Name evaluation strategy

In the previous example, we have seen that we have achieved the *non-strict* evaluation but we have not yet checked whether we did accomplished the *call by need* evaluation other than the *call by name*. To show this, we can experiment with the following program:

```
Program = [
  ["double", "x"] = "x" + "x",
  ["main"       ] = ("double", 1 + 2)
].
```

Listing 5.11: Double

which will result `result(6)`.

Actually, this program is the same program we have seem in Section 4.1. In short, we are to share the sub-expression `1 + 2` and should not calculate this twice. Thus, we will be performing the addition twice; in the `double` function, `3 + 3`, and for the applied expression, `1 + 2`. Here is a list of the transition rules of G-machine.

```
----->pushGlobal
----->unwindGlobalWithEmptyDump
----->pushInt
----->pushInt
----->pushGlobal
----->mkap
----->mkap
----->pushGlobal
----->mkap
----->eval
----->unwindApp
----->unwindGlobalWithNonEmptyDump
----->push
----->eval
----->unwindApp
----->unwindApp
----->unwindGlobalWithNonEmptyDump
----->push
----->eval
----->unwindInt
----->push
```

```
----->eval
----->unwindInt
----->add           % performing the "1 + 2"
----->update
----->pop0
----->unwindInt
----->push
----->eval
----->unwindInt
----->addSameNode   % performing the "3 + 3"
----->update
----->pop0
----->unwindInt
----->update
----->pop0
----->terminateWithNum
```

Notice that we have performed the addition only twice, at the `add` and the `addSameNode`. We have defined the rules differently since the former will match the different nodes and the latter will match the same nodes. The rules are the following:

```
% Perform the addition with the different nodes as the arguments.
add @@
code = [add|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N = N0 + N1
    | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum
(N).

% Perform the addition with the same node as the arguments.
addSameNode @@
code = [add|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- N = N0 + N0
    | code = Is, stack = [!A|S], !A0 = nNum(N0), !A = nNum(N).
```

Listing 5.12: The transition rule for the addition in G-machine

We can observe that the G-machine was not sharing an expression in the addition of the first time, when it is performing `1 + 2`, but it was sharing the expression when we perform the second addition `3 + 3`. This can be also checked with the visualizer, *Graphene*[11]. Here is a shot of the visualizer.

The result of the $1 + 2$, 3 is shared

The next instruction (`addSameNode`) generates the 6 ($= 3 + 3$) and push its address on the top of the stack.

5.2.5. Structured Data: Infinite List

So far, we have observed that we have accomplished the non-strict evaluation. Here comes a more practical examples: *an infinite list*.

The following example calculates the factorial number using an infinite list `[1, 2, 3, 4, 5, ...]`. The function `from` generates the infinite list. And the `timesN list n` multiples the first `n` of the list `list`. Here, `eConstr(2, 2)` is a data *constructor* and can be read as `Cons` for this time as we have already described in Section 4.3. The `alt(2, ["h", "t"], "h" * (("timesN", "t"), "n" - 1))` performs the *pattern matching*. This checks the *tag* of the structured data and binds the components of the data to the variables, for this time, `"h"` and `"t"`.

```
Program = [
  ["from", "n"] = ((eConstr(2, 2), "n"), ("from", "n" + 1)),
  ["timesN", "list", "n"] = if( 0 < "n",
  case( "list",
  [ alt(1, [], 1),      % Nil
    alt(2, ["h", "t"], % Cons
    "h" * (("timesN", "t"), "n" - 1)
    ) ]
  ),
  1
  ),
  ["main"] = (("timesN", ("from", 1)), 5)
].
```

Listing 5.13: Calculating the factorial using an infinite list

Again, we can obtain `result(120)`.

A more interesting example exploiting an infinite list is the *Sieve of Eratosthenes*. This is a well-known algorithm to obtain prime numbers. The program is the following:

```
Program = [
  ["main"] = (("getNth", 5), ("sieve", ("from", 2))),
  ["from", "n"] = ((eConstr(2, 2), "n"), ("from", "n" + 1)),
  ["sieve", "xs"] = case("xs", [
    alt(1, [], eConstr(1, 0)),
    alt(2, ["p", "ps"],
      ((eConstr(2, 2),
        "p"),
       ("sieve", (("filter", ("nonMultiple", "p")), "ps")))
    )]),
  ["filter", "predicate", "xs"] = case("xs", [
    alt(1, [], eConstr(1, 0)),
    alt(2, ["p", "ps"],
      let(["rest" = (("filter", "predicate"), "ps")],
        if(("predicate", "p"),
           ((eConstr(2, 2), "p"), "rest"),
           "rest"
        ))
    )]),
  ["nonMultiple", "p", "n"] = (("n" / "p") * "p" \= "n"),
  ["getNth", "n", "xs"] = case("xs", [
    alt(1, [], -1), % Error
    alt(2, ["p", "ps"],
      if(0 < "n",
         (("getNth", "n" - 1), "ps"),
         "p"
      )
    )])
].
```

Listing 5.14: The Sieve of Eratosthenes

Where `getN` returns the `5`th element of the sieve, `from` generates an infinite list `[2, 3, 4, 5, 6, ...]`, `sieve` excludes the multiple of the head from the given list, `filter` filters an list with a function `predicate` and `nonMultiple` determines whether the second argument is a multiple of the first.

We have also counted the number of the steps of our G-machine took in the former program. Notice that this is not the number of the steps of HyperLMNtal. It took 36504 steps to obtain the 33th prime number, 131. Here we can observe that this algorithm takes about $\mathcal{O}(N \log(\log N))$ steps in Figure 5.4.

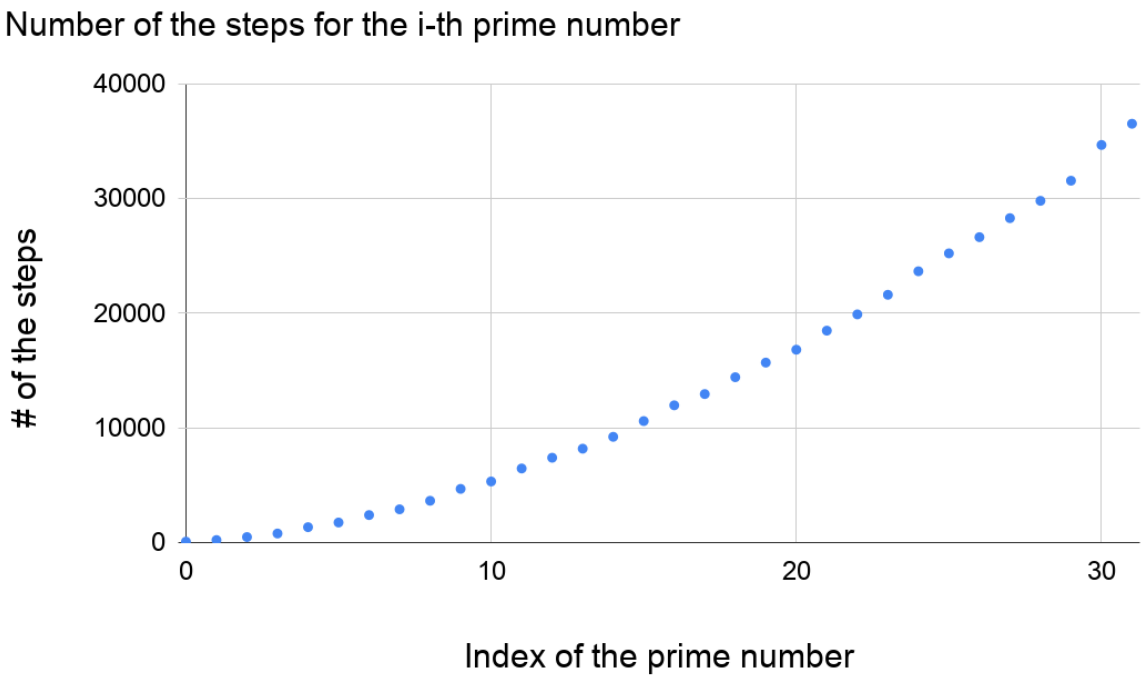


Figure 5.3: Number of the steps of the implemented G-machine and the index of the prime number

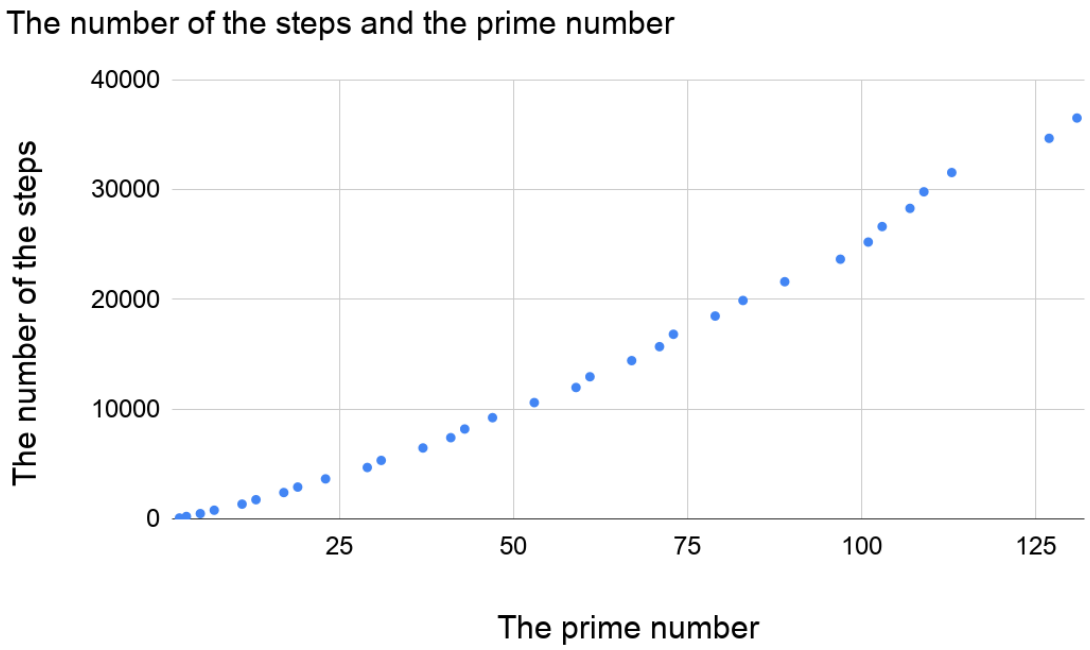


Figure 5.4: Number of the steps of the implemented G-machine and the prime number obtained

Chapter 6.

