A Functional Language with Hypergraphs as First-Class Data

A Thesis Submitted to the Department of Computer Science and Communications Engineering,
the Graduate School of Fundamental Science and Engineering of Waseda University
in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Submission Date: January 23rd, 2023

Jin Sano
(5121F053–7)
Advisor: Prof. Kazunori Ueda
Research guidance: Research on Parallel Knowledge Information Processing

Abstract

A Hypergraph is a generalized concept that encompasses more complex data structures than trees, such as difference lists, doubly-linked lists, skip lists, and leaf-linked trees. Normally, these structures are handled with destructive assignments to heaps, which is opposed to a purely functional programming style. These low-level operations are tiresome and prone to errors and are difficult to be verified.

To propose a new language to overcome the situation, we firstly discuss a hypergraph transformation language, HyperLMNtal. The language is a simple term language with syntax-directed semantics. As far as we have surveyed, this is the only calculus that has syntax-directed semantics which handle hypergraph matching and rewriting. We carefully observe the properties of HyperLMNtal; especially, the relation with terms and the denoted hypergraphs. This gives a theoretical foundation for the semantics of the new functional language and the type system we will introduce later.

Then, we propose a new functional language, λ_{GT}, that employs hypergraphs as immutable, first-class data and supports pattern matchings for them. In λ_{GT}, we do not rewrite one global heap with destructive assignment. Instead, hypergraphs are immutable local values that can be bound to variables, decomposed by pattern matchings with possibly multiple wildcards, in which the matched subgraphs may be used separately, passed as inputs of functions, and composed to construct larger graphs. To formalize the language, we incorporate the syntax and the semantics of HyperLMNtal into a call-by-value λ-calculus.

Finally, we construct a new type system, F_{GT}, for the λ_{GT} language. In F_{GT}, we define the type of graphs using graph grammar. This can be regarded as an extension of regular tree grammar, on which algebraic data types are based. Our approach is in contrast with the analysis of pointer manipulation programs using separation logic, shape analysis, etc. in that (i) we do not consider destructive operations but pattern matchings over graphs provided by the new higher-level language that abstract pointers and heaps away and that (ii) we pursue what properties can be established automatically using a rather simple typing framework.

Acknowledgements

I would like to thank my advisor, Prof. Kazunori Ueda, for not just simply giving me useful advice consistently but also letting me cultivate a new research area instead of taking over the existing work in our lab or forcing a direction. I always felt like I was treated as an independent researcher at the forefront — in fact, becoming a researcher with a brand new theory and application was my greatest dream ever since my childhood — which motivated me a lot. I must also apologise for almost all of my work being slow and unsteady and express my heartfelt gratitude for the time and effort spent on support despite this.

My colleague Kunihiro Hata always saved me with his keen insight and extensive experience in algebra. Much of the foundation of the work in Sections 2.3 to 2.5, in particular, is due to his helpful advice. Every time, no matter how vague or elementary my questions are, he kindly tells me his insightful thoughts, and we think together. Besides, you are a great person. I appreciate your extreme gentleness and decent attitude every day. If everyone had co-workers as great as you, no one would ever complain about work again. Thank you so much for being there for me.

I always admire my friend Masaki Nakata for his exceptional work ethic and incredible enthusiasm. I have never seen such a hard-working individual. It's truly inspiring. Although we have been in different environments for a few years, he keeps noticing me and keeping me motivated. Every time I got stuck or felt lazy, I imagined that you would sit next to me as you had done two years ago, overcome every obstacle, and show progress.

I could not live without having lunch with members of our lab, including Ren Imagawa. You make coming to work each day such a pleasure. To be honest, most days, I come to the lab for lunch. I cannot imagine that we would ever have such a wonderful time again. I will always remember my best time at this lab.

Mr Naoki Yamamoto, I must apologise for my rude and challenging demeanour. Forgive me; I greatly respect you, and that's why I want to surpass you. I am so confident that you will deserve the degree. Work hard!

I am always grateful for my family and their continued encouragement and support. I owe my life to so many people, and I wish them all the best.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Jin Sano
January 23rd, 2023

Contents

1

Introduction

Hypergraphs are a generalized concept that encompasses more complex data structures than trees, such as difference lists, doubly-linked lists, skip lists [Pug90], and leaf-linked trees (Figure 1.1).

Normally, these structures are handled with destructive assignments to heaps, which is opposed to a purely functional programming style. These low-level operations are tiresome and prone to errors and are difficult to be verified.

Without garbage collection, programming with heaps and pointers may lead to significant bugs with memory. Some of these bugs can be detected with type systems. For example, some garbage-free imperative programming languages, such as Rust, utilize a linear type to ensure memory safety. This works when using tree-based data structures, but when they come with sharing, we need to pay costs for shared pointers to ensure memory safety, which can also easily leak memory when they have cycles.

On the other hand, many functional languages support Algebraic Data Types (ADTs). Compared to a low-level programming with heaps and pointers, this allows more declarative programming and makes it easier to read and write programs, and also makes verification easier. In fact, using ADTs, we can recursively define the shapes of data structures and can use them with a guarantee of the shapes by type systems. We can handle them purely, without destructive assignments, with pattern matchings, in contrast with programming with heaps and pointers. However, with ADTs, we can construct only trees, and more complex data structures cannot be handled.

Many of the practical functional programming languages support reference types (e.g., `ref` type in OCaml, `IORef` type in Haskell) and we can construct more complicated structures than trees using these. However, this style implies imperative programming with destructive assignments, this is again tiresome and prone to errors as well as pointer programming. Furthermore, although the type systems and garbage collectors provide pointer safety, they only check the type of the referenced local values, not the global shape of structures, which is not sufficient when handling complex structures, considering that handling these with low-level operations is prone to errors.

To overcome the situation, we aim to incorporate *Graph Transformation* [Roz97] to a functional language. In this research, we propose a new purely functional language, λ_{GT}, that employs hypergraphs as immutable, first-class data and supports pattern matchings for them, and a new type system F_{GT} for the language.

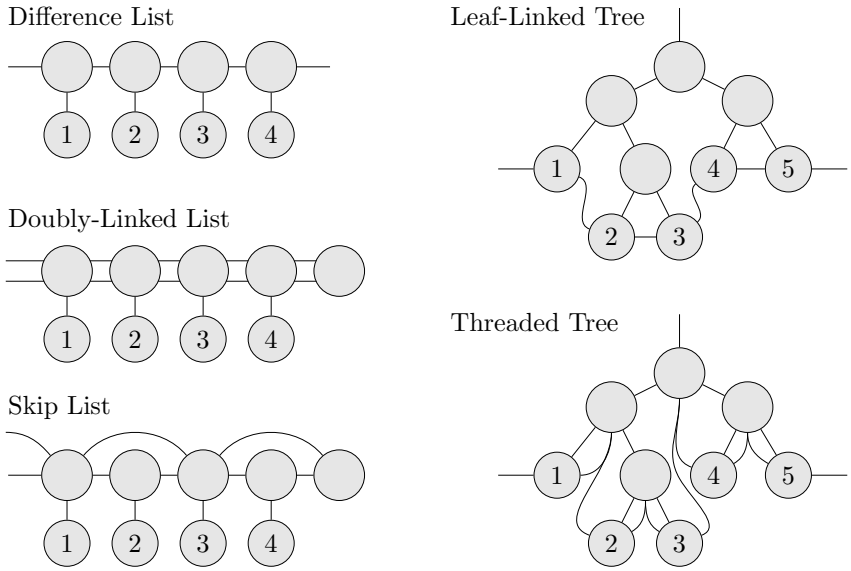


Figure 1.1: Examples of complex graph structures

Our approach is in contrast with the analysis of pointer manipulation programs using separation logic [Rey02], shape analysis, etc. in that (i) we consider graph structures formed by higher-level languages that abstract pointers and heaps away and guarantee low-level invariants such as the absence of dangling pointers and that (ii) we pursue what properties can be established automatically using a rather simple typing framework.

1.1 HyperLMNtal: Hypergraph Transformation Formalism

Graph Transformation Systems (GTSs) are computational models and programming (or modeling) languages based on graphs and their rewritings [Ehr+06; Roz97]. Of various GTSs, HyperLMNtal [UO12] is a rewriting language that supports hypergraphs. With hypergraphs, we can express structures more complex than trees, e.g., difference lists, doubly-linked lists, skip lists, and leaf-linked trees.

HyperLMNtal allows us to handle these data structures declaratively with rewrite rules that are activated by pattern matching. Furthermore, GTS has cultivated a unique style of type checking frameworks such as Structured Gamma [FM98]. However, GTSs are in general based on destructive rewriting and do not support higher-order functions. In contrast, functional languages basically work with immutable data structures and support higher-order functions, making them highly modular. This motivates us to study how we can incorporate the data structure of HyperLMNtal into the λ-calculus.

1.2 The λ_{GT} Language and the Type System F_{GT}

We propose a new functional language, λ_{GT}, that features hypergraphs as a first-class data structure. The λ_{GT} language allows us to handle complex data structures declaratively

with a static type system. Intuitively, the core language is a call-by-value λ-calculus that employs hypergraphs as values and supports pattern matching for them.

In order to formalize hypergraphs in a syntax-directed manner, we employ the techniques developed in a hypergraph rewriting language HyperLMNtal [UO12; SU21]. While various different formalisms have been proposed to handle the shapes of graphs, including bisimulation (to handle "equivalence" of cyclic structures) and morphism (in a category-theoretic approach), we believe that our approach enables type checking relatively easily. We also propose a new type-checking algorithm that automatically performs this verification using structural induction.

1.3 Contributions

The main contributions of this paper are threefold.

1. We investigate the properties of a proposed hypergraph transformation formalism, HyperLMNtal.
   (a) We redefine the syntax and the semantics of HyperLMNtal carefully. We have already proposed these in our previous work [San21; SU21]. However, the definition has a problem with a substitution, which is critical since this leads to all the terms being congruent with the term in which all the hyperlinks have the same name. We have also proved some properties of *fusions*, that connects hyperlinks, and *free links* (See Section 2.2 for the definitions of these) that characterize these as an equivalence relation (Section 2.2.2).
   (b) We define the denoted hypergraphs by HyperLMNtal terms and define a mapping from HyperLMNtal terms to the hypergraphs: *t2gs*. Since HyperLMNtal has a notion of *fusions* and *free links* (Section 2.2), which plays a key role in subgraph matching and the composition of graphs, the definition of translated hypergraphs is not trivial. We extend the existing hypergraph formalism to meet our calculus.
   (c) We prove that structurally congruent terms are mapped to isomorphic graphs using *t2gs*: i.e., the mapping satisfies soundness.
2. We propose λ_{GT}, a purely functional language that handles data structures beyond algebraic data types.
   (a) We propose the syntax and the semantics of the language and examine their validity with some running examples.
   (b) We implement a *reference interpreter*, a reference implementation of the language. We believe this is usable for further investigation, including in the design of *real languages* based on λ_{GT}. The interpreter is written in only 500 lines of OCaml [Ler+22] code, which is strikingly concise.

3. We propose a new type system F_{GT} for the λ_{GT} language.
   (a) We describe the syntax and the typing rules of the basic F_{GT} and prove some properties including soundness.
   (b) We extend the typing framework for the λ_{GT} language so that can successfully handle more manipulations of graphs, including which could not be handled in a previous study, Structured Gamma.

1.4 Thesis Roadmap

The rest of this thesis is organized as follows. Chapter 2 describes HyperLMNtal, a calculus model based on hypergraph transformation. Chapter 3 gives the syntax and the operational semantics of the proposing language λ_{GT}, and describes our reference interpreter. Chapter 4 describes F_{GT}, a new type system proposed for λ_{GT}. We firstly introduce the basic system and extend it later to cover more powerful operations. Finally, Chapter 5 concludes the dissertation and hints possible future work.

1.5 Syntactic Conventions

Throughout the paper, we use the following syntactic conventions.

For some syntactic entity E, $\overrightarrow{E}$ stands for a sequence $E_1, \ldots, E_n$ for some n (≥ 0). When we wish to mention the indices explicitly, $E_1, \ldots, E_n$ will also be denoted as $\overrightarrow{E_i}^i$. The length of the sequence $\overrightarrow{E}$ is denoted as $|\overrightarrow{E}|$.

For a set S, the form $S\{s\}$ stands for the set S such that $s \in S$ (or equivalently, $S = S \cup \{s\}$).

For some syntactic entities E, p and q, a substitution $E[q/p]$ stands for E with all the (free) occurrences of p replaced by q. An explicit definition will be given if the substitution should be capture-avoiding. For substitutions of hyperlinks, we use a slightly different syntax $E\langle q/p\rangle$ for clarity.

In order to focus on novel and/or non-obvious aspects of the language, constructs and properties that can be defined/derived in the same manner as those of standard functional languages will be described rather briefly.

We denote a reflexive transitive closure of a relation R as R^* and reflexive transitive symmetric closure of R as $R^{\equiv}$. We may denote an equivalence relation with a quotient set.

We may denote by $[x \mapsto y]$ as a first-class function such as $[x \mapsto y](z) \stackrel{\text{def}}{=} z[y/x]$. Notice that a function $\{x \mapsto y\}$ is slightly different from this: the domain of the latter is a singleton set only consists of x.

Hereinafter we may simply refer to hyperlinks as *links*, and hypergraphs as *graphs*.

2

HyperLMNtal: A Hypergraph Transformation Formalism

2.1 Introduction

A hypergraph is a generalization of a graph in which an edge can join any number of vertices. Hypergraphs are important both in theory and application [Ehr+06; Roz97] since hypergraphs allow us to model various data structures in programming, especially those that use heaps and pointers in imperative programming, in a highly general setting. Since most of the programming languages and verification frameworks are designed as term languages, a simple term language and its semantics are desirable both for programming and modelling with these data structures and for the verification of the programs that use such data structures.

Since (hyper)graphs and their operations are more complex than trees, it is not sufficient to define the syntax of a term language; in order to provide terms with an appropriate interpretations as graphs, we need to define an *equivalence relation*, for example with a finite set of rules over terms, which is far from trivial.

It is known that there exists a simple term language, 2pdom-algebra [LP17], that can express graphs whose treewidth is up to 2. Recently, the language proved to be sound and complete using Coq [DP20]. Since we want to enable dealing with more complex data structures than what 2pdom-algebra can express, in this paper, we investigate the property of HyperLMNtal [UO12], what which is a term language that can express hypergraphs with arbitrary treewidth.

HyperLMNtal is extended from LMNtal [Ued09]. LMNtal is a programming language based on hierarchical graph rewriting. Flat LMNtal is a subset of LMNtal which does not allow a hierarchy of graphs. Links in graphs that LMNtal handles are restricted to have at most two endpoints. On the other hand, HyperLMNtal allows hyperlinks, apart from normal links, which can interconnect an arbitrary number of endpoints. Flat HyperLMNtal [SU21; San21] is a subset of HyperLMNtal that disallows normal links and hierarchies of hypergraphs: the data structure of Flat HyperLMNtal is formed only by hyperlinks and nodes.

In the previous study, we proposed a calculus for Flat HyperLMNtal [SU21; San21], which is a simple term language with syntax-directed semantics. Hereinafter we simply

refer to our calculus as *HyperLMNtal*. The design of HyperLMNtal is based on Process Algebra [SW01] and features the notions of *free names* and *bindings* (ν), which are common in programming languages, with syntax-directed semantics. As far as we have surveyed, HyperLMNtal is the only language that has syntax-directed semantics which handles hypergraph matching and rewriting. Since most of the calculi for the basis of programming languages, such as λ-calculus and IMP, are defined as Structural Operational Semantics (SOS) [Plo04], it would be smoother to incorporate the calculus for HyperLMNtal than existing graph transformation formalisms based on algebraic approaches [Roz97] to other calculi with SOS when we incorporate hypergraphs into those calculi.

Intuitively, hypergraphs denoted by the language have labelled vertices and (hyper)edges connected to vertices through *ports*. Such graphs are called *labelled port graphs* [FP18], but ours extend the edges to hyperedges. The notion of a port is important to model data structures that appear in programming since it corresponds to accessors of `structs` or records.

HyperLMNtal defines a *congruence* relation over terms with a finite set of axioms: *Structural Congruence* rules. The calculus was proposed for giving a semantics of the implementation of the programming language and a model checker [GHU11]. The implementation assumes that the congruence relation on terms is equivalent to graph isomorphism on heaps but this is yet to be proved. Also, using the calculus, we proposed a new purely functional language λ_{GT} and a type system F_{GT} [SYU23]. We aim to give a theoretical foundation on the semantics of λ_{GT} and the F_{GT} at the same time.

Contributions

The main contributions in this chapter are the following:

1. We redefine the syntax and the semantics of HyperLMNtal carefully. We have already proposed these in our previous work [San21; SU21]. However, the definition has a problem with a substitution, which is critical since this leads to all the terms being congruent with the term in which all the hyperlinks have the same name. We have also proved some properties of *fusions*, that connects hyperlinks, and *free links* (See Section 2.2 for the definitions of these) that characterise these as an equivalence relation (Section 2.2.2).

2. We define the denoted hypergraphs by HyperLMNtal terms and define a mapping from HyperLMNtal terms to the hypergraphs: *t2gs*. Since HyperLMNtal has a notion of *fusions* and *free links* (Section 2.2), which plays a key role in subgraph matching and the composition of graphs, the definition of translated hypergraphs is not trivial. We extend the existing hypergraph formalism to meet our calculus.

3. We prove that structurally congruent terms are mapped to isomorphic graphs using *t2gs*: i.e., the mapping satisfies soundness.

Chapter Map

The rest of this chapter is organized as follows. Section 2.2 introduces HyperLMNtal, a simple term language to denote hypergraphs. After describing the syntax, We give the definition of the equivalence of terms by defining *Structural Congruence* rules on terms. We investigate the properties of *fusions* and *free links* that characterize these as an equivalence relation in Section 2.2.2. We also define a *reduction relation*, which relies on pattern matching on hypergraphs. Section 2.3 describes the hypergraphs denoted by HyperLMNtal. We also give the definitions of the equivalence of hypergraphs by defining *Graph Isomorphism* on hypergraphs. Section 2.4 defines some operations on graphs and a mapping from HyperLMNtal term to hypergraphs; *t2gs*. Section 2.5 proves that the structurally congruent terms are mapped to graph-isomorphic graphs using the function *t2gs*: the *soundness* of the function. Section 2.6 describes related work and Section 2.7 gives a summary and future work.

2.2 HyperLMNtal

HyperLMNtal is extended from LMNtal [Ued09]. LMNtal is a computational model and a programming language based on hierarchical graph rewriting. Flat LMNtal is a subset of LMNtal which does not allow a hierarchy of graphs. Links in graphs that LMNtal handles are restricted to have at most two endpoints. On the other hand, HyperLMNtal [UO12] allows hyperlinks, apart from normal links, which can interconnect an arbitrary number of endpoints. Flat HyperLMNtal is a subset of HyperLMNtal that disallow normal links and hierarchies of hypergraphs: the data structure of Flat HyperLMNtal is formed only by hyperlinks and nodes.

The design of HyperLMNtal is based on Process Algebra [SW01] and features the notions of *free names* and *bindings*, which are common in programming languages, with syntax-directed semantics. These characteristics are advantageous when giving operational semantics structurally. As far as we have surveyed, Flat HyperLMNtal is the only computational model that has syntax-directed semantics which handles hypergraph matching and rewriting. Since the λ-calculus and many other computational models derived from the λ-calculus are are defined as Structural Operational Semantics (SOS) [Plo04], it would be smoother to incorporate Flat HyperLMNtal than other graph transformation formalisms based on algebraic approaches [Roz97].

The following subsections are based on Flat HyperLMNtal, except that hypergraphs and rewrite rules are separated from each other for the sake of formulation. Hereinafter we simply refer to this language as *HyperLMNtal*, hyperlinks as *links*, and hypergraphs as *graphs*.

2.2 Syntax of Graphs and Rewriting Rules

Before moving on to the definitions of HyperLMNtal, we would like to introduce some basic concepts.

Definition 2.2.1.

1. $\mathbb{V}$ is a denumerable set of *vertices*. We use V to denote a finite subset of it and v to denote its elements.

2. $\mathbb{X}$ is a denumerable set of *link names*. We use $\mathbf{X}$ to denote a finite subset of it and X, Y, Z, W to denote its elements.

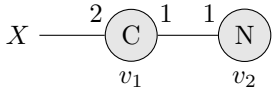
3. A *port* is the pair of a vertex and a non-zero natural number such as $\langle v, i\rangle \in V \times \mathbb{N}_{>0}$. Labelled port graphs [FP18] generalise the ports by allowing them to be named by elements of an arbitrary finite set. However, we can easily encode an element of the set with a natural number and generalise our framework in a straightforward manner.

4. $\mathbb{P}$ is a possibly infinite set of *atom names*, which are labels that are assigned to vertices. We use p to denote its elements.

The syntax of HyperLMNtal is given in Figure 2.1. $\mathbf{0}$ denotes an empty graph. $p(\overrightarrow{X})$ is an *atom*. Intuitively, it is a *node* of a data structure with label p and links $\overrightarrow{X}$; for example, $\mathrm{Nil}\,(X)$, $\mathrm{Cons}\,(Y, Z, X)$, etc. $X \bowtie Y$ is called a *fusion*, which fuses the link X and the link Y into a single link. (G, G) is a composition or gluing of graphs. $\nu X.G$ hides the link X.

We abbreviate $\nu X_1 \ldots \nu X_n.G$ to $\nu X_1 \ldots X_n.G$, which can be denoted as $\nu\overrightarrow{X}.G$. The pair of the name p and the arity $n = |\overrightarrow{X}|$ of an atom $p(\overrightarrow{X})$ is referred to as the *functor*[1] of the atom and is written as p/n.

The set of free link names in hypergraph G is denoted as $fn(G)$, which is defined inductively in Figure 2.2.

Example 2.2.1 (A Singleton List in HyperLMNtal Term)**.** $\nu Y.(\mathrm{Cons}\,(Y, X), \mathrm{Nil}\,(Y))$. *Informally, this corresponds to the following:* 

, *where* C *stands for* Cons *and* N *for* Nil.

Definition 2.2.2 (Abbreviation)**.** We introduce the following abbreviation schemes:

1. A nullary atom $p()$ can be simply written as p.

2. Term Notation: $\nu Y.(p(\overrightarrow{X}, Y, \overrightarrow{Z}), q(\overrightarrow{W}, Y))$ where $Y \notin \{\overrightarrow{X}, \overrightarrow{Z}, \overrightarrow{W}\}$ can be written as $p(\overrightarrow{X}, q(\overrightarrow{W}), \overrightarrow{Z})$.

[1] Synonym of function symbol and function object; not to be confused with functors in category theory.

Graph	G	$::=$	$\mathbf{0}$	Null
		$\mid$	$p(\overrightarrow{X})$	Atom
		$\mid$	$X \bowtie Y$	Fusion
		$\mid$	(G, G)	Molecule
		$\mid$	$\nu X.G$	Hyperlink Creation
Rewrite Rule	r	$::=$	$G \longrightarrow G$	Rule

Figure 2.1: Syntax of HyperLMNtal

$$
\begin{aligned}
fn(\mathbf{0}) &\stackrel{\text{def}}{=} \varnothing \\
fn(p(\overrightarrow{X})) &\stackrel{\text{def}}{=} \{\overrightarrow{X}\} \\
fn(X \bowtie Y) &\stackrel{\text{def}}{=} \{X, Y\} \\
fn((G_1, G_2)) &\stackrel{\text{def}}{=} fn(G_1) \cup fn(G_2) \\
fn(\nu X.G) &\stackrel{\text{def}}{=} fn(G) \setminus \{X\}.
\end{aligned}
$$

Figure 2.2: The set of free link names

Rules have the form $G \longrightarrow G$. The two Gs are called the left-hand side (LHS) and the right-hand side (RHS), respectively.

Definition 2.2.3 (Syntactic condition on rules)**.** A rule $G_1 \longrightarrow G_2$ should satisfy $fn(G_1) \supseteq fn(G_2)$.

The condition indicates that we must denote a new hyperlink in the scope of a ν (new) on the RHS of a rule.

The semantics of HyperLMNtal comes with two major ingredients, structural congruence $\equiv$ and reduction relation $\rightsquigarrow$ on graphs.

2.2 Structural Congruence

Since graphs are more complex than trees, it is not sufficient to define the syntax of the term language but we need to define an equivalence relation. For the purpose, we exploit the notion of *Structural Congruence*, which originates from process algebra. Structural congruence defines what terms (represented in the syntax of Figure 2.1) are essentially the same.

Definition 2.2.4 (Link Substitution)**.** $G\langle Y_1, \ldots, Y_n / X_1, \ldots, X_n \rangle$ is a *link substitution* that replaces all free occurrences of X_i with Y_i as defined in Figure 2.3. Here, the

$$X\langle \overrightarrow{Z}/\overrightarrow{Y}\rangle = \begin{cases} Z_i & \text{if } X = Y_i \\ X & \text{if } X \notin \{\overrightarrow{Y}\} \end{cases}$$

$$\mathbf{0}\sigma = \mathbf{0}$$

$$p(\overrightarrow{X})\sigma = p(X_1\sigma, \ldots, X_n\sigma)$$

$$(G_1, G_2)\sigma = (G_1\sigma, G_2\sigma)$$

$$(\nu X.G)\langle \overrightarrow{Z}/\overrightarrow{Y}\rangle = \begin{cases} \nu X.G\langle \overrightarrow{Z'}/\overrightarrow{Y'}\rangle & \text{if } X = Y_i \\ & \quad \wedge\ \overrightarrow{Z'} = Z_1, \ldots, Z_{i-1}, Z_{i+1}, \ldots, Z_n \\ & \quad \wedge\ \overrightarrow{Y'} = Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n \\ \nu X.G\langle \overrightarrow{Z}/\overrightarrow{Y}\rangle & \text{if } X \notin \{\overrightarrow{Y}\} \wedge X \notin \{\overrightarrow{Z}\} \\ \nu W.((G\langle W/X\rangle)\langle \overrightarrow{Z}/\overrightarrow{Y}\rangle) & \text{if } X \notin \{\overrightarrow{Y}\} \wedge X \in \{\overrightarrow{Z}\} \\ & \quad \wedge\ W \notin fn(G) \wedge W \notin \{\overrightarrow{Z}\} \wedge W \neq X \end{cases}$$

Figure 2.3: Hyperlink Substitution

$X_1, \ldots, X_n$ should be mutually distinct. Note that, if a free occurrence of X_i occurs at a location where Y_i would not be free, α-conversion may be required.

Definition 2.2.5 (Structural Congruence)**.** We define the relation $\equiv$ on graphs as the minimal equivalence relation satisfying the rules shown in Figure 2.4. Two graphs related by $\equiv$ are essentially the same and are convertible to each other in zero steps.

(E1), (E2) and (E3) are the characterization of molecules as multisets. (E4) and (E5) are structural rules that make $\equiv$ a congruence. (E6) and (E7) are concerned with fusions. (E7) says that a closed fusion is equivalent to $\mathbf{0}$. (E6) is an absorption law of $\bowtie$, which says that a fusion can be absorbed by connecting hyperlinks. Because of the symmetry of $\bowtie$, (E6) says that an atom can emit a fusion as well. (E8), (E9) and (E10) are concerned with hyperlink creations.

We give two important theorems showing that the symmetry of $\bowtie$ and α-conversion can be derived from the rules of Figure 2.4.

Theorem 2.2.1 (Symmetry of $\bowtie$)**.**

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

Proof. See Chapter 3 of [San21]. □

(E1)	$(\mathbf{0}, G)$	$\equiv$	G	
(E2)	(G_1, G_2)	$\equiv$	(G_2, G_1)	
(E3)	$(G_1, (G_2, G_3))$	$\equiv$	$((G_1, G_2), G_3)$	
(E4)	$G_1 \equiv G_2$	$\Rightarrow$	$(G_1, G_3) \equiv (G_2, G_3)$	
(E5)	$G_1 \equiv G_2$	$\Rightarrow$	$\nu X.G_1 \equiv \nu X.G_2$	
(E6)	$\nu X.(X \bowtie Y, G)$	$\equiv$	$\nu X.G\langle Y/X\rangle$	where $X \in fn(G) \vee Y \in fn(G)$
(E7)	$\nu X.\nu Y.X \bowtie Y$	$\equiv$	$\mathbf{0}$	
(E8)	$\nu X.\mathbf{0}$	$\equiv$	$\mathbf{0}$	
(E9)	$\nu X.\nu Y.G$	$\equiv$	$\nu Y.\nu X.G$	
(E10)	$\nu X.(G_1, G_2)$	$\equiv$	$(\nu X.G_1, G_2)$	where $X \notin fn(G_2)$

Figure 2.4: Structural congruence on HyperLMNtal graphs

Thus, (E6) can be used also when we have a local link on the right-hand side of $\bowtie$.

Theorem 2.2.2 (α-conversion of hyperlinks)**.** *Bound link names are α-convertible in HyperLMNtal, i.e.,*

$$\nu X.G \equiv \nu Y.G\langle Y/X\rangle \text{ where } Y \notin fn(G)$$

Proof. See Chapter 3 of [San21]. □

Properties of Fusions

Here, we show that fusions simply define an equivalence relation in the congruence rules. These properties are not trivial since X, Y and Z are free links and a fusion cannot fuse free links (Notice that (E6) fuses X to Y where X is a local link).

Firstly, Theorem 2.2.3 shows that we can rename any link X to Y, and vice versa, if there exists a fusion $X \bowtie Y$, even if X and Y are both free link names. For example, $(X \bowtie L, \mathrm{Leaf}(\mathrm{Zero}, X, R)) \equiv (X \bowtie L, \mathrm{Leaf}(\mathrm{Zero}, L, R))$. Also, $(X \bowtie X, X \bowtie Y) \equiv (X \bowtie Y, X \bowtie Y)$ holds as a corollary, which we will use later to prove Theorem 2.2.5.

Theorem 2.2.3 (Renaming free links using a fusion)**.** $(X \bowtie Y, G\langle X/Z\rangle) \equiv (X \bowtie Y, G\langle Y/Z\rangle)$.

Proof.

	$(X \bowtie Y, G\langle X/Z\rangle)$
$\equiv_{\text{(E6)}}$	$\nu Z.(Z \bowtie X, (Z \bowtie Y, G))$
$\equiv_{\text{(E2), (E3), (E4)}}$	$\nu Z.(Z \bowtie Y, (Z \bowtie X, G))$
$\equiv_{\text{(E6)}}$	$\nu Z.(Y \bowtie X, G\langle Y/Z\rangle)$
$\equiv_{\text{(Symmetricity of fusion)}}$	$\nu Z.(X \bowtie Y, G\langle Y/Z\rangle)$
$\equiv_{\text{(Absorb unused name binding)}}$	$(X \bowtie Y, G\langle Y/Z\rangle)$

□

An equivalence relation should satisfy reflexivity, which can be understood as Theorem 2.2.4.

Theorem 2.2.4 (Reflexivity of fusions)**.** *For all G, such that $X \in fn(G)$, $G \equiv (X \bowtie X, G)$.*

The condition $X \in fn(G)$ is necessary not to change the set of free link names in congruent terms.