Further Discussion

In this chapter, we will give a further discussion based on the experience of the implementation in the former chapter.

6.1. Proposal for abbreviation rules for a hyperlink

We observed that the hyperlinks often connect 2 atoms in the practice. In LMNtal, a local link, a link with just 2 end points, can be abbreviated. For example, `a(X, Y, Z), b(W, Y)` can be abbreviated as `a(X, b(W), Z)`. Thus, we may introduce the same strategy into hyperlinks. For example, as follows:

In the proposed HyperLMNtal syntax,

$$\nu !H.(p(X_1, \ldots, !H, \ldots, X_n), q(Y_1, \ldots, Y_{n-1}, !H))$$

can be abbreviated as

$$p(X_1, \ldots, X_{i-1}, !q(Y_1, \ldots, Y_{n-1}), X_{i+1}, \ldots, X_n)$$

Using this abbreviation, for example, a program `a(!X), b(!X)` can also be written as `a(!b)`. Furthermore, a hyperlink connected list

```
!X1 = [A1|!X2],
!X2 = [A2|!X3],
!X3 = [A3|!X4],
!X4 = [].
```

can be abbreviated as

```
!X1 = [A1|![A2|![A3|![]]]].
```

We may introduce a more ambitious abbreviation scheme for a list. For example, we may like to write the former list as

```
!X = [A1!A2!A3![]]
```

6.2. Abstraction in the visualizer and the output of the HyperLMNtal runtime environment

In the previous chapter, we have seen the screenshot of the visualizer[11] and the output of the HyperLMNtal runtime environment[15]. However, they were actually simplified for this purpose. It was very hard to observe the desired information when using these tools. To tell the truth, we did not use the visualizer when implementing this. Here is the screenshot of the Graphene in Figure 6.1. Its is a state in the calculation process of factorial of 4.

As you can see, this is that easy-to-understand output. This is simply because that it output too much information. Thus, we can improve this by abstracting the state and reducing the information.

One possible way to achieve this is to write some rules for the abstraction and apply them to each of the states in the execution steps. For example, in G-machine, we have a global node, which has a pretty long code, a list of instructions, which does not changes through the execution process. Therefore, we may like to discard all the information about the global nodes. Then for example, we can have a rule as the following:

```
clearGlobalNode @@
!A = nGlobal(Arity, Code)
  :- int(Arity), ground(Code)|.
```

Applying the rule will let the output much clearer. However, we may want to restore the previous state.

For example, we may do not want to delete the global node entirely but want to simplify to one atom with the following rule and want to restore it if the user clicked the simplified atom in the visualizer.

```
abstractGlobalNode @@
!A = nGlobal(Arity, Code)
  :- int(Arity), ground(Code)|
    | !A = nGlobal.
```

Of course we cannot restore the previous graph if we simply discarded it. We should probably store the previous graph. However, even we did so, it is not obvious to restore the graph that has degenerated to the atom. Well there is no certain way in my thought but maybe the technique in the *Reversible computing* help. We may like to restrict the degenerating rule syntactically, for example, the grammar used in the LMNtal Shape Type[17] or CSLMNtal[18], in order to achieve this.

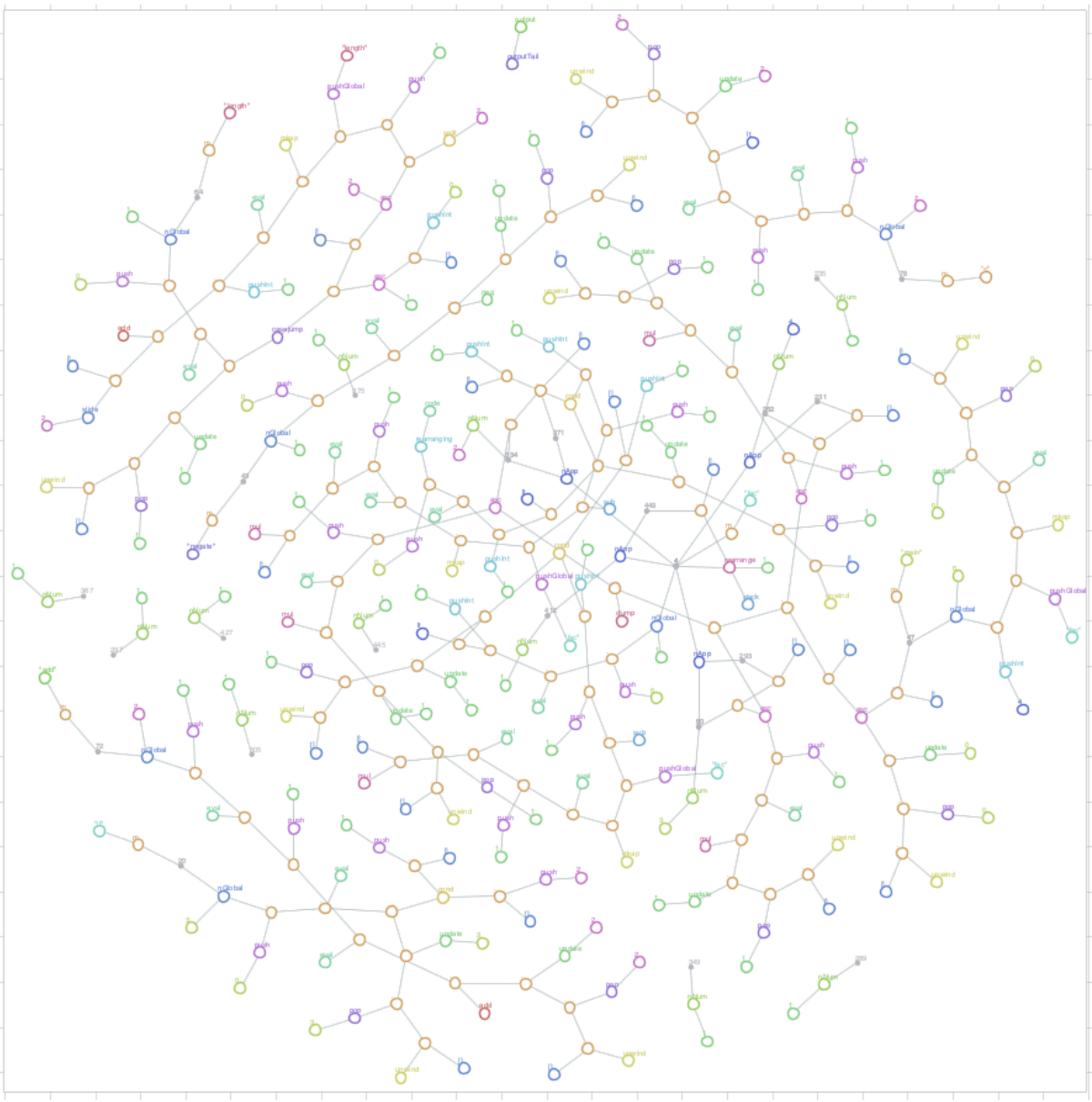


Figure 6.1: A state in the calculation process of factorial of 4

6.3. Else If statement in a rule

In LMNtal, we cannot use the *otherwise* pattern. That is, there does not exists the *else if statements*. The runtime environment SLIM[15] does attempt the matching and the rewriting from the top of the rules to the bottoms. And this, else if statement, can be actually achieved by just denoting rules downwards. However, this is not guaranteed as a language feature: this is just a behavior of the processing system.

Therefore, in the implementation of the compiler for G-machine, we have written all the *otherwise* patterns. However, this is very inconvenient.

Also, we have observed that the existence of the rule that matches partially affects the execution time heavily. The program used in the experiment if the following:

```
incr(n(1)).

% Partially matches
irrelevantRule1 @@
incr(a) :- .

% Does not matches at all
irrelevantRule2 @@
decr(a) :- .

increment @@
incr(n(N)) :- N < 1000000, N1 = N + 1 | incr(n(N1)).
```

Listing 6.1: The benchmark program

In our experiment, the program is compiled with the options `--slimcode -O3 --use-swaplink`. And experimented in the environment shown in the Table 6.1. Figure 6.2 shows the execution time of the program.

Table 6.1.: Experiment Conditions

Processor	2.2 GHz Dual-Core Intel Core i7
Memory	8 GB 1600 MHz DDR3
LMNtal Version	LMNtal Compiler 1.50
SLIM Version	2.4.0

We can clearly see that the number of the partially matching rule make the performance worse. This is a very big issue since we often write the partially matching rules. For example, in the G-machine we have implemented, we firstly search for the atom `code` and this eventually succeeds, since we always have the atom. However, we might backtrack

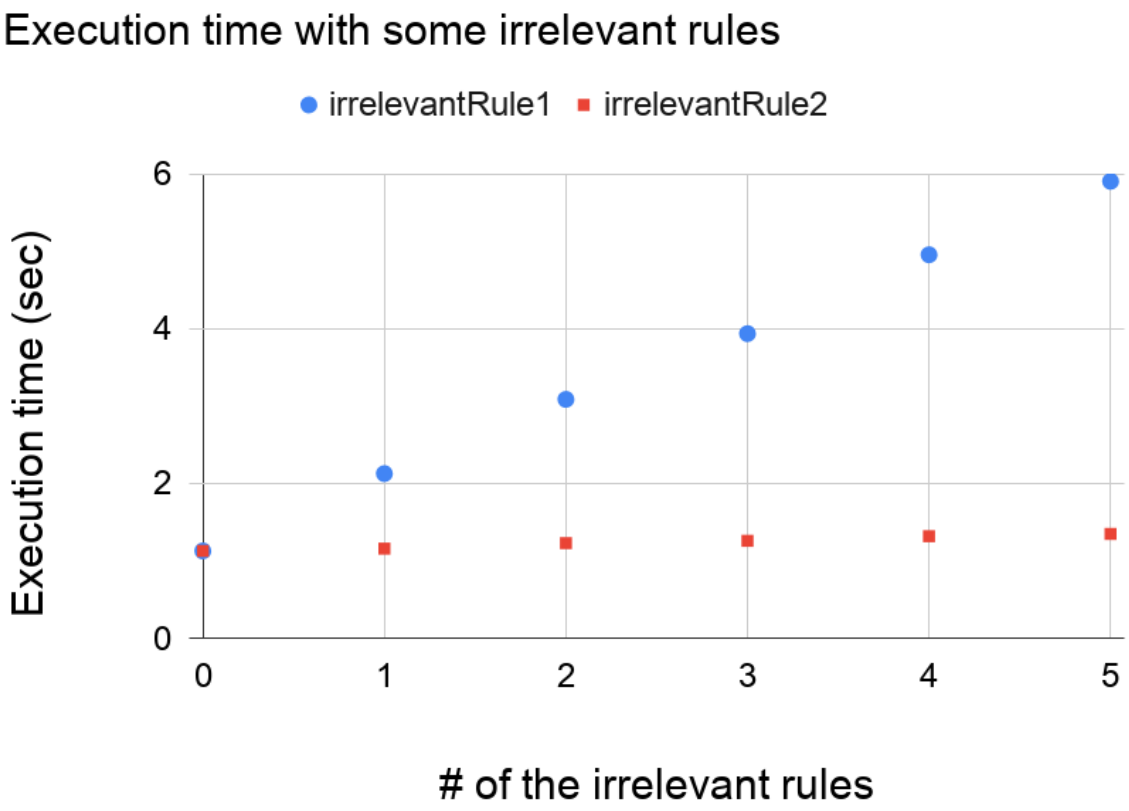


Figure 6.2: Execution time with the arbitrary number of the irrelevant rules

(Rule)	*rule*	::=	P `:-` $[rule_1$ `||` ... `||` $rule_n]$	$n > 1$	Else If statement
		|	P `:-` P `@@` *RuleName*		Rule

Figure 6.3: Syntax of Flat HyperLMNtal

depending on the head of the list of the code, for example, `unwind`, `push`, `add`, etc. If we failed to apply the rule, then it was a rule that partially matched. And this happens so many times. Thus, this problem affects the overall performance.

Incorporating the *else if statement* gives one solution to this problem. For the example above, we do not want to search for the `code` again and again. CSLMNtal[18] has a rule with multiple guards but we want to exploit the else if statement in more general matching. Thus, we may introduce a new syntax as the Figure 6.3:

For example, the following program

```
% Make application spines
makeSpineApp @@
H = [ makeSpine((E1, E2)) | T]
:- H = [ makeSpine(E1), E2 | T].

% The default cases (other than application) for the "makeSpine"
makeSpineDefaultInt @@
R = makeSpine(Int)
:- int(Int) | R = Int.

makeSpineDefaultVar @@
R = makeSpine(Var)
:- string(Var) | R = Var.

makeSpineDefaultCase @@
R = makeSpine(case(E, Alts))
:- R = case(E, Alts).

makeSpineDefaultIf @@
R = makeSpine(if(E1, E2, E3))
:- R = if(E1, E2, E3).

makeSpineDefaultPlus @@
R = makeSpine(E1 + E2)
:- R = E1 + E2.

makeSpineDefaultTimes @@
R = makeSpine(E1 * E2)
:- R = E1 * E2.

makeSpineDefaultMinus @@
R = makeSpine(E1 - E2)
:- R = E1 - E2.

makeSpineDefaultDiv @@
```

```
R = makeSpine(E1 / E2)
:- R = E1 / E2.

makeSpineDefaultNeg @@
R = makeSpine(negate(E))
:- R = negate(E).

makeSpineDefaultLessThan @@
R = makeSpine(E1 < E2)
:- R = (E1 < E2).

makeSpineDefaultNeq @@
R = makeSpine(E1 \= E2)
:- R = (E1 \= E2).

makeSpineDefaultLet @@
R = makeSpine(let(Defns, E))
:- R = let(Defns, E).

makeSpineDefaultLetRec @@
R = makeSpine(letrec(Defns, E))
:- R = letrec(Defns, E).

makeSpineDefaultConstr @@
R = makeSpine(eConstr(Tag, Arity))
:- R = eConstr(Tag, Arity).
```

Listing 6.2: makeSpine

can be rewritten as the following

```
% Make spines
H = [ makeSpine(Exp) | T] :- [
  % Make application spines
  Exp = (E1, E2) :- H = [ makeSpine(E1), E2 | T] @@ makeSpineApp
  % The default cases (other than application) for the "makeSpine"
  || :- H = [Exp | T] @@ makeSpineDefault
]
```

Listing 6.3: makeSpine with an else if syntax

which is certainly easy to read and write.

We can implement this easily; just adding a `branch` instruction to the HyperLMNtal runtime environment SLIM.

6.4. Directed HyperLMNtal

The hyperlinks in HyperLMNtal are undirected. However, considering to deal with the directed hyperlinks is important from the view of (i) the efficiency (the one-way pointer is more efficient that the mutual links) and (ii) the programmer to program with directed edges. Thus, we introduce a language extended HyperLMNtal, Directed HyperLMNtal.

Figure 6.4 is a instructions obtained by compiling the HyperLMNtal program

```
a(!H), b(!H) :- .
```

. Notice that we should perform `findatom` twice to obtain the atoms "a" and "b" which are connected by a hyperlink. That is, we search two atoms randomly and checks whether it is connected to the same hyperlink (by the instruction `samefunc`). This is obviously inefficient. We should let this to execute `findatom` once and then `dereference` through the connected hyperlink. To implement this, it will be much easier if we let hyperlink as an one-way pointer, which is referred as a backward hyperarc in [19], rather than an undirected hyper edge.

Capability typing[19] gives one solution to this problem but it is too much strict. Since the capability typing does not allow a port to be connected to a hyperlink that the number of the end-points changes through the calculation process.

Example 6.1: List traversal

he following program is il-typed in the capability typing system but it does not yields any null/dangling pointers and should be allowed.

$$\overline{R} \mapsto walk(Prev), \overline{Prev} \mapsto cons(Elem, Next)$$
$$\vdash \overline{R} \mapsto walk(Next), \overline{Prev} \mapsto cons(Elem, Next)$$

Thus, we shall firstly introduce the Directed HyperLMNtal, which simply let hyperlinks in our HyperLMNtal to be directed. Then discuss the condition that the Directed HyperLMNtal should satisfy. Again, we shall call a hyperlink as a link for the simplicity from now on in this section.

6.4.1. Syntax and operational semantics of Directed HyperLMNtal

X denotes a link name and p denotes an atom name. The only reserved name is $\mapsto$. The syntax is given in Figure 6.5. An atom $\overline{X} \mapsto Y$ is called a *indirection*. Given a rule $(P \vdash Q)$, rules must not appear in P and it should satisfy $fn(P) \supseteq fn(Q)$. Notice that the name of the head and the tail of the link are in the same name space. For example, the free link names of $p(X, Y, \overline{Z})$ is $\{X, Y, Z\}$.