Proof.

$$
\begin{aligned}
& G \\
\equiv_{\text{(Absorb unused name binding)}} \quad & \nu Z.G \\
\equiv_{\text{(E6)}} \quad & \nu Z.(Z \bowtie X, G) \\
\equiv_{\text{(E6)}} \quad & \nu Z.(Z \bowtie Z, (Z \bowtie X, G)) \\
\equiv_{\text{(E2), (E3), (E4), (E5)}} \quad & \nu Z.(Z \bowtie X, (Z \bowtie Z, G)) \\
\equiv_{\text{(E6)}} \quad & \nu Z.(X \bowtie X, G) \\
\equiv_{\text{(Absorb unused name binding)}} \quad & (X \bowtie X, G)
\end{aligned}
$$

□

Also, we show that the same fusions can be contracted as in Theorem 2.2.5. This characterise fusions to be not a multiset but a set; i.e., the multiplicity of fusions does not matter.

Theorem 2.2.5 (Fusion Contraction)**.** $X \bowtie Y \equiv (X \bowtie Y, X \bowtie Y)$.

Proof.

$$
\begin{aligned}
& X \bowtie Y \\
\equiv_{\text{Theorem 2.2.4}} \quad & (X \bowtie X, X \bowtie Y) \\
\equiv_{\text{(E2), Theorem 2.2.3}} \quad & (X \bowtie Y, X \bowtie Y)
\end{aligned}
$$

□

Finally, we show that fusions satisfies transitivity as in Theorem 2.2.6.

Theorem 2.2.6 (Transitivity of fusions)**.** $(X \bowtie Y, Y \bowtie Z) \equiv (X \bowtie Z, (X \bowtie Y, Y \bowtie Z))$.

Proof.

$$
\begin{aligned}
& (X \bowtie Y, Y \bowtie Z) \\
\equiv_{\text{Theorem 2.2.4}} \quad & (X \bowtie X, (X \bowtie Y, Y \bowtie Z)) \\
\equiv_{\text{(E2), (E3), (E4), Theorem 2.2.3}} \quad & (X \bowtie Y, (X \bowtie Y, Y \bowtie Z)) \\
\equiv_{\text{(E2), (E3), (E4), Theorem 2.2.3}} \quad & (X \bowtie Z, (X \bowtie Y, Y \bowtie Z))
\end{aligned}
$$

□

2.2 Reduction Relation

We give the reduction relation of HyperLMNtal that defined the small-step semantics of the language. Note, however, that λ_{GT} described in the next subsection has its own operational semantics without incorporating the reduction relation described here. We nevertheless introduce the reduction relation of HyperLMNtal here because it serves as the basis of the graph types of λ_{GT} described in Chapter 4.

Definition 2.2.6 (Reduction Relation)**.** For a set $P \stackrel{\text{def}}{=} \{\overrightarrow{r}\}$ of rewrite rules, the reduction relation $\rightsquigarrow_P$ on graphs is defined as the minimal relation satisfying the rules in Figure 2.5.

$$\text{(R1)} \quad \frac{G_1 \rightsquigarrow_P G_2}{(G_1, G_3) \rightsquigarrow_P (G_2, G_3)}$$

$$\text{(R2)} \quad \frac{G_1 \rightsquigarrow_P G_2}{\nu X.G_1 \rightsquigarrow_P \nu X.G_2}$$

$$\text{(R3)} \quad \frac{G_1 \equiv G_2 \qquad G_2 \rightsquigarrow_P G_3 \qquad G_3 \equiv G_4}{G_1 \rightsquigarrow_P G_4}$$

$$\text{(R4)} \quad \frac{(G_1 \longrightarrow G_2) \in P}{G_1 \rightsquigarrow_P G_2}$$

Figure 2.5: Reduction relation on HyperLMNtal graphs

2.3 The Denoted Hypergraphs

In this section, we introduce the hypergraphs denoted by the language.

The graph denoted by the language has labelled vertices and edges connected to vertices through *ports*. Such graphs are called labelled port graphs [FP18], but ours extend the edges to hyperedges.

Our graphs also have a notion of *free links*, which are links that can be connected to the outer graphs. This naturally extends the existing definitions of hypergraphs [Ehr+06].

Definition 2.3.1 (Graph-Theoretic Graph)**.** A graph-theoretic graph is represented by a quadruple $\mathscr{G} \stackrel{\text{def}}{=} \langle V, F, L, \mathscr{L} \rangle$, where:

1. V is a set of vertices,
2. the triple $F \stackrel{\text{def}}{=} \langle \mathbf{X}, R, \pi \rangle$, denotes *free links* where:
   (a) $\mathbf{X}$ is a set of free link names,
   (b) R is an equivalence relation on a set of free link names $\mathbf{X}$, and
   (c) $\pi : \mathbf{X} \to \mathscr{P}(V \times \mathbb{N}_{>0})$ is a function from link names to sets of ports.

3. $L \subset \mathscr{P}(V \times \mathbb{N}_{>0})$ is a set of *local links*, quotient set of ports, and

4. $\mathscr{L} : V \to \mathbb{P}$ is a labelling function.

The definition of hypergraphs in graph theory [Ehr+06] is basically a triple of vertices, hyperedges and a labelling function as follows: $\langle V, E, \mathscr{L} \rangle$ where $E \subseteq \mathscr{P}(V)$. Notice that $L \subset \mathscr{P}(V \times \mathbb{N}_{>0})$ in our graphs generalises E in hypergraphs in graph theory. Also, our graphs have the notion of free links F, which does not exist in the definition of hypergraphs in graph theory. Our graphs also feature the notion of ports. Thus, our graphs are more generalised than that in graph theory.

Since R defines the equivalence classes of free link names, i.e., sets of the free links that are connected, and π defines the set of ports that a free link is connected to, π should return the same set of ports if and only if it took free link names that are in the same equivalence class. Or more formally, π and R should satisfy the relations in Definition 2.3.2.

Definition 2.3.2 (Well-definedness condition of π)**.** π and R in Definition 2.3.1should satisfy $X_1 \, R \, X_2 \Rightarrow \pi(X_1) = \pi(X_2)$ and $X_1 \, \neg R \, X_2 \Rightarrow \pi(X_1) \cap \pi(X_2) = \varnothing$.

Example 2.3.1 (A Singleton List in Graph-Theoretic Graph notation)**.** *The graph-theoretic graph of Example 2.2.1 is*

$$\langle \{v_1, v_2\}, \langle \{X\}, \{\{X\}\}, \{X \mapsto \{\langle v_1, 2\rangle\}\}\rangle, \{\{\langle v_1, 1\rangle, \langle v_2, 1\rangle\}\}, \{v_1 \mapsto \mathrm{Cons}, v_2 \mapsto \mathrm{Nil}\}\rangle$$

in which the equivalence relation is represented with a quotient set as $\{\{X\}\}$.

For graph $\mathscr{G}$, we denote $fn_{\mathscr{G}}(\mathscr{G})$ to obtain the set of free link names in $\mathscr{G}$: i.e., $fn_{\mathscr{G}}(\langle V, \langle \mathbf{X}, R, \pi \rangle, L, \mathscr{L} \rangle) \stackrel{\text{def}}{=} \mathbf{X}$. We use $\mathbf{G}$ to denote a *non-empty* set of (possibly infinite) graphs. $\mathbb{G}$ is a *non-empty* set of *all* graphs.

We define a notation for an empty set of free links and empty graph since they are likely to be identity elements of the operations we will define later.

Definition 2.3.3 (Empty set of free links and empty graph)**.** $F^0 \stackrel{\text{def}}{=} \langle \varnothing, \varnothing, \varnothing \rangle$, $\mathscr{G}^0 \stackrel{\text{def}}{=} \langle \varnothing, F^0, \varnothing, \varnothing \rangle$.

Mappings of Graphs

In order to discuss the properties of graphs denoted by HyperLMNtal, we define a mapping of graphs.

We firstly introduce some general combinators for convenience. Notice that these are just syntactic sugars that can be replaced with verbose expressions that do not use combinators. In accordance with the conventions of functional languages, parentheses are sometimes omitted in the application, and function application is left-associative.

Definition 2.3.4 (General Combinators)**.**

1. $app_i\ f\ \left\langle \overrightarrow{x_j}^j \right\rangle \stackrel{\text{def}}{=} \left\langle \overrightarrow{y_j}^j \right\rangle$ where $y_j = \begin{cases} f(x_j) & \text{if } j = i \\ x_j & \text{if } j \neq i. \end{cases}$
2. $map_{\mathtt{S}}\ f\ S \stackrel{\text{def}}{=} \{f(s) \mid s \in S\}$.
3. $map_{\mathtt{R}}\ f\ R \stackrel{\text{def}}{=} \{\langle f(x), f(y)\rangle \mid \langle x, y\rangle \in R\}$.
4. $fst \left\langle \overrightarrow{x_i}^i \right\rangle \stackrel{\text{def}}{=} x_1$, $snd \left\langle \overrightarrow{x_i}^i \right\rangle \stackrel{\text{def}}{=} x_2$, and $thd \left\langle \overrightarrow{x_i}^i \right\rangle \stackrel{\text{def}}{=} x_3$.

Theorem 2.3.1 ($map_{\mathtt{S}}\ f$ is homomorphic over $(\mathscr{P}(S), \cup, \varnothing)$)**.** *For an arbitrary set S and a function f, $map_{\mathcal{S}}\ f\ (S_1 \cup S_2) = (map_{\mathcal{S}}\ f\ S_1) \cup (map_{\mathcal{S}}\ f\ S_2)$ holds for any set $S_1, S_2 \subseteq S$.*

Proof. Trivial from the definition. □

Definition 2.3.5 (Restriction of a relation)**.** $R \setminus x \stackrel{\text{def}}{=} \{\langle y, y'\rangle \mid \langle y, y'\rangle \in R, y \neq x, y' \neq x\}$.

Notice that if R is an equivalence relation on a set S, $R \setminus x$ is an equivalence relation on a set $S \setminus \{x\}$.

Using the combinators we have defined in Definition 2.3.4, we define some mappings for graphs as follows:

Definition 2.3.6 (Graph Mappings)**.**

1. $map^V_{\mathscr{G}}\ f\ \langle V, F, L, \mathscr{L}\rangle \stackrel{\text{def}}{=} \left\langle map_{\mathtt{S}}\ f\ V, map^V_F\ f\ F, (map_{\mathtt{S}} \circ map_{\mathtt{S}} \circ app_1)\ f\ L, \mathscr{L} \circ f^{-1}\right\rangle$ where $map^V_F\ f\ \langle \mathbf{X}, R, \pi\rangle \stackrel{\text{def}}{=} \langle \mathbf{X}, R, ((map_{\mathtt{S}} \circ app_1)\ f) \circ \pi\rangle$ where f must be a bijection.
2. $map^X_{\mathscr{G}}\ f\ \langle V, F, L, \mathscr{L}\rangle \stackrel{\text{def}}{=} \left\langle V, map^X_F\ f\ F, L, \mathscr{L}\right\rangle$ where $map^X_F\ f\ \langle \mathbf{X}, R, \pi\rangle \stackrel{\text{def}}{=} \langle \mathbf{Y}, R', \pi'\rangle$ where $\mathbf{Y} \stackrel{\text{def}}{=} map_{\mathtt{S}}\ f\ \mathbf{X}$, $R' \stackrel{\text{def}}{=} (map_{\mathtt{R}}\ f\ R)^{\equiv}$, and $\pi' : \mathbf{Y} \to V \times \mathbb{N}_{>0}, Y \mapsto \bigcup_{X \in f^{-1}([Y]_{R'})} \pi(X)$.
3. $map^X_{\mathbf{G}} \stackrel{\text{def}}{=} map_{\mathtt{S}} \circ map^X_{\mathscr{G}}$.

$map^V_{\mathscr{G}}$ simply applies f to all the vertices that appear in the given graph. $map^X_{\mathscr{G}}$ applies f to the free link names in the given graph, but since the f may *merge* the equivalence classes of free link names, we need to redefine them carefully. R' is the smallest equivalence relation that contains $(map_{\mathtt{R}}\ f\ R)$. Since R is an equivalence relation and is symmetric, $(map_{\mathtt{R}}\ f\ R)$ is also symmetric, thus $(map_{\mathtt{R}}\ f\ R)^{\equiv} = (map_{\mathtt{R}}\ f\ R)^*$. Then π' is defined according to R' to meet Definition 2.3.2. If f is a bijection, the operation can be much simpler: $R' \stackrel{\text{def}}{=} (map_{\mathtt{R}}\ f\ R)$ and $\pi' \stackrel{\text{def}}{=} \pi \circ f^{-1}$ by Definition 2.3.2.

We check that the operation preserves the well-definedness property of π (Definition 2.3.2).

Theorem 2.3.2. *π satisfies the well-definedness condition (Definition 2.3.2) after $map^V_{\mathscr{G}}$ and $map^X_{\mathscr{G}}$. Or more formally, for all $\mathscr{G}$, f, and $\langle V, \langle \mathbf{X}, R, \pi\rangle, L, \mathscr{L}\rangle \stackrel{\text{def}}{=} map^V_{\mathscr{G}}\ f\ \mathscr{G}$, $X_1\ R\ X_2 \Rightarrow \pi(X_1) = \pi(X_2)$ and $X_1\ \neg R\ X_2 \Rightarrow \pi(X_1) \cap \pi(X_2) = \varnothing$ holds, and so does with $map^X_{\mathscr{G}}$.*

Proof. It is trivial for $map^V_{\mathscr{G}}\ f$ since f is a bijection. For $map^X_{\mathscr{G}}\ f$, $Y_1\ R'\ Y_2 \Rightarrow \pi'(Y_1) = \pi'(Y_2)$ is trivial from the definition and $Y_1\ \neg R'\ Y_2 \Rightarrow \pi(Y_1) \cap \pi(Y_2) = \varnothing$ also holds since $\forall X_1, X_2 \in \mathbf{X}, Y_1 = f(X_1), Y_2 = f(X_2).(X_1\ R\ X_2 \Rightarrow Y_1\ (map_{\mathrm{R}}\ f\ R)\ Y_2 \Rightarrow Y_1\ R'\ Y_2)$, $\forall Y_1, Y_1 \in \mathbf{Y}, X_1 \in f^{-1}([Y_1]_{R'}), X_1 \in f^{-1}([Y_2]_{R'}).(Y_1\ \neg R'\ Y_2 \Rightarrow X_1\ \neg R\ X_2 \Rightarrow \pi_1(X_1) \cap \pi_2(X_2) = \varnothing)$. □

Example 2.3.2 (A Singleton List)**.**

$\mathscr{G}_1 \stackrel{\text{def}}{=} \langle \{v_1, v_2\}, \langle \{X\}, \{\{X\}\}, \{X \mapsto \{\langle v_1, 2\rangle\}\}\rangle, \{\{\langle v_1, 1\rangle, \langle v_2, 1\rangle\}\}, \{v_1 \mapsto \mathrm{Cons}, v_2 \mapsto \mathrm{Nil}\}\rangle$,

$\mathscr{G}_2 \stackrel{\text{def}}{=} \langle \{v_3, v_4\}, \langle \{X\}, \{\{X\}\}, \{X \mapsto \{\langle v_3, 2\rangle\}\}\rangle, \{\{\langle v_3, 1\rangle, \langle v_4, 1\rangle\}\}, \{v_3 \mapsto \mathrm{Cons}, v_4 \mapsto \mathrm{Nil}\}\rangle$

then $map^V_{\mathscr{G}}\ f_V\ \mathscr{G}_1 = \mathscr{G}_2$ *where* $f_V = \{v_1 \mapsto v_3, v_2 \mapsto v_4\}$.

Theorem 2.3.3. $map^V_{\mathscr{G}}\ f\ \mathscr{G}^0 = \mathscr{G}^0$ *and* $map^X_{\mathscr{G}}\ f\ \mathscr{G}^0 = \mathscr{G}^0$ *for any* f.

Proof. Trivial from the definitions. □

Using the mappings we have defined so far, we define a graph isomorphism, $\mathscr{G} \cong \mathscr{G}'$, as in Definition 2.3.7. For example, $\mathscr{G}_1$ and $\mathscr{G}_2$ in Example 2.3.2 are graph-isomorphic, i.e., $\mathscr{G}_1 \cong \mathscr{G}_2$.

Definition 2.3.7 (Graph isomorphism)**.** $\mathscr{G} \cong \mathscr{G}'$ iff $\exists f_V.(\mathscr{G}' = map^V_{\mathscr{G}}\ f_V\ \mathscr{G})$ where $f_V : fst(\mathscr{G}) \to fst(\mathscr{G}')$ is a bijection.

2.4 Hypergraph Operations and a Translator From Terms to Graphs

In this section, *terms* simply refers to HyperLMNtal graphs, and *graphs* refers to graph-theoretic graphs.

We define some operations on graphs and discuss their properties. Then, define how to transform HyperLMNtal terms to graph-theoretic graphs.

2.4 Graph Operations

We define some operations on graphs and discuss their properties.

Merging Free Links

To compose graphs from subgraphs, intuitively, we need to be able to connect edges. This corresponds to the operations that we introduce in the following:

Definition 2.4.1 (Merging free links)**.** $\langle \mathbf{X}_1, R_1, \pi_1\rangle + \langle \mathbf{X}_2, R_2, \pi_2\rangle \stackrel{\text{def}}{=} \langle \mathbf{X}_1 \cup \mathbf{X}_2, R, \pi\rangle$ where $R \stackrel{\text{def}}{=} (R_1 \cup R_2)^*$ and $\pi : \mathbf{X}_1 \cup \mathbf{X}_2 \to \mathscr{P}\left(\left(\bigcup_{X_1 \in \mathbf{X}_1} \pi_1(X_1)\right) \sqcup \left(\bigcup_{X_2 \in \mathbf{X}_2} \pi_2(X_2)\right)\right)$, $X \mapsto \left(\bigcup_{X_1 \in \mathbf{X}_1 . X_1 R X} \pi_1(X_1)\right) \sqcup \left(\bigcup_{X_2 \in \mathbf{X}_2 . X_2 R X} \pi_2(X_2)\right)$.

The operation can be done if and only if the π_1 and π_2 satisfies the following: $\forall ports_1 \in \pi_1(\mathbf{X}_1), ports_2 \in \pi_2(\mathbf{X}_2).(ports_1 \cap ports_2 = \varnothing)$.

Here, R is the smallest equivalence relation that contains $R_1 \cup R_2$. Notice that the operation is defined only when all the ports in the images of π_1 and π_2 are disjoint. This condition is required to ensure the property Definition 2.3.2 after the operation as we will prove in Theorem 2.4.7. We also use a summation notation as follows: $\sum_{1 \le i \le n} F_i \stackrel{\text{def}}{=} F_1 + \cdots + F_n$.

Example 2.4.1.

$$\begin{aligned}
&\big\langle \{X_1, X_2, X_3, X_4, X_5, X_6\}, \{\{X_1, X_2\}, \{X_3, X_4\}, \{X_5, X_6\}\}, \\
&\quad \{X_1, X_2 \mapsto \{\langle v_1, 1\rangle\}, X_3, X_4 \mapsto \{\langle v_1, 2\rangle, \langle v_2, 2\rangle\}, X_5, X_6 \mapsto \{\langle v_2, 1\rangle\}\}\big\rangle \\
+ \quad &\big\langle \{X_2, X_6, X_7\}, \{\{X_2, X_6\}, \{X_7\}\}, \{X_2, X_6 \mapsto \{\langle v_3, 1\rangle\}, X_7 \mapsto \{\langle v_4, 1\rangle\}\}\big\rangle \\
= &\big\langle \{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}, \{\{X_1, X_2, X_5, X_6\}, \{X_3, X_4\}, \{X_7\}\}, \\
&\quad \{X_1, X_2, X_5, X_6 \mapsto \{\langle v_1, 1\rangle, \langle v_2, 1\rangle, \langle v_3, 1\rangle\}, X_3, X_4 \mapsto \{\langle v_1, 2\rangle, \langle v_2, 2\rangle\}, X_7 \mapsto \{\langle v_4, 1\rangle\}\}\big\rangle
\end{aligned}$$

We check that the operation preserves the well-definedness property of π (Definition 2.3.2).

Theorem 2.4.1. *π satisfies the well-definedness property (Definition 2.3.2) after the free link merge.*

Proof. $X\ R'\ X' \Rightarrow \pi(X) = \pi(X')$ is trivial from the definition and $X\ \neg R'\ X' \Rightarrow \pi(X) \cap \pi(X') = \varnothing$ is also trivial with the side condition of the operation. □

Theorem 2.4.2. *$+$ over $\mathbb{F}$ is commutative, associative, and has an identity element F^0.*

Proof. Straightforward using the property Definition 2.3.2. □

Theorem 2.4.3 ($map_F^V\ f$ from/to $(\mathbb{F}, +, F^0)$ is an endomorphism)**.** *$map_F^V\ f\ (F_1 + F_2) = map_F^V\ f\ F_1 + map_F^V\ f\ F_2$.*

Proof. For $\mathbf{X}$ and R, by definition, $map_F^V\ f$ does not change the values. Thus, the theorem holds trivially. For π, the theorem holds because $((map_{\mathtt{s}} \circ app_i)\ f, \varnothing)$ is an automorphism over $(\mathscr{P}(V \times \mathbb{N}_{>0}), \cup, \varnothing)$ by Theorem 2.3.1. □

Graph Composition

Using the operation Definition 2.4.1, we can compose graphs from subgraphs as follows:

Definition 2.4.2 (Graph Composition)**.**
$\langle V_1, F_1, L_1, \mathscr{L}_1\rangle + \langle V_2, F_2, L_2, \mathscr{L}_2\rangle \stackrel{\text{def}}{=} \langle V_1 \sqcup V_2, F_1 + F_2, L_1 \sqcup L_2, \mathscr{L}_1 \sqcup \mathscr{L}_2\rangle$ where $V_1 \cap V_2 = \varnothing$.

Notice that when V_1 and V_2 are disjoint, L_1 and L_2, domains of $\mathscr{L}_1$ and $\mathscr{L}_2$, and the ports in the images of $\pi_1 \stackrel{\text{def}}{=} thd(F_1)$ and $\pi_2 \stackrel{\text{def}}{=} thd(F_2)$ are disjoint, respectively, which is the required condition in $+$ over $\mathbb{F}$, too.

Theorem 2.4.4. *$+$ over $\mathbb{G}$ is commutative, associative, and has an identity element $\mathscr{G}^0$.*

Proof. Straightforward by Theorem 2.4.2.

□

Theorem 2.4.5 ($map^V_{\mathscr{G}}\ f_V$ from/to $(\mathbb{G}, +, \mathscr{G}^0)$ is an endomorphism)**.**
$map^V_{\mathscr{G}}\ f\ (\mathscr{G}_1 + \mathscr{G}_2) = (map^V_{\mathscr{G}}\ f\ \mathscr{G}_1) + (map^V_{\mathscr{G}}\ f\ \mathscr{G}_2)$ *where* $f(fst(\mathscr{G}_1)) \cap f(fst(\mathscr{G}_2)) = \varnothing$.

Proof. For V and L, $+$ performs set union and $map^V_{\mathscr{G}}$ performs $map_{\mathtt{S}}$, thus the theorem holds by Theorem 2.3.1. For $\mathscr{L}$, the theorem holds since $(\mathscr{L}_1 \sqcup \mathscr{L}_2) \circ f^{-1} = (\mathscr{L}_1 \circ f^{-1}) \sqcup (\mathscr{L}_2 \circ f^{-1})$. For F, the theorem holds by Theorem 2.4.3.

□

Since we need to compose all the graphs in the sets of graphs to define a mapping from HyperLMNtal terms to graphs, we define a notation for that as follows:

Definition 2.4.3 (Graph Set Union)**.**
$\mathbf{G}_1 \otimes \mathbf{G}_2 \stackrel{\text{def}}{=} \{\mathscr{G}_1 + \mathscr{G}_2 \mid \mathscr{G}_1 \in \mathbf{G}_1, \mathscr{G}_2 \in \mathbf{G}_2, fst(\mathscr{G}_1) \cap fst(\mathscr{G}_2) = \varnothing\}$

Theorem 2.4.6. $(\mathscr{P}(\mathbb{G}), \otimes, \{\mathscr{G}^0\})$ *is a commutative monoid.*

Proof. Straightforward from Theorem 2.4.4. □

Notice that $\varnothing$ is not a unit of the monoid since $\mathbf{G} \otimes \varnothing = \varnothing$, and $\mathbf{G} \otimes \varnothing = \mathbf{G}$ does not always hold, either.

Link Name Hiding

Since the HyperLMNtal terms have a notion of a scope of link names, we need to define an operation that corresponds to it.

Definition 2.4.4 (Link Name Hiding)**.**

$$\langle V, \langle \mathbf{X}, R, \pi\rangle, L, \mathscr{L}\rangle \setminus X \stackrel{\text{def}}{=} \begin{cases} \langle V, \langle \mathbf{X}, R, \pi\rangle, L, \mathscr{L}\rangle & (X \notin \mathbf{X}) \\ \langle V, \langle \mathbf{X}', R', \pi'\rangle, L', \mathscr{L}\rangle & (X \in \mathbf{X}) \end{cases}$$

where $\mathbf{X}' \stackrel{\text{def}}{=} \mathbf{X} \setminus \{X\}$, $R' \stackrel{\text{def}}{=} R \setminus X$, $\pi' \stackrel{\text{def}}{=} \pi|_{\mathbf{X}'}$, and $L' \stackrel{\text{def}}{=} \begin{cases} L & \exists X' \in \mathbf{X}'. X\ R\ X' \\ L \cup \{\pi(X)\} & \text{otherwise.} \end{cases}$

We discuss the properties with the link name hiding operation. First of all, π satisfies the well-definedness property (Definition 2.3.2) after the link name hiding.

Theorem 2.4.7. *For any* $\mathscr{G}$, X *and* $\langle V, \langle \mathbf{X}, R, \pi\rangle, L, \mathscr{L}\rangle \stackrel{\text{def}}{=} \mathscr{G} \setminus X$, $X_1\ R\ X_2 \Rightarrow \pi(X_1) = \pi(X_2)$ *and* $X_1\ \neg R\ X_2 \Rightarrow \pi(X_1) \cap \pi(X_2) = \varnothing$.

Proof. Trivial from the definition. □

We discuss the distributivity of graph composition and link name hiding. This property does not always hold as shown in the following:

Remark 1. *There exists a* $\mathscr{G}_1, \mathscr{G}_2$, *and* X *such that* $fst(\mathscr{G}_1) \cap fst(\mathscr{G}_2) = \varnothing$ *and* $(\mathscr{G}_1 + \mathscr{G}_2) \setminus X \neq (\mathscr{G}_1 \setminus X) + (\mathscr{G}_2 \setminus X)$*; for example, graphs* $\mathscr{G}_1$ *and* $\mathscr{G}_2$ *which are connected with the link* X*.*

We need some restriction for the property to hold.

Theorem 2.4.8. $(\mathscr{G}_1 + \mathscr{G}_2) \setminus X = (\mathscr{G}_1 \setminus X) + \mathscr{G}_2$ *if* $X \notin fn_{\mathscr{G}}(\mathscr{G}_2)$*.*

To prove the theorem, we need to prove some lemmas.

Lemma 2.4.1. *Let* R_1 *and* R_2 *be equivalence relations on sets* S_1 *and* S_2*, respectively, then* $(R_1 \cup R_2)^* \setminus x = ((R_1 \setminus x) \cup R_2)^*$ *holds if* $x \notin S_2$*.*

Proof. $(R_1 \cup R_2)^* \setminus x \supseteq ((R_1 \setminus x) \cup R_2)^*$ is trivial from the definition. By definition, $(R_1 \cup R_2)^* \setminus x = \{\langle y_0, y_n\rangle \mid y_0 \neq x, y_n \neq x, y_0\ (R_1 \cup R_2) \cdots (R_1 \cup R_2)\ y_n\}$. Here, for all y_{i-1}, y_i, y_{i+1} such that $y_{i-1}\ (R_1 \cup R_2)\ y_i\ (R_1 \cup R_2)\ y_{i+1}$, if $y_i = x$ then since $x \notin S_2$, we need to use R_1 before and after y_i, that is, they should satisfy $y_{i-1}\ R_1\ y_i\ R_1\ y_{i+1}$. Since R_1 is an equivalence relation and is transitive, $y_{i-1}\ R_1\ y_{i+1}$ holds: we can omit any y_i $(0 < i < n)$ such that $y_i = x$ in $y_0 (R_1 \cup R_2) \cdots (R_1 \cup R_2) y_n$. Therefore, $(R_1 \cup R_2)^* \setminus x \subseteq ((R_1 \cup R_2) \setminus x)^* = ((R_1 \setminus x) \cup R_2)^*$. □

Lemma 2.4.2. *Let* R_1 *and* R_2 *be equivalence relations on sets* S_1 *and* S_2*, respectively. Then,* $x\ (R_1 \cup R_2)^*\ y \Rightarrow \exists y' \in S_1.(y' \neq x \wedge x\ R_1\ y' \wedge y'\ (R_1 \cup R_2)^*\ y)$ *if* $x \notin S_2$ *and* $y \neq x$*.*

Then, we prove the Theorem 2.4.8.

Proof of Theorem 2.4.8. Let $x\ (R_1 \cup R_2)^*\ y$ be such that $x \notin S_2$ and $y \neq x$, which is equivalent to $x\ (R_1 \cup R_2)\ y_1\ (R_1 \cup R_2) \cdots (R_1 \cup R_2)\ y_n$ where $y_n = y$. Since $x \notin S_2$, y_1 should satisfy $x\ R_1\ y_1$. If $y_1 = x$, then $x\ R_1\ y_1\ (R_1 \cup R_2) \cdots (R_1 \cup R_2)\ y_n$ can be rewritten be $x\ R_1\ y_1\ R_1\ y_2\ (R_1 \cup R_2) \cdots (R_1 \cup R_2)\ y_n$. Repeating the operation less than n times, we can obtain y' such that $y' \neq x \wedge x\ R_1 \cdots R_1\ y'\ (R_1 \cup R_2) \cdots (R_1 \cup R_2)\ y_n$ since $y_n \neq x$. Since R_1 is an equivalence relation and is transitive, $x\ R_1\ y'$ holds. □

Proof. Since $\setminus X$ only changes F and L, we only need to consider these. Let the free links of $\mathscr{G}_i$ as $\langle \mathbf{X}_i, R_i, \pi_i\rangle$. For $\mathbf{X}$, the theorem holds trivially. For R, it follows from Lemma 2.4.1. Let $R \stackrel{\text{def}}{=} (R_1 \cup R_2)^*$.

For π, for all $Y \in (\mathbf{X}_1 \cup \mathbf{X}_2)$ such that $X \neq Y$, if $X\ R\ Y$ then by Lemma 2.4.2, there exists $Y' \in \mathbf{X}_1$ such that $Y' \neq X \wedge X\ R_1\ Y' \wedge Y'\ R\ Y$ since X is not in the support set of R_2. Therefore, by Definition 2.3.2, $\pi_1(X) = \pi_2(Y')$. Thus, since $Y' \in \mathbf{X}_1 \setminus \{X\} \wedge Y'(R \setminus X)Y$, $\pi_1(X) = \pi_2(Y') \subseteq \bigcup_{Y_1 \in \mathbf{X}_1 \setminus \{X\}.Y_1 (R \setminus X) Y} \pi_1(Y_1)$. Therefore, $\bigcup_{Y_1 \in \mathbf{X}_1.Y_1 R Y} \pi_1(Y_1) =$

$$\bigcup_{Y_1 \in \mathbf{X}_1 \setminus \{X\}.Y_1 (R \setminus X) Y} \pi_1(Y_1) \cup \begin{cases} \pi_1(Y_1) & \text{if } X\ R\ Y \\ \varnothing & \text{otherwise} \end{cases} = \bigcup_{Y_1 \in \mathbf{X}_1 \setminus \{X\}.Y_1 (R \setminus X) Y} \pi_1(Y_1),$$

thus the theorem holds.