We define the relation $\equiv$ on processes as the minimal equivalence relation satisfying the rules shown in Figure 6.6. Where $P[Y/X]$ is a link name substitution that replaces

```
spec        [1, 5]
findatom    [1, 0, 'a'_1]
derefatom   [3, 1, 0]
ishlink     [3]
isunary     [3]
findatom    [2, 0, 'b'_1]
derefatom   [4, 2, 0]
isunary     [4]
samefunc    [3, 4]
jump        [L103, [0], [1, 2, 3, 4], []]
```

Figure 6.4: Example of an intermediate code of HyperLMNtal

(Process)	P	$::=$	$\mathbf{0}$		Null
		$\mid$	$p(L_1, \ldots, L_m)$	$m \geq 0$	Atom
		$\mid$	(P, P)		Molecule
		$\mid$	$\nu X.P$		Link Creation
		$\mid$	$(P \vdash P)$		Rule
(Link)	L	$::=$	X		Tail
		$\mid$	$\overline{X}$		Head

Figure 6.5: Syntax of Directed HyperLMNtal

(E1) $(\mathbf{0}, P) \equiv P$

(E2) $(P, Q) \equiv (Q, P)$

(E3) $(P, (Q, R)) \equiv ((P, Q), R)$

(E4) $P \equiv P' \Rightarrow (P, Q) \equiv (P', Q)$

(E5) $P \equiv Q \Rightarrow \nu X.P \equiv \nu X.Q$

(E6) $\nu X.(\overline{X} \mapsto Y, P) \equiv \nu X.P[Y/X]$
where $X \in fn(P) \vee Y \in fn(P)$

(E7) $\nu X.\nu Y.\overline{X} \mapsto Y \equiv \mathbf{0}$

(E8) $\nu X.\mathbf{0} \equiv \mathbf{0}$

(E9) $\nu X.\nu Y.P \equiv \nu Y.\nu X.P$

(E10) $\nu X.(P, Q) \equiv (\nu X.P, Q)$
where $X \notin fn(Q)$

Figure 6.6: Structural congruence on Directed HyperLMNtal processes

all free occurrences of X with Y (and $\overline{X}$ with $\overline{Y}$). If a free occurrence of X occurs in a location where Y would not be free, α-conversion may be required.

We define the reduction relation $\longrightarrow$ on processes as the minimal relation satisfying the rules in Figure 6.7.

6.4.2. Conditions for Directed HyperLMNtal

Functional (right-unique) condition

To let a hyperlink to be a backward hyperarc, we should let the head of the link to occur just once. Thus, we can add a following condition.

> **Functional condition:** For all link named $X \in fn(P)$ in a process P, the *head* of the link $(\overline{X})$ must not occur free more than once in P.

Notice the congruence rules should be applied preserving this condition. For example, we cannot transform $\nu X.p(\overline{X})$ to $\nu X.(\overline{X} \mapsto Y, p(\overline{X}))$ with the reversed rule of (E6).

By introducing this condition, the indirection (link fusion) is no longer symmetry. Since we have proved the symmetry of the fusion via the process $\nu Z.(Z \bowtie X, Z \bowtie Y)$, which is now disallowed in Directed HyperLMNtal because $\nu Z.(\overline{Z} \mapsto X, \overline{Z} \mapsto Y)$, in which the head of a link Z, $\overline{Z}$, occurs twice, is prohibited[1].

[1]However, we cannot say it is asymmetry just because we cannot prove the symmetricity.

$$\text{(R1)} \quad \frac{P \longrightarrow P'}{(P, Q) \longrightarrow (P', Q)}$$

$$\text{(R2)} \quad \frac{P \longrightarrow P'}{\nu X.P \longrightarrow \nu X.P'}$$

$$\text{(R3)} \quad \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)} \quad (P, (P \vdash Q)) \longrightarrow (Q, (P \vdash Q))$$

Figure 6.7: Reduction relation on Directed HyperLMNtal processes

Serial (left-total) condition

We definitely do not want to allow null/dangling pointers. To achieve this, we should add a condition as the following.

Serial condition 1 (pre-revised): For all the process $\nu X.P$ where $X \in fn(P)$ in a given process, the *head* of the X must exist in P .

However, this is insufficient considering the congruence rules. For example, $\nu X.(\overline{X} \mapsto X, p(X))$ satisfies the both functional and serial conditions, but the congruent process $\nu X.p(X)$ obtained by applying the rule (E6) does not satisfy this. To resolve this, we should first consider a process without indirection. The serial condition is now revised as the following.

Serial condition 2 (revised):
This condition is satisfied if the *head* of the link X exists in P and the indirection $\overline{X} \mapsto Y$ (whose X occurs free in P) does not occur in P for all the process $\nu X.P$ where $X \in fn(P)$ in the given process.
Also, Q satisfies this condition if P satisfies this condition where $P \equiv Q$.

Intuitively, this means that if you follow the link whose scope is closed from a port of an atom except an indirection possibly via indirections, you will always reach a port (head of the link) of an atom except an indirection. We shall call it "reachable".

An additional condition for a rule

We should add an additional condition to store the serial condition after reduction.

An additional condition for a rule: If the *head* of a link X occurs free in P, the *head* of the link X must also occur free in Q.

Theorem 6.1: The preservation of serial condition in a reduction

P where $Q \mapsto P$ satisfies serial condition if the rule used in (R4) does not have an indirection.

Proof 6.1

Firstly, we do not have to worry about the local link (the link but with the free link name). The local links on LHS should match local links outside of the rule "completely". And there should be no chance that the link would be unreachable. Since we cannot even start traversing. The local links on RHS will generate a new local links, which should have satisfied the serial condition. Thus it will obviously satisfy the condition after generation by the rule. For the free links, we have to need to pay more attention. Is there a chance that the head of a free link would disappear? Well, this is denied by the condition we added on a rule; We are to store the head of the links in the rewriting process. If we are allowed to have an indirection in the rule, then it may disappear after rewriting and the process may cannot satisfy the serial condition. However, we just have prohibited this and that would not occur.

An example that collapses

Firstly, we shall introduce an abbreviation rule for the convenience.

$\nu X.(p(\ldots, X, \ldots), q(\ldots, \overline{X}))$ where the number of occurrences of the link named X is 2 can be written as $p(\ldots, q(\ldots), \ldots)$.
For example, $\nu X.(\overline{R} \mapsto X, append(L_1, L_2, \overline{X}))$ can be abbreviated as $\overline{R} \mapsto append(L_1, L_2)$

We have shown that a rule without an indirection will be a safe to rewrite a process. What about a rule with indirection ? Sadly, it yields an annoying problem.

Example 6.2

The append of the nil and the list should just return the latter list.

$$\overline{R} \mapsto append(nil, L) \vdash \overline{R} \mapsto L$$

What happens if we apply the rule to the process ?

$$\overline{R} \mapsto append(nil, R), p(R)$$

Then it will be like the following.

$$\begin{aligned}
&\nu R.(\overline{R} \mapsto \mathit{append}(\mathit{nil}, R), p(R), (\ldots \vdash \ldots)) \\
&\equiv_{(\mathrm{E7})} \nu R.\nu L.(\overline{L} \mapsto R, \overline{R} \mapsto \mathit{append}(\mathit{nil}, L), p(R), (\ldots \vdash \ldots)) \\
&\longrightarrow \nu R.\nu L.(\overline{L} \mapsto R, \overline{R} \mapsto L, p(R), (\ldots \vdash \ldots)) \\
&\equiv_{(\mathrm{E7})} \nu R.(\overline{R} \mapsto R, p(R), (\ldots \vdash \ldots)) \\
&\equiv_{(\mathrm{E7})} \nu R.p(R), (\ldots \vdash \ldots)
\end{aligned}$$

Here the link R is now unreachable, which is surely undesirable.

The solutions

As shown in former example, the indirection from a free link to a free link is somewhat dangerous. How should we avoid this ?

There should be at least 2 solutions.

- Prohibit the aliasing on the right-hand side of the rule.

  Then, for example, the above append-nil-rule can be rewritten with 2 rules

  $$\begin{aligned}
  &(\overline{R} \mapsto \mathit{append}(\mathit{nil}, \mathit{cons}(H, T)) \vdash \overline{R} \mapsto \mathit{cons}(H, T)), \\
  &(\overline{R} \mapsto \mathit{append}(\mathit{nil}, \mathit{nil}) \vdash \overline{R} \mapsto \mathit{nil})
  \end{aligned}$$

  This is the simplest solution though it limits the expressiveness power.

- Check the rules (and the processes) can be well-typed with the capability-typing as following.

  Then, for example, the above append-nil-rule will be il-typed.

  $$\begin{aligned}
  &\overline{R} \mapsto \mathit{append}(\mathit{nil}, R), p(R), && \textcircled{1} \\
  &(\overline{R} \mapsto \mathit{append}(\mathit{nil}, L) \vdash \overline{R} \mapsto L) && \textcircled{2}
  \end{aligned}$$

  By KCL, this should satisfy

  $$\begin{aligned}
  &-\mathit{append}/3 + \mathit{append}/2 + p/1 = 0 && \text{by } \textcircled{1} \\
  &\mathit{append}/3 = \mapsto /1 && \text{by } \textcircled{2} \\
  &\mathit{append}/2 = \mapsto /2 && \text{by } \textcircled{2}
  \end{aligned}$$

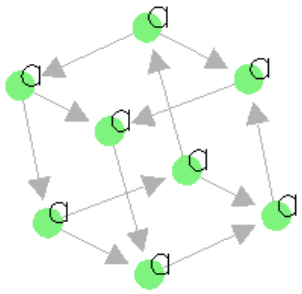
  By Conn, $\mapsto /1 = \mapsto /2$, therefore, $p/1 = 0$, which is il-typed.

6.4.3. Implementation

We have implemented a proof of concept of Directed HyperLMNtal. The Github repository is `https://github.com/sano-jin/vertex`. Current implementation lacks the capability typing system and restricted so that the head of the link can only appear at the left-hand side of `->` ($\mapsto$). However, we have implemented programs including G-machine and FL interpreter and are confident with its usefulness.