For L, by Lemma 2.4.2, $X\ R\ X' \Rightarrow \exists X'' \in \mathbf{X}_1.(X'' \neq X \wedge X\ R_1\ X'' \wedge X''\ R\ X')$ since X is not in the support set of R_2. Therefore, $\exists X' \in (\mathbf{X}_1 \cup \mathbf{X}_2) \setminus \{X\}.X\ R\ X' \Rightarrow \exists X'' \in$

$\mathbf{X}_1 \setminus \{X\}.X\ R_1\ X''$. Since the converse is trivial, $\exists X' \in (\mathbf{X}_1 \cup \mathbf{X}_2) \setminus \{X\}.X\ R\ X' = \exists X'' \in \mathbf{X}_1 \setminus \{X\}.X\ R_1\ X''$, which makes the side conditions for L in the link name hidings equivalent. □

$map^V_{\mathscr{G}}$ and $\setminus X$ is commutative without imposing any restrictions.

Theorem 2.4.9. *$map^V_{\mathscr{G}}\ f\ (\mathscr{G} \setminus X) = (map^V_{\mathscr{G}}\ f\ \mathscr{G}) \setminus X$ for any $\mathscr{G}$, and X.*

Proof. Trivial if $X \notin \mathbf{X}$. Case $X \in \mathbf{X}$. For V and $\mathscr{L}$, since $\setminus X$ does not change the values, the theorem holds. For $\mathbf{X}$ and R, since $map^V_{\mathscr{G}}\ f$ does not change the values, the theorem holds. For π', since $(((map_{\mathtt{S}} \circ app_1)\ f) \circ (\pi|'_{\mathbf{X}})) = (((map_{\mathtt{S}} \circ app_1)\ f) \circ \pi)|'_{\mathbf{X}}$, the theorem holds. For L, the theorem holds trivially if the value of L does not change in link name hiding. Otherwise, the theorem holds since by Theorem 2.3.1, $(map_{\mathtt{S}} \circ map_{\mathtt{S}} \circ app_1)\ f\ (L \cup \{\pi(X)\}) = ((map_{\mathtt{S}} \circ map_{\mathtt{S}} \circ app_1)\ f\ L) \cup \{(((map_{\mathtt{S}} \circ app_1)\ f) \circ \pi)(X)\}$. □

Theorem 2.4.10. *$\mathscr{G}^0 \setminus X = \mathscr{G}^0$ for any X.*

Proof. Trivial from the definition. □

Lemma 2.4.3. *$fn_{\mathscr{G}}(\mathscr{G} \setminus X) = fn_{\mathscr{G}}(\mathscr{G}) \setminus \{X\}$ for any X and $\mathscr{G}$.*

Proof. Trivial from the definition. □

Theorem 2.4.11. *$(\mathscr{G} \setminus X) \setminus Y = (\mathscr{G} \setminus Y) \setminus X$*

Proof. Trivial if $X = Y$. Otherwise, *case* $X \notin fn_{\mathscr{G}}(\mathscr{G})$: by Lemma 2.4.3 and the premise, $X \notin fn_{\mathscr{G}}(\mathscr{G} \setminus Y)$. Therefore, $(\mathscr{G} \setminus X) \setminus Y = \mathscr{G} \setminus Y = (\mathscr{G} \setminus Y) \setminus X$. *Case* $Y \notin fn_{\mathscr{G}}(\mathscr{G})$: same as the case above. *Case* $X, Y \in fn_{\mathscr{G}}(\mathscr{G})$: let $\langle V, \langle \mathbf{X}', R', \pi' \rangle, L', \mathscr{L} \rangle = \langle V, \langle \mathbf{X}, R, \pi \rangle, L, \mathscr{L} \rangle \setminus X$, then $\langle V, \langle \mathbf{X}', R', \pi' \rangle, L', \mathscr{L} \rangle \setminus Y = \langle V, \langle \mathbf{X}'', R'', \pi'' \rangle, L'', \mathscr{L} \rangle$ where

1. $\mathbf{X}'' \stackrel{\text{def}}{=} \mathbf{X}' \setminus \{Y\} = (\mathbf{X} \setminus \{X\}) \setminus \{Y\} = \mathbf{X} \setminus \{X, Y\}$,
2. $R'' \stackrel{\text{def}}{=} \{\langle Z, Z' \rangle \mid Z, Z' \in \mathbf{X}''.Z\ R'\ Z'\}$
   $= \{\langle Z, Z' \rangle \mid Z, Z' \in \mathbf{X}''.(Z, Z' \in \mathbf{X}' \wedge Z\ R\ Z')\}$ $\because$ Definition of R'
   $= \{\langle Z, Z' \rangle \mid Z, Z' \in \mathbf{X}''.Z\ R\ Z'\}$ $\because Z, Z' \in \mathbf{X}'' \Rightarrow Z, Z' \in \mathbf{X}'$,
3. $\pi'' \stackrel{\text{def}}{=} \pi'|_{\mathbf{X}''} = (\pi|_{\mathbf{X}'})|_{\mathbf{X}''} = \pi|_{\mathbf{X}''}$ $(\because \mathbf{X}'' \subset \mathbf{X}')$, and
4. $L'' \stackrel{\text{def}}{=} \begin{cases} L' & \exists Y' \in \mathbf{X}''.Y\ R'\ Y \\ L' \cup \{\pi(Y)\} & \text{otherwise.} \end{cases}$
   $= \begin{cases} L' & P_X \wedge P_Y \\ L' \cup \{\pi(X)\} & \neg P_X \wedge P_Y \\ L' \cup \{\pi(Y)\} & P_X \wedge \neg P_Y \\ L' \cup \{\pi(X)\} \cup \{\pi(Y)\} & \neg P_X \wedge \neg P_Y \end{cases}$

   where $P_X \stackrel{\text{def}}{=} \exists X' \in \mathbf{X}'.X\ R\ X'$ and $P_Y \stackrel{\text{def}}{=} \exists Y' \in \mathbf{X}''.Y\ R'\ Y'$.

Since $X \neq Y$, $X \in \mathbf{X}' \Leftrightarrow X \in \mathbf{X}''$. Therefore, $P_X \Leftrightarrow \exists X' \in \mathbf{X}''.X\ R\ X'$.

By definition of R', $P_Y \Leftrightarrow \exists Y' \in \mathbf{X}''.(Y, Y' \in \mathbf{X}'.Y\ R\ Y')$. Since $Y \in \mathbf{X}$ and $X \neq Y$, $Y \in \mathbf{X}'$ always holds. Also, $Y' \in \mathbf{X}'' \Rightarrow Y' \in \mathbf{X}'$. Thus, $P_Y \Leftrightarrow \exists Y' \in \mathbf{X}''.Y\ R\ Y'$.

Since the definitions of $\mathbf{X}'', R'', \pi'', L''$ are symmetric with respect to X and Y, the theorem holds. □

Since we need to α-convert link names later, we need some theorems to handle these.

Theorem 2.4.12 (Alpha-Conversion)**.** $\left(map^X_{\mathscr{G}}\ [X \mapsto Y]\ \mathscr{G}\right) \setminus Y = \mathscr{G} \setminus X$ *if* $Y \notin fn_{\mathscr{G}}(\mathscr{G})$ *for any* X, Y*, and* $\mathscr{G}$*.*

Proof. Since $Y \notin fn_{\mathscr{G}}(\mathscr{G})$, $[X \mapsto Y]$ can be replaced with a bijection of which domain is $fn_{\mathscr{G}}(\mathscr{G})$. Thus, $map^X_{\mathscr{G}}$ can be much simpler as we have explained earlier and the result follows straightforwardly. □

Theorem 2.4.13. $\left(map^X_{\mathscr{G}}\ [X \mapsto Y]\ \left(map^X_{\mathscr{G}}\ [Y \mapsto Z]\ \mathscr{G}\right)\right) \setminus Z = (\mathscr{G} \setminus X)$ *if* $X \neq Y \wedge Z \neq Y \wedge Z \notin fn_{\mathscr{G}}(\mathscr{G})$*. for any* X, Y, Z*, and* $\mathscr{G}$*.*

Proof. We firstly prove $\left(map^X_{\mathscr{G}}\ [X \mapsto Y]\ \left(map^X_{\mathscr{G}}\ [Y \mapsto Z]\ \mathscr{G}\right)\right) \setminus Z = map^X_{\mathscr{G}}\ [X \mapsto Y]\ (\mathscr{G} \setminus Y)$ if $X \neq Y \wedge Z \neq Y \wedge Z \notin fn_{\mathscr{G}}(G)$. and then use Theorem 2.4.12.

For the former, since $Z \notin fn_{\mathscr{G}}(\mathscr{G})$, $[Y \mapsto Z]$ can be replaced with a bijection as in Theorem 2.4.12. Thus, $map^X_{\mathscr{G}}$ can be much simpler as we have explained earlier and the results follows straightforwardly. Since we have replaced Y with Z, $Y \notin fn_{\mathscr{G}}(map^X_{\mathscr{G}}\ [Y \mapsto Z]\ \mathscr{G})$, thus we can use Theorem 2.4.12 later. □

Since $Z \neq X \wedge Z \neq Y$, $\setminus Z$ and $map^X_{\mathscr{G}}\ [X \mapsto Y]$ commutes as follows:

Theorem 2.4.14. $(map^X_{\mathscr{G}}\ [X \mapsto Y]\ \mathscr{G}) \setminus Z = map^X_{\mathscr{G}}\ [X \mapsto Y]\ (\mathscr{G} \setminus Z)$*. if* $Z \neq X \wedge Z \neq Y$ *for any* X, Y, Z*, and* $\mathscr{G}$*.*

Proof. The theorem holds straightforwardly from the definitions of the operations. □

HyperLMNtal terms includes a fusion, which corresponds to the following graph:

Definition 2.4.5 (Fusion in Graph-Theoretic Graphs)**.**

$\underline{X \bowtie Y} \stackrel{\text{def}}{=} \{\langle \varnothing, \langle \{X, Y\}, \{\{X, Y\}\}, \{X \mapsto \varnothing, Y \mapsto \varnothing\}\rangle, \varnothing, \varnothing\rangle\}$

We prove that composing a fusion is equivalent to connect free link X to Y by renaming X to Y.

Theorem 2.4.15. $(\underline{X \bowtie Y} + \mathscr{G}) \setminus X = \left(map^X_{\mathscr{G}}\ [X \mapsto Y]\ \mathscr{G}\right) \setminus X$ *where* $X \in fn_{\mathscr{G}}(\mathscr{G}) \vee Y \in fn_{\mathscr{G}}(\mathscr{G})$ *for any* X, Y*, and* $\mathscr{G}$*.*

Notice that $\underline{X \bowtie Y} + \mathscr{G}$ can be done always without imposing a restriction on vertices since $\underline{X \bowtie Y}$ does not have vertices.

Proof. The theorem holds straightforwardly from the definitions. □

2.4 Term to Graphs Translator

We introduce a function to transform a term to graphs. Intuitively, this function returns a set of all the isomorphic graphs that corresponds to the given term.

Definition 2.4.6 (Term to Graphs)**.**

$$
\begin{array}{lcl}
t2gs(p(\overrightarrow{X_i}^i)) & \stackrel{\text{def}}{=} & \{\langle\{v\}, \sum_i \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v, i\rangle\}\}\rangle\}, \varnothing, \{v \mapsto p\}\rangle \mid v \in \mathbb{V}\} \\
t2gs((G, G')) & \stackrel{\text{def}}{=} & t2gs(G) \otimes t2gs(G') \\
t2gs(\nu X.G) & \stackrel{\text{def}}{=} & \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(G)\} \\
t2gs(\mathbf{0}) & \stackrel{\text{def}}{=} & \{\mathscr{G}^0\} \\
t2gs(X \bowtie Y) & \stackrel{\text{def}}{=} & \{\underline{X \bowtie Y}\}
\end{array}
$$

Theorem 2.4.16 (Link Substitution)**.** *$t2gs(G\langle Y/X\rangle) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G)$ for any X, Y, and G.*

Proof. We prove this by induction on the structure of G.

Case $G = \mathbf{0}$.
$t2gs(\mathbf{0}\langle Y/X\rangle) = t2gs(\mathbf{0}) = \{\mathscr{G}^0\} = map^X_{\mathbf{G}}\ [X \mapsto Y]\ \{\mathscr{G}^0\} = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(\mathbf{0})$.

Case $G = p(\overrightarrow{X_i}^i)$.
$t2gs(p(\overrightarrow{X_i}^i)\langle Y/X\rangle) = t2gs(p(\overrightarrow{X_i\langle Y/X\rangle}^i)) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(p(\overrightarrow{X_i}^i))$.

Case $G = X \bowtie Y$.
Follows straightforwardly from the definition.

Case $G = (G_1, G_2)$.
$t2gs((G_1\langle Y/X\rangle, G_2\langle Y/X\rangle)) = t2gs(G_1\langle Y/X\rangle) \otimes t2gs(G_2\langle Y/X\rangle) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G_1) \otimes map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G_2) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ (t2gs(G_1) \otimes t2gs(G_2)) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G)$ where $t2gs(G_1\langle Y/X\rangle) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G_1)$ and $t2gs(G_2\langle Y/X\rangle) = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G_2)$ by induction hypothesis.

Case $G = \nu Z.G'$.

We split the cases according to the cases in the hyperlink substitution Definition 2.2.4.

Case $Z = X$. Since $X \notin fn_{\mathscr{G}}(\mathscr{G} \setminus Z)$ for any $\mathscr{G}$, $map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(\nu Z.G') = t2gs(\nu Z.G')$. Therefore, the theorem holds.

Case $Z \neq X \wedge Z \neq Y$. By Theorem 2.4.14 and induction hypothesis, $t2gs(G\langle Y/X\rangle) = t2gs(\nu Z.G'\langle Y/X\rangle) = \{\mathscr{G}' \setminus Z \mid \mathscr{G}' \in t2gs(G'\langle Y/X\rangle)\} = \{\mathscr{G}' \setminus Z \mid \mathscr{G}' \in map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G')\} = map^X_{\mathbf{G}}\ [X \mapsto Y]\ \{\mathscr{G}' \setminus Z \mid \mathscr{G}' \in t2gs(G')\} = map^X_{\mathbf{G}}\ [X \mapsto Y]\ t2gs(G)$.

Case $Z \neq X \wedge Z = Y$.

$$
\begin{aligned}
&\mathit{t2gs}((\nu Z.G')\langle Y/X\rangle)\\
&= \mathit{t2gs}(\nu W.(G'\langle W/Z\rangle)\langle Y/X\rangle) \quad \text{case } Z \neq X \wedge Z = Y \wedge W \notin \mathit{fn}_{\mathscr{G}}(G') \wedge W \neq Y\\
&= \left\{\mathscr{G}' \setminus W \;\middle|\; \mathscr{G}' \in \mathit{map}^X_{\mathbf{G}}\ [X \mapsto Y]\ (\mathit{map}^X_{\mathbf{G}}\ [Z \mapsto W]\ \mathit{t2gs}(G'))\right\} \quad \because \text{Induction Hypothesis}\\
&= \mathit{map}^X_{\mathbf{G}}\ [X \mapsto Y]\ \{\mathscr{G}' \setminus Z \mid \mathscr{G}' \in \mathit{t2gs}(G')\} \quad \because \text{Theorem 2.4.13}\\
&= \mathit{map}^X_{\mathbf{G}}\ [X \mapsto Y]\ \mathit{t2gs}(\nu Z.G').
\end{aligned}
$$

□

2.5 Soundness of the Translation From Terms to Graphs

In this section, we prove that the structurally congruent terms are mapped to isomorphic graphs using the function *t2gs*; i.e., the *soundness* of the function.

We firstly prove that, for an term G, all the graphs in the set which is returned by the function *t2gs* are graph-isomorphic.

Theorem 2.5.1 (Well-definedness of the *t2gs* function)**.** $\forall \mathscr{G}, \mathscr{G}' \in \mathit{t2gs}(G).\mathscr{G} \cong \mathscr{G}'$.

Proof. We use induction on the structure of G.

Case $G = \mathbf{0}$ or $G = X \bowtie Y$.

Since $\mathit{t2gs}(G)$ is a singleton set, the theorem holds trivially.

Case $G = p(\overrightarrow{X})$.

Let $\mathscr{G} = \langle\{v\}, F, \varnothing, \{v \mapsto p\}\rangle$ where $F = \sum_i \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v, i\rangle\}\}\rangle\}$ and $\mathscr{G} = \langle\{v'\}, F', \varnothing, \{v' \mapsto p\}\rangle$ where $F' = \sum_i \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v', i\rangle\}\}\rangle\}$. Let $f_V \stackrel{\text{def}}{=} \{v \mapsto v'\}$, be a bijection.

The theorem holds straightforwardly for those other than F. For $\mathit{map}^V_F\ f\ F = F'$, the theorem holds as follows:

$$
\begin{aligned}
&\mathit{map}^V_F\ f\ \textstyle\sum_i \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v, i\rangle\}\}\rangle\}\\
&= \textstyle\sum_i \mathit{map}^V_F\ f\ \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v, i\rangle\}\}\rangle\} \quad \because \text{Theorem 2.4.3}\\
&= \textstyle\sum_i \{\langle\{X_i\}, \{\{X_i\}\}, \{X_i \mapsto \{\langle v', i\rangle\}\}\rangle\}
\end{aligned}
$$

Case $G = (G_1, G_2)$.

By definition of $(\mathscr{P}(\mathbb{G}), \otimes, \{\mathscr{G}^0\})$,

1. $\mathscr{G} \in \mathit{t2gs}(G_1) \otimes \mathit{t2gs}(G_2)$ satisfies $\mathscr{G} = \mathscr{G}_1 + \mathscr{G}_2$ where $\mathscr{G}_1 \in \mathit{t2gs}(G_1) \wedge \mathscr{G}_2 \in \mathit{t2gs}(G_2) \wedge \mathit{fst}(\mathscr{G}_1) \cap \mathit{fst}(\mathscr{G}_2) = \varnothing$, and
2. $\mathscr{G}' \in \mathit{t2gs}(G_1) \otimes \mathit{t2gs}(G_2)$ satisfies $\mathscr{G}' = \mathscr{G}'_1 + \mathscr{G}'_2$ where $\mathscr{G}'_1 \in \mathit{t2gs}(G_1) \wedge \mathscr{G}'_2 \in \mathit{t2gs}(G_2) \wedge \mathit{fst}(\mathscr{G}'_1) \cap \mathit{fst}(\mathscr{G}'_2) = \varnothing$.

By induction hypothesis, there exist bijections f_{V1}, f_{V2} such that $\mathscr{G}'_1 = \mathit{map}^V_{\mathscr{G}}\ f_{V1}\ \mathscr{G}_1$ and $\mathscr{G}'_2 = \mathit{map}^V_{\mathscr{G}}\ f_{V2}\ \mathscr{G}_2$.

Since $fst(\mathscr{G}_1) \cap fst(\mathscr{G}_2) = \varnothing$, $\mathrm{dom}(f_{V1}) \cap \mathrm{dom}(f_{V2}) = \varnothing$, and since $fst(\mathscr{G}'_1) \cap fst(\mathscr{G}'_2) = \varnothing$, $\mathrm{img}(f_{V1}) \cap \mathrm{img}(f_{V2}) = \varnothing$.

Since an union of bijections whose domains and images are disjoint is a bijection, $f_V \stackrel{\text{def}}{=} f_{V1} \sqcup f_{V2}$ is a bijection.

By Theorem 2.4.5, $map^V_{\mathscr{G}}\ f_V\ \mathscr{G} = map^V_{\mathscr{G}}\ f_V\ (\mathscr{G}_1 + \mathscr{G}_2) = (map^V_{\mathscr{G}}\ f_V\ \mathscr{G}_1) + (map^V_{\mathscr{G}}\ f_V\ \mathscr{G}_2) = (map^V_{\mathscr{G}}\ f_{V1}\ \mathscr{G}_1) + (map^V_{\mathscr{G}}\ f_{V2}\ \mathscr{G}_2) = \mathscr{G}'_1 + \mathscr{G}'_2 = \mathscr{G}'$.

Therefore, there exists a bijection, f_V, such that $map^V_{\mathscr{G}}\ f_V\ \mathscr{G} = \mathscr{G}'$.

Case $G = \nu X.G'$**.**

By definition of $t2gs$, $\mathscr{G}_1, \mathscr{G}_2 \in t2gs(G)$ satisfies $\mathscr{G}_1 = \mathscr{G}'_1 \setminus X$ and $\mathscr{G}_2 = \mathscr{G}'_2 \setminus X$ where $\mathscr{G}'_1, \mathscr{G}'_2 \in t2gs(G')$.

Here, by induction hypothesis, For all $\mathscr{G}'_1, \mathscr{G}'_2 \in t2gs(G')$, there exists a bijection f_V such that $\mathscr{G}'_2 = map^V_{\mathscr{G}}\ f_V\ \mathscr{G}'_1$.

Therefore, by Theorem 2.4.9, $\mathscr{G}_2 = \mathscr{G}'_2 \setminus X = (map^V_{\mathscr{G}}\ f\ \mathscr{G}'_1) \setminus X = map^V_{\mathscr{G}}\ f\ (\mathscr{G}'_1 \setminus X) = map^V_{\mathscr{G}}\ f\ \mathscr{G}_1$.

Thus, there exists a bijection, f_V, such that $map^V_{\mathscr{G}}\ f_V\ \mathscr{G}_1 = \mathscr{G}_2$.

□

We then prove that $t2gs$ returns the same set of graphs if it takes structurally congruent terms.

Theorem 2.5.2 (Structural Congruence ⇒ Graph Isomorphism)**.** $G \equiv G' \Rightarrow t2gs(G) = t2gs(G')$

Proof. We use induction on the structure of terms and split the cases by the rule in Definition 2.2.5.

Case (E1). $t2gs((\mathbf{0}, G)) = t2gs(\mathbf{0}) \otimes t2gs(G) = \{\mathscr{G}^0\} \otimes t2gs(G) = t2gs(G)$ by Theorem 2.4.6.

Case (E2). $t2gs((G_1, G_2)) = t2gs(G_1) \otimes t2gs(G_2) = t2gs(G_2) \otimes t2gs(G_1) = t2gs((G_2, G_1))$. by Theorem 2.4.6.

Case (E3). $t2gs((G_1, (G_2, G_3))) = t2gs(G_1) \otimes t2gs((G_2, G_3)) = t2gs(G_1) \otimes (t2gs(G_2) \otimes t2gs(G_3)) = (t2gs(G_1) \otimes t2gs(G_2)) \otimes t2gs(G_3) = t2gs((G_1, G_2)) \otimes t2gs(G_3) = t2gs(((G_1, G_2), G_3))$. by Theorem 2.4.6.

Case (E4). By induction hypothesis,begin $t2gs(G_1) = t2gs(G_2)$. Therefore, $t2gs((G_1, G_3)) = t2gs(G_1) \otimes t2gs(G_3) = t2gs(G_2) \otimes t2gs(G_3) = t2gs((G_2, G_3))$.

Case (E5). By induction hypothesis, $t2gs(G_1) = t2gs(G_2)$. Therefore, $t2gs(\nu X.G_1) = \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(G_1)\} = \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(G_2)\} = t2gs(\nu X.G_1)$.

Case (E6). $t2gs(\nu X.(X \bowtie Y, G)) = \{(\underline{X \bowtie Y} + \mathscr{G}) \setminus X \mid \mathscr{G} \in t2gs(G)\} = \left\{\left(map^X_{\mathscr{G}}\ \mathrm{id}_{X \mapsto Y}\ \mathscr{G}\right) \setminus X \;\middle|\; \mathscr{G} \in t2gs(G)\right\} = t2gs(\nu X.G\langle Y/X\rangle)$ by Theorem 2.4.16 and Theorem 2.4.15
where $X \in fn_{\mathscr{G}}(\mathscr{G}) \vee Y \in fn_{\mathscr{G}}(\mathscr{G})$ by the side condition of the rule, $X \in fn(G) \vee Y \in fn(G)$.

Case (E7). $\nu X.\nu Y.X \bowtie Y \equiv \mathbf{0}$ $t2gs(\nu X.\nu Y.X \bowtie Y) = \mathscr{G}^0 = t2gs(\mathbf{0})$.

Case (E8). $t2gs(\nu X.\mathbf{0}) = \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(\mathbf{0})\} = \{\mathscr{G} \setminus X \mid \mathscr{G} \in \{\mathscr{G}^0\}\} = \{\mathscr{G}^0 \setminus X\} = \{\mathscr{G}^0\} = t2gs(\mathbf{0})$ by Theorem 2.4.10.

Case (E9). By Theorem 2.4.11, $t2gs(\nu X.\nu Y.G) = \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(\nu Y.G)\} = \{\mathscr{G} \setminus X \mid \mathscr{G} \in \{\mathscr{G} \setminus Y \mid \mathscr{G} \in t2gs(G)\}\} = \{(\mathscr{G} \setminus Y) \setminus X \mid \mathscr{G} \in t2gs(G)\} = \{(\mathscr{G} \setminus X) \setminus Y \mid \mathscr{G} \in t2gs(G)\} = \{\mathscr{G} \setminus Y \mid \mathscr{G} \in \{\mathscr{G} \setminus X \mid \mathscr{G} \in t2gs(G)\}\} = \{\mathscr{G} \setminus Y \mid \mathscr{G} \in t2gs(\nu X.G)\} = t2gs(\nu Y.\nu X.G)$.

Case (E10). $\nu X.(G_1, G_2) \equiv (\nu X.G_1, G_2)$ where $X \notin fn_{\mathscr{G}}(G_2)$
By Theorem 2.4.8, and the side condition of the rule, $X \notin fn_{\mathscr{G}}(G_2)$,
$t2gs(\nu X.(G_1, G_2)) = \{(\mathscr{G}_1 + \mathscr{G}_2) \setminus X \mid \mathscr{G}_1 \in t2gs(G_1), \mathscr{G}_2 \in t2gs(G_2), fst(\mathscr{G}_1) \cap fst(\mathscr{G}_2) = \varnothing\}$
$= \{(\mathscr{G}_1 \setminus X) + \mathscr{G}_2 \mid \mathscr{G}_1 \in t2gs(G_1), \mathscr{G}_2 \in t2gs(G_2), fst(\mathscr{G}_1) \cap fst(\mathscr{G}_2) = \varnothing\} = t2gs((\nu X.G_1, G_2))$.

Since $\equiv$ is defined as a reflective transitive closure of the relation in Definition 2.2.5, the theorem holds by using the rules repeatedly.

□

2.6 Related Work

Since graphs and their operations are more complex than trees, there are diverse formalisms for graphs and the equivalence definitions.

It is known that there exists a simple term language, 2pdom-algebra [LP17], that can denote labelled graphs whose treewidth is up to 2. The terms written in the language can be translated to graphs using the function g. The language has a finite set of axioms that defines the congruence over terms. In [LP17], the authors have proved that all the congruent terms are translated to graph-isomorphic graphs and the terms that are mapped to graphs-isomorphic graphs are congruent; i.e., the translation satisfies both soundness and completeness. Recently, the language has formally proved to be complete using Coq [DP20].

On the other hand, HyperLMNtal deals with a superclass of the graphs denoted by 2pdom-algebra. Since HyperLMNtal deals with hypergraphs, does not impose the treewidth limitation, has a notion for ports, which is important for the application in programming and verification, and has a notion of fusions and free links, which plays a big role for subgraph matchings.

UnCAL [BFS00] is a calculus of a query language for graph-based databases. The language features recursive operation over graphs. The equivalence of the graphs is defined with a bisimulation.

In [HMA18], the authors have axiomatized the equivalence relations of UnCAL graphs. Our research investigates the axiomatization of the equivalence based on graph-isomorphism, which is different from those based on bisimulation. For example, there exist cyclic graphs which are distinguished in our axioms but are handled as the same graphs in UnCAL.

In [Bas94], the authors describe that the equivalence problem of the terms containing otherwise uninterpreted associative, commutative, and associative-commutative function symbols and commutative variable-binding operators is polynomially equivalent to the graph isomorphism problem. They have described proof for the soundness and the completeness of the translation, but the discussion focuses on the computational complexity and lacks a precise definition of the translator function from terms to graphs.

Also, the terms in [Bas94] do not have the notion of fusion in HyperLMNtal. Since HyperLMNtal and λ_{GT} utilizes fusion for the matching, this is important.

In [KM09], the authors discuss that the structural congruence in process algebra is equivalent to graph isomorphism. Since our calculus is based on process algebra, this can be an important related work. This paper also focuses on computational complexity and utilizes the theorem in [Bas94], but they do not prove that the structural congruent terms are graph-isomorphic in a rigorous manner.

Separation Logic is a verification framework for imperative languages using heaps and pointers. Since heaps and pointers can be generalized with hypergraphs, the logic is closely related to HyperLMNtal. In particular, Symbolic Heap, the subset of separation logic, also introduces the notion of graphs and associates the logic with it. Separation Logic [Rey02] does not axiomatize the equivalence of heaps but defines it with an isomorphism with a bijection of addresses. However, axiomatizing isomorphism of hypergraphs facilitates inductive arguments and is advantageous for verification.

2.7 Further Work

We have proved the soundness but we leave the proof of completeness theorem, i.e., the terms mapped to graph isomorphic graphs by the function are structurally congruent, for future work. We are planning to prove this in the following steps.

We first define a function to transform hypergraphs to HyperLMNtal terms, *g2ts*, and prove that the graph-isomorphic graphs are mapped to structurally congruent terms by the function *g2ts*. Since the structural congruence rules enable the reordering of atoms (vertices) and α-conversion of local link names, we believe that this can be proved without difficulty. Then, we are to prove that the term transformed from a term using *t2gs* and *g2ts* are structurally congruent; $\forall G.(\forall \mathscr{G} \in g2ts(G).(\forall G' \in t2gs(\mathscr{G}).(G \equiv G')))$. This may be difficult depending on the definition of *g2ts*. If we can prove the former 2 theorems,

we can prove the *completeness* theorem by combining the theorems.

Formal proof using proof assistants such as Coq as in 2pdom-albegra[DP20] is left as future work. We believe there is no big hurdle since the definitions of HyperLMNtal and graphs and their operations are given formally and are not that complicated though it may require a huge amount of effort.

Our ultimate goal is to clarify the relation between existing graph transformation formalisms and our calculus. In this paper, we only discuss the relationship between structural congruence and graph isomorphisms. HyperLMNtal is capable of matching and rewriting graphs using structural congruence. We believe that it is necessary to investigate and prove how this relates to the subgraph isomorphism and pushout in existing graph transformations. In addition, we are considering utilizing the results of this study as a basis for the implementation of λ_{GT} and a more powerful verification on F_{GT}.

3

λ_{GT}: A Functional Language with Hypergraphs as First-Class Data

3.1 Introduction

λ_{GT} is a functional language to support graphs as first-class data. λ_{GT} enables us to construct and pattern match graphs as well as Algebraic Data Types. The key features of λ_{GT} are the following.

Data structures more complex than trees. Algebraic Data Types (ADT) in purely functional languages can only represent tree structures. On the other hand, in λ_{GT}, not just lists and trees but also difference lists, skip lists [Pug90], doubly linked lists, leaf linked trees, and threaded trees, etc, can be handled succinctly.

Graphs as first-class data. Not pointers or references to a global heap but graphs are first-class, i.e., values, in this language. That is, graphs can be dynamically created, graphs can be bound to variables, be input and output of functions, and be dynamically discarded.

Powerful pattern matching mechanism. When matching ADTs in functional languages, we can only use the patterns that allow only the matching of a fixed region near the root of the structure. On the other hand, we enabled more powerful matching based on Graph Transformation [Ehr+06; Roz97].

Syntax-driven semantics. To establish the semantics of λ_{GT}, we incorporated that of HyperLMNtal [UO12; SU21] into call-by-value λ calculus. Unlike definitions common in conventional graph transformations, such as the triplet of a set of vertices, a set of sets of vertices (hyperlinks), and a labelling function, the graphs in HyperLMNtal can be constructed compositionally from subgraphs. These syntax and semantics follow the style of π-calculus [SW01] rather than traditional algebraic graph transformation formalism. This makes it easier to incorporate graphs into λ-calculus.

Type System. The type system of λ_{GT} incorporates graph grammar and use infinite decent. We will introduce this later in Chapter 4.

First-class functions. Functions are first-class data in λ_{GT} as well as in other functional languages.

Immutability. Graphs are immutable in this language. We do not rely on destructive rewriting but employ immutable composing and decomposing with pattern matchings.

Contributions

In this chapter, we propose a purely functional language, λ_{GT}, that handles data structures beyond algebraic data types; hypergraphs. Since hypergraphs can be more complex than trees, it requires non-trivial formalism. In this chapter, we define the formal syntax and semantics of λ_{GT} and examine their validity with some running examples.

Since its formalism is more complicated than ordinary functional languages, its implementation should be also more complicated. λ_{GT} is even more advanced than ordinary graph transformation systems. Graph transformation systems rewrite one global graph with rewriting rules. On the other hand, in λ_{GT}, graphs are immutable local values that can be bound to variables, decomposed by pattern matchings with possibly multiple wildcards, in which the matched subgraphs may be used separately, passed as inputs of functions, and composed to construct larger graphs. Therefore, it is not trivial to implement the language.

Thus, we implemented a *reference interpreter*, a reference implementation of the language. We believe this is usable for further investigation, including in the design of *real languages* based on λ_{GT}. The interpreter is written in only 500 lines of OCaml [Ler+22] code, which is strikingly concise[1].

Chapter Map

The rest of this chapter is organized as follows. Section 3.2 introduces λ_{GT} informally. Section 3.3 gives the formal syntax and the operational semantics of λ_{GT}. In Section 3.4, we give a more detailed explanation of the examples we have introduced informally in Section 3.2. Section 3.5 describes our reference interpreter. Since our language has a new type system we will introduce in the next chapter, which is also a key element of the novelties of our language, we do not discuss related work in this chapter but in the next chapter.

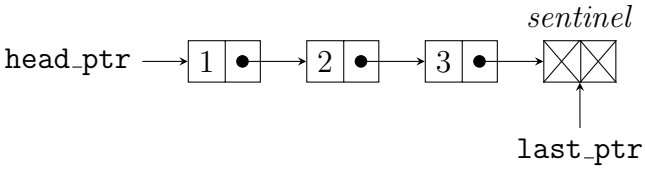
3.2 Informal Introduction to λ_{GT}

In this section, we introduce λ_{GT} *informally*. We will give the formal syntax and semantics of the language later in Section 3.3.

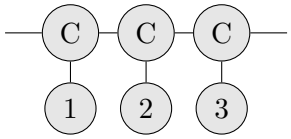
Consider the case where we want to enable adding an element to the end of a list efficiently. In imperative languages, we will prepare a pointer that points to the address of the last node (sentinel node) of the list. Adding a new element to the list can be done with the destructive assignment to the sentinel node with a new number and the address to the newly created sentinel node. We also need to update `last_ptr` to point to the new

[1]The source code is available at https://github.com/sano-jin/lambda-gt-alpha. We have also implemented a visualizing tool that runs on a browser, which is available at https://sano-jin.github.io/lambda-gt-online/.

sentinel node. These low-level operations are tiresome and prone to errors, e.g. we can easily forget to update `last_ptr`.

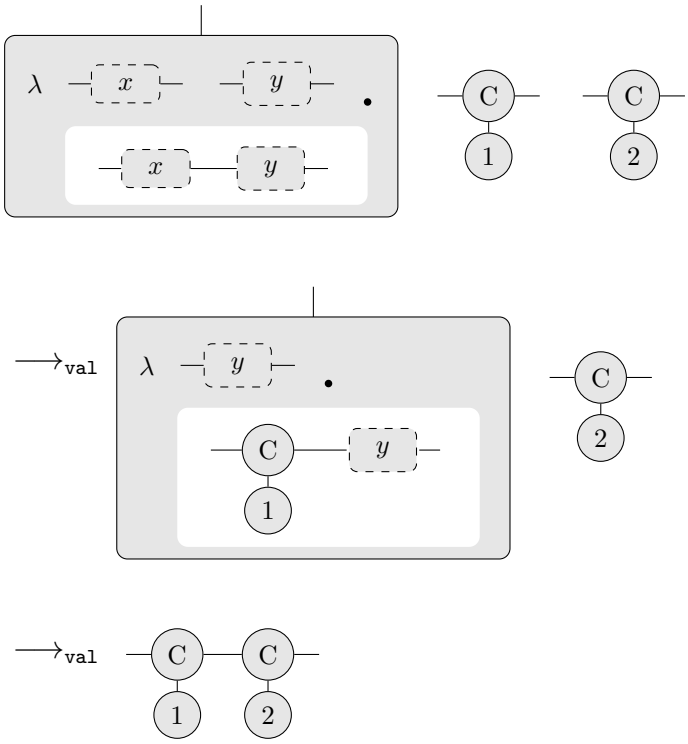


In λ_{GT}, such data structure can be abstracted to a *difference list*; a list with a link to the last node, as follows. Adding a new element to the list can be understood as concatenating a singleton list to the list. Note that the free links and the indices of ports are omitted in the graphical representation for the sake of simplicity.



(1)

We can represent a program with a function that takes two difference lists and returns the concatenated difference list as the following. Notice that the input and output of the function are not pointers to a global heap but graphs as local *values*. λ_{GT} can handle graphs as first-class data (i.e., values) in such a manner.



Although we will not discuss in detail in this chapter (See Chapter 4), we have also proposed a new type system for the language, which is extended from the typing framework

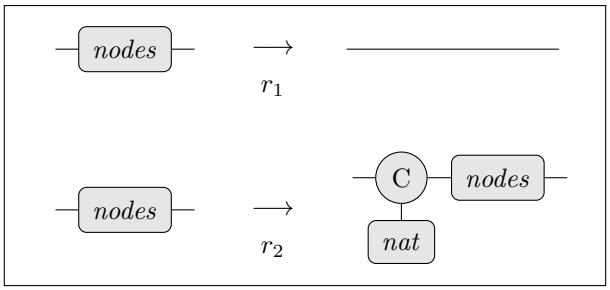
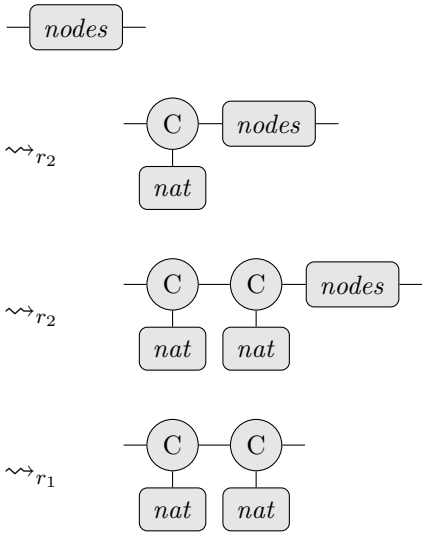


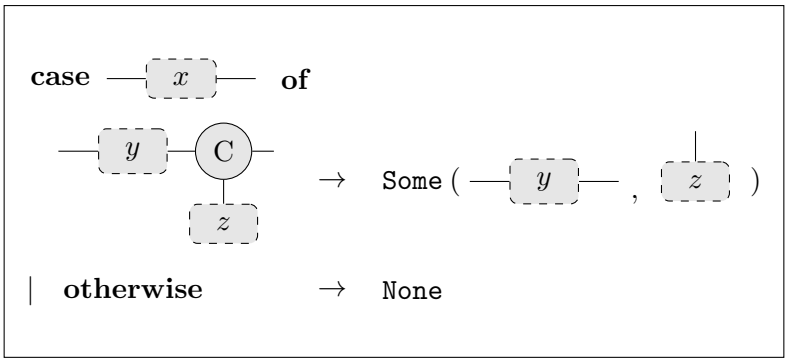
Figure 3.1: Production rules for a difference list.

for graph transformation based languages [FM97; FM98; YU21]. In this type system, the types of graphs are defined using graph grammar. For example, the type of a difference list can be defined using the production rules in Figure 3.1.

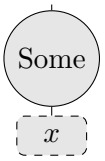
Informally, we can say graph has a type if we can obtain the graph from the type applying production rules zero or more times. The following example shows that we can obtain a difference list with two elements using the production rules.



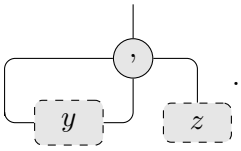
λ_{GT} can not only handle graphs as input/output of functions, but also able to pattern match graphs. This is more powerful than those of traditional functional languages with Algebraic Data Types. With Algebraic Data Types, normally, only the root of a tree can be matched. Taking the last element of a list needs iterating from the head of the list. On the other hand, in λ_{GT}, we can pop the last element in one step.

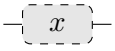


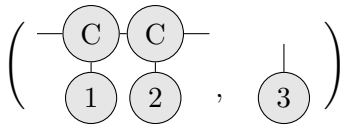
Here, we used an `option` type (`Some` and `None`) and a tuple to return both the popped list and the element. λ_{GT} we will describe in the next section does not have `option` type and tuple explicitly. However, they can be easily introduced directly to the language or encoded without extension.

For example, `Some` x can be encoded as 

and 

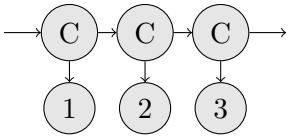
as 

. If we have bound (1) to 

, then the obtained result will be 

.

The values in λ_{GT} are undirected graphs. However, it will be suitable for the links in the graphs to be directed when compiling to an impure functional language program using reference types. In λ_{GT}, we can easily encode directed edges. The links in the difference lists we have introduced can be regarded as directed edges from left to right or from up to down.



$(1-b)$

However, if the list consists only of forward pointers, it will be difficult to match backward efficiently such as matching to the last node of the list. Therefore, we consider making it a doubly linked list as the following. We can easily rewrite the program to handle this.

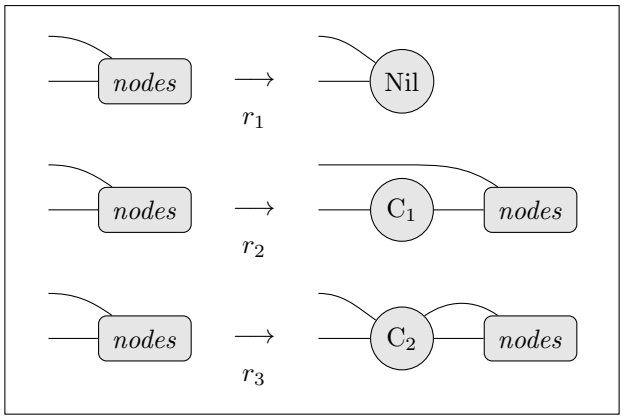
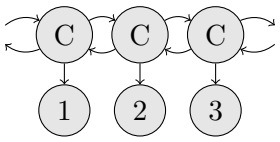


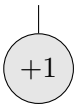
Figure 3.2: Production rules for 2-level skip list.

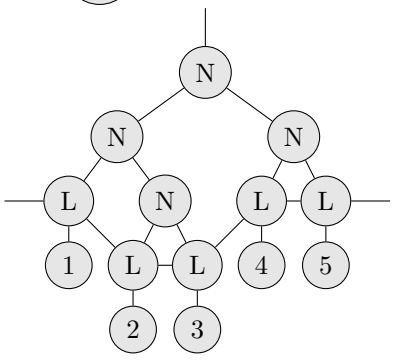


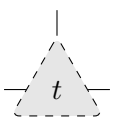
λ_{GT} can handle not only difference lists but also various data structures. Skip list is a list with extra edges, as shown in Figure 3.4. The extra edges can be used to make searching more efficient. In λ_{GT}, the skip list of level 2 can be expressed as shown in Figure 3.5, using the production rules in Figure 3.2.

Suppose we want to represent a skip list of arbitrary level. A skip list using a list of links to the nodes to be skipped can be represented as in Figure 3.6. Figure 3.3 shows the production rules for such lists. The rules exploit difference lists to link to a skipped node after Forking. The list of skipping links is terminated with the neXt atom.

For operations that cannot be performed by simply decomposing and composing graphs, we can prepare atoms to behave as markers and use them for matching. For example, a map function that applies a function to the element of the leaves in a leaf-linked tree can be expressed as Table 3.7.

Here, if 

is a function that returns the successor of a given number, and we have bound 

to 

, then evaluating the program will result in

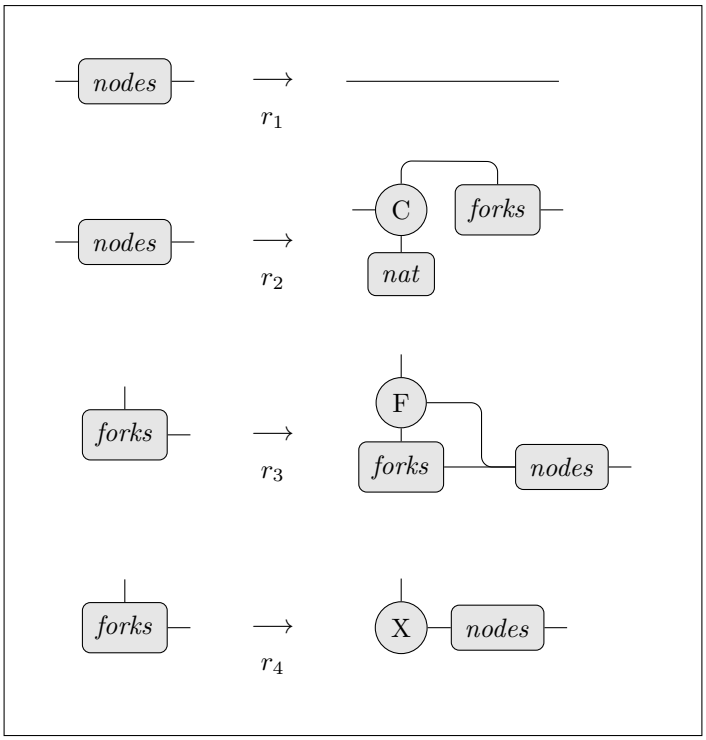


Figure 3.3: Production rules for an arbitrary-level skip list.

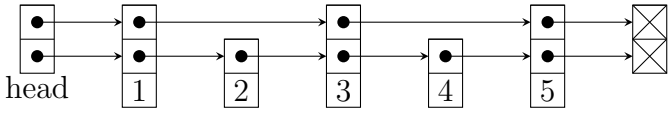
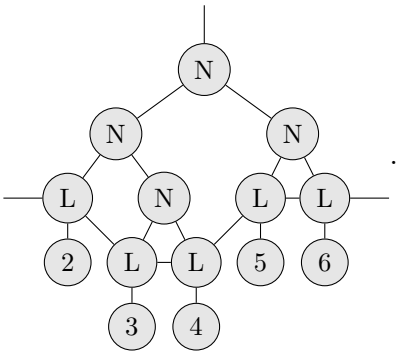


Figure 3.4: 2-level skip list with heaps and pointers.



3.3 Syntax and Semantics of λ_{GT}

This section describes the syntax and the semantics of λ_{GT}, which is a small, call-by-value functional language that employs hypergraphs as values and supports pattern matching for them. The main design issue is how to represent and manipulate hypergraphs in the setting of a functional language and how to let hypergraphs and abstractions co-exist in a unified framework.

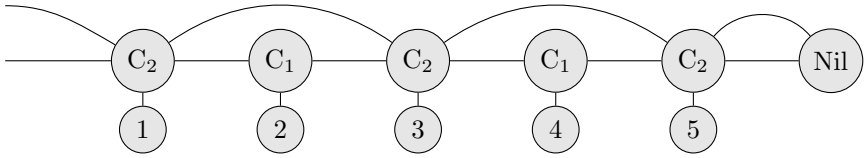


Figure 3.5: 2-level skip list in λ_{GT}.

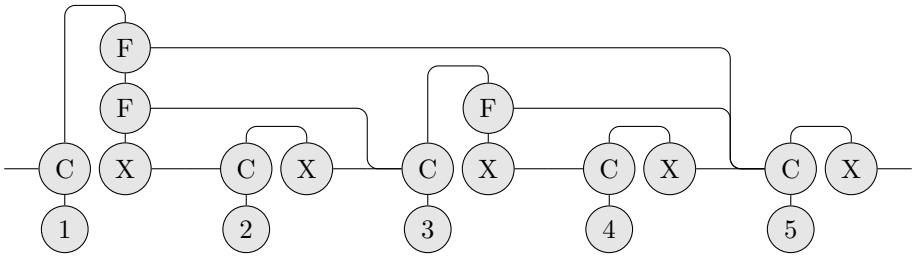


Figure 3.6: n-level skip list ($n = 3$).

3.3 Syntax of λ_{GT}

The λ_{GT} language is composed of the following syntactic categories.

- X denotes a *Link Name*.
- C denotes a *Constructor Name*.
- x denotes a *Graph Context Name*.

The syntax of the language is given in Figure 3.8. T is a *template* of a graph. It extends graphs in HyperLMNtal defined in Figure 2.1 with *graph contexts*. A graph context $x[\overrightarrow{X}]$, where $\overrightarrow{X}$ is a sequence of different links, is a wildcard in pattern matching corresponding to a variable in functional languages, It matches any graph with free links $\overrightarrow{X}$. Free links of a graph could be thought of as named parameters (or 'access points') of the graph. $C(\overrightarrow{X})$ is a constructor atom. Intuitively, it is a node of a data structure with links $\overrightarrow{X}$. We allow λ-abstractions as the names of atoms in graph templates T (and its subclass G to be defined shortly). The λ-abstraction atoms have the form $(\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Y})$. Intuitively, the atom takes a graph with free links $\overrightarrow{X}$, binds it to the graph context $x[\overrightarrow{X}]$ and returns the value (defined in Figure 3.9) obtained by evaluating the expression e with the bound graph context. Notice that the λ-abstraction $(\lambda\, x[\overrightarrow{X}].e)$ is *just the name of an atom*: λ-abstraction atoms can be incorporated into data structures just like atoms with constructor names. This is how λ_{GT} supports first-class functions in a graph setting. The free link(s) $\overrightarrow{Y}$ of the atom can be used to connect the atom to other structures such as lists to form a graph structure containing a first-class function. The links $\overrightarrow{X}$ and the links appearing in the graphs of the body expression e are *not* the free links of the atom.

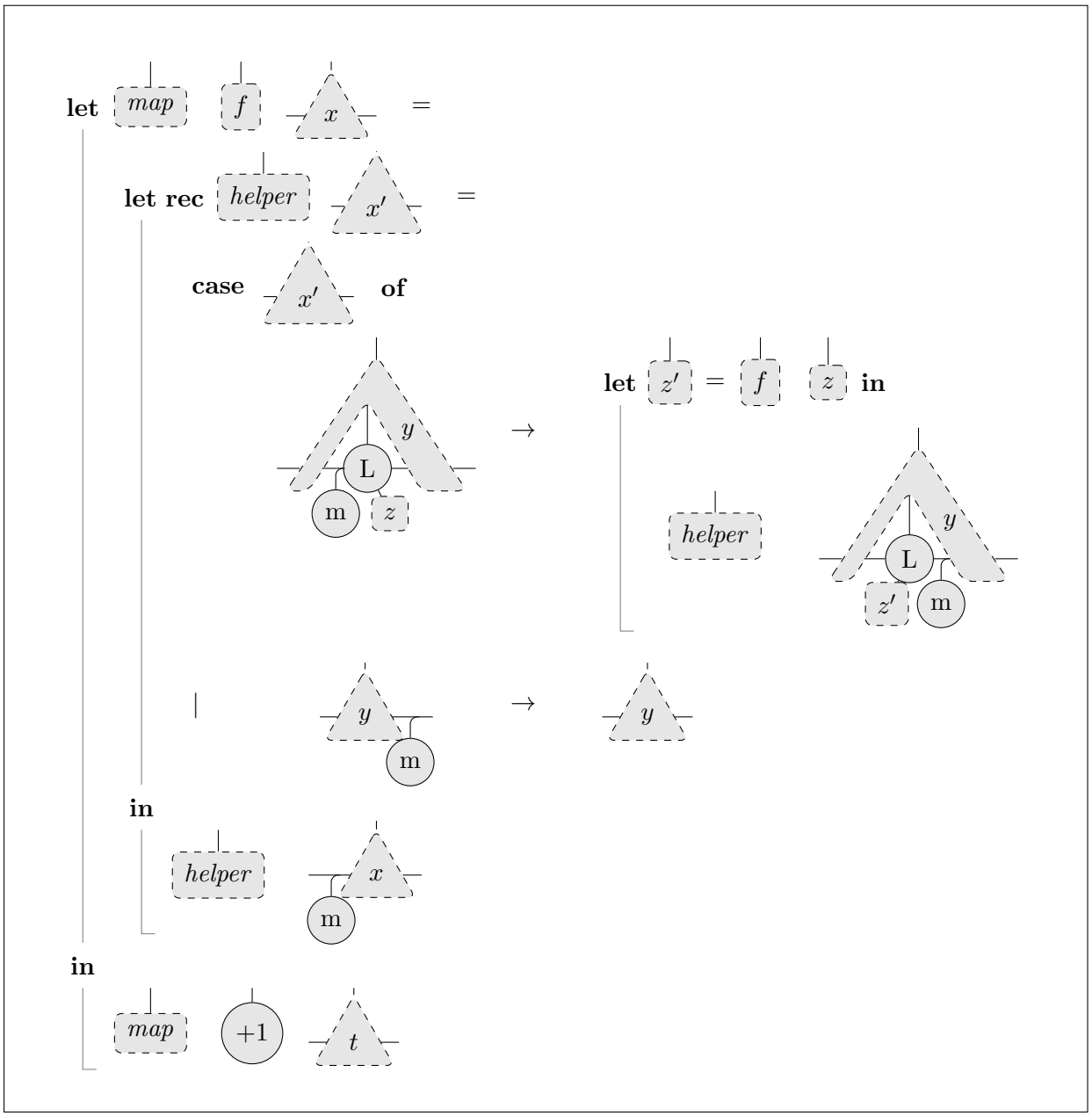


Figure 3.7: A map function for leaf linked trees.

(**case** e_1 **of** $T \to e_2$ | **otherwise** $\to e_3$) evaluates e_1, checks whether this matches the graph template T, and reduces to e_2 or e_3. The details are described in sections 3.3.2–3.3.2. The case expression covers just two cases in pattern matching, but we can nest the expression to handle more cases. (e_1 e_2) is an application.

Note that some graph rewriting languages including Interaction Nets [Mac06] and HyperLMNtal have encodings of the λ-calculus [Mac04; YU16] in which both abstractions and applications are *encoded* using explicit graph nodes and (hyper) links. In contrast, λ_{GT} features abstractions and applications at the language level so as to retain the standard framework of functional languages.

Graph Template

T	::=	$\mathbf{0}$	Null
	|	$x[\overrightarrow{X}]$	Graph Context
	|	$v\,(\overrightarrow{X})$	Atom
	|	$X \bowtie Y$	Fusion
	|	(T, T)	Molecule
	|	$\nu X.T$	Hyperlink Creation

Atom Name

v	::=	C	Constructor Name
	|	$\lambda\, x[\overrightarrow{X}].e$	Abstraction

Expression

e	::=	T	Graph
	|	$\mathbf{case}\ e\ \mathbf{of}\ T \to e \mid \mathbf{otherwise} \to e$	Case
	|	$(e\ e)$	Application

Figure 3.8: Syntax of λ_{GT}

G stands for a *value* of the language λ_{GT}, which is T not containing graph contexts. Henceforth, we may call both G and T a *graph* when the distinction is not important.

Definition 3.3.1 (Syntactic condition on expressions)**.** A λ-abstraction atom is not allowed to appear in the pattern T of the case expression $\mathbf{case}\ e_1\ \mathbf{of}\ T \to e_2 \mid \mathbf{otherwise} \to e_3$.

Definition 3.3.2 (Abbreviation rules for graph contexts)**.** We introduce the following abbreviation schemes to graph contexts as well as we have done to atoms.

1. The parentheses of nullary graph contexts can be abbreviated. For example, $x[]$ can be abbreviated as x.

2. Term Notation: $\nu X.(v\,(\ldots, X, \ldots), x[\ldots, X])$ can be abbreviated as $v\,(\ldots, x[\ldots], \ldots)$. The same can be done for embedding atoms (or graph contexts) in the argument of a graph context (or atoms), respectively.

Definition 3.3.3 (Free functors of an expression)**.** We define free functors of an expression e, $\mathit{ff}(e)$, in Figure 3.10. Free functors are not to be confused with free link names.

Value

G	$::=$	$\mathbf{0}$	Null
	$\mid$	$v\,(\overrightarrow{X})$	Atom
	$\mid$	$X \bowtie Y$	Fusion
	$\mid$	(G, G)	Molecule
	$\mid$	$\nu X.G$	Hyperlink Creation

Figure 3.9: Value of λ_{GT}

$$
\begin{aligned}
ff(\textbf{case } e_1 \textbf{ of } T \to e_2 \mid \textbf{otherwise} \to e_3) = \\
ff(e_1) \cup (ff(e_2) \setminus ff(T)) \cup ff(e_3)
\end{aligned}
$$

$$
\begin{aligned}
ff((e_1\ e_2)) &= ff(e_1) \cup ff(e_2) \\
ff(x[\overrightarrow{X}]) &= \{x/|\overrightarrow{X}|\} \\
ff(v\,(\overrightarrow{X})) &= \varnothing \\
ff(X \bowtie Y) &= \varnothing \\
ff((\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Y})) &= ff(e) \setminus \{x/|\overrightarrow{X}|\} \\
ff((T_1, T_2)) &= ff(T_1) \cup ff(T_2) \\
ff(\nu X.T) &= ff(T)
\end{aligned}
$$

Figure 3.10: Free functors of an expression

3.3 Operational Semantics of λ_{GT}

First, we define the congruence rules ($\equiv$) and the link substitutions, $T\langle Y/X\rangle$ and $G\langle Y/X\rangle$, for T and G in the same manner as we have defined in Section 2.2. Although there is no graph context in Flat HyperLMNtal, the link substitution for $x[\overrightarrow{X}]$ in T can be defined in the same way as the one for atoms in HyperLMNtal.

Graph Substitution

We define *graph substitution*, which replaces a graph context whose functor occurs free by a given subgraph. The substitution avoids clashes with any bound functors by implicit α-conversion (capture-avoiding substitution). Graph substitution is not to be confused with *hyperlink substitution*. Intuitively, hyperlink substitution just reconnects hyperlinks. On the other hand, graph substitution performs deep copying at the semantics level (though it could or should be implemented with sharing whenever possible).

We define capture-avoiding substitution θ of a graph context $x[\overrightarrow{X}]$ with a template

$$
\begin{aligned}
(T_1, T_2)\theta &= (T_1\theta, T_2\theta) \\
(\nu X.T)\theta &= \nu X.T\theta \\
(x[\overrightarrow{X}])[T/y[\overrightarrow{Y}]] &= \\
&\quad \text{if } x/|\overrightarrow{X}| = y/|\overrightarrow{Y}| \text{ then } T\left\langle \overrightarrow{X}/\overrightarrow{Y} \right\rangle \\
&\quad \text{else } x[\overrightarrow{X}] \\
(C(\overrightarrow{X}))\theta &= C(\overrightarrow{X}) \\
(X \bowtie Y)\theta &= X \bowtie Y \\
((\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Z}))[T/y[\overrightarrow{Y}]] &= \\
&\quad \text{if } x/|\overrightarrow{X}| = y/|\overrightarrow{Y}| \text{ then } (\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Z}) \\
&\quad \text{else if } x/|\overrightarrow{X}| \notin \mathit{ff}(e) \text{ then } (\lambda\, x[\overrightarrow{X}].e[T/y[\overrightarrow{Y}]])(\overrightarrow{Z}) \\
&\quad \text{else } (\lambda\, z[\overrightarrow{X}].e[z[\overrightarrow{X}]/x[\overrightarrow{X}]][T/y[\overrightarrow{Y}]])(\overrightarrow{Z}) \\
&\qquad\qquad \text{where } z/|\overrightarrow{X}| \notin \mathit{ff}(e). \\
(\textbf{case } e_1 \textbf{ of } T \to e_2 \mid \textbf{otherwise} \to e_3)\theta & \\
&= \textbf{case } e_1\theta \textbf{ of } T \to e_2\theta \mid \textbf{otherwise} \to e_3\theta \\
(T_1\ T_2)\theta &= (T_1\theta\ \ T_2\theta)
\end{aligned}
$$

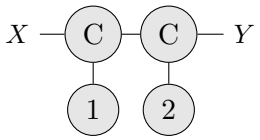
Figure 3.11: Graph Substitution

T in e, written $e[T/x[\overrightarrow{X}]]$, as in Figure 3.11. The definition is standard except that it handles the substitution of the free links of graph contexts in the third rule.

Matching

We say that T matches a graph G if there exists graph substitutions θ such that $G \equiv T\overrightarrow{\theta}$. The graphs in the range of substitutions should not contain free occurrence of graph contexts: i.e., the substitution should be ground. Since the matching of λ_{GT} does not involve abstractions (by Def. 3.3.1), in which case G of λ_{GT} is essentially the same as G of HyperLMNtal, we employ the $\equiv$ defined in Figure 2.4. For example, Figure 3.12 shows the matching of a difference list.