Visualizer :

```
\X Y Z W U V S T.(
   X -> a(Z, W),
   Y -> a(W),
   Z -> a(U, V),
   W -> a(V),
   U -> a(S, T),
   V -> a(T),
   S -> a(X, Y),
   T -> a(Y)
)
```

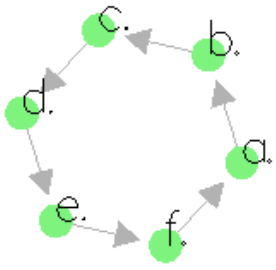


Non-deterministic execution and the visualizer for the state transition graph:

```
% Cycling program
%
% This program never terminates with the
   'normal execution'
% However, the 'non-deterministic
  execution' will result the finite states
% and it shows the ring forming
  transitons.

% The initial graph
a.

% Cycing rules.
a :- b.
b :- c.
c :- d.
d :- e.
e :- f.
f :- a.
```



Chapter 7.

Summary and Conclusion

7.1. Summary

The contributions of this research are basically the following 2:

- We formalized the syntax and the semantics of HyperLMNtal
- We implemented G-machine and a compiler for it in HyperLMNtal

Formalizing the syntax and the semantics of HyperLMNtal

HyperLMNtal is extended from the graph rewriting language/calculus model LMNtal. However, it lacked the rigid definition: it was more an extension of the implementation than on the calculus model. The semantics of LMNtal features fine-grained concurrency based on local rewriting. However, since we could not determine the locality of the hyperlink in HyperLMNtal, we couldn't incorporate it into the LMNtal semantics. Thus, we first introduced a scope (link creation) adopted from the π-calculus[10] and defined the locality of a hyperlink to formalize the syntax/semantics. Now, HyperLMNtal is not just a programming language extended from the basic calculation model, but also a concurrent calculation model based on strict and formal definitions.

Implementing G-machine in HyperLMNtal

In our research, we implemented a compiler which translates the source language, the core language, into the execution code for G-machine and G-machine, in HyperLMNtal. We have succeeded to implement the compiler in 404 lines and G-machine in 570 lines and showed that we can implement the language processing system that handles complex data structures in graph rewriting language tersely. In addition, we achieved to visualize G-machine using the HyperLMNtal visualizer[11][12].

7.2. Future tasks

We can possibly prove the correctness of the encodings of λ-calculus, System F and so on in HyperLMNtal with the formalized semantics. For example, in the encodings of λ-calculus,

α-equivalent expressions should be encoded to the structural congruent processes. And the β-converted expression should be encoded into those of the reduced process in our new semantics. We believe that we can give a more formal and simple proof without referring the graph homomorphism.

For the G-machine we have implemented, we can firstly eliminate the duplication and a deletion of a large graph and make it more efficient: currently we are copying the "code" each time when we `pushGlobal` but this can be eliminated by replacing the "code" with a "pointer" to the code; instruction pointer. Then we may compare with the implementation in other languages. Also, there exists a more efficient G-machines, spineless G-machine[20] and spineless tag-less G-machine[21] (this is the machine used in Glaskaw Haskell Compiler) and a Parallel G-machine [22] so we may like to implement these. With a new semantics, HyperLMNtal is now a concurrent calculus model with a fine-grained concurrency. Thus, we believe that it is a great significance to demonstrate how parallel G-machine can be modeled in HyperLMNtal.

Acknowledgments

Throughout the writing of this paper I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Ueda. I could not even start my research without your patient support.

In addition, I would like to thank my parents for the encouragement.

Finally, I would like to thank my colleagues, Masaki Nakata, Kunihiro Hata, Uzawa Seishiro and Kota Fukui and also other members in our lab including Yamamoto-san, Yamada-san, Tsunekawa-san and Walter-san.

January 2021, Jin Sano

Bibliography

[1] 小久保晃. LMNtal 処理系を利用したコンパイラの設計と実装, 2014.

[2] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. WORLD SCIENTIFIC, 1997.

[3] Richard B. Kieburtz. The g-machine: A fast, graph-reduction evaluator. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pp. 400–413, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[4] Kazunori Ueda and Seiji Ogawa. Hyperlmntal: An extension of a hierarchical graph rewriting model. *KI - Kunstliche Intelligenz*, Vol. 26, No. 1, pp. 27–36, 2 2012.

[5] Kazunori Ueda. Lmntal as a hierarchical logic programming language. *Theoretical Computer Science*, Vol. 410, No. 46, pp. 4784 – 4800, 2009. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

[6] Christopher Bak. Gp 2: Efficient implementation of a graph programming language. September 2015.

[7] Detlef Plump. From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, Vol. 88, pp. 154 – 173, 2017.

[8] Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *SIGSAM Bull.*, No. 15, p. 13–27, July 1970.

[9] Ian Zerny. On graph rewriting, reduction, and evaluation in the presence of cycles. *Higher Order Symbol. Comput.*, Vol. 26, No. 1–4, p. 63–84, December 2013.

[10] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, USA, 2001.

[11] 谷口直輝. 動的階層グラフ可視化ソフトウェア Graphene の設計と実装. 修士論文, 早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻, 2014.

[12] 綾野貴之, 堀泰祐, 岩澤宏希, 小川誠司, 上田和紀. 統合開発環境による LMNtal モデル検査. コンピュータ ソフトウェア, Vol. 27, No. 4, pp. 197–214, 2010.

[13] 上田和紀, 加藤紀夫. 言語モデル LMNtal. コンピュータ ソフトウェア, Vol. 21, No. 2, pp. 126–142, 2004.

[14] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀. 階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal. コンピュータ ソフトウェア, Vol. 25, No. 1, pp. 124–150, 2008.

[15] 石川力, 堀泰祐, 村山敬, 岡部亮, 上田和紀. 軽量な LMNtal 実行時処理系 SLIM の設計と実装. 情報処理学会第 70 回全国大会講演論文集, pp. 153–154, 2008.

[16] SL Peyton Jones, DR Lester, and Simon Peyton Jones. *Implementing functional languages: a tutorial*. Prentice Hall, 1 1992.

[17] 吉元佑介. グラフ書換え言語 LMNtal におけるグラフ型検査およびルール型保存性検査. 修士論文, 早稲田大学基幹理工学研究科情報理工・情報通信専攻, 2014.

[18] 奈良耕太. 再帰的な文脈パターンマッチング機能を持つグラフ書き換え言語の設計と効率的な実装手法. 修士論文, 早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻, 2015.

[19] Kazunori Ueda. *Towards a Substrate Framework of Computation*, pp. 341–366. Springer Berlin Heidelber, Berlin, Heidelberg, 2014.

[20] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, p. 244–258, New York, NY, USA, 1988. Association for Computing Machinery.

[21] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, Vol. 2, pp. 127–202, 04 1992.

[22] Hugh Kingdon, David R. Lester, and Geoffrey L. Burn. The hdg-machine: A highly distributed graph-reducer for a transputer network. *Comput. J.*, Vol. 34, No. 4, p. 290–301, August 1991.

Appendix A.

Source codes

A.1. Implementation of the compiler for G-Machine in HyperLMNtal

```
/*
  G−Machine Compiler on LMNtal
*/


{ compiler.
    {'$callback'(zerostep).

  % Some helper functions for the compiler.

  appendCons @@
  R = append([H|T], L)
  :- R = [H | append(T, L)].

  appendNil @@
  R = append([], L)
  :- R = L.

  % zip with an incresing number N
  zipNCons @@
  R = zip([V|Vs], N)
  :- Nplus1 = N + 1
     | R = [asc(V, N) | zip(Vs, Nplus1) ].

  zipNNil @@
  R = zip([], N)
  :- int(N)
     | R = [].

  addNtoRightElemOfAscListCons @@
  R = addNtoRightElemOfAscList([asc(V, Vi) | T], N)
  :- ViPlusN = Vi + N
     | R = [asc(V, ViPlusN) | addNtoRightElemOfAscList(T, N)].

  addNtoRightElemOfAscListNil @@
  R = addNtoRightElemOfAscList([], N)
  :- int(N)
```

```
    | R = [].

reverse @@
R = reverse(List)
:- R = revAppend(List, []).

revAppendCons @@
R = revAppend([H|T], Acc)
:- R = revAppend(T, [H|Acc]).

revAppendNil @@
R = revAppend([], Acc)
:- R = Acc.

length_init @@
R = length(L)
:- R = length(L, 0).

length_cons @@
R = length([H|L], N)
:- Nplus1 = N + 1, ground(H)
   | R = length(L, Nplus1).

length_nil @@
R = length([], N)
:- R = N.

mapDefVarCons @@
R = mapDefVar([Var = Def|T])
:- ground(Def)
   | R = [Var | mapDefVar(T)].

mapDefVarNil @@
R = mapDefVar([])
:- R = [].


% Main compiling functions starts from here

% Compiles the super combinators.
% This turns the super combinator expression to the global node
% with the code (the compiled body of the super combinator) embedded in it.
% Also, adds the
% "<name of the super combinator> -> <the address of the global node>"
% binding "m(Name, Address)" to the global environment.
compileSC @@
compileSC = [ [Name|Vs] = Body | SCs ]
:- ground(Vs)
   | compileSC = SCs,
     m(Name, !A),
     !A = compileR(Body, zip(Vs, 0), length(Vs)).

compiledSC @@
compileSC = [] :- .

compileR @@
R = compileR(Exp, Env, Arity)
```

```
:- int(Arity)
   | R = nGlobal(Arity,
     [ compileE(Exp, Env),
       update(Arity),
       pop(Arity),
       unwind
     ]).


% The "compileE" compiles the expression and turns it to the code.
% compileE(<expression to compile>, <environment>, <de bruijn index (based index)>)

% If the expression is an integer, the code should be the "pushInt"
compileEInt @@
R = compileE(Int, Env)
:- int(Int), ground(Env)
   | R = pushInt(Int).

% let
compileELet @@
H = [compileE(let(Defs, E), Env) | T ]
:- ground(Defs), ground(Env)
   | H = [ compileLetDefs(Defs, Env),
     compileE(E,
        % new environmemt
        append(zip(reverse(mapDefVar(Defs)), 0),
         addNtoRightElemOfAscList(Env, length(Defs)))
     ),
     slide(length(Defs)) | T ].


% letrec
compileELetRec @@
H = [compileE(letrec(Defs, E), Env) | T ]
:- ground(Defs), ground(Env)
   | H = [ alloc(length(Defs)),
     compileLetRecDefs(Defs,
           % new environmemt
           append(zip(reverse(mapDefVar(Defs)), 0),
            addNtoRightElemOfAscList(Env, length(Defs))),
           length(Defs)
     ),
     compileE(E,
        % new environmemt
        append(zip(reverse(mapDefVar(Defs)), 0),
         addNtoRightElemOfAscList(Env, length(Defs)))
     ),
     slide(length(Defs)) | T ].


% Compile primitive operators
compileEAdd @@
H = [ compileE(E0 + E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     add | T ].
```

```
compileEMul @@
H = [ compileE(E0 * E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     mul | T ].

compileESub @@
H = [ compileE(E0 - E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     sub | T ].

compileEDiv @@
H = [ compileE(E0 / E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     div | T ].

compileENegate @@
H = [ compileE(negate(E0), Env) | T ]
:- H = [ compileE(E0, Env),
   neg | T ].

compileELessThan @@
H = [ compileE(E0 < E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     lt | T ].

compileENeq @@
H = [ compileE(E0 \= E1, Env) | T ]
:- ground(Env)
   | H = [ compileE(E1, Env),
     compileE(E0, addNtoRightElemOfAscList(Env, 1)),
     neq | T ].

compileEIf @@
H = [ compileE(if(E0, E1, E2), Env) | T ]
:- ground(Env)
   | H = [ compileE(E0, Env),
     cond([ compileE(E1, Env) ], [ compileE(E2, Env) ]) | T ].

% The default cases.
% (The default cases are handled with compileC).

compileEVariable @@
H = [ compileE(Var, Env) | T]
:- string(Var)
   | H = [ compileC(Var, Env), eval | T].

compileEApp @@
H = [ compileE((E0, E1), Env) | T]
```

```
   :- H = [ compileC((E0, E1), Env), eval | T].

   compileEConstr @@
   H = [ compileE(eConstr(Tag, Arity), Env) | T]
   :- H = [ compileC(eConstr(Tag, Arity), Env),
      eval | T].

   % Mark 6
   compileECase @@
   H = [ compileE(case(E, Alts), Env) | T ]
   :- ground(Env)
      | H = [ compileE(E, Env),
        casejump(compileD(Alts, Env)) | T ].

   % Compile alternatives in the case expression
   % "D" scheme
   compileDCons @@
   R = compileD([Alt | Alts], Env)
   :- ground(Env)
      | R = [ compileA(Alt, Env) | compileD(Alts, Env) ].

   compileDNil @@
   H = compileD([], Env)
   :- ground(Env)
      | H = [].

   % "A" scheme
   compileACons @@
   R = compileA(alt(Tag, Vars, Body), Env)
   :- ground(Vars)
      | R = asc(Tag,
          [ split(length(Vars)),
            compileE(Body,
         append(zip(Vars, 0),
                addNtoRightElemOfAscList(Env, length(Vars)))

            ),
            slide(length(Vars)) ]
   ).


   % The "compileC" compiles the expression and turns it to the code.
   % compileC(<expression to compile>, <environment>, <de bruijn index (based index)>)

   % Looks up and the get the de bruijn index
   % (... based index considering the change of the length of the stack
   % through pushes).
   compileLocalVar_lookup @@
   R = compileC(Var1, [asc(Var2, Vi) | Vs])
   :- string(Var1), Var1 \== Var2, int(Vi)
      | R = compileC(Var1, Vs).

   % If the variable is in the environment, the variable is a local variable and
   % it can be "pushed" with the de bruijn index (based index) from the stack.
   compileLocalVar_resolve @@
   R = compileC(Var1, [asc(Var2, Vi) | Vs])
   :- string(Var1), Var1 == Var2, ground(Vs)
```

```
    | R = push(Vi).

% If the variable is NOT in the environment, the variable is a free variable and
% it should be "pushed" from the global environment through "pushGlobal".
compileFreeVar @@
R = compileC(Var, [])
:- string(Var)
   | R = pushGlobal(Var).


% Arithmetic operators
compileCSAdd @@
R = compileC(E0 + E1, Env)
:- R = compileC(("+", E0), E1), Env).

compileCSub @@
R = compileC(E0 - E1, Env)
:- R = compileC(("-", E0), E1), Env).

compileCMul @@
R = compileC(E0 * E1, Env)
:- R = compileC(("*", E0), E1), Env).

compileCDiv @@
R = compileC(E0 / E1, Env)
:- R = compileC(("/", E0), E1), Env).

compileCNeg @@
R = compileC(negate(E), Env)
:- R = compileC(("negate", E), Env).

compileCLt @@
R = compileC(E0 < E1, Env)
:- R = compileC(("<", E0), E1), Env).

compileCNeq @@
R = compileC(E0 \= E1, Env)
:- R = compileC(("\=", E0), E1), Env).

compileCIf @@
R = compileC(if(E0, E1, E2), Env)
:- R = compileC((("if", E0), E1), E2), Env).

% If the expression is an integer, the code should be the "pushInt"
compileCInt @@
R = compileC(Int, Env)
:- int(Int), ground(Env)
   | R = pushInt(Int).


% let
compileCLet @@
H = [compileC(let(Defs, E), Env) | T ]
:- ground(Defs), ground(Env)
   | H = [ compileLetDefs(Defs, Env),
     compileC(E,
        % new environmemt
```

```
          append(zip(reverse(mapDefVar(Defs)), 0),
           addNtoRightElemOfAscList(Env, length(Defs)))
      ),
      slide(length(Defs)) | T ].

compileLetDefsCons @@
H = [compileLetDefs([Var = Def | Defs], Env) | T]
:- ground(Env), string(Var)
   | H = [ compileC(Def, Env),
     compileLetDefs(Defs, addNtoRightElemOfAscList(Env, 1)) | T].

compileLetDefsNil @@
H = [compileLetDefs([], Env) | T]
:- ground(Env)
   | H = T.


% letrec
compileCLetRec @@
H = [compileC(letrec(Defs, E), Env) | T ]
:- ground(Defs), ground(Env)
   | H = [ alloc(length(Defs)),
     compileLetRecDefs(Defs,
           % new environmemt
           append(zip(reverse(mapDefVar(Defs)), 0),
            addNtoRightElemOfAscList(Env, length(Defs))),
           length(Defs)
      ),
      compileC(E,
         % new environmemt
         append(zip(reverse(mapDefVar(Defs)), 0),
          addNtoRightElemOfAscList(Env, length(Defs)))
      ),
      slide(length(Defs)) | T ].


compileLetRecDefsCons @@
H = [compileLetRecDefs([Var = Def | Defs], Env, N) | T]
:- ground(Env), string(Var), N_1 = N - 1
   | H = [ compileC(Def, Env), update(N_1),
     compileLetRecDefs(Defs, Env, N_1) | T].

compileLetRecDefsNil @@
H = [compileLetRecDefs([], Env, 0) | T]
:- ground(Env)
   | H = T.


% Mark 6

compileCNullaryConstr @@
R = compileC(eConstr(Tag, 0), Env)
:- ground(Env)
   | R = pack(Tag, 0).


% Application is compiled into the postfix-notated code
```

```
% like "(E0, E1) -> [E1, E0, mkap]".
% However, the de bruijn index (based index) on the E0 should be increased by 1
% with the consideration of the increasement of the length of the stack.
compileCApp @@
H = [ compileCApp((E0, E1), Env) | T ]
:- ground(Env)
   | H = [ compileC(E1, Env),
     compileC(E0, addNtoRightElemOfAscList(Env, 1)),
     mkap | T ].

compileCAppCheckSaturated @@
R = compileC((E0, E1), Env)
:- ground(E0), ground(E1)
   | R = checkSaturated( saturatedCons([ makeSpine((E0, E1)) ]),
       [ makeSpine((E0, E1)) ],
       (E0, E1),
       Env
).

% Check checkSaturated or not
R = checkSaturated(X =:= Y, Spine, (E0, E1), Env)
:- X =:= Y, ground(E0), ground(E1)
   | R = compileCS(reverse(Spine), Env).

R = checkSaturated(X =:= Y, Spine, (E0, E1), Env)
:- X =\= Y, ground(Spine)
   | R = compileCApp((E0, E1), Env).

R = checkSaturated(notSaturated, Spine, (E0, E1), Env)
:- ground(Spine)
   | R = compileCApp((E0, E1), Env).


% CompileCS
% Compiles the constructor followed by applications
compileCSConstr @@
R = compileCS([ eConstr(Tag, Arity) ], Env)
:- ground(Env)
   | R = pack(Tag, Arity).

compileCSSpines @@
H = [ compileCS([E1, E2 | Es], Env) | T]
:- ground(Env)
   | H = [ compileC(E1, Env),
     compileCS([E2|Es], addNtoRightElemOfAscList(Env, 1)) | T].


% Auxiliary functions for compiling "pack"

% Make application spines
isConstrSaturated @@
R = saturatedCons([eConstr(Tag, Arity) | T])
:- int(Tag)
   | R = (Arity =:= length(T)).


% The default cases (other than constructor) for the "isConstrSaturated"
```

```
isConstrSaturatedDefaultNil @@
R = saturatedCons([])
:- R = notSaturated. % This may not happen and not needed.

isConstrSaturatedDefaultInt @@
R = saturatedCons([Int | T])
:- int(Int), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultVar @@
R = saturatedCons([Var | T])
:- string(Var), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultPlus @@
R = saturatedCons([E1 + E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultMinus @@
R = saturatedCons([E1 - E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultTimes @@
R = saturatedCons([E1 * E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultDiv @@
R = saturatedCons([E1 / E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultNeg @@
R = saturatedCons([negate(E) | T])
:- ground(E), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultLessThan @@
R = saturatedCons([E1 < E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultNeq @@
R = saturatedCons([E1 \= E2 | T])
:- ground(E1), ground(E2), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultIf @@
R = saturatedCons([if(E1, E2, E3) | T])
:- ground(E1), ground(E2), ground(E3), ground(T)
   | R = notSaturated.

isConstrSaturatedDefaultLet @@
R = saturatedCons([let(Defns, E) | T])
:- ground(Defns), ground(E), ground(T)
```

```
    | R = notSaturated.

isConstrSaturatedDefaultLetRec @@
R = saturatedCons([letrec(Defns, E) | T])
:- ground(Defns), ground(E), ground(T)
    | R = notSaturated.

isConstrSaturatedDefaultCase @@
R = saturatedCons([case(E, Alts) | T])
:- ground(E), ground(Alts), ground(T)
    | R = notSaturated.


% Make application spines
makeSpineApp @@
H = [ makeSpine((E1, E2)) | T]
:- H = [ makeSpine(E1), E2 | T].

% The default cases (other than application) for the "makeSpine"
makeSpineDefaultInt @@
R = makeSpine(Int)
:- int(Int) | R = Int.

makeSpineDefaultVar @@
R = makeSpine(Var)
:- string(Var) | R = Var.

makeSpineDefaultCase @@
R = makeSpine(case(E, Alts))
:- R = case(E, Alts).

makeSpineDefaultIf @@
R = makeSpine(if(E1, E2, E3))
:- R = if(E1, E2, E3).

makeSpineDefaultPlus @@
R = makeSpine(E1 + E2)
:- R = E1 + E2.

makeSpineDefaultTimes @@
R = makeSpine(E1 * E2)
:- R = E1 * E2.

makeSpineDefaultMinus @@
R = makeSpine(E1 - E2)
:- R = E1 - E2.

makeSpineDefaultDiv @@
R = makeSpine(E1 / E2)
:- R = E1 / E2.

makeSpineDefaultNeg @@
R = makeSpine(negate(E))
:- R = negate(E).

makeSpineDefaultLessThan @@
R = makeSpine(E1 < E2)
```

```
  :- R = (E1 < E2).

  makeSpineDefaultNeq @@
  R = makeSpine(E1 \= E2)
  :- R = (E1 \= E2).

  makeSpineDefaultLet @@
  R = makeSpine(let(Defns, E))
  :- R = let(Defns, E).

  makeSpineDefaultLetRec @@
  R = makeSpine(letrec(Defns, E))
  :- R = letrec(Defns, E).

  makeSpineDefaultConstr @@
  R = makeSpine(eConstr(Tag, Arity))
  :- R = eConstr(Tag, Arity).

    }.
}.
```