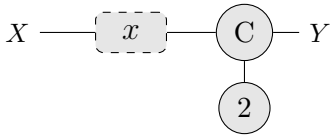
Note that the matching of λ_{GT} is not subgraph matching (as is standard in graph rewriting systems) but the matching with the entire graph G (as is standard in pattern matching of functional languages). For this reason, the free link names appearing in a template T must exactly match the free links in the graph G to be matched. This is to be contrasted with free links of HyperLMNtal rules that are effectively α-convertible since the rules can match subgraphs by supplementing fusion atoms ([SU21, section 4.4]). The matching can be done non-deterministic. We are planning to put constraints over the

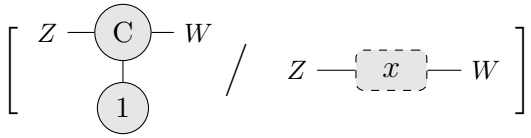


G

$$
\begin{array}{l}
\nu Z.(\\
\quad \nu Z_1.(\mathrm{Cons}(Z_1, Z, X), 1(Z_1)), \\
\quad \nu Z_2.(\mathrm{Cons}(Z_2, Y, Z), 2(Z_2)) \\
)
\end{array}
$$

$$\equiv$$





T $\qquad\qquad$ θ

$$
\begin{array}{ll}
\nu Z.(& \\
\quad x[Z, X], & \left[\nu Z_1.(\mathrm{Cons}(Z_1, Z, W), 1(Z_1)) \Big/ x[Z, W]\right] \\
\quad \nu Z_2.(\mathrm{Cons}(Z_2, Y, Z), 2(Z_2)) & \\
) &
\end{array}
$$

Figure 3.12: Example of the graph matching

$$\frac{G \equiv T\,\overrightarrow{\theta}}{(\textbf{case}\ G\ \textbf{of}\ T \to e_2 \mid \textbf{otherwise} \to e_3) \longrightarrow_{\texttt{val}} e_2\,\overrightarrow{\theta}}\ \text{Rd-Case1}$$

$$\frac{\neg\exists\,\overrightarrow{\theta}.G \equiv T\,\overrightarrow{\theta}}{(\textbf{case}\ G\ \textbf{of}\ T \to e_2 \mid \textbf{otherwise} \to e_3) \longrightarrow_{\texttt{val}} e_3}\ \text{Rd-Case2}$$

$$\frac{fn(G) = \{\overrightarrow{X}\}}{((\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Y})\ G) \longrightarrow_{\texttt{val}} e[G/x[\overrightarrow{X}]]}\ \text{Rd-}\beta$$

$$\frac{e \longrightarrow_{\texttt{val}} e'}{E[e] \longrightarrow_{\texttt{val}} E[e']}\ \text{Rd-Ctx}$$

Figure 3.13: Reduction relation of λ_{GT}

graph templates in case expressions to ensure deterministic matching but it is a future task.

Reduction

We choose the call-by-value evaluation strategy. The reason we did not choose call-by-need (or call-by-name) is to avoid infinite graphs to use infinite-descent in the verification later in Section 4.4.

In order to define the small-step reduction relation, we extend the syntax with evaluation contexts defined as follows:

$$E \quad ::= \quad [] \quad \mid \quad (\textbf{case}\ E\ \textbf{of}\ T \to e \mid \textbf{otherwise} \to e) \quad \mid \quad (E\ e) \quad \mid \quad (G\ E) \quad \mid \quad T$$

As usual, $E[e]$ stands for E whose hole is filled with e.

We define the reduction relation in Figure 3.13.

Definition 3.3.4 (Abbreviation rules for λ-abstraction atom)**.** We introduce a shorthand notation similar to the λ-calculus.

1. Application is left-associative.
2. $(\lambda\, x[\overrightarrow{X}].(\lambda\, y[\overrightarrow{Y}].e)(\overrightarrow{Z}))(\overrightarrow{Z})$ can be abbreviated as $(\lambda\, x[\overrightarrow{X}]\, y[\overrightarrow{Y}].e)(\overrightarrow{Z})$.

$$
\begin{aligned}
&\textbf{let } \mathit{append}[Z] = (\lambda\, x[Y,X]\ y[Y,X].\, x[y[Y],X])(Z)\\
&\quad \textbf{in } \mathit{append}[Z]\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X)\\
\longrightarrow_{\mathtt{val}}\ &(\lambda\, x[Y,X]\ y[Y,X].\, x[y[Y],X])(Z)\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X)\\
\longrightarrow_{\mathtt{val}}\ &(\lambda\, y[Y,X].\, x[y[Y],X])(Z)[\mathrm{Cons}(1,Y,X)/x[Y,X]]\ \ \mathrm{Cons}(2,Y,X)\\
=\ \ &(\lambda\, y[Y,X].\, \mathrm{Cons}(1,y[Y],X))(Z)\ \ \mathrm{Cons}(2,Y,X)\\
\longrightarrow_{\mathtt{val}}\ &(\mathrm{Cons}(1,y[Y],X))(X)[\mathrm{Cons}(2,Y,X)/y[Y,X]]\\
=\ \ &\mathrm{Cons}(1,\mathrm{Cons}(2,Y),X)
\end{aligned}
$$

Figure 3.14: An example of reduction: append operation on difference lists

3. $((\lambda\, x[\overrightarrow{X}].e_1)(\overrightarrow{Y})e_2)$ can be abbreviated as **let** $x[\overrightarrow{X}] = e_2$ **in** e_1. The $\overrightarrow{Y}$ will disappear immediately after evaluating the expression, doing nothing, in β-reduction. Thus, we omit the links in the abbreviation.

For example, we can describe a program to append two singleton difference lists as follows (detailed description of difference lists will be given in Example 4.2.2):

$$
\begin{aligned}
&\textbf{let } \mathit{append}[Z] =\\
&\quad (\lambda\, x[Y,X]\ y[Y,X].\\
&\qquad\quad x[y[Y],X]\\
&\quad)(Z)\\
&\textbf{in } \mathit{append}[Z]\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X)
\end{aligned}
$$

We show the whole process of reduction of this program in Figure 3.17 and graphically in Figure 3.15, which we have already shown in Section 3.2 informally, omitting free link names and port indices.

Firstly, the λ-abstraction atom is bound to the graph context $\mathit{append}[Z]$[2]. The bound λ-abstraction atom is a function that takes two difference lists, both having X and Y as free links, and returns their concatenation also having X and Y as its free links.

[2]It may appear that the Z of $\mathit{append}[Z]$ does not play any role in this example. However, such a link becomes necessary when the *append* is made to appear in a data structure (e.g., as in $\nu Z.(\mathrm{Cons}(Z,Y,X), \mathit{append}[Z])$). This is why λ-abstraction atoms are allowed to have argument links. Once such a function is accessed and β-reduction starts, the role of Z ends, while the free links *inside* the abstraction atom start to play key roles.

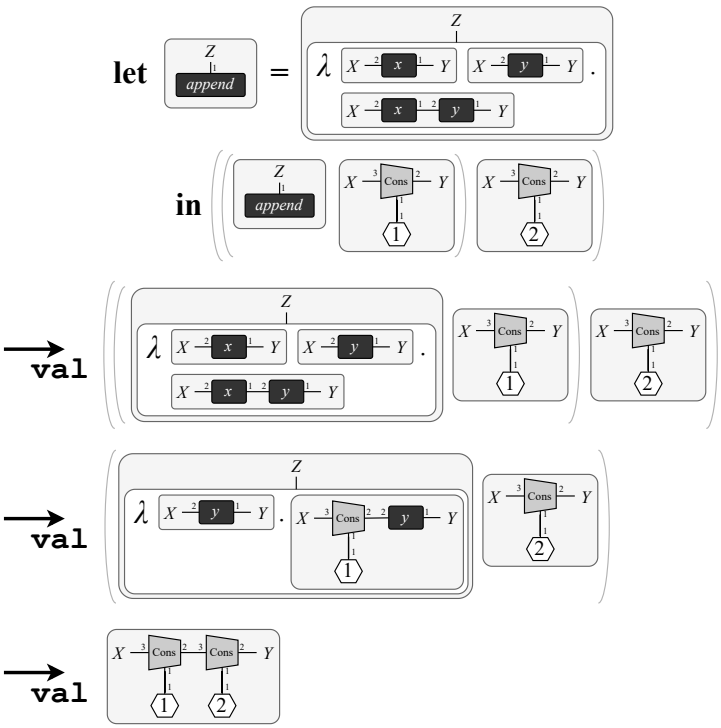


Figure 3.15: Visualized version of the reduction process in Figure 3.17.
Small numbers around a non-unary atom indicate the ordering of arguments, and a small dot among edges stands for a *fusion* of the edges.

A program that pops the last element of a difference list can be described as follows.

$$
\begin{aligned}
&\textbf{let } pop[Z] = \\
&\quad (\lambda\, x[Y, X]. \\
&\qquad \textbf{case } x[Y, X] \textbf{ of} \\
&\qquad\quad y[\mathrm{Cons}(z, Y), X] \to y[Y, X] \\
&\qquad\quad \mid \textbf{otherwise} \to x[Y, X] \\
&\quad)(Z) \\
&\textbf{in } pop[Z]\ \ \mathrm{Cons}(1, \mathrm{Cons}(2, Y), X)
\end{aligned}
$$

This will result in $\mathrm{Cons}(1, Y, X)$.

3.4 Program Examples in Detail

This section describes some of the programs we have introduced informally in Section 3.2 in detail with the formal semantics in order to discuss the implementation algorithm in later sections.

For the sake of explanation, the graph of evaluation results may be rewritten using the structural congruence rules.

$$
\begin{array}{l}
\textbf{let } \mathit{append}[Z] = \\
\quad (\lambda\, x[Y,X]. \\
\quad\quad (\lambda\, y[Y,X]. \\
\quad\quad\quad \nu Z.(x[Z,X], y[Y,Z]) \\
\quad)(Z))(Z) \\
\textbf{in } \mathit{append}[Z]\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X)
\end{array}
$$

Figure 3.16: Append operation on difference lists

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{let } \mathit{append}[Z] = (\lambda\, x[Y,X].\, (\lambda\, y[Y,X].\, x[y[Y],X])(Z))(Z)} \\
\multicolumn{3}{l}{\quad \textbf{in } \mathit{append}[Z]\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X)} \\
\longrightarrow_{\mathtt{val}} & (\lambda\, x[Y,X].\, (\lambda\, y[Y,X].\, x[y[Y],X])(Z))(Z)\ \ \mathrm{Cons}(1,Y,X)\ \ \mathrm{Cons}(2,Y,X) \\
\longrightarrow_{\mathtt{val}} & (\lambda\, y[Y,X].\, x[y[Y],X])(Z)[\mathrm{Cons}(1,Y,X)/x[Y,X]]\ \ \mathrm{Cons}(2,Y,X) \\
= & (\lambda\, y[Y,X].\, \mathrm{Cons}(1,y[Y],X))(Z)\ \ \mathrm{Cons}(2,Y,X) \\
\longrightarrow_{\mathtt{val}} & (\mathrm{Cons}(1,y[Y],X))(X)[\mathrm{Cons}(2,Y,X)/y[Y,X]] \\
= & \mathrm{Cons}(1,\mathrm{Cons}(2,Y),X)
\end{array}
$$

Figure 3.17: An example of reduction: append operation on difference lists

Graphs as Inputs and Output of a Function

For example, we can describe a program to append two singleton difference lists as in Figure 3.16.

We show the whole process of reduction of this program in Figure 3.17, whose process is already shown graphically in Section 3.2.

Firstly, the λ-abstraction atom is bound to the graph context $\mathit{append}[Z]$[3]. The bound λ-abstraction atom is a function that takes two difference lists, both having X and Y as free links, and returns their concatenation also having X and Y as its free links.

Pattern Matching Graphs

Figure 3.18 shows the program, a slightly simplified version of the program introduced in Section 3.2, that matches the difference list and removes the last node. If $\mathrm{Cons}(1,\mathrm{Cons}(2,Y),X)$ is bound to $x[Y,X]$, this will result in $\mathrm{Cons}(1,Y,X)$.

[3]It may appear that the Z of $\mathit{append}[Z]$ does not play any role in this example. However, such a link becomes necessary when the *append* is made to appear in a data structure (e.g., as in $\nu Z.(\mathrm{Cons}(Z,Y,X), \mathit{append}[Z])$). This is why λ-abstraction atoms are allowed to have argument links. Once such a function is accessed and β-reduction starts, the role of Z ends, while the free links *inside* the abstraction atom start to play key roles.

```
let pop[Z] =
  (λ x[Y, X].
    case x[Y, X] of
      y[Cons(z, Y), X] → y[Y, X]
    | otherwise → x[Y, X]
  )(Z)
in pop[Z]  x[Y, X]
```

Figure 3.18: Pop operation on a difference list

Without the term-notation abbreviation, the graph template used in the matching can be written as $\nu W.(y[W, X], \mathrm{Cons}(Z, Y, W), z[Z])$. The matching in this template proceeds as the following, which is mostly the same as we have explained in Figure 3.12 in the previous section.

$$\begin{aligned} & \mathrm{Cons}(1, \mathrm{Cons}(2, Y), X) \\ \equiv\ & \nu W Z.(y[W, X], \mathrm{Cons}(Z, Y, W), z[Z]) \\ & \quad [\mathrm{Cons}(1, W, X)/y[W, X],\ 2(Z)/z[Z]] \end{aligned}$$

In order to implement the language precisely, we also need to consider the *corner case*; for example, the matching which exploits *fusion*.

Consider if a singleton list $\nu Z.(\mathrm{Cons}(Z, Y, X), 1(Z))$ is bound to $x[Y, X]$. We need a subgraph that has free links W and X, the free links of the graph context $y[W, X]$ in the graph template, which does not exist in the list. Thus the matching would not proceed without supplying subgraphs.

This time, we need to firstly *supply a fusion atom*. Then we can match $y[W, X]$ to the supplied fusion atom. The matching proceeds as follows.

$$\begin{aligned} & \mathrm{Cons}(1, Y, X) \\ \equiv\ & \nu W Z.(\underline{W \bowtie X}, \mathrm{Cons}(Z, Y, X), 1(Z)) \\ =\ & \nu W Z.(y[W, X], \mathrm{Cons}(Z, Y, W), z[Z]) \\ & \quad [W \bowtie X/y[W, X],\ 1(Z)/z[Z]] \end{aligned}$$

Therefore, the program will result in $Y \bowtie X$.

The supplied fusion does not always appear in the result; it may be absorbed and does not appear explicitly in the return value. Consider the program in Figure 3.19 that cycles the elements by taking the last node of the difference list and reconnecting it to the head. In this program, the supplied fusion can be absorbed.

If $\mathrm{Cons}(1, Y, X)$ is bound to $x[Y, X]$, this will result in $\mathrm{Cons}(1, Y, X)$. The pattern matching proceeds in the same way as in the previous program. Thus we will obtain

```
let rotate[Z] =
  (λ x[Y, X].
    case x[Y, X] of
      y[Cons(z, Y), X] → Cons(z, y[Y], X)
    | otherwise → x[Y, X]
  )(Z)
in rotate[Z]  x[Y, X]
```

Figure 3.19: Rotate operation on a difference list

the substitution $[W \bowtie X/y[W, X],\ 1(Z)/z[Z]]$. Substituting the graph template on the right-hand side of $\rightarrow$ will result in $\mathrm{Cons}(1, Y, X)$ as follows.

$$\begin{aligned} & \nu W Z.(\mathrm{Cons}(Z, W, X), z[Z], y[Y, W]) \\ & \quad [W \bowtie X/y[W, X],\ 1(Z)/z[Z]] \\ = \ & \nu W Z.(W \bowtie X, \mathrm{Cons}(Z, Y, X), 1(Z)) \\ \equiv \ & \mathrm{Cons}(1, Y, X) \end{aligned}$$

The program using the map function for leaf-linked trees in Figure 3.7 in Section 3.2 also uses the matching with supplying fusions.

Such fusion-complementary matching does not appear in ordinary ADT matching in functional languages. Also, formalization using fusions is not common in ordinary graph transformations, as well as the tools based on them. Thus, it is a *challenge* to deal with it.

3.5 Reference Interpreter

In this study, we have implemented a reference interpreter: the POC of the λ_{GT} language. We firstly describe the motivation to implement a reference interpreter, then explain our implementation, and give some discussion on our implementation.

3.5 Motivation

We implement a *reference interpreter*, a reference implementation of the language, which has several potential uses as follows.

For research of the design of real languages. λ_{GT} is a computational model with a new concept. While the operational semantics are defined, this only defines the behavior as a computational model, and does not define how we can implement data structures and perform efficient pattern matching. It is ultimately up to the language designer to decide how to implement this.

If general hypergraphs are handled purely and no restrictions are placed on pattern matching to them, an efficient implementation will be difficult. Therefore, in designing a real language, it is realistic to put in appropriate restrictions while using the type system as support. However, these constraints should not exclude practicality. Actual programming using the reference interpreter is useful to determine whether or not it is practical to include constraints.

For testing future implementations. We intend to build a more efficient implementation in the future. However, since we are dealing with complex data structures, we need to do low-level programming making full use of pointers at the meta-level, which is not easy. Therefore, it is assumed that development will proceed with testing. To test the results, it is useful to have an implementation that outputs the correct results, even if the execution efficiency is poor.

To develop applications using λ_{GT}. It is better to have a runtime to find and test programs that can be written concisely using λ_{GT}.

To develop tools. This study pioneers a method for representing graphs in terms of terms, and gives a semantics based on them. This is advantageous in terms of semantics and verification. However, it is not clear whether it is easy for users to write. If we are dealing with graphs, it is considered more intuitive to be able to draw them graphically. Therefore, the editor of λ_{GT} may be GUI-based. It would be advantageous to be able to actually run the tool when developing tools.

3.5 Implementation

Overview

The goal of this study is to implement as simple as possible, without regard to efficiency. Our implementation consists of only 500 lines of OCaml code as shown in Table 3.1. This is about half of the lines of a reference interpreter of a graph transformation-based language GP 2[Bak15], which is about 1,000 lines of Haskell code [Bak+15]. This is striking considering that our language does not only support graph transformation but we have incorporated it into a functional language without sacrificing functional language features such as higher-order functions.

The interpreter is composed of pure functions without destructive operations. We used lists to represent graphs.

Parser

Our current implementation is not intended to provide a complete language that can be used in real-world programming. The final design of the concrete syntax is left to the designer of the actual language. There is even a possibility of providing a graphical UI and not allowing the language to be written in texts. In this study, we gave a concrete

Table 3.1: LOC of the interpreter

File	LOC
eval/match_ctxs.ml	79
parser/parser.mly	70
parser/lexer.mll	51
eval/syntax.ml	47
eval/eval.ml	43
eval/pushout.ml	42
eval/match_atoms.ml	36
eval/preprocess.ml	36
parser/syntax.ml	16
eval/match.ml	11
parser/parse.ml	4
bin/main.ml	3
SUM	438

syntax that is easy to parse. The syntax is not sophisticated enough for programmers to write easily. However, this still satisfies our purpose.

We define the concrete syntax as follows:

- Link name starts from a capital letter with an underscore as a prefix. For example, `_X`.
- $\nu X_1 \ldots \nu X_n.$ is written as `nu _X1 ... _Xn.`.
- λ-abstraction atom $(\lambda\, x[\overrightarrow{X}].e)(\overrightarrow{Y})$ is written with `<` and `>` as follows:

  `<\x[_X1, ..., _Xn]. e> (_Y1, ..., _Ym)`.
- The graph templates appears in expressions should be surrounded with `{` and `}`.

For example, The example Figure 3.7 we have introduced in Section 3.2 can be written in the concrete syntax as in Figure 3.20.

The interpreter preprocesses graphs before moving on to the evaluation. In our implementation, values, i.e., graphs, are represented with lists of atoms without link creations inside; i.e., prenex normal form. We call them *host graphs*. Links are classified into free links with `string` names and local links α-converted to fresh `int`eger ids. In this thesis, we denote L_i for the local link with the id i and F_X for the free link with the name X.

Fusion atoms with local link(s) are absorbed beforehand. Currently, we assign numbers starting from 1,000 to the local links in the template to avoid conflict with the host graphs

```
let f[_X] =
  {<\x[_X]. {nu _X1 _X2. (Succ (_X1, _X), x[_X1])}>(_X)}
in

let map[_X] =
  {<\f[_X].{<\x[_L, _R, _X].
  let rec helper[_X] x2[_L, _R, _X] =
    case {x2[_L, _R, _X]} of
      {nu _L2 _R2 _X2 _X3. (
        y[_L, _R, _X, _L2, _R2, _X2],
        Leaf (_X3, _L2, _R2, _X2),
        z[_X3],
        M (_L2)
      )} ->
        let z2[_X] = {f[_X]} {z[_X]} in
        {helper[_X]}
        {nu _L2 _R2 _X2 _X3 _X4. (
          y[_L, _R, _X, _L2, _R2, _X2],
          Leaf (_X3, _L2, _R2, _X2),
          z2[_X3],
          M (_R2)
        )}
    | otherwise -> case {x2[_L, _R, _X]} of
      { y[_L, _R, _X], M (_R) } -> { y[_L, _R, _X] }
    | otherwise -> {Error, x2[_L, _R, _X]}
  in {helper[_X]} {x[_L, _R, _X], M (_L)}
  >(_X)}>(_X)} in

{map[_X]}
{f[_X]}
{
nu _X1 _X2 _X3 _X4 _X5. (
  Node (_X1, _X2, _X),
  Leaf (_X4 ,_L, _X3, _X1),
  Zero (_X4),
  Leaf (_X5, _X3, _R, _X2),
  Zero (_X5)
)
}
```

Figure 3.20: Map leaves of a leaf-linked tree.

whose local links are assigned numbers starting from 0[4].

The interpreter transforms graph templates to a pair of the list of the atoms and the list of the graph contexts. For example, the graph template $y[\mathrm{Cons}(z, Y), X]$, the pattern in the Figure 3.18, is transformed into the following.

$$\begin{aligned} \langle\, &[\mathrm{Cons}(L_{1001}, F_Y, L_{1000})]\,, \\ &[y\,[L_{1000}, F_X]\,, z\,[L_{1001}]]\,\rangle \end{aligned} \tag{3.1}$$

Whereas the host graph $\mathrm{Cons}(1, Y, X)$ is transformed into the following.

$$[\mathrm{Cons}(L_0, F_Y, F_X), 1(L_0)] \tag{3.2}$$

This preprocessing occurs every time evaluating the expression. This can be easily optimized to be memorized. However we focused more on the simplicity of the implementation.

Matching Graphs

The interpreter firstly tries to (i) match all the atoms in the template to the host graph, (ii) supply fusions if necessary, and then (iii) match the graph contexts to the rest of the host graph. The interpreter backtracks if the consequent matching failes.

(i) *Matching atoms.*

We take an atom from the head of the list of atoms in the graph template and try to find the corresponding atom from the host graph. If the matching succeeds, the atom in the host graph is removed.

To match an atom, we need to check that they have the same name and the correspondence of links. Since the local link names are α-convertible, it is not sufficient only to check that they have the same link names. Therefore, we exploit a *link environment*, a mapping from the local link names in the template to the link names in the host graph.

Using structural congruence rules such as (E6), we can *fuse* local link names. Therefore, it seems that different link names can be mapped to the same link name, and vice-versa. However, since we have absorbed all the fusion atoms that have local links in the graph template, the former situation is impossible. Thus the link environment is functional, i.e., the same link names are mapped to the same link name. Notice the latter is still possible since we can supply fusion atoms to the host graph in the matching.

The matching of link names using the link environment proceeds as in Figure 4.13. Free links in the template match links with the same names in the host graph (line 7–8). On the other hand, the matching of local links in the template (line 10–14) is more flexible, since we can α-convert them.

For example, the atom $\mathrm{Cons}(L_{1001}, F_Y, L_{1000})$ in the template (3.1) can be matched to the atom $\mathrm{Cons}(L_0, F_Y, F_X)$ in the host graph (3.1) with link environment

$$\{L_{1000} \mapsto F_X, L_{1001} \mapsto L_0\} \tag{3.3}$$

[4]This may be too ad-hoc solution but is simple and does work in our examples.

and we have

$$[1(L_0)] \tag{3.4}$$

left in the host graph.

(ii) *Supplying Fusions.*

After all the atoms in the template have matched, we substitute link names in the host graph with inverse of the obtained link environment. For example, the rest host graph (3.4) becomes the following.

$$[1(L_{1001})] \tag{3.5}$$

We supply fusion atoms to the host graph using the link environment obtained in the matchings of atoms. If the mapping is not injective, then we should supply fusion atoms to the host graph. For example, if we obtain the link environment

$$\{L_{1000} \mapsto L_0, L_{1001} \mapsto L_0\}$$

then we should supply $L_{1000} \bowtie L_{1001}$.

If a local link in the template is mapped to a free link in the host graph, then we also need to supply a fusion since a local link does not match a free link without such treatment. For example, in the link environment (3.3), we have a mapping $\{L_{1000} \mapsto F_X\}$. Thus, we should supply $L_{1000} \mapsto F_X$ and obtain

$$[1(L_{1001}), L_{1000} \bowtie F_X] \tag{3.6}$$

as the rest subgraph.

Since we have preprocessed the template to have no fusion atoms with local links, this fusion-supplying task can be performed after all the atoms have matched. However, since graph contexts can match with fusions, we need to perform the task before moving on to the matching of graph contexts.

(iii) *Matching Graph Contexts.*

Limitation. We place the following two limitations on the graph contexts to make the implementation simple: (i) the graph that a graph context can match is connected and (ii) the local links of a graph context are connected to atom(s). The matching that does not satisfy these two constraints can be highly non-deterministic, which we believe is not practical and thus is not the main scope of our language.

The matching of a graph context $x[\overrightarrow{X}]$ with subgraph G, initially Null, proceeds as follows.

1. Find all the atom(s) with link $\overrightarrow{Y}$ where $\{\overrightarrow{Y}\} \cap \{\overrightarrow{X}\} \neq \varnothing$. Add the atoms to G.
2. Update $\overrightarrow{X}$ with $\{\overrightarrow{Y}\} \setminus \{\overrightarrow{X}\}$ and iterate from (1) again.
3. If we obtain no more newly added atom, then check the free links of G is the same as the links of the graph context $x[\overrightarrow{X}]$.

```
1  let check_link
2      σ                                          (* Link environment *)
3      X                                          (* Link in the template *)
4      Y                                          (* Link in the host graph *)
5  =
6      match (X, Y) with
7      | (F_X, F_Y) →
8          if X = Y then Some σ else None
9      | (F_X, L_i) → None
10     | (L_i, Y) →
11         if L_i ↦ Z ∈ σ then
12             if Y = Z then Some σ
13             else None
14         else Some (σ ∪ {L_i ↦ Y})
```

Figure 3.21: Link name matching

For example, $y[L_{1000}, F_X]$ and $z[L_{1001}]$, the graph contexts in (3.1), can match host graphs $L_{1000} \bowtie F_X$ and $1(L_{1001})$, the subgraphs in (3.6), respectively. After the matching, we obtain the following graph substitution.

$$[L_{1000} \bowtie F_X / y\,[L_{1000}, F_X]\,, 1(L_{1001}) / z\,[L_{1001}]] \tag{3.7}$$

Graph Substitution

Graph substitution can be done by adding the matched atom(s) G to the host graph, the list of atoms. However, we need to take care of link names. As we have defined in Figure 3.11 in Section 3.3, we need to substitute the link names of G with the link names of the graph context in an evaluating template. Therefore, if we have matched G with $x[\overrightarrow{X}]$ and the template has $x[\overrightarrow{Y}]$, we need to update G with substitution $\langle Y_1/X_1\rangle \ldots \left\langle Y_{|\overrightarrow{Y}|}/X_{|\overrightarrow{X}|}\right\rangle$. We also need to reassign ids for the local links since composing graphs may result in a conflict of ids.

With the obtained graph substitution (3.7), we can instantiate the template on the right-hand side of $\rightarrow$ in Figure 3.18, $[y[F_Y, F_X]]$, which will result in

$$[F_Y \bowtie F_X] \tag{3.8}$$

Our implementation absorbs fusions as much as possible after graph substitutions. The fusion atom in (3.8) cannot be absorbed since both its links are free links. On the other hand, the example in Figure 3.19 will result in

$$\begin{bmatrix} L_{1000} \bowtie F_X, \\ \mathrm{Cons}(L_{1001}, F_Y, L_{1000}), \\ 1(L_{1001}) \end{bmatrix}$$

which will be normalized and reassigned ids to obtain the following.

$$[\mathrm{Cons}(L_0, F_Y, F_X), 1(L_0)]$$

The Evaluator

The evaluator is implemented just as these for functional languages. It takes an environment, a mapping from graph contexts to the matched subgraphs, and an expression to evaluate.

3.5 Discusson

The soundness of the matching seems to be trivial but the precise discussion or formal verification is a matter for the future.

The pattern matching we have implemented in the interpreter is not complete; i.e., there exists a graph that should be matched but failed in our current implementation. For example, $(X \bowtie L, \mathrm{Leaf}(\mathrm{Zero}, X, R))$ should be able to be matched with $(X \bowtie L, \mathrm{Leaf}(\mathrm{Zero}, L, R))$ since they are congruent by Theorem 2.2.3. However, Figure 4.13 matches a free link name with that which has precisely the same name, thus the example fails because X cannot be matched with L. It seems to be not that difficult to deal with this example, but to design a complete matching algorithm and verify it, we need to investigate more on the theoretical foundation of HyperLMNtal congruence.

The implementation in this study is only a Proof of Concept: execution efficiency is not considered. To improve execution performance to the same level as the corresponding imperative code, it is necessary to develop static analysis. We are planning to extend the type system to check the direction (polarity) of links [Ued14], and then perform ownership checking [DM05]. Then, we develop a method to a transpile to a lower-level code using reference types in functional languages such as OCaml or an imperative code with pointers.

3.5 Related Work

There are several languages based on graph transformations. For example, AGG [RET12], GAMMA [BL93], Structured Gamma [FM98], GP [MP08], GP 2 [Bak15], GROOVE [Gha+12], GrGen.NET [JBK10], (Hyper) LMNtal [Ued09; UO12; SU21], PORGY [Fer+14], and PROGRES [SWZ97]. However, as far as we know, few published implementations have focused on simplicity.

HyperLMNtal, which is the language we have incorporated, has the compiler [LMN] and the runtime SLIM [SLI; GHU11]. The compiler is written in Java in around 12,000 lines and the runtime is written in C++ in around 47,000 lines. SLIM is highly optimized and enables non-deterministic execution and model checking, which is out of the scope of our language and implementation. Thus, we cannot say our implementation surpasses it. Even so, the contrast with the 500 lines of OCaml code for our interpreter is conspicuous.

GP 2 has a reference interpreter [Bak+15]. This is written in around 1,000 lines of Haskell sources without sacrificing performance significantly. Though we sacrificed performance, we have implemented the language that exceeds graph transformation in about half of the lines in OCaml.

4

F_{GT}: A Type System for λ_{GT}

4.1 Introduction

In this chapter, we propose a new type system, F_{GT}, for the λ_{GT} language. In F_{GT}, we define the type of graphs using graph grammar. This can be regarded as an extension of regular tree grammar, on which algebraic data types are based. We also develop a new type-checking algorithm that automatically performs this verification using structural induction. Our approach is in contrast with the analysis of pointer manipulation programs using separation logic [Rey02], shape analysis, etc. in that (i) we consider graph structures formed by higher-level languages that abstract pointers and heaps away and guarantee low-level invariants such as the absence of dangling pointers and that (ii) we pursue what properties can be established automatically using a rather simple typing framework.

Contributions

The main contributions in this chapter are twofold.

1. We define the syntax and the typing rules of the basic F_{GT} and prove some properties including soundness.
2. We extend the typing framework for the λ_{GT} language so that can successfully handle more manipulations of graphs, including which could not be handled in a previous study, Structured Gamma.

Chapter Map

The rest of this chapter is organized as follows. Section 4.2 introduces the new type system, F_{GT} proposed for λ_{GT}. Section 4.3 extends the system F_{GT} to cover powerful operations based on graph transformation. Section 4.4 discusses the algorithm for the extended F_{GT}. Section 4.5 describes related work. Section 4.6 describes related work.

4.2 Type System

In this subsection, we propose a type system, F_{GT}, for the λ_{GT} language. We define the type of graphs using graph grammar. This can be regarded as an extension of regular

Atom Name for Types

τ	::=	α	Type Variable
	|	$\tau\,(\overrightarrow{X}) \to \tau\,(\overrightarrow{X})$	Arrow

RHS of Production Rules

$\mathcal{T}$	::=	$\tau\,(\overrightarrow{X})$	Type Atom
	|	$C(\overrightarrow{X})$	Constructor Atom
	|	$X \bowtie Y$	Fusion
	|	$(\mathcal{T}, \mathcal{T})$	Molecule
	|	$\nu X.\mathcal{T}$	Hyperlink Creation

Production Rule

r	::=	$\alpha(\overrightarrow{X}) \longrightarrow \mathcal{T}$	Production Rule

Figure 4.1: Syntax of F_{GT}

tree grammar, on which algebraic data types are based.

4.2 Syntax and Rules for F_{GT}

Let α be a syntactic category denoting the identifier of a type name. The syntax of types is given in Figure 4.1. It can be observed that the definition of a type employs both inductive definition (standard in programming languages) and production rules (standard in formal grammar). The reason for doing so is that, unlike ADTs, types of graphs cannot be defined inductively in general. Thus we employed generative grammar as a well-established formalism for defining graphs. Integrating it into F_{GT} is the research question of the present work.

We extend the λ-expression $\lambda\, x[\overrightarrow{X}].e$ with type annotation $\tau\,(\overrightarrow{X})$ as $\lambda\, x[\overrightarrow{X}] : \tau\,(\overrightarrow{X}).e$.

Definition 4.2.1 (Abbreviation rule for an arrow atom)**.** We introduce a shorthand notation similar to an arrow in the typed λ-calculus, that is,

$$(\tau_1(\overrightarrow{X}) \to (\tau_2(\overrightarrow{Y}) \to \tau_3(\overrightarrow{Z}))(\overrightarrow{W}))(\overrightarrow{W})$$

can be abbreviated as

$$(\tau_1(\overrightarrow{X}) \to \tau_2(\overrightarrow{Y}) \to \tau_3(\overrightarrow{Z}))(\overrightarrow{W}).$$

Definition 4.2.2 (Syntactic constraints)**.** A production rule $\alpha(\overrightarrow{X}) \longrightarrow \mathcal{T}$ should satisfy $fn(\mathcal{T}) = \{\overrightarrow{X}\}$.

Let Γ be a *typing context* which is a set of the form $x\,[\overrightarrow{X}] : \tau\,(\overrightarrow{X})$, where the x's are mutually distinct and t should be a type variable or an arrow. The typing relation $(\Gamma, P) \vdash e : \tau\,(\overrightarrow{X})$ denotes that e has the type $\tau\,(\overrightarrow{X})$ under the type environment Γ and a set P of production rules, whose typing rules are defined as follows.

Definition 4.2.3 (Rules for F_{GT})**.** Typing rules for F_{GT} is given in Figure 4.2.

Ty-App, Ty-Arrow, and Ty-Var are essentially the same as other functional languages except that the type of F_{GT} is written as an atom with free links. Ty-Var gets the type of the variable from the type environment. Ty-Cong incorporates the structural congruence rules. Ty-Alpha α-converts the free link names of both the graph and its type. This rule corresponds to the fact that the free link names in the rules of HyperLMNtal are (theoretically) α-convertible. Ty-Prod incorporates production rules to the type system. Ty-Case is also defined in the same manner as in other functional languages, where Γ' is a type environment that maps types from all the graph contexts appearing in T, which we will describe in detail in Section 4.2.4.

4.2 Examples

In this subsection, we introduce some of the production rules, which we believe describes many of the types of the data structures for programming in practice.

Example 4.2.1 (Type of a natural number)**.** *The type of a natural number connected to a free link X can be denoted as $nat\,(X)$, where the production rules are follows.*

$$\begin{aligned} nat\,(X) &\longrightarrow \mathrm{Zero}(X) \\ nat\,(X) &\longrightarrow \mathrm{Succ}(nat, X) \end{aligned}$$

(Recall that the RHS of the latter rule is a shorthand of $\nu N.\mathrm{Succ}(N, X), nat\,(N)$ (Def. 2.2.2).)

Algebraic data types (ADTs) can be easily expressed in the same way as in this example: our language and the type system is a natural extension of functional languages and their type systems.

Example 4.2.2 (Type of a difference list)**.** *The λ_{GT} language can handle some data structures that algebraic data types cannot handle. A difference list can be understood as a list with an additional link to the last element. This is a popular data structure since the early days of logic programming in which the links are represented as logical variables. It allows us to append two lists in constant time. In functional programming, a difference list can be implemented using a higher-order function that receives a subsequent list and returns the entire list, but we wish to represent such data structures in the first-order setting.*

The production rules for a difference list can be defined as follows.

$$\begin{aligned} nodes\,(Y, X) &\longrightarrow X \bowtie Y \\ nodes\,(Y, X) &\longrightarrow \mathrm{Cons}(nat, nodes\,(Y), X) \end{aligned}$$

$$\frac{(\Gamma, P) \vdash e_1 : (\tau_1(\overrightarrow{X}) \to \tau_2(\overrightarrow{Y}))(\overrightarrow{Z}) \qquad (\Gamma, P) \vdash e_2 : \tau_1(\overrightarrow{X})}{(\Gamma, P) \vdash (e_1\ e_2) : \tau_2(\overrightarrow{Y})}\ \text{Ty-App}$$

$$\frac{((\Gamma, x[\overrightarrow{X}] : \tau_1(\overrightarrow{X})), P) \vdash e : \tau_2(\overrightarrow{Y})}{(\Gamma, P) \vdash (\lambda\, x[\overrightarrow{X}] : \tau_1(\overrightarrow{X}).e)(\overrightarrow{Z}) : (\tau_1(\overrightarrow{X}) \to \tau_2(\overrightarrow{Y}))(\overrightarrow{Z})}\ \text{Ty-Arrow}$$

$$\frac{}{(\Gamma\{x[\overrightarrow{X}] : \tau\,(\overrightarrow{X})\}, P) \vdash x[\overrightarrow{X}] : \tau\,(\overrightarrow{X})}\ \text{Ty-Var}$$

$$\frac{(\Gamma, P) \vdash T : \tau\,(\overrightarrow{X}) \qquad T \equiv T'}{(\Gamma, P) \vdash T' : \tau\,(\overrightarrow{X})}\ \text{Ty-Cong}$$

$$\frac{(\Gamma, P) \vdash T : \tau\,(\overrightarrow{X})}{(\Gamma, P) \vdash T\langle Z/Y\rangle : \tau\,(\overrightarrow{X})\langle Z/Y\rangle}\ \text{Ty-Alpha}$$

where $Z \notin \mathit{fn}(T)$

$$\frac{(\Gamma, P) \vdash T_1 : \tau_1(\overrightarrow{X_1}) \quad \ldots \quad (\Gamma, P) \vdash T_n : \tau_n(\overrightarrow{X_n})}{(\Gamma, P\{\alpha(\overrightarrow{X}) \longrightarrow \mathcal{T}\}) \vdash \mathcal{T}[T_1/\tau_1(\overrightarrow{X_1}), \ldots, T_n/\tau_n(\overrightarrow{X_n})] : \alpha(\overrightarrow{X})}\ \text{Ty-Prod}$$

where $\tau_i(\overrightarrow{X_i})$ are all the type atoms appearing in $\mathcal{T}$

$$\frac{(\Gamma, P) \vdash e_1 : \tau_1(\overrightarrow{X})\ ((\Gamma, \Gamma'), P) \vdash e_2 : \tau_2(\overrightarrow{Y})\ (\Gamma, P) \vdash e_3 : \tau_2(\overrightarrow{Y})}{(\Gamma, P) \vdash (\textbf{case}\ e_1\ \textbf{of}\ T \to e_2 \mid \textbf{otherwise} \to e_3) : \tau_2(\overrightarrow{Y})}\ \text{Ty-Case}$$

Figure 4.2: Typing rules for F_{GT}

Example 4.2.3 (Typing a difference list with functions)**.** *Since λ_{GT} and its type system F_{GT} treat functions as first-class citizens, it is even possible to have a difference list with functions as its elements. Figure 4.3 shows that the graph $G = \mathrm{Cons}(succ, Y, X)$ has*

$$\dfrac{\dfrac{}{(\Gamma\{succ[Z_1] : (nat\,(X) \to nat\,(X))(Z_1)\}, P) \vdash succ[Z_1] : (nat\,(X) \to nat\,(X))(Z_1)}\ \text{Ty-Var} \quad \dfrac{\dfrac{}{(\Gamma, P\{P_1\}) \vdash X \bowtie Y : nodes\,(Y, X)}\ \text{Ty-Prod}}{(\Gamma, P) \vdash Z_2 \bowtie Y : nodes\,(Z_2, X)}\ \text{Ty-Alpha}}{\dfrac{(\Gamma, P\{P_2\}) \vdash G' : nodes\,(Y, X) \quad \text{where} \quad G' = \nu Z_1 Z_2.(\mathrm{Cons}(Z_1, Z_2, X), succ[Z_1], Z_2 \bowtie Y),\ G \equiv G'}{(\Gamma, P) \vdash G : nodes\,(Y, X) \quad \text{where} \quad G = \mathrm{Cons}(succ, Y, X)}\ \text{Ty-Cong}}\ \text{Ty-Prod}$$

Figure 4.3: Type checking a difference list

type $nodes\,(Y, X)$ *under type environment* $\Gamma = succ[Z_1] : (nat\,(X) \to nat\,(X))(Z_1)$ *and production rules* $P = \{P_1, P_2\}$ *where*

$$\begin{array}{lcll} nodes\,(Y, X) & \longrightarrow & X \bowtie Y & \cdots\ P_1 \\ nodes\,(Y, X) & \longrightarrow & \mathrm{Cons}(nat\,(X) \to nat\,(X), nodes\,(Y), X) & \cdots\ P_2 \end{array}$$

Example 4.2.4 (Type of a doubly-linked difference list)**.** *A doubly-linked difference list is a list with four free links, two different links for each end. Although the (hyper)links of* λ_{GT} *and HyperLMNtal are undirected, we are interested in using them to model directed hyperlinks (roughly corresponding to pointers in imperative languages) that are to be 'followed' in one direction. As with difference lists, the addition of elements to the tail of the list can be done in constant time, as desired in representing deques. Of course, doubly-linked lists that are not difference lists can also be handled in an obvious way.*

$$\begin{array}{lcl} nodes\,(F', B, B', F) & \longrightarrow & F \bowtie B, B' \bowtie F' \\ nodes\,(F', B, B', F) & \longrightarrow & \nu X.\mathrm{Cons}(nat, F', nodes\,(X, B, B'), X, F) \end{array}$$

Example 4.2.5 (Type of difference skip lists)**.** *By extending the type definition of difference lists, the type of unbounded-level skip lists can be defined. This implies that we can also define a type for skip lists with a nil node at the end and/or whose level is fixed.*

$$\begin{array}{lcl} nodes\,(Y, X) & \longrightarrow & X \bowtie Y \\ nodes\,(Y, X) & \longrightarrow & \mathrm{Cons}(nat, forks\,(Y), X) \\ forks\,(Y, X) & \longrightarrow & \mathrm{Next}(nodes\,(Y), X) \\ forks\,(Y, X) & \longrightarrow & \nu Z.\mathrm{Fork}(Z, forks\,(Z), X), nodes\,(Y, Z) \end{array}$$

We also show the visualized version of production rules in Figure 4.4 and an example difference skip list in Figure 4.5.

Example 4.2.6 (Type of a leaf-linked tree)**.** *A leaf-linked tree is a graph with three free links (say* X, L, R*) which is a tree whose root is represented by* X *and whose leaves form a difference list represented by* L *and* R*.*

$$\begin{array}{lcl} lltree\,(L, R, X) & \longrightarrow & L \bowtie X, \mathrm{Leaf}(nat, R, X) \\ lltree\,(L, R, X) & \longrightarrow & \nu Y.\mathrm{Node}(lltree\,(L, Y), lltree\,(Y, R), X) \end{array}$$

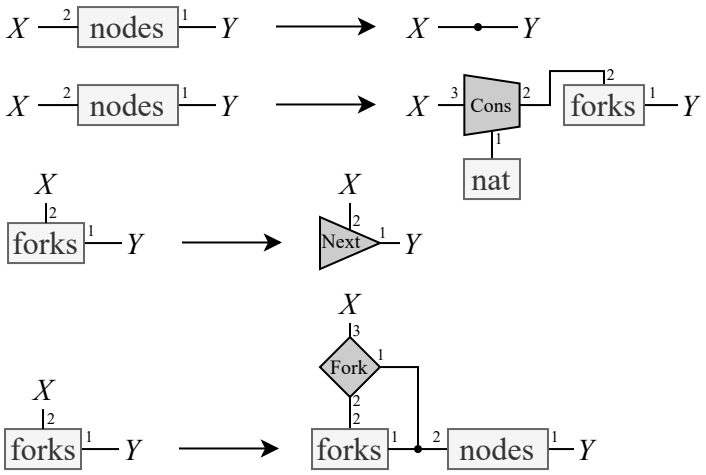


Figure 4.4: The production rules for the type of difference skip list

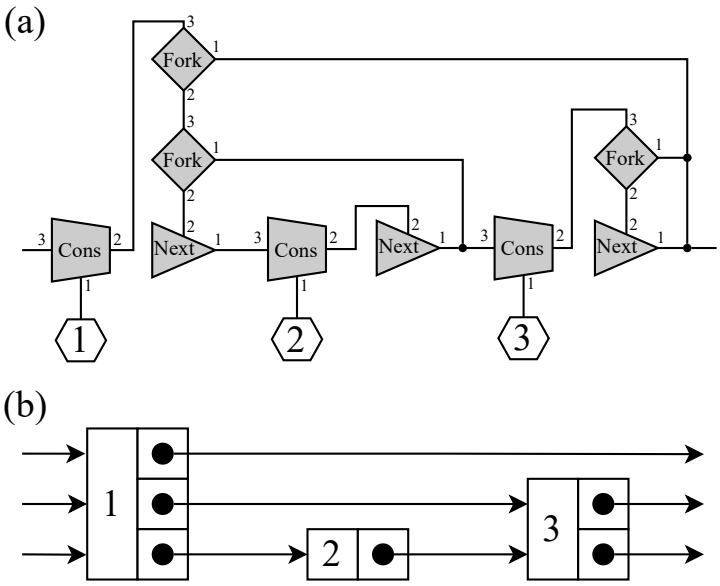


Figure 4.5: An example of a skip list:(a) in our framework, (b) with pointers

Example 4.2.7 (Type of a threaded tree)**.** *A threaded tree is somewhat similar to a leaf-linked tree but each non-terminal node has access to the the rightmost leaf the left subtree and the leftmost leaf of the right subtree.*

$$\begin{aligned} \mathit{thtree}\,(L, R, X) &\longrightarrow L \bowtie X, \mathrm{Leaf}(\mathit{nat}, R, X) \\ \mathit{thtree}\,(L, R, X) &\longrightarrow \mathrm{Node}(\mathit{thtree}\,(L, X), \mathit{thtree}\,(X, R), X) \end{aligned}$$

4.2 Properties of F_{GT}

This section discusses some properties of λ_{GT} and F_{GT}. As mentioned in Section 3.3, we keep the language small to focus on the handling of graph structures, more specifically the handling of graphs by pattern matching with graph contexts. In particular, it has no explicit mechanism (such as **let rec** or **fix**) to deal with recursive functions. This is

because those features can be achieved essentially in the same way as other functional languages do.

Soundness of F_{GT}

Lemma 4.2.1 (Progress)**.** *If* $(\varnothing, P) \vdash e : \tau\,(\overrightarrow{X})$*, then* e *is a value or* $\exists e'.e \longrightarrow_{\texttt{val}} e'$*.*

Proof. By induction on the derivation of $(\varnothing, P) \vdash e : \tau\,(\overrightarrow{X})$. Notice that the only new extension from other functional languages in expressions (Figure 3.8) is Case, and the Case expression is never stuck because if matching fails; it just branches to **otherwise** and evaluation proceeds. □

Lemma 4.2.2 (Substitution)**.** *If* $(\Gamma, P) \vdash e_1 : \tau_1(\overrightarrow{Y_1})$ *and* $((\Gamma, x[\overrightarrow{X_1}] : \tau_1(\overrightarrow{Y_1})), P) \vdash e_2 : \tau_2(\overrightarrow{Y_2})$ *then* $(\Gamma, P) \vdash e_2[e_1/x[\overrightarrow{X_1}]] : \tau_2(\overrightarrow{Y_2})$

Proof. By induction on the derivation of $((\Gamma, x[\overrightarrow{X_1}] : \tau_1(\overrightarrow{Y_1})), P) \vdash e_2 : \tau_2(\overrightarrow{Y_2})$. □

Lemma 4.2.3 (Preservation)**.** *If* $(\Gamma, P) \vdash e : \tau\,(\overrightarrow{X})$ *and* $e \longrightarrow_{\texttt{val}} e'$*, then* $(\Gamma, P) \vdash e' : \tau\,(\overrightarrow{X})$*.*

Proof. Proved using the Lemma 4.2.2. □

Theorem 4.2.1 (Soundness)**.** *If* $(\varnothing, P) \vdash e : \tau\,(\overrightarrow{X})$*, and* $e \longrightarrow^*_{\texttt{val}} e'$ *then* e' *is a value or* $\exists e''.e' \longrightarrow_{\texttt{val}} e''$*.*

Proof. Follows from Lemma 4.2.1 and Lemma 4.2.3. □

Relation with Graph Reduction

Structured Gamma [FM98] is a first-order graph rewriting system developed to represent and reason about the shapes of pointer data structures. The framework of Structured Gamma was then adapted to LMNtal (whose graph structures are dual to those of Structured Gamma, roughly speaking) to design and implement LMNtal ShapeType [YU21]. Despite several syntactic variations (such as the duality of nodes/links and the presence/absence of hyperlinks), Structured Gamma and LMNtal ShapeType can (essentially) handle graphs of λ_{GT} without λ-abstraction atoms. The typing relation à la Structured Gamma and LMNtal ShapeType is defined as follows.

Definition 4.2.4 (Typing relation in Structured Gamma/LMNtal ShapeType)**.** $P \vdash \tau : \alpha(\overrightarrow{X})$ iff $\alpha(\overrightarrow{X}) \rightsquigarrow^*_P \tau$ and τ does not contain type variables or arrow atoms

We have shown that the typing relation in our type system F_{GT} subsumes the one in Structured Gamma in the following sense.

Theorem 4.2.2 (F_{GT} and HyperLMNtal reduction)**.**

$$(\Gamma, P) \vdash T : \tau\,(\overrightarrow{X}) \Leftrightarrow \tau\,(\overrightarrow{X}) \rightsquigarrow_P^* T[\overrightarrow{\tau_i(\overrightarrow{Y_i})/x_i[\overrightarrow{X_i}]}^{i}\ \overrightarrow{\tau_i(\overrightarrow{Z_i})/(\lambda\ldots)_i(\overrightarrow{W_i})}^{i}]$$

where

- $\Gamma = \overrightarrow{x_i[\overrightarrow{X_i}] : \tau_i(\overrightarrow{X_i})}^{i}$,
- $(\lambda\ldots)_i(\overrightarrow{W_i})$ *are all the* λ*-abstraction atoms in* T*, and*
- $(\Gamma, P) \vdash (\lambda\ldots)_i(\overrightarrow{W_i}) : \tau_i(\overrightarrow{Z_i})$

Proof. For $\Rightarrow$, we can prove by induction on the last applied F_{GT} rules. For $\Leftarrow$, We prove by induction on the length of the reduction $\rightsquigarrow_P^*$. □

Note that if no graph contexts or λ-expressions appear in T, by Theorem 4.2.2, the typing relation in F_{GT} is equivalent to the one in Structured Gamma. In other words, our type system is an extension of Structured Gamma to allow graph contexts and λ-abstraction atoms. This allows us to take advantage of research results on Structured Gamma, its derivative LMNtal ShapeType, and parsing of graphs using graph grammar.

Example 4.2.8 (Theorem 4.2.2 on the difference list example)**.** *Here, we see that Theorem 4.2.2 holds on Example 4.2.3. Recall that* $(succ[Z_1] : (nat\,(X) \to nat\,(X))(Z_1), P) \vdash \mathrm{Cons}(succ, Y, X) : nodes\,(Y, X)$ *holds in* F_{GT}*, which can also be shown using HyperLMNtal reduction as follows.*

$$\begin{aligned}
&nodes\,(Y, X)\\
&\rightsquigarrow_{P_2} \nu Z_1 Z_2.(\mathrm{Cons}(Z_1, Z_2, X), (nat\,(X) \to nat\,(X))(Z_1), nodes\,(Y, Z_2))\\
&\rightsquigarrow_{P_1} \nu Z_1 Z_2.(\mathrm{Cons}(Z_1, Z_2, X), (nat\,(X) \to nat\,(X))(Z_1), Z_2 \bowtie Y)\\
&\equiv \nu Z_1.(\mathrm{Cons}(Z_1, Y, X), (nat\,(X) \to nat\,(X))(Z_1))\\
&= \mathrm{Cons}(succ, Y, X)[(nat\,(X) \to nat\,(X))(Z_1)/succ[Z_1]]
\end{aligned}$$

4.2 Type Checking Case Expressions

λ_{GT} allows pattern matching of graphs. In pattern matching, graph contexts can be used as wildcards. Since a graph context can match any graph as long as the sets of free links are the same, we cannot naively ensure that the type of the graph matches the intended type of the context. Therefore, we allow the typing annotation of graph contexts.

To allow type annotation in pattern matching, we extend the syntax of the graph template T. A type annotation $T : \tau\,(\overrightarrow{X})$ ensures that the type of the graph matched with T is of type $\tau\,(\overrightarrow{X})$.

To evaluate pattern matching with annotations, we extend the matching mechanism. $\mathrm{Match}(G, T, \overrightarrow{\theta})$ denotes that (i) the graph context T can match the graph G with graph

substitutions $\overrightarrow{\theta}$ and that (ii) each subgraph of G matched by a subcontext of T satisfies the type constraint attached to the subcontext. Match$(G, T, \overrightarrow{\theta})$ is defined inductively as in Figure 4.6. It is a straightforward inductive argument to see that Figure 4.6 extends the matching defined in Section 3.3.2 with the rule Mt-Ty for type checking.

The type annotations that do not match the type definitions of production rules could be reported as bugs, which could be analyzed easily and statically. Also, for simplicity, henceforth we will assume that all the graph contexts are type-annotated and make it a future task to support unannotated graph contexts.

The matching can be non-deterministic; that is, given a graph and a pattern, there may in general be more than one way in which graph contexts in the pattern are bound to subgraphs. However, the non-determinacy of the matching does not affect the soundness of the type system since the system proves that every execution path is type-safe.

The program that pops the last element of a difference list we have introduced in Section 3.3.2 can be handled with type-annotations as follows.

$$
\begin{array}{l}
(\Gamma, P) \vdash \\
\quad (\lambda\, x[Y, X] : \mathit{nodes}\,(Y, X). \\
\qquad \textbf{case}\ x[Y, X]\ \textbf{of} \\
\qquad\quad \nu Z_1.Z_2.(y[Z_1, X] : \mathit{nodes}\,(Z_1, X), \\
\qquad\qquad\qquad \mathrm{Cons}(Z_2, Y, Z_1), \\
\qquad\qquad\qquad z[Z_2] : \mathit{nat}\,(Z_2)) \\
\qquad\qquad \to y[Y, X] \\
\qquad\quad \mid \textbf{otherwise} \to x[Y, X] \\
\quad)(Z)
\end{array}
\quad : \quad
\begin{array}{l}
(\mathit{nodes}\,(Y, X) \to \\
\quad \mathit{nodes}\,(Y, X) \\
)(Z)
\end{array}
$$

This can be typed using Ty-Case where the Γ' stands for the annotated typing relations $y[Z_1, X] : \mathit{nodes}\,(Z_1, X), z[Z_2] : \mathit{nat}\,(Z_2)$.

4.3 Extending the Type System

In this subsection, we deal with an example which the type system in Section 4.2 fails to verify. The type system in Section 4.2 was actually for *parsing* when dealing with graphs; it just checks if the graph can be generated from the annotated type variable atom, i.e., the start symbol. Algebraic data types can be handled in this manner because they can only be generated according to the grammar that defines the type. However, in the case of graphs, more powerful operations are possible, for example the concatenation of difference lists. In this subsection, we propose an extended verification framework to deal with such cases.

$$\frac{fn(G) = \{\overrightarrow{X}\}}{\text{Match}(G, x[\overrightarrow{X}], [G/x[\overrightarrow{X}]])}\ \text{Mt-Var}$$

$$\frac{}{\text{Match}(G, G, [])}\ \text{Mt-Triv}$$

$$\frac{\text{Match}(G_1, T_1, \overrightarrow{\theta_1}) \qquad \text{Match}(G_2, T_2, \overrightarrow{\theta_2})}{\text{Match}((G_1, G_1), (T_1, T_2), \overrightarrow{\theta_1}\overrightarrow{\theta_2})}\ \text{Mt-Mol}$$

where $\text{dom}(\overrightarrow{\theta_1}) \cap \text{dom}(\overrightarrow{\theta_2}) = \varnothing$

$$\frac{\text{Match}(G, T, \overrightarrow{\theta})}{\text{Match}(\nu X.G, \nu X.T, \overrightarrow{\theta})}\ \text{Mt-}\nu$$

$$\frac{\text{Match}(G_1, T, \overrightarrow{\theta}) \qquad G_1 \equiv G_2}{\text{Match}(G_2, T, \overrightarrow{\theta})}\ \text{Mt-Cong}$$

$$\frac{\text{Match}(G, T, \overrightarrow{\theta}) \qquad G : \tau\,(\overrightarrow{X})}{\text{Match}(G, (T : \tau\,(\overrightarrow{X})), \overrightarrow{\theta})}\ \text{Mt-Ty}$$

Figure 4.6: Matching with a template and graph substitutions

4.3 Motivation

As a running example, we consider a typed version of the following program for appending two difference lists introduced in Section 3.3.2.

$$\begin{array}{l}(\lambda\, x[Y, X] : \mathit{nodes}\,(Y, X)\\ \quad y[Y, X] : \mathit{nodes}\,(Y, X).\\ \quad\quad x[y[Y], X]\\ )(Z)\end{array}$$

It seems natural that the following typing relation holds, where $\mathit{append}[Z]$ is the λ-

abstraction atom above.

$$(\Gamma, P) \vdash \\ \quad append[Z] : (nodes\,(Y, X) \to nodes\,(Y, X) \to nodes\,(Y, X))(Z)$$

However, this program cannot be verified by directly using the rules in the type system in Section 4.2.

Theorem 4.3.1. *The append operation on difference lists fails to verify on the previously defined F_{GT}.*

Proof. We need to prove

$$((x[Y, X] : nodes\,(Y, X), y[Y, X] : nodes\,(Y, X)), P) \vdash \\ \quad x[y[Y], X] : nodes\,(Y, X)$$

to verify the present example. Theorem 4.2.2 states that, if we can successfully prove the typing relation using F_{GT}, we should be able to prove $nodes\,(Y, X) \rightsquigarrow_P^* nodes\,(nodes\,(Y), X)$. However, applying the production rules of difference lists cannot increase the number of $nodes/2$ atoms. Therefore, applying the production rules to the annotated type variable atom $nodes\,(Y, X)$ will never yield $nodes\,(nodes\,(Y), X)$. □

However, it is obvious that appending two difference lists returns a difference list, and this operation should be supported. We extend the previously defined F_{GT} to enable such verification.

4.3 Extension on F_{GT}

We start with the first attempt of the extension.

Definition 4.3.1 (Extension on F_{GT} (Unrefined))**.** For a graph template T, it is sufficient if the typing succeeds after replacing each graph context in T by all possible values of the types attached to the graph context, or more formally, as in Figure 4.7.

The (apparently intuitive) rule in Definition 4.3.1 has $\Rightarrow$ on the antecedent and the typing relation we are going to define (the parameter of the generating function) appears on the left-hand side of the $\Rightarrow$. Unfortunately, then, we cannot ensure the monotonicity of the generating function and the existence of a least fixed point, which is the typing relation we want to define.

Now we consider how to fix this, which we have found is not trivial or straightforward. If we define a typing relation, say R_0, without Ty-Subst and define the left-hand side of the $\Rightarrow$ of Ty-Subst with R_0, we can ensure the monotonicity of the generating function and the typing relation becomes well-defined.

First, we prepare two sets of typing rules, one with all the :'s in Figure 4.2 rewritten as $:_0$ and the other as $:_1$. Then, we can define R_0 only with typing rules with $:_0$, which is well-defined.

$$\frac{\forall \overrightarrow{G_i}.\left(\left(\bigwedge^{i}\left((\varnothing, P) \vdash G_i : \tau_i(\overrightarrow{X_i})\right)\right) \Rightarrow (\varnothing, P) \vdash T\,\overrightarrow{\left[G_i/x_i[\overrightarrow{X_i}]\right]}^{i} : \tau\,(\overrightarrow{X})\right)}{\left(\overrightarrow{x_i[\overrightarrow{X_i}] : \tau_i(\overrightarrow{X_i})}^{i}, P\right) \vdash T : \tau\,(\overrightarrow{X})}\ \text{Ty-Subst (unrefined)}$$

$$\text{where } \overrightarrow{x_i[\overrightarrow{X_i}]}^{i} = \mathit{ff}(T)$$

Figure 4.7: Extension on F_{GT} (unrefined)

$$\frac{\forall \overrightarrow{G_i}.\left(\left(\bigwedge^{i}\left((\varnothing, P) \vdash G_i :_0 \tau_i(\overrightarrow{X_i})\right)\right) \Rightarrow (\varnothing, P) \vdash T\,\overrightarrow{\left[G_i/x_i[\overrightarrow{X_i}]\right]}^{i} :_1 \tau\,(\overrightarrow{X})\right)}{\left(\overrightarrow{x_i[\overrightarrow{X_i}] :_1 \tau_i(\overrightarrow{X_i})}^{i}, P\right) \vdash T :_1 \tau\,(\overrightarrow{X})}\ \text{Ty-Subst (unrefined 2)}$$

$$\text{where } \overrightarrow{x_i[\overrightarrow{X_i}]}^{i} = \mathit{ff}(T)$$

Figure 4.8: Extension on F_{GT} (unrefined 2)

Next, we define the typing relation, say R_1, using typing rule with $:_1$ and the rule in Figure 4.8 (Ty-Subst with $:_0$ and $:_1$). Since the left-hand side of $\Rightarrow$ in Figure 4.8 uses the already defined R_0, it can be interpreted as a monotonic function and the typing relation R_1 is well-defined.

However, we cannot ensure the soundness if we define Ty-Subst in such a way because the antecedent of the rule ensures the safety if G_i has a type $\tau_i(\overrightarrow{X_i})$ in R_0 but the antecedent does not ensure the safety when G_i has a type $\tau_i(\overrightarrow{X_i})$ only in R_1. Since R_1 can handle more programs than R_0, this may violate the soundness of the system.

For example, consider the case where G_i is a graph containing a function that concatenates difference lists and $\tau_i(\overrightarrow{X_i})$ contains an arrow type for a function that takes two difference lists (as curried arguments) and return a difference list. Since it is not verifiable in R_0 that a function that concatenates difference lists returns a difference list, we can make the left-hand side of $\Rightarrow$ on Ty-Subst false. In such a case, the antecedent of Ty-Subst is satisfied no matter what the right-hand side of $\Rightarrow$ is. Thus, we cannot ensure safety for

the case where we bound a graph containing a function that concatenates difference lists. However, since the consequent of Ty-Subst uses $:_1$, it allows the $x_i[\overrightarrow{X_i}]$ to be bound to a graph that includes a function that concatenates difference lists.