A.2. Implementation of G-Machine in HyperLMNtal

```
/*
  G-Machine on LMNtal
  Mark 6
*/

{ gm.
    {'$callback'(zerostep).
  % Some helper functions for the compiler.

  appendCons @@
  R = append([H|T], L)
  :- R = [H | append(T, L)].

  appendNil @@
  R = append([], L)
  :- R = L.

  revAppendCons @@
  R = revAppend([H|T], Acc)
  :- R = revAppend(T, [H|Acc]).

  revAppendNil @@
  R = revAppend([], Acc)
  :- R = Acc.

  reverse @@
  R = reverse(List)
  :- R = revAppend(List, []).
```

```
length_init @@
R = length(L)
:- R = length(L, 0).

length_cons @@
R = length([H|L], N)
:- Nplus1 = N + 1, ground(H)
   | R = length(L, Nplus1).

length_nil @@
R = length([], N)
:- R = N.
  }.


  % G-Machine

  % Looks up the global node and pushes the address to the top of the stack.
  pushGlobal @@
  code = [pushGlobal(F)|Is], stack = S, m(MF, !A)
  :- F == MF
     | code = Is, stack = [!A|S], m(MF, !A).

  % Makes the number node and pushes the address.
  pushInt @@
  code = [pushInt(N)|Is], stack = S
  :- int(N)
     | code = Is, stack = [!A|S], !A = nNum(N).

  % Make application node of the top 2 addresses on the stack.
  mkap @@
  code = [mkap|Is], stack = [!A1, !A2|S]
  :- code = Is, stack = [!A|S], !A = nApp(!A1, !A2).

  % Push the address of the local variable on the Nth element on the stack.
  % Put the stopper "pushing" on the head of the code when pushing.
  code = [push(N)|Is], stack = S
  :- code = [pushing|Is], stack = [lookupStack(N)|S].

  {'$callback'(zerostep).
% Traverse the stack to accomplish the push of the Nth element on the stack.
pushLookup @@
H = [lookupStack(N), A|T]
:- N > 0, N_1 = N - 1
   | H = [A, lookupStack(N_1)|T].

% Push the Nth element on the stack and clear the stopper "pushing".
pushResolve @@
code = [pushing|Is], stack = S, H = [lookupStack(0), !A|T]
:- code = Is, stack = [!A|S], H = [!A|T].
  }.

  % Update the Nth element of the stack.
  update @@
  code = [update(N)|Is], stack = [!A|S]
  :- code = [updating|Is], stack = updateStack(N, !A, S).
```

```
    {'$callback'(zerostep).
  updateN @@
  H = updateStack(N, A, [Ai|S])
  :- N > 0, N_1 = N - 1
     | H = [Ai|updateStack(N_1, A, S)].

  updateOnApp @@
  H = updateStack(0, !A, [!Ai|S]), !Ai = nApp(!AL, !AR), code = [updating|Is]
  :- H = [!Ai|S], !Ai >< !A, code = Is.

  updateOnGlobal @@
  H = updateStack(0, !A, [!Ai|S]), !Ai = nGlobal(VN, Body), code = [updating|Is]
  :- ground(Body), int(VN)
     | H = [!Ai|S], !Ai >< !A, code = Is.

  % Mark 6 (implementation with no confidence)
  updateOnConstr @@
  H = updateStack(0, !A, [!Ai|S]), !Ai = nConstr(Tag, Args), code = [updating|Is]
  :- int(Tag), ground(Args)
     | H = [!Ai|S], !Ai >< !A, code = Is.


  % Update the indirection node pointing null.
  % This is for the "let rec" (Mark 3).
  updateOnNull @@
  H = updateStack(0, !A, [!Null|S]), !Null = null, code = [updating|Is]
  :- H = [!A|S], !Null >< !A, code = Is.
   }.


    % clear the top N element of the stack.
    {'$callback'(zerostep).
  popN @@
  code = [pop(N)|Is], stack = [!A0|S]
  :- N > 0, N_1 = N - 1
     | code = [pop(N_1)|Is], stack = S.
    }.

    popO @@
    code = [pop(0)|Is]
    :- code = Is.


    % Unwind application node.
    unwindApp @@
    code = [unwind], stack = [!A|S], !A = nApp(!A1, !A2)
    :- code = [unwind], stack = [!A1, !A|S], !A = nApp(!A1, !A2).

    % Unwind the number node.
    unwindInt @@
    code = [unwind], stack = [!A|S], dump = [asc(I2, S2)|D], !A = nNum(N)
    :- int(N), ground(S)
       | code = I2, stack = [!A|S2], dump = D, !A = nNum(N).

    % Unwind the global node and pushes the codes to the code.
    % (3.19)
    unwindGlobalWithEmptyDump @@
```

```
   code = [unwind], stack = [!A0|S], dump = [], !A0 = nGlobal(VN, Cs)
   :- ground(Cs), int(VN)
      | code = rearranging(Cs),
    stack = rearrange(VN, !A0, S),
    dump = [],
    !A0 = nGlobal(VN, Cs).

   % Unwind the global node and pushes the codes to the code.
   % (3.29)
   unwindGlobalWithNonEmptyDump @@
   code = [unwind], stack = [!A0|S], dump = [asc(I2, S2)|D], !A0 = nGlobal(VN, Cs
)
   :- ground(S), int(VN)
      | code = unwindingGlobal(length(S), VN),
    stack = [!A0|S],
    dump = [asc(I2, S2)|D],
    !A0 = nGlobal(VN, Cs).

    {'$callback'(zerostep).
   % (3.29)
   unwindGlobalWithNonEmptyDumpKLessThanN @@
   code = unwindingGlobal(K, N), dump = [asc(I2, S2)|D]
   :- K < N
      | code = unwindingGlobalsVariables(I2, S2),
        dump = D.

   unwindingGlobalsVariabes @@
   code = unwindingGlobalsVariables(I2, S2), stack = [!A0, !A1|S]
   :- code = unwindingGlobalsVariables(I2, S2), stack = [!A1|S].

   unwindingGlobalsVariabesFinish @@
   code = unwindingGlobalsVariables(I2, S2), stack = [!Ak]
   :- code = I2, stack = [!Ak|S2].


   % (3.29)
   unwindGlobalWithNonEmptyDumpKGreaterThanN @@
   code = unwindingGlobal(K, N), stack = [!A0|S], !A0 = nGlobal(VN, Cs)
   :- K >= N, ground(Cs), int(VN)
      | code = rearranging(Cs),
        stack = rearrange(VN, !A0, S),
        !A0 = nGlobal(VN, Cs).
     }.

   % Unwind the constructor node
   % Mark 6
   unwindNConstrWithNonEmptyDump @@
   code = [unwind], stack = [!A|S], dump = [asc(I2, S2)|D], !A = nConstr(Tag, Args)
   :- ground(S)
      | code = I2, stack = [!A|S2], dump = D, !A = nConstr(Tag, Args).


     % Rearrange the pointers on the stack.
     % Replace the pointer to the application node with the pointer to its right child.
     {'$callback'(zerostep).
   rearrange @@
   H = rearrange(VN, !Ai_1, [!A|T]), !A = nApp(AL, !AR)
```

```
  :- VN > 0, VN_1 = VN - 1
     | H = [!AR|rearrange(VN_1, !A, T)], !A = nApp(AL, !AR) .

  finishRearrange @@
  code = rearranging(CS), H = rearrange(0, !An, T),
  :- code = CS, H = [!An|T].
  }.

  % Evaluate the arguments of the primitive operators
  eval @@
  code = [eval|Is], stack = [A|S], dump = D
  :- code = [unwind], stack = [A], dump = [asc(Is, S)|D].

  % Arithmetic instructions
  add @@
  code = [add|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N = N0 + N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(N).

  addSameNode @@
  code = [add|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- N = N0 + N0
     | code = Is, stack = [!A|S], !A0 = nNum(N0), !A = nNum(N).

  sub @@
  code = [sub|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N = N0 - N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(N).

  subSameNode @@
  code = [sub|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- N = N0 - N0
     | code = Is, stack = [!A|S], !A0 = nNum(N0), !A = nNum(N).

  mul @@
  code = [mul|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N = N0 * N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(N).

  mulSameNode @@
  code = [mul|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- N = N0 * N0
     | code = Is, stack = [!A|S], !A0 = nNum(N0), !A = nNum(N).

  div @@
  code = [div|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N1 =\= 0,  N = N0 / N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(N).

  divSameNode @@
  code = [div|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- N0 =\= 0, N = N0 / N0
     | code = Is, stack = [!A|S], !A0 = nNum(N0), !A = nNum(N).

  neg @@
  code = [neg|Is], stack = [!A0|S], !A0 = nNum(N0)
  :- N = - N0
```

```
      | code = Is, stack = [!A|S], !A= nNum(N), !A0 = nNum(N0).


  % Comparison instructions
  lessThanTrue @@
  code = [lt|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N0 < N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(1).

  lessThanFalse @@
  code = [lt|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N0 >= N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(0).

  lessThanFalseSameNode @@
  code = [lt|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- code = Is, stack = [!A|S], !A0= nNum(N0), !A = nNum(0).


  neqTrue @@
  code = [neq|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N0 =\= N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(1).

  neqFalse @@
  code = [neq|Is], stack = [!A0, !A1|S], !A0= nNum(N0), !A1 = nNum(N1)
  :- N0 =:= N1
     | code = Is, stack = [!A|S], !A0= nNum(N0), !A1 = nNum(N1), !A = nNum(0).

  neqFalseSameNode @@
  code = [neq|Is], stack = [!A0, !A0|S], !A0= nNum(N0)
  :- code = Is, stack = [!A|S], !A0= nNum(N0), !A = nNum(0).


  % Condition instruction
  condTrue @@
  code = [cond(I1, I2)|Is], stack = [!A|S], !A= nNum(1)
  :- ground(I2)
     | code = append(I1, Is), stack = S, !A = nNum(1).

  condFalse @@
  code = [cond(I1, I2)|Is], stack = [!A|S], !A= nNum(0)
  :- ground(I1)
     | code = append(I2, Is), stack = S, !A = nNum(0).


  % Slides the top of the stack N and discard the slided elements.
  slide @@
  code = [slide(N)|Is]
  :- int(N)
     | code = [sliding(N)|Is].

  {'$callback'(zerostep).
slideN @@
code = [sliding(N)|Is], stack = [!A0, !A1|S]
:- N > 0, N_1 = N - 1
   | code = [sliding(N_1)|Is], stack = [!A0|S].
```

```
  slide0 @@
  code = [sliding(0)|Is]
  :- code = Is.
    }.

    % Alloc
    % Allocates N "null pointer"s.
    % These "null pointers" will be updated and replaces with a "real" addresses
    % (so there is no need for worrying for the null pointer traversing).
    {'$callback'(zerostep).
  allocN @@
  code = [alloc(N)|Is], stack = S
  :- N > 0, N_1 = N - 1
     | code = [alloc(N_1)|Is], stack = [!Null|S], !Null = null.
    }.

    alloc0 @@
    code = [alloc(0)|Is], stack = S
    :- code = Is, stack = S.


    % Mark 6
    % Implementation of data types

    % Pack constructor with the arguments
    pack @@
    code = [pack(Tag, Arity) | Is]
    :- code = [packing(Tag, Arity, []) | Is].

    {'$callback'(zerostep).
  packN @@
  R = packing(Tag, Arity, Args), stack = [A|S]
  :- Arity > 0, Arity_1 = Arity - 1
     | R = packing(Tag, Arity_1, [A|Args]), stack = S.

  pack0 @@
  code = [packing(Tag, 0, Args) | Is], stack = S
  :- code = Is, stack = [!A|S], !A = nConstr(Tag, reverse(Args)).
    }.

    % Casejump
    % A jump instructions for the case expression
    {'$callback'(zerostep).
  casejumpLookup @@
  code = [casejump([asc(Tag1, I2) | Cases]) | Is],
  stack = [!A|S],
  !A = nConstr(Tag2, SS)
  :- Tag1 =\= Tag2, ground(I2)
     | code = [casejump(Cases) | Is],
       stack = [!A|S],
       !A = nConstr(Tag2, SS).
    }.

    casejumpResolve @@
    code = [casejump([asc(Tag1, I2) | Cases]) | Is],
    stack = [!A|S],
```

```
    !A = nConstr(Tag2, SS)
    :- Tag1 =:= Tag2, ground(Cases)
       | code = append(I2, Is),
   stack = [!A|S],
   !A = nConstr(Tag2, SS).

    % Split
    code = [split(N) | Is], stack = [!A|S], !A = nConstr(Tag, Args)
    :- ground(Args), int(N) % There may be no need for the N
       | code = Is,
   stack = append(Args, S),
   !A = nConstr(Tag, Args).


    % Garbage collector (based on reference counting)
    {'$callback'(zerostep).
  gcNApp @@
  !A = nApp(!E1, !E2)
  :- num(!A) =:= 1
     | .

  gcNNum @@
  !A = nNum(N)
  :- num(!A) =:= 1, int(N)
     | .

  gcNConstr @@
  !A = nConstr(Tag, Args)
  :- num(!A) =:= 1, int(Tag), ground(Args)
     | .
    }.
}.
```