Therefore, we need a more refined framework that allows the indices we have attached to : previously to be different for each type. Accordingly, we introduce the notion of *ranks* for types and Ty-Subst.

If the typing on the left-hand side of $\Rightarrow$ does not use Ty-Subst, which we will define, it can be interpreted as a monotonic function and is well-defined. Therefore, we introduce ranks into Ty-Subst so that the left-hand side typing of $\Rightarrow$ can only use Ty-Subst with a lower rank that has already been defined.

We first introduce ranks to the type. We denote the type with Rank n ($n \geq 0$) as $\tau^n(\overrightarrow{X})$. We extend the rules in Figure 4.2 so that the types have ranks.

Definition 4.3.2 (Rules for F_{GT} with ranks)**.** Typing rules for F_{GT} with ranks are given in Figure 4.9. Notice that we have added a new typing rule Ty-Sub, a rule for subtyping.

Definition 4.3.3 (Extension on F_{GT} (Refined))**.** The refined version of Definition 4.3.1 is shown in Figure 4.10.

Proposition 4.3.1. *The typing rules are well-defined even if we add Definition 4.3.3.*

Proof. In Definition 4.3.3, the ranks of the type on the left-hand side of $\Rightarrow$, n_i, are always smaller than the rank of the type $\tau^n(\overrightarrow{X})$ on the consequent of the rule, n. The typing rules in Figure 4.9 are defined so that the ranks do not increase when we read the rules upwards. Thus, the typing relation used for the left-hand side of $\Rightarrow$ in the antecedent of Ty-Subst (of rank n) can be established using Ty-Subst with smaller ranks m ($m < n$) only (which may actually be used when, for example, the G_i's contain abstraction atoms). Suppose all typing relations involving smaller ranks are well-defined. Then, since the typing relation we are about to define does not appear in the left-hand side of $\Rightarrow$, we can ensure the well-definedness of the typing relation involving ranks up to n. Because the typing relation containing types with rank 0 only does not involve Ty-Subst and is therefore well-defined, by mathematical induction on rank, we can define a typing relation for all ranks. □

The existence of the typing rule defined in Definition 4.3.3 does not violate the soundness since using the rule ensures that a program can be typed without such a rule for all the possible graphs bound to graph contexts.

Let us consider the typing of a function that concatenates difference lists. From now on, we omit the "$(\varnothing, P) \vdash$" for brevity. Suppose we have already proven the following (we will prove this in Section 4.3.3).

$$\begin{aligned}&\forall G_1, G_2.\Big(G_1 : nodes^n(Y,X) \wedge G_2 : nodes^m(Y,X)\\ &\Rightarrow \nu Z.(x[Z,X], y[Y,Z])\left[G_1/x[Y,X]\right]\left[G_2/x[Y,X]\right]\\ &\quad : nodes^{\max(n,m)}(Y,X)\Big)\end{aligned} \tag{4.1}$$

$$\frac{(\Gamma, P) \vdash e_1 : (\tau_1{}^n(\overrightarrow{X}) \to \tau_2{}^m(\overrightarrow{Y}))^m(\overrightarrow{Z}) \qquad (\Gamma, P) \vdash e_2 : \tau_1{}^n(\overrightarrow{X})}{(\Gamma, P) \vdash (e_1\ e_2) : \tau_2{}^m(\overrightarrow{Y})}\ \text{Ty-App}$$

$$\frac{((\Gamma, x[\overrightarrow{X}] : \tau_1{}^n(\overrightarrow{X})), P) \vdash e : \tau_2{}^m(\overrightarrow{Y}) \qquad n \leq m}{(\Gamma, P) \vdash (\lambda\, x[\overrightarrow{X}] : \tau_1{}^n(\overrightarrow{X}).e)(\overrightarrow{Z}) : (\tau_1{}^n(\overrightarrow{X}) \to \tau_2{}^m(\overrightarrow{Y}))^m(\overrightarrow{Z})}\ \text{Ty-Arrow}$$

$$\frac{}{(\Gamma\{x[\overrightarrow{X}] : \tau^n\,(\overrightarrow{X})\}, P) \vdash x[\overrightarrow{X}] : \tau^n\,(\overrightarrow{X})}\ \text{Ty-Var}$$

$$\frac{(\Gamma, P) \vdash T : \tau^n\,(\overrightarrow{X}) \qquad T \equiv T'}{(\Gamma, P) \vdash T' : \tau^n(\overrightarrow{X})}\ \text{Ty-Cong}$$

$$\frac{(\Gamma, P) \vdash T : \tau^n(\overrightarrow{X})}{(\Gamma, P) \vdash T\langle Z/Y\rangle : \tau^n(\overrightarrow{X})\langle Z/Y\rangle}\ \text{Ty-Alpha}$$

where $Z \notin fn(T)$

$$\frac{(\Gamma, P) \vdash T_1 : \tau_1{}^{n_1}(\overrightarrow{X_1}) \quad \ldots \quad (\Gamma, P) \vdash T_n : \tau_m{}^{n_m}(\overrightarrow{X_m})}{(\Gamma, P\{\alpha(\overrightarrow{X}) \longrightarrow \mathcal{T}\}) \vdash \mathcal{T}[T_1/\tau_1(\overrightarrow{X_1}), \ldots, T_m/\tau_m(\overrightarrow{X_m})] : \alpha^{\max n_i}(\overrightarrow{X})}\ \text{Ty-Prod}$$

where $\tau_i(\overrightarrow{X_i})$ are all the type atoms appearing in $\mathcal{T}$

$$\frac{\begin{array}{c}((\Gamma, \Gamma'), P) \vdash e_2 : \tau_2{}^m(\overrightarrow{Y}) \\ (\Gamma, P) \vdash e_1 : \tau_1{}^n(\overrightarrow{X}) \qquad (\Gamma, P) \vdash e_3 : \tau_2{}^m(\overrightarrow{Y}) \quad n \leq m\end{array}}{(\Gamma, P) \vdash (\textbf{case}\ e_1\ \textbf{of}\ T \to e_2 \mid \textbf{otherwise} \to e_3) : \tau_2{}^m(\overrightarrow{Y})}\ \text{Ty-Case}$$

$$\frac{(\Gamma, P) \vdash e : \tau^n\,(\overrightarrow{X}) \qquad n < m}{(\Gamma, P) \vdash e : \tau^m(\overrightarrow{X})}\ \text{Ty-Cong}$$

Figure 4.9: Typing rules for F_{GT} with ranks

Then, we can type the function using Ty-Sub, Ty-Subst (rank $\max(n, m) + 1$), and Ty-Arrow as shown in Figure 4.11.

$$\frac{\forall \overrightarrow{G_i}.\left(\left(\bigwedge^{i}\left((\varnothing, P) \vdash G_i : {\tau_i}^{n_i}(\overrightarrow{X_i})\right)\right) \Rightarrow (\varnothing, P) \vdash T\overrightarrow{\left[G_i/x_i[\overrightarrow{X_i}]\right]}^{i} : \tau^n\,(\overrightarrow{X})\right)}{\left(\overrightarrow{x_i[\overrightarrow{X_i}] : {\tau_i}^{n_i}(\overrightarrow{X_i})}^{i}, P\right) \vdash T : \tau^n\,(\overrightarrow{X})}\ \text{Ty-Subst (rank } n\text{)}$$

where $n = \max n_i + 1$, $\overrightarrow{x_i[\overrightarrow{X_i}]}^{i} = \mathit{ff}(T)$.

Figure 4.10: Extension on F_{GT} (refined)

Since the type system is monomorphic, we cannot type the following program.

$$\begin{aligned}
&((\ f \quad : (\mathit{nodes}^n(Y,X) \to \quad ,\\
&\qquad\qquad (\mathit{nodes}^m(Y,X) \to\\
&\qquad\qquad\quad \mathit{nodes}^{\max(n,m)+1}(Y,X)\\
&\qquad\qquad)^{\max(n,m)+1}(Z)\\
&\qquad\quad)^{\max(n,m)+1}(Z)\\
&\quad x[Y,X]\ :\ \mathit{nodes}^l(Y,X)\\
&),P)\\
&\vdash \quad f\ \ x[Y,X]\ \ (f\ \ x[Y,X]\ \ x[Y,X]) : \tau^k(Y,X)
\end{aligned}$$

We need to satisfy $n = \max(n,m) + 1$, which is unsatisfiable.

Such programs can be typed introducing polymorphism for ranks. However, this paper does not go into this and leaves it as future work.

4.3 Proving the Antecedent of the Rule

In order to apply Definition 4.3.3 to the present example, we need to prove that, for any graphs to which $x[Y,X]$ and $y[Y,X]$ can be mapped, the substituted result must have the type $\mathit{nodes}^n(Y,X)$, that is,

$$\begin{aligned}
&\forall G_1, G_2.((G_1 : \mathit{nodes}^n(Y,X) \wedge G_2 : \mathit{nodes}^m(Y,X))\\
&\Rightarrow \nu Z.(x[Z,X], y[Y,Z])[G_1/x[Y,X]][G_2/y[Y,X]]\\
&\qquad = \nu Z.(G_1\langle Z/Y\rangle, G_2\langle Z/X\rangle) : \mathit{nodes}^{\max(n,m)}(Y,X)).
\end{aligned}$$

The above can be rewritten using Ty-Alpha as follows.

$$\begin{aligned}
&\forall G_1, G_2.(G_1 : \mathit{nodes}^n(Z,X) \wedge G_2 : \mathit{nodes}^m(Y,Z)\\
&\qquad \Rightarrow \nu Z.(G_1, G_2) : \mathit{nodes}^{\max(n,m)}(Y,X))
\end{aligned} \tag{4.2}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{Equation (4.1)}}\ \text{Ty-Sub}}{\forall G_1, G_2.\Big(G_1 : nodes^n(Y,X) \wedge G_2 : nodes^m(Y,X) \Rightarrow \nu Z.(x[Z,X], y[Y,Z])\big[G_1/x[Y,X]\big]\big[G_2/x[Y,X]\big] : nodes^{\max(n,m)+1}(Y,X)\Big)}\ \text{Ty-Subst}}{((x[Y,X] : nodes^n(Y,X), y[Y,X] : nodes^m(Y,X)), P) \vdash \nu Z.(x[Z,X], y[Y,Z]) : nodes^{\max(n,m)+1}(Y,X)}\ \text{(rank } \max(n,m)+1\text{)}}{(x[Y,X] : nodes^n(Y,X), P) \vdash (\lambda\, y[Y,X] : nodes^m(Y,X).\ \nu Z.(x[Z,X], y[Y,Z])\ )(Z) : (nodes^m(Y,X) \to nodes^{\max(n,m)+1}(Y,X))^{\max(n,m)+1}(Z)}\ \text{Ty-Arrow}$$

$$\dfrac{}{(\varnothing, P) \vdash (\lambda\, x[Y,X] : nodes^n(Y,X).\ (\lambda\, y[Y,X] : nodes^m(Y,X).\ \nu Z.(x[Z,X], y[Y,Z])\ )(Z)\ )(Z) : (nodes^n(Y,X) \to (nodes^m(Y,X) \to nodes^{\max(n,m)+1}(Y,X))^{\max(n,m)+1}(Z))^{\max(n,m)+1}(Z)}\ \text{Ty-Arrow}$$

Figure 4.11: Type checking a function that concatenate difference lists

We prove this by induction on the derivation of the antecedents. To do this, we need a lemma and a theorem.

Lemma 4.3.1. *For $G : \alpha^n(\overrightarrow{X})$, the rule Ty-Prod with $\alpha/|\overrightarrow{X}|$ on its LHS is used in the derivation. Furthermore, only Ty-Cong and Ty-Alpha are used after the last application of Ty-Prod.*

Proof. Suppose we build a proof tree of $G : \alpha^n(\overrightarrow{X})$ bottom-up. Since G is a value, we can only use Ty-Cong, Ty-Alpha, Ty-Subst, Ty-Sub and Ty-Prod until a λ-abstraction atom appears. Ty-Alpha, Ty-Subst, Ty-Sub, and Ty-Cong only inherit the annotated type from the antecedent (although they may changes the rank) so they alone cannot make the annotated type a type variable atom. If Ty-Prod does not appear but a λ-abstraction atom appears and Ty-Arrow is used, the annotated type becomes an arrow and not a type variable. Therefore, there must exist a Ty-Prod whose annotated type has the functor $\alpha/|\overrightarrow{X}|$. □

Theorem 4.3.2. *For $G : \alpha^n(\overrightarrow{Y})$, if the production rule used by last Ty-Prod was $\alpha(\overrightarrow{X}) \longrightarrow \mathcal{T}$, there exists $\overrightarrow{G_j}^j$ such that $G \equiv \mathcal{T}'[\overrightarrow{G_j/\tau_j\,(\overrightarrow{X_j})}^j]$ where*

- $\mathcal{T}' = \mathcal{T}\overrightarrow{\langle Y_i/X_i\rangle}^i$,

- $\tau_j\,(\overrightarrow{X_j})$ *are all the type atoms appearing in* $\mathcal{T}'$,
- $\overrightarrow{G_j : \tau_j{}^{n_j}(\overrightarrow{X_j})}^{j}$, *and*
- $\max n_j = n$.

Proof. By induction on the derivation of $G : \alpha^n(\overrightarrow{Y})$ after the last application of Ty-Prod using Lemma 4.3.1. □

Consider the case if the rule Ty-Prod used last in the derivation of $G_1 : nodes^n(Z, X)$ was the one with the following production rule.

$$nodes(X_2, X_1) \longrightarrow \nu X_3 X_4.(\mathrm{Cons}(X_3, X_4, X_1), nat\,(X_3), nodes\,(X_2, X_4)),$$

Then, by Theorem 4.3.2, we can decompose the graph $G_1 \equiv \nu X_3 X_4.(\mathrm{Cons}(X_3, X_4, X), G_3, G_4)$ into G_3 and G_4, where $G_3 : nat^0(X_3)$ and $G_4 : nodes^n(Z, X_4)$. And we can proceed verification by checking if the target graph has the desirable type for all possible values of G_3 and G_4.

We prove Equation (4.2) by induction on the derivation of $G_1 : nodes^n(Z, X)$. We split the cases based on the rule Ty-Prod used last in the derivation and decompose the graph using Theorem 4.3.2.

For brevity, we denote the graph G of the type $\alpha^n(\overrightarrow{X})$ as $\underline{\alpha}^n(\overrightarrow{X})$ and omit $\forall G$. Then Equation (4.2) can be rewritten as

$$\nu Z.(\underline{nodes}_1^n(Z, X), \underline{nodes}_2^m(Y, Z)) : nodes^{\max(n,m)}(Y, X).$$

The inference rule (or rule *scheme*, precisely speaking) that splits the cases by the last application of Ty-Prod to derive $\underline{\beta}_j^n$ in G is expressed in the following form. Here, G_i is the graph such that the last production rule used in the derivation of $\underline{\beta}_j^n$ is P_i.

$$\frac{G_1 : \alpha^l(\overrightarrow{X}) \quad \ldots \quad G_m : \alpha^l(\overrightarrow{X})}{G : \alpha^l(\overrightarrow{X})}\ \mathrm{Case}\ \underline{\beta}_j^n$$

The concatenation of difference lists can be verified as shown in Figure 4.12, where the arrow $\hookleftarrow$ refers to using the induction hypothesis.

4.4 Automatic Verification on the Extended Type System

In Section 4.3, we typed the program by manually applying structural induction to the target program. In this subsection, we describe a method to do this automatically. From now on, we handle the cases where ranks are all zero and omit them. Extending the algorithm to handle general rank is future work.

We construct a proof tree like what we have shown in Figure 4.12 bottom-up. Given $G : \alpha(\overrightarrow{X})$, we can use the following strategies to verify those programs.

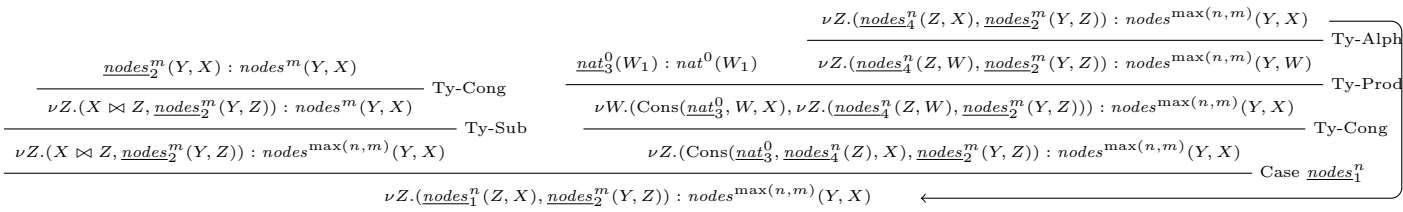


Figure 4.12: Verifying concatenation of difference lists

Ty-Prod: If we get a constructor atom C/n from G, we can check whether an annotated type name atom $\alpha(\overrightarrow{X})$ can be derived from the target graph using Prod with a production rule with $\alpha(\overrightarrow{X})$ on the LHS and C/n on the RHS. However, in order to use a production rule, the subgraphs in G must have the types necessary for the derivation. For this reason, the type checker is performed inductively on the subgraphs.

Case $\underline{\beta}_i$: If we get a type annotated graph $\underline{\beta}_i$ from G, we decompose it using Theorem 4.3.2. Then check if G with its subgraph $\underline{\beta}_i$ thus decomposed has type $\alpha(\overrightarrow{X})$.

$\hookleftarrow$: Induction hypotheses are used when applicable.

However, it is not that easy to do this automatically. Especially for more complex examples.

1. We cannot easily separate a graph into subgraphs when using a production rule. It is difficult to automatically separate and guess the type of a subgraph, prove it as a subproblem, and proceed with the proof using it without any prior preparation.

2. The possibility that links may be *fused* later makes it difficult to get the correspondence of link names in the target graph and the applying production rule.

   Remember the production rules for leaf-linked trees in Example 4.2.6. Here, we want to type check the following graph.

   $$\nu Y.\mathrm{Node}(L, \mathrm{lltree}(Y,R), X), \mathrm{Leaf}(\mathrm{nat}, L, Y) : \mathrm{lltree}(L,R,X)$$

   In this example, we try to apply the second rule

   $$\mathrm{lltree}(L,R,X) \longrightarrow \nu Y.\mathrm{Node}(\mathrm{lltree}(L,Y), \mathrm{lltree}(Y,R), X).$$

   In this rule, the first link of the atom Node/3 is a(n anonymous) local link, say Y_1, but the corresponding link in the target graph is the free link L. Therefore, it is necessary to proceed with the information that Y_1 will be fused to L later, and to check that the fusion occurs before the local link Y_1 leaves the scope. Implementing this becomes complex with more similar cases and is not that easy. In addition, it is not trivial to add the structural induction hypothesis in this process and apply it.

3. A strategy is also needed for the decomposition of annotated contexts. In Figure 4.12, we decomposed $\underline{nodes}_1(Z, X)$. If we decomposed $\underline{nodes}_2(Y, Z)$, we would not get the form to which induction hypothesis can be applied and verification would fail.

Therefore, we restrict the production rules to facilitate disassembly into subgraphs by introducing the notion of a *root link*. Also, fusions are absorbed first to prevent link fusion from occurring later. And we decompose annotated contexts from the one holding a free root link (e.g., $\underline{nodes}_1$ holding X in the proof goal of Figure 4.12).

4.4 Constraints on Production Rules

The type system F_{GT} defined so far has imposed no restriction on production rules, even disconnected graphs (multisets) could be handled. However, here, we design the type system to efficiently support data structures of practical importance.

In order to handle graphs inductively with production rules easier, we introduce the notion of root links.

Definition 4.4.1 (Root link)**.** We call the last link of each atom as its *root link*.

We give a restriction on production rules so that we can find a spanning tree of a graph by traversing the root links. Since a spanning tree can be found for any connected graph, we can arrange the ordering of links of individual atoms in such a way that the root links form the edges of a spanning tree. Thus the restriction on production rules will not essentially sacrifice the expressive power of the data structure for practical programs. We call a link X the *root link of a graph* G if every atom in the graph can be reached through their root links from X.

Definition 4.4.2 (Constraints on production)**.** A production rule should have the form $\alpha(\overrightarrow{X}, R) \longrightarrow \tau$, where the τ should be one of the following.

1. one or more fusions.

2. has one constructor atom $C(\overrightarrow{Y}, R)$, zero or more type variable atoms $\alpha_i(\overrightarrow{Y_i})$, zero or more fusions, and zero or more arrow atoms and satisfies all the following conditions.

   (a) The root link R of $C(\overrightarrow{Y}, R)$ occurs free in τ.

   (b) The root link R_i of a type variable atom $\alpha_i(\ldots, R_i)$ should satisfy $R_i \in \{\overrightarrow{Y}\}$ and all the R_i's are mutually distinct.

All the examples we have introduced in Section 4.2.2 satisfy these constraints. Therefore, we claim that most of the practical examples are covered even with the restrictions.

4.4 Fusion Elimination

Since fusion ($\bowtie$) is difficult to handle, we attempt to eliminate fusions ($\bowtie$) except when they are generated directly from the annotated type variable atom by merging of production rules.

Definition 4.4.3 (Fusion elimination)**.** Let $P_{\bowtie}$ denote the set of production rules that include fusion. And let $\overline{P_{\bowtie}}$ denote the set of production rules without fusion. For each production rule in $\overline{P_{\bowtie}}$, we apply the production rules in $P_{\bowtie}$ to the type variable atom in the RHS of the rule. This is done in n^2 ways for n type variable atoms to cover all combinations. We add the newly created rules, which includes the original one, to P'. We also add the rules that have no type variable atoms on RHS to P'. If there exist rules in P' and $P_{\bowtie}$ which have the annotated type variable $\alpha(\overrightarrow{X})$ on the LHS, we add the rule whose LHS are replaced with $\alpha_{\bowtie}(\overrightarrow{X})$ to P'.

Finally, we replace the annotated type variable $\alpha(\overrightarrow{X})$ with $\alpha_{\bowtie}(\overrightarrow{X})$.

We have observed that it is not always possible to eliminate fusion in this way. However, all of our practical examples can be successfully transformed by this method. A more refined method of fusion elimination and a rigorous proof that the production rules obtained by this operation are equivalent to the original ones will be the subject of future work.

If fusion elimination succeeds, we can say that fusion will not appear "later" when the production rule is applied backwards (Ty-Prod). On the other hand, we cannot deny the possibility of occurrence of unabsorbable fusion when applying production rules to decompose graphs (Case). However, this did not happen in our examples.

Once we eliminate fusions, it will be easy to check the correspondence of links. Firstly, we α-convert link names so that all the link names are distinct. Then, the correspondence of links in the target graph and the annotated type can be checked as follows. If they are free links, check if they have the same name. If the links are local links, we check the correspondence between the link in the target graph and the link in the annotated type based on mapping. If the correspondence has not yet been established, add a new correspondence. If the correspondence is already in place, we check that it is satisfied. Figure 4.13 shows the algorithm to check the correspondence of links.

4.4 The Algorithm

It will be a little troublesome to implement the backward application of a production rule to handle the reverse execution of Ty-Prod. Thus, we will first apply the production rule to the annotated type and then remove the constructor atom both on the target graph and the annotated type. Note that this will result in allowing graphs in the annotation during the execution of this algorithm, which we refer to as an *annotated graph*.

Figure 4.14 shows the outline of the algorithm. The function $\mathit{check}(G, \alpha(\overrightarrow{X}, R), P)$ checks that $(\varnothing, P) \vdash G : \alpha(\overrightarrow{X}, R)$ where G possibly includes $\underline{\beta}(\overrightarrow{Y})$; type annotated graph

```
1  let check_link_name
2      L                                   (* A set of local links of the target graph *)
3      f    (* A mapping from the links in annotation to the links in the target graph
           *)
4      ( X,                                (* The link in the target graph *)
5       Y                                  (* The link in the annotation *)
6      )
7  =
8      if X ∉ L then
9          if X ∉ dom(f) ∧ X = Y then Some f else None
10     else
11         if Y ↦ None ∈ f then Some (f updated with Y ↦ Some X)
12         else if Y ↦ Some X ∈ f then Some f
13         else None
```

Figure 4.13: Check link name

G_β where $G_\beta : \beta(\overrightarrow{Y})$. The algorithm runs recursively with *helper* function (line 6) on the atoms/type annotated graph with a root link R of the target graph G and the annotated graph $\mathcal{T}$.

Line 12 checks that the graph G has type $\mathcal{T}$ *trivially*. For example, G maybe the type annotated graph whose annotated type was $\mathcal{T}$ or a λ-abstraction atom, whose typing relation can be checked as the same as the other functional language (except that we may need to apply this algorithm recursively for the graphs in its body expression).

From line 13, we split the cases by the atom with the root link of the target graph and the annotated graph. If both atoms have constructor names with the same functor, then we remove the atoms and run the algorithm recursively to all the subgraphs traversable from their arguments.

If the atom in the annotated graph is a type variable atom $\alpha(\overrightarrow{Y})$, then we first try to use induction hypotheses H (line 23 and line 28). Notice that we can use congruence rules (Ty-Cong) and α-conversion of free links (Ty-Alpha) to absorb the syntactic difference between $(G : \mathcal{T})$ and hypothesis in H.

If we cannot prove it by the hypothesis, then we should proceed with the construction of the proof tree with Ty-Prod or Case. If the root of the target graph is a constructor atom $C_G(\overrightarrow{X})$ (line 22), then we apply the production rules whose LHS is $\alpha/|\overrightarrow{Y}|$ and check there *exists* a way to successfully construct a sub-proof. Notice that we add the current typing relation to the induction hypotheses. If the root of the target graph is a type annotated graph $\underline{\beta}(\overrightarrow{X})$ (line 27), then we decompose the graph using the production rules of last Ty-Prod and check *all* of them satisfies the type.

Although we did not mention it in our pseudocode but we need to make sure that

the links $\overrightarrow{X}$ and $\overrightarrow{Y}$ have a proper correspondence using the function we have shown in Figure 4.13.

Theorem 4.4.1. *The algorithm in Figure 4.14 is sound.*

Proof. This is straightforward since we are constructing a proof tree. There is a concern that soundness may be violated when the induction hypothesis is used, but this is not a problem. This is because the size of the graph gets strictly smaller when the type checker applies the structural induction. The structural induction hypothesis is added on line 26, where a production rule is applied to the annotation, and the root of the annotated graph always becomes a constructor atom. Therefore, the type checker does not proceed to the cases except in line 14 in the recursion, and if this branch succeeds, the constructor atom is removed, reducing the size of the graph. Therefore, it is sound by the infinite descent method. □

4.5 Related Work

Since graphs and its operations are more complex than trees, there are diverse formalisms for graphs and graph types.

4.5 Functional Language with Graphs

FUnCAL [MA17] is a functional language that supports graphs as a first-class data structure. This language is based on an existing graph rewriting language, UnCAL . In UnCAL (and FUnCAL), graphs may include back edges and their equality is defined based on bisimulation. FUnCAL comes with its type system but does not support pattern matching for user-defined data types, which classic functional languages support for ADTs.

Functional programming with structured graphs [OC12] can express recursive graphs using recursive functions, i.e., **let rec** statements. Since they employ ADTs as the basic structure, they can enjoy type-based analysis based on the traditional type system. On the other hand, we can do further detailed type analysis by our language and type system.

Initial algebra semantics for cyclic sharing tree structures [Ham10] discusses how to express graphs by λ-expressions. However, there is a large gap between λ-expressions and pointer structures. On the other hand, we defined a graph based on nodes and hyperedges, which has a clear correspondence to a pointer structure. This style is rather suitable for future implementation. In addition, they do not support user-defined graph types or verification based on them.

4.5 Typing Frameworks for Graphs

Structured Gamma [FM98] is a typing framework for graphs, in which types are defined by production rules in context-free graph grammar. Shape Types [FM97] are similar but the following restrictions are imposed on type definitions to ensure completeness of type

checking: (i) the state space of type checking must be confluent, and (ii) graphs supplemented during the type checking must consist only of a finite number of symbols. With context-free graph grammar, we can express a broad and expressive class of types. However, type checking becomes harder and hence it does not cover some practical operations. For example, the concatenation of difference lists and the pop operation from the tail of them cannot be checked by either Shape Types or Structured Gamma. In this research, we restrict the target grammar so that we can verify practical operations by structural induction.

With Graph Types [KS93], we can define types of algebraic data structures accompanied by *extra edges*, where the destination of an extra edge is specified by a *routing expression*. A routing expression is a regular expression over small-step traverse operations, which describes the relative position of the destination of an extra edge, and the actual destination can be automatically computed based on it. In addition, Graph Types provide a decidable monadic second-order logic on the types as a way of formal verification and automatic program generation. For example, a constant-time concatenation of doubly-linked lists as modification of pointers can be deduced by the logic.

Our type system F_{GT} and Graph Types share the ideas that typed graphs consist of a canonical spanning tree and auxiliary edges, and types are defined by production rules. On the other hand, auxiliary edges and their modification are *computed* based on routing expressions in Graph Types, whereas they are described by users and *verified* by the types in our method. In addition, pattern matching based on the types can be described in our language λ_{GT}.

4.5 Separation Logic

Our approach is in contrast with the analysis of pointer manipulation programs using Separation Logic [Rey02], shape analysis [WSR00], etc.

Firstly, the target languages differ in many ways. Separation Logic and shape analysis normally handle low-level imperative programs using heaps and pointers. In contrast, we dispense with destructive operations and adopt pattern matching over graphs provided by the new higher-level language λ_{GT}, which abstracts address, pointers and heaps away, and features hyperlinks and operations on them including fusion and hiding.

Secondly, we pursue a lightweight, automatic type system for functional languages rather than Hoare-style general verification for imperative languages. Separation Logic allows us to use *pure formulae* that represent various non-spatial properties. The only thing that seems to correspond to pure formulae in our type system is fusion (which can be regarded as $x = y$ in Separation Logic). This design choice reflects the fact that our goal is not a formal system for software verification but a programming language and its type system.

The problem discussed in Section 4.3, verification of an inductively defined structure with structural induction, is close to the entailment problem of inductive predicates with

symbolic heaps in Separation Logic, sometimes referred to as *SLRD* (Separation Logic with Recursive Definitions). Cyclist [BGP12] performs automatic verification of the problem. However, the algorithm requires dynamic checking of the soundness condition. On the other hand, we have restricted graph grammar and proved the soundness statically as a (meta-)theorem. Antonopoulos et al. [Ant+14] show that the entailment problem of general SLRD is undecidable. Therefore, decision procedures for them impose some restrictions on SLRD. Iosif et al. [IRS13] propose a sub-class of SLRD, SLRD_{btw}, which handles graphs with bounded treewidth. The restrictions imposed on the recursive definitions are similar to the restrictions we have introduced in Section 4.4.1. However, they do not allow empty graphs and cannot handle a difference list without elements. Tatsuta et al. [TNK19] has imposed further restriction to SLRD_{btw} which corresponds to the notion of *root link* in ours. A precise comparison of the algorithms in [TNK19] and our technique will be the subject of future work.

4.6 Further Work

Finally, we address future work for the type system that is not mentioned in previous subsections.

4.6 Extend the Type System to Handle Untyped Graph Contexts

In this paper, we introduced dynamic type checking (Section 4.2.4) and excluded untyped graph contexts. However, verification with untyped graph contexts is necessary not just to reduce the programmer's extra effort since there exist programs that cannot be succinctly handled without untyped graph contexts. For example, matching the leftmost leaf in a leaf-linked tree is possible in λ_{GT} using a template consisting of the leftmost leaf and an untyped graph context for the rest of the tree. However, we cannot denote the type of the untyped graph context using the type of the leaf-linked tree because it is not a tree.