Appendix B.

Experiments

input	result	number of the steps of G-machine
1	3	98
2	5	168
3	9	308
4	15	518
5	25	868
6	41	1428
7	67	2338
8	109	3808
9	177	6188
10	287	10038

Figure B.1: The number of the steps took in `nfib`

The index of the prime	the prime	the number of the steps of G-machine
0	2	79
1	3	218
2	5	479
3	7	788
4	11	1341
5	13	1746
6	17	2395
7	19	2896
8	23	3641
9	29	4678
10	31	5323
11	37	6456
12	41	7393
13	43	8182
14	47	9215
15	53	10588
16	59	11961
17	61	12942
18	67	14411
19	71	15684
20	73	16809
21	79	18470
22	83	19887
23	89	21596
24	97	23645
25	101	25206
26	103	26619
27	107	28276
28	109	29785
29	113	31538
30	127	34655
31	131	36504

Figure B.2: The number of the steps took in the Sieve of Eratosthenes

Number of the irrelevant rules	execution time (sec) with the irrelevantRule1	execution time (sec) with the irrelevantRule2
0	1.13	1.13
1	2.13	1.16
2	3.09	1.23
3	3.94	1.26
4	4.96	1.32
5	5.91	1.35

Figure B.3: The execution time with the patial maching rules