4.6 Extension on the Type System: Polymorphism and Type Inference

The proposed type system F_{GT} is monomorphic. We can only define difference lists with a specific element type, though introducing generic data types as in other functional languages could be done in the same way.

However, for more complex data structures, introducing polymorphism may be not that straightforward since we have introduced more powerful operations than the other languages such as concatenation of difference lists. In λ_{GT}, concatenation of difference lists can be done without explicitly handling constructor atoms, which may be typeable as a generic function. However, since operations on data structures may not result in data structures of the same type, we may need to verify programs with the type information of the inputs, which seems to be a little incompatible with polymorphism.

The same thing can be said for type inference. Since we allow powerful operations over data structures without explicitly denoting constructor names, it may be more difficult than in other functional languages and may require some non-obvious ingenious techniques.

```
1  let check (
2      G,                                                    (* Target graph *)
3      α(X⃗, R),                                             (* Annotated type atom *)
4      P                                                     (* Production rules *)
5  ) =
6      let rec helper (
7          R,                                                (* Root link *)
8          G,                                                (* Target subgraph *)
9          𝒯,                                                (* Annotated graph *)
10         H                                                 (* Induction hypotheses *)
11     ) =
12         if trivially G : 𝒯 then true
13         match (v (X⃗, R) or α(X⃗, R) in G, τ (X⃗, R) in 𝒯) with
14         | C_G(X⃗, R), C_𝒯(Y⃗, R) →
15             if C_G/|X⃗| ≠ C_𝒯/|Y⃗| then false
16             else
17                 ∀i.
18                 if X_i and Y_i are the roots of the non-empty subgraph G_i and 𝒯_i
                    then
19                     helper (R, G_i, 𝒯_i, H)
20                 else
21                     X_i and Y_i are not the root of atoms in G and 𝒯
22         | C_G(X⃗), α(Y⃗) →
23             (G : 𝒯) ∈ H ∨
24             ∃(β(Z⃗) ⟶ 𝒯') ∈ P such that
25             α/|Y⃗| = β/|Z⃗| ∧
26             helper (R, G, 𝒯'⟨Y_i/Z_i⟩^i, {G : 𝒯} ∪ H)
27         | β̲(X⃗), α(Y⃗) →
28             (G : 𝒯) ∈ H ∨
29             ∀(r with β/|X⃗| on LHS ∈ P).
30             helper (R, G decomposed β̲(X⃗) with r, 𝒯, H)
31         | otherwise → false
32     helper (R, G, α(X⃗, R), ∅)
```

Figure 4.14: Graph type checker

5

Conclusions and Further Work

5.1 Conclusions

In this study, we proposed a new functional language λ_{GT} that handles graphs as a first-class data structure with declarative operations based on graph transformation.

First, we investigated the property of a simple term language, HyperLMNtal, to denote labelled hypergraphs with ports with a finite set of axioms to denote the congruence of terms. We defined a mapping from HyperLMNtal terms to hypergraphs; *t2gs* and proved that the structural congruent terms are mapped to graph isomorphic graphs using *t2gs*; i.e., the mapping satisfies soundness. In other words, we have precisely discussed the relationship between the syntax for introducing (hyper)graphs into a programming language and mathematically defined hypergraphs. Since very few languages have incorporated hypergraphs, this is an important study for the foundation of its (denotational) semantics.

Second, we formalized the formal syntax and semantics of λ_{GT} in a syntax-directed manner, incorporating HyperLMNtal into a call-by-value λ-calculus. We have also implemented a reference interpreter.

Third, we developed a new type system F_{GT} that employs HyperLMNtal rules as production rules to deal with data structures more complex than trees. We firstly introduced the basic type system and then extended the type system to support more powerful verification such as concatenation of difference lists. We have also developed an algorithm to automatically verify programs with the extended type system using structural induction.

Finally, we address future work that is not mentioned in previous subsections.

5.2 Further Work

Our goal is to construct a solid theoretical foundation for the new generation of language that comes after Rust. Our current results so far are not sufficient to fulfil the purpose.

First, we need more investigation into HyperLMNtal. We have already proved the soundness of the denoted hypergraphs. However, we have left the completeness, which we believe is necessary to examine the completeness of the matching in the implementation. Furthermore, the λ_{GT} language does not just test that the given two graphs are congruent, it tries to find a substitution to make them congruent. Thus, we need to investigate the

property with substitutions.

Second, we need to extend the type system more. The type system currently relies on dynamic type checking to ensure type safety in pattern matchings. This forces runtime costs depending on the size of graphs, which is *unacceptable* since we would lose one of the main advantages, runtime efficiency, to use rich data structures. Also, we need to investigate the way to warn against `non-exhaustive matching`s and `redundant matching`s.

Third, we construct a compiler to enable a more efficient execution with the same level of efficiency as the corresponding imperative code. The implementation of a reference interpreter in this study is only a Proof of Concept: execution efficiency is not considered. To improve execution performance to the same level as the corresponding imperative code, it is necessary to develop static analysis more. We are planning to extend the type system to check the direction (polarity) of links [Ued14], and then perform ownership checking [DM05]. Then, we develop a method to transpile to a lower-level code using reference types in functional languages such as OCaml or an imperative code with pointers.

Bibliography

[Ant+14] Timos Antonopoulos et al. "Foundations for Decision Problems in Separation Logic with General Inductive Predicates". In: *Proc. FoSSaCS 2014*. Vol. 8412. Lecture Notes in Computer Science. Springer, 2014, pp. 411–425. DOI: 10.1007/978-3-642-54830-7_27 (cit. on p. 80).

[Bak+15] Christopher Bak et al. "A Reference Interpreter for the Graph Programming Language GP 2". In: *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015*. Ed. by Arend Rensink and Eduardo Zambon. Vol. 181. EPTCS. 2015, pp. 48–64. DOI: 10.4204/EPTCS.181.4. URL: https://doi.org/10.4204/EPTCS.181.4 (cit. on pp. 48, 55).

[Bak15] Christopher Bak. "GP 2: efficient implementation of a graph programming language". PhD thesis. Department of Computer Science, The University of York, 2015. URL: https://etheses.whiterose.ac.uk/12586/ (cit. on pp. 48, 54).

[Bas94] David A. Basin. "A term equality problem equivalent to graph isomorphism". In: *Information Processing Letters* 51.2 (1994), pp. 61–66. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(94)00084-0. URL: https://www.sciencedirect.com/science/article/pii/0020019094000840 (cit. on p. 26).

[BFS00] Peter Buneman, Mary Fernandez, and Dan Suciu. "UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion". In: *The VLDB Journal* 9.1 (Mar. 2000), pp. 76–110. ISSN: 1066-8888. DOI: 10.1007/s007780050084. URL: https://doi.org/10.1007/s007780050084 (cit. on p. 26).

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. "A Generic Cyclic Theorem Prover". In: *Proc. APLAS 2012*. Vol. 7705. Lecture Notes in Computer Science. Springer, 2012, pp. 350–367 (cit. on p. 80).

[BL93] Jean-Pierre Banâtre and Daniel Le Métayer. "Programming by Multiset Transformation". In: *Commun. ACM* 36.1 (Jan. 1993), pp. 98–111. ISSN: 0001-0782. DOI: 10.1145/151233.151242. URL: https://doi.org/10.1145/151233.151242 (cit. on p. 54).

[DM05] Werner Dietl and Peter Müller. "Universes: Lightweight Ownership for JML." In: *Journal of Object Technology* 4 (Oct. 2005), pp. 5–32. DOI: 10.5381/jot.2005.4.8.a1 (cit. on pp. 54, 84).

[DP20] Christian Doczkal and Damien Pous. "Completeness of an Axiomatization of Graph Isomorphism via Graph Rewriting in Coq". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 325–337. ISBN: 9781450370974. DOI: 10.1145/3372885.3373831. URL: https://doi.org/10.1145/3372885.3373831 (cit. on pp. 5, 25, 27).

[Ehr+06] Hartmut Ehrig et al. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-31187-4. DOI: 10.1007/3-540-31188-2 (cit. on pp. 2, 5, 13, 14, 29).

[Fer+14] Maribel Fernández et al. "Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY". In: *Language, Life, Limits*. Ed. by Arnold Beckmann, Erzsébet Csuhaj-Varjú, and Klaus Meer. Cham: Springer International Publishing, 2014, pp. 183–193. ISBN: 978-3-319-08019-2 (cit. on p. 54).

[FM97] Pascal Fradet and Daniel Le Métayer. "Shape types". In: *Proc. POPL'97*. ACM. 1997, pp. 27–39. DOI: 10.1145/263699.263706 (cit. on pp. 32, 78).

[FM98] Pascal Fradet and Daniel Le Métayer. "Structured Gamma". In: *Science of Computer Programming* 31.2 (1998), pp. 263–289. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(97)00023-3 (cit. on pp. 2, 32, 54, 63, 78).

[FP18] Maribel Fernández and Bruno Pinaud. "Labelled Port Graph – A Formal Structure for Models and Computations". In: *Electronic Notes in Theoretical Computer Science* 338 (Oct. 2018), pp. 3–21. DOI: 10.1016/j.entcs.2018.10.002 (cit. on pp. 6, 8, 13).

[Gha+12] A.H. Ghamarian et al. "Modelling and analysis using GROOVE". In: *STTT* 14.1 (2012), pp. 15–40. DOI: 10.1007/s10009-011-0186-x (cit. on p. 54).

[GHU11] Masato Gocho, Taisuke Hori, and Kazunori Ueda. "Evolution of the LMNtal Runtime to a Parallel Model Checker". In: *Computer Software* 28.4 (2011), 4_137–4_157. DOI: 10.11309/jssst.28.4_137 (cit. on pp. 6, 54).

[Ham10] Makoto Hamana. "Initial Algebra Semantics for Cyclic Sharing Tree Structures". In: *Log. Methods Comput. Sci.* 6.3 (2010). URL: http://arxiv.org/abs/1007.4266 (cit. on p. 78).

[HMA18] Makoto Hamana, Kazutaka Matsuda, and Kazuyuki Asada. "The algebra of recursive graph transformation language UnCAL: complete axiomatisation and iteration categorical semantics". In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 287–337. DOI: 10.1017/S096012951600027X (cit. on p. 26).

[IRS13] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. “The Tree Width of Separation Logic with Recursive Definitions”. In: *Automated Deduction – CADE-24*. 2013, pp. 21–38. ISBN: 978-3-642-38574-2 (cit. on p. 80).

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. “GrGen.NET: The expressive, convenient and fast graph rewrite system”. In: *International Journal on Software Tools for Technology Transfer* 12.3 (2010), pp. 263–271. ISSN: 1433-2779. DOI: 10.1007/s10009-010-0148-8 (cit. on p. 54).

[KM09] Victor Khomenko and Roland Meyer. “Checking pi-Calculus Structural Congruence is Graph Isomorphism Complete”. In: July 2009, pp. 70–79. DOI: 10.1109/ACSD.2009.8 (cit. on p. 26).

[KS93] Nils Klarlund and Michael I. Schwartzbach. “Graph Types”. In: *Proc. POPL’93*. ACM. Charleston, South Carolina, USA, 1993, pp. 196–205. ISBN: 0-89791-560-7. DOI: 10.1145/158511.158628 (cit. on p. 79).

[Ler+22] Xavier Leroy et al. “The OCaml system release 4i̇4”. In: *INRIA* 3 (2022) (cit. on pp. 3, 30).

[LMN] LMNtal. https://github.com/lmntal/lmntal-compiler. (Visited on 08/10/2022) (cit. on p. 54).

[LP17] Enric Cosme Llópez and Damien Pous. “K4-free Graphs as a Free Algebra”. In: *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*. Ed. by Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin. Vol. 83. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrumfuer Informatik, 2017, 76:1–76:14. ISBN: 978-3-95977-046-0. DOI: 10.4230/LIPIcs.MFCS.2017.76. URL: http://drops.dagstuhl.de/opus/volltexte/2017/8088 (cit. on pp. 5, 25).

[MA17] Kazutaka Matsuda and Kazuyuki Asada. “A Functional Reformulation of UnCAL Graph-Transformations: Or, Graph Transformation as Graph Reduction”. In: *Proc. POPL’97*. Paris, France: ACM, 2017, pp. 71–82. ISBN: 9781450347211. DOI: 10.1145/3018882.3018883. URL: https://doi.org/10.1145/3018882.3018883 (cit. on p. 78).

[Mac04] Ian Mackie. “Efficient λ-Evaluation with Interaction Nets”. In: *Proc. RTA 2004*. Springer, 2004, pp. 155–169. ISBN: 978-3-540-25979-4 (cit. on p. 37).

[Mac06] Ian Mackie. “Encoding Strategies in the Lambda Calculus with Interaction Nets”. In: *Proc. IFL 2005*. Springer, 2006, pp. 19–36. ISBN: 978-3-540-69175-4 (cit. on p. 37).

[MP08] G. Manning and D. Plump. “The GP programming system”. In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008), volume 10 of Electronic Communications of the EASST*. 2008 (cit. on p. 54).

[OC12] Bruno C.d.S. Oliveira and William R. Cook. "Functional Programming with Structured Graphs". In: *SIGPLAN Not.* 47.9 (2012), pp. 77–88. ISSN: 0362-1340. DOI: 10.1145/2398856.2364541. URL: https://doi.org/10.1145/2398856.2364541 (cit. on p. 78).

[Plo04] Gordon Plotkin. "A Structural Approach to Operational Semantics". In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139. DOI: 10.1016/j.jlap.2004.05.001 (cit. on pp. 6, 7).

[Pug90] William Pugh. "Skip lists: A probabilistic alternative to balanced trees". In: *Commun. ACM* 33.6 (1990), pp. 668–676. DOI: 10.1145/78973.78977 (cit. on pp. 1, 29).

[RET12] Olga Runge, Claudia Ermel, and Gabriele Taentzer. "AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations". In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by Andy Schürr, Dániel Varró, and Gergely Varró. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 81–88. ISBN: 978-3-642-34176-2 (cit. on p. 54).

[Rey02] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proc. LICS 2002*. IEEE. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cit. on pp. 2, 26, 57, 79).

[Roz97] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1997. DOI: 10.1142/3303 (cit. on pp. 1, 2, 5–7, 29).

[San21] Jin Sano. "Implementing G-Machine in HyperLMNtal". https://arxiv.org/abs/2103.14698. Bachelor's Thesis. Waseda University, 2021. arXiv: 2103.14698. URL: https://arxiv.org/abs/2103.14698 (cit. on pp. 3, 5, 6, 10, 11).

[SLI] SLIM. https://github.com/lmntal/slim. (Visited on 08/10/2022) (cit. on p. 54).

[SU21] Jin Sano and Kazunori Ueda. "Syntax-driven and compositional syntax and semantics of Hypergraph Transformation System". In: *Proc. 38th JSSST Annual Conference (JSSST 2021)*. 2021 (cit. on pp. 3, 5, 6, 29, 40, 54, 91).

[SU22] Jin Sano and Kazunori Ueda. "A functional language with graphs as first-class data". In: *Proc. 39th JSSST Annual Conference (JSSST 2022)*. 2022 (cit. on p. 91).

[SU23] Jin Sano and Kazunori Ueda. "Axiomatizing Hypergraph Isomorphism". In: *Special Interest Group on Programming and Programming Language*. 2023 (cit. on p. 91).

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. USA: Cambridge University Press, 2001. ISBN: 0521781779 (cit. on pp. 6, 7, 29).

[SWZ97] Andy Schürr, Andreas J. Winter, and Albert Zündorf. "The PROGRES Approach: Language and Environment". In: *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific, 1997. Chap. 13, pp. 487–550. ISBN: 9810228848. DOI: 10.1142/9789812384720_0002 (cit. on p. 54).

[SYU23] Jin Sano, Naoki Yamamoto, and Kazunori Ueda. "Type Checking Data Structures More Complex Than Trees". In: *Journal of Information Processing* 31.1 (Jan. 2023). ISSN: 1882-7802. URL: http://id.nii.ac.jp/1001/00223349/ (cit. on pp. 6, 91).

[TNK19] Makoto Tatsuta, Koji Nakazawa, and Daisuke Kimura. "Completeness of Cyclic Proofs for Symbolic Heaps with Inductive Definitions". In: *APLAS 2019*. Vol. 11893. Lecture Notes in Computer Science. Springer, 2019, pp. 367–387. DOI: 10.1007/978-3-030-34175-6_19. URL: https://doi.org/10.1007/978-3-030-34175-6%5C_19 (cit. on p. 80).

[Ued09] Kazunori Ueda. "LMNtal as a hierarchical logic programming language". In: *Theoretical Computer Science* 410.46 (2009), pp. 4784–4800. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2009.07.043 (cit. on pp. 5, 7, 54).

[Ued14] Kazunori Ueda. "Towards a Substrate Framework of Computation". In: *Concurrent Objects and Beyond*. Vol. 8665. LNCS. Springer, 2014, pp. 341–366. ISBN: 978-3-662-44471-9. DOI: 10.1007/978-3-662-44471-9_15 (cit. on pp. 54, 84).

[UO12] Kazunori Ueda and Seiji Ogawa. "HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model". In: *KI - Künstliche Intelligenz* 26.1 (2012), pp. 27–36. ISSN: 1610-1987. DOI: 10.1007/s13218-011-0162-3 (cit. on pp. 2, 3, 5, 7, 29, 54).

[WSR00] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. "Shape Analysis". In: *Compiler Construction, 9th International Conference, CC 2000*. Vol. 1781. Lecture Notes in Computer Science. Springer, 2000, pp. 1–17. DOI: 10.1007/3-540-46423-9_1. URL: https://doi.org/10.1007/3-540-46423-9%5C_1 (cit. on p. 79).

[YU16] Alimujiang Yasen and Kazunori Ueda. "Hypergraph Representation of Lambda-Terms". In: *Proc. 10th Int. Symp. on Theoretical Aspects of Software Engineering (TASE 2016)*. 2016, pp. 113–116. DOI: 10.1109/TASE.2016.25 (cit. on p. 37).

[YU21] Naoki Yamamoto and Kazunori Ueda. "Engineering Grammar-based Type Checking for Graph Rewriting Languages". In: *Proc. Twelfth International Workshop on Graph Computation Models (GCM 2021)*. June 2021, pp. 93–114 (cit. on pp. 32, 63).

Publications

1. 佐野仁, 上田和紀. HyperLMNtal を用いた G-Machine の実装. 第 23 回プログラミングおよびプログラミング言語ワークショップ (ポスター発表). 2021.

2. Jin Sano and Kazunori Ueda. "Syntax-driven and compositional syntax and semantics of Hypergraph Transformation System". In: *Proc. 38th JSSST Annual Conference (JSSST 2021)*. 2021. Student encouragement award.

3. 佐野仁, 上田和紀, 参照を用いたデータ構造の形状のユーザ定義の型に基づく型検査, 第 24 回プログラミングおよびプログラミング言語ワークショップ (ポスター発表), 2022.

4. Jin Sano and Kazunori Ueda. "A functional language with graphs as first-class data". In: *Proc. 39th JSSST Annual Conference (JSSST 2022)*. 2022. Presentation award.

5. Jin Sano, Naoki Yamamoto, and Kazunori Ueda. "Type Checking Data Structures More Complex Than Trees". In: *Journal of Information Processing* 31.1 (Jan. 2023). ISSN: 1882-7802. URL: http://id.nii.ac.jp/1001/00223349/.

6. Jin Sano and Kazunori Ueda. "Axiomatizing Hypergraph Isomorphism". In: *Special Interest Group on Programming and Programming Language*. 2023

A

Proof of properties of HyperLMNtal

Lemma A.0.1 (Elimination of ν which bounds no link name)**.**

$$\nu X.G \equiv G \ \textit{where}\ X \notin \mathit{fn}(G)$$

Proof.

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

□

Lemma A.0.2 (Elimination of a futile link substitution)**.**

$$G\langle X/X\rangle = G$$

This is not as obvious as it may seem. The reason is that it cannot be naïvely ruled out that a α-conversion may be performed during the hyperlink assignment, resulting in a congruent but syntactically different graphs.

Proof. We prove by induction on graphs. It is trivial for $\mathbf{0}$, $p(X_1, \ldots, X_m)$, (G, Q), $(G \longrightarrow Q)$.

Case $\nu Y.G$:

$$(\nu Y.G)\langle X/X\rangle \stackrel{\mathrm{def}}{=} \begin{cases} \nu Y.G & \text{if } Y = X \\ \nu Y.G\langle X/X\rangle & \text{if } Y \neq X \\ = \nu Y.G \because \text{induction hypothesis} & \end{cases}$$

Since $Y \neq X \wedge Y = X$ can never happen, there is no possibility of α-conversion of links (which could have resulted in loss of syntactic equality) to avoid variable capture.

□

Proof of Theorem 2.2.2. We are using Lemma A.0.1 and Lemma A.0.2. We consider the case where the free hyperlink to be substituted appears and the case where it does not. The latter case seems obvious, but it is not because of the possibility of α-conversion due to hyperlink substitution. We prove the former first, and then transform the latter into a form that allows us to use the former.

Case $X \in fn(G)$:

$\nu X.\nu Y.(Y \bowtie X, (X \bowtie Y, G))$
$\equiv_{\text{E5, E3}} \nu X.\nu Y.((Y \bowtie X, X \bowtie Y), G)$
$\equiv_{\text{E5, E10}} \nu X.(\nu Y.(Y \bowtie X, X \bowtie Y), G)$
$\quad \because Y \notin fn(G)$
$\equiv_{\text{E5, E6}} \nu X.(\nu Y.X \bowtie X, G)$
$\quad \because (X \bowtie Y)\langle X/Y\rangle = X \bowtie X$
$\equiv_{\text{E5, E10}} \nu X.\nu Y.(X \bowtie X, G)$
$\quad \because Y \notin fn(G)$
$\equiv_{\text{E9}} \nu Y.\nu X.(X \bowtie X, G)$
$\equiv_{\text{E6, Lemma A.0.2}} \nu Y.\nu X.G$
$\quad \because G\langle X/X\rangle = G$
$\equiv_{\text{Lemma A.0.1}} \nu X.G$
$\quad \because Y \notin fn(G)$

and

$\nu X.\nu Y.(Y \bowtie X, (X \bowtie Y, G))$
$\equiv_{\text{E2, E3, E5, E9}} \nu Y.\nu X.(X \bowtie Y, (Y \bowtie X, G))$
$\equiv_{\text{E5, E6}} \nu Y.\nu X.(Y \bowtie Y, G\langle Y/X\rangle)$
$\equiv_{\text{E5, Lemma A.0.1}} \nu Y.(Y \bowtie Y, G\langle Y/X\rangle)$
$\quad \because X \notin fn((Y \bowtie Y, G\langle Y/X\rangle))$
$\equiv_{\text{E6}} \nu Y.G\langle Y/X\rangle$
$\quad \because$ by Lemma A.0.2$(G\langle Y/X\rangle)\langle Y/Y\rangle = G\langle Y/X\rangle$

Thus, $\nu X.G \equiv \nu Y.G\langle Y/X\rangle$

Case $X \notin fn(G)$:

In this case, we use the previous proof by first adding a free hyperlink X using (E7).

$$
\begin{aligned}
&\nu X.G \\
&\equiv_{\text{Lemma A.0.1}} G \\
&\equiv_{\text{E1}} (\mathbf{0}, G) \\
&\equiv_{\text{E4, E7}} (\nu X.\nu X.X \bowtie X, G) \\
&\equiv_{\text{E4, Lemma A.0.1}} (\nu X.X \bowtie X, G) \\
&\qquad \because X \notin \mathit{fn}(\nu X.X \bowtie X) \\
&\equiv_{\text{E10}} \nu X.(X \bowtie X, G) \\
&\qquad \because X \notin \mathit{fn}(G) \\
&\equiv_{\text{The formar proof}} \nu Y.(X \bowtie X, G)\langle Y/X\rangle \\
&\qquad \because X \in \mathit{fn}((X \bowtie X, G)) \\
&= \nu Y.(Y \bowtie Y, G\langle Y/X\rangle) \\
&\equiv_{\text{E10}} (\nu Y.Y \bowtie Y, G\langle Y/X\rangle) \\
&\qquad \because X \notin \mathit{fn}(G), \text{ thus } Y \notin \mathit{fn}(G\langle Y/X\rangle) \\
&\equiv_{\text{E4, Lemma A.0.1}} (\nu Y.\nu Y.Y \bowtie Y, G\langle Y/X\rangle) \\
&\equiv_{\text{E7}} (\mathbf{0}, G\langle Y/X\rangle) \\
&\equiv_{\text{E1}} G\langle Y/X\rangle \\
&\equiv_{\text{Lemma A.0.1}} \nu Y.G\langle Y/X\rangle
\end{aligned}
$$

□

Proof of Theorem 2.2.1.

$$
\begin{aligned}
&\nu Z.(Z \bowtie X, Z \bowtie Y) \\
&\equiv_{\text{E6}} \nu Z.(X \bowtie Y) \\
&\qquad \because (Z \bowtie Y)\langle X/Z\rangle = X \bowtie Y \\
&\equiv_{\text{Lemma A.0.1}} X \bowtie Y
\end{aligned}
$$

and

$$
\begin{aligned}
&\nu Z.(Z \bowtie X, Z \bowtie Y) \\
&\equiv_{\text{E2, E5}} \nu Z.(Z \bowtie Y, Z \bowtie X) \\
&\equiv_{\text{E6}} \nu Z.(Y \bowtie X) \\
&\qquad \because (Z \bowtie X)\langle Y/Z\rangle = Y \bowtie X \\
&\equiv_{\text{Lemma A.0.1}} Y \bowtie X
\end{aligned}
$$

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

□

B

Proof of properties of F_{GT}

Theorem 4.1 (Soundness of F_{GT}) can be derived in the same way as in the ordinary type systems for functional languages, so we omit the precise proof. Theorem 4.2 and Theorem 5.2 have a proof specific to F_{GT}, which is supplemented in this appendix.

B.1 Theorem 4.2 (F_{GT} and HyperLMNtal reduction)

Lemma B.1.1. *If* $G_1 \rightsquigarrow_P^* G_2$ *then* $G_1\langle Y/X\rangle \rightsquigarrow_P^* G_2\langle Y/X\rangle$

Proof. By (R1), (R2), (R3) and $G_1 \rightsquigarrow_P G_2$, we can show $\nu X.(X \bowtie Y, G_1) \rightsquigarrow_P \nu X.(X \bowtie Y, G_2)$. Thus $G_1\langle Y/X\rangle \rightsquigarrow_P G_2\langle Y/X\rangle$ by (R3). Then we can obtain $G_1\langle Y/X\rangle \rightsquigarrow_P^* G_2\langle Y/X\rangle$ by induction on the length of the reduction $\rightsquigarrow_P^*$ □

Proof of Theorem 4.2.2. We denote $\overrightarrow{[\tau_i(\overrightarrow{Y_i})/x_i[\overrightarrow{X_i}]]}^{i}$ as θ_x and $\overrightarrow{[\tau_i(\overrightarrow{Z_i})/(\lambda\ldots)_i(\overrightarrow{W_i})]}^{i}$ as θ_λ.

We firstly prove $\Rightarrow$. We split the cases by the last applied F_{GT} rules.

Case Ty-Ctx:

$T = x[\overrightarrow{X}]$ and $x[\overrightarrow{X}] : \tau\,(\overrightarrow{X}) \in \Gamma$. Thus $T[\tau\,(\overrightarrow{X})/x[\overrightarrow{X}], \ldots]\theta_\lambda = \tau\,(\overrightarrow{X}) \rightsquigarrow_P^* \tau\,(\overrightarrow{X})$.

Case Ty-Arrow:

$T = (\lambda\ldots)(\overrightarrow{X})$ where $(\Gamma, P) \vdash (\lambda\ldots)(\overrightarrow{X}) : \tau\,(\overrightarrow{X})$. Thus $T\theta_x[\tau\,(\overrightarrow{X})/(\lambda\ldots)(\overrightarrow{X})] = \tau\,(\overrightarrow{X}) \rightsquigarrow_P^* \tau\,(\overrightarrow{X})$.

Case Ty-Cong:

Suppose the antecedent of Ty-Cong was $(\Gamma, P) \vdash T' : \tau\,(\overrightarrow{X})$ where $T \equiv T'$. By induction hypothesis, $\tau\,(\overrightarrow{X}) \rightsquigarrow_P^* T'\theta_x\theta_\lambda$. Since $T\theta_x\theta_\lambda \equiv T'\theta_x\theta_\lambda$, we can show $\tau\,(\overrightarrow{X}) \rightsquigarrow_P^* T\theta_x\theta_\lambda$ using (R3).

Case Ty-Alpha:

Suppose the antecedent of Ty-Alpha was $(\Gamma, P) \vdash T' : \tau\,(\overrightarrow{X'})$ where $T = T'\langle Y/X\rangle$ and $\tau\,(\overrightarrow{X}) = \tau\,(\overrightarrow{X'})\langle Y/X\rangle$. By induction hypothesis, $\tau\,(\overrightarrow{X'}) \rightsquigarrow_P^* T'\theta_x\theta_\lambda$. Here, we can show that $T\theta_x\theta_\lambda = T'\theta_x\theta_\lambda\langle Y/X\rangle$. Therefore, by Lemma B.1.1, $\tau\,(\overrightarrow{X'})\langle Y/X\rangle \rightsquigarrow_P^* T\theta_x\theta_\lambda\langle Y/X\rangle$.

Case Ty-Prod:

> Suppose the antecedents of Ty-Prod was $\overrightarrow{(\Gamma, P) \vdash T_i : \tau_i(\overrightarrow{X_i})}^{i}$ where $T = \mathcal{T}[\overrightarrow{T_i/\tau_i(\overrightarrow{X_i})}^{i}]$ By induction hypothesis, $\tau_i(\overrightarrow{X_i}) \rightsquigarrow_P^* T_i\theta_x\theta_{\lambda i}$. Therefore, using (R1), (R2), and (R3), we can show $\mathcal{T}' \rightsquigarrow_P^* \mathcal{T}'[T_i\theta_x\theta_{\lambda i}/\tau_i(\overrightarrow{X_i})]$ for any $\mathcal{T}'$. Thus, we can have $\tau_i(\overrightarrow{X_i}) \rightsquigarrow_P^* \mathcal{T}_0 \rightsquigarrow_P^* \cdots \rightsquigarrow_P^* \mathcal{T}_n$ where $\mathcal{T}_i$ is inductively defined as $\mathcal{T}_0 = \mathcal{T}$ and $\mathcal{T}_{i+1} = \mathcal{T}_i[T_i\theta_x\theta_{\lambda i}/\tau_i(\overrightarrow{X_i})]$, in which $\mathcal{T}_n = T\theta_x\theta_\lambda$.

Then, we prove $\Leftarrow$. by induction on the length of the reduction $\rightsquigarrow_P^*$. We denote $\overrightarrow{[x_i[\overrightarrow{X_i}]/\tau_i(\overrightarrow{Y_i})]}^{i}$ as θ_x^{-1} and $\overrightarrow{[(\lambda\ldots)_i(\overrightarrow{W_i})/\tau_i(\overrightarrow{Z_i})]}^{i}$ as θ_λ^{-1}. Then, the proposition can be rewritten as

$$\tau\,(\overrightarrow{X}) \rightsquigarrow_P^* \mathcal{T} \Rightarrow (\Gamma, P) \vdash \mathcal{T}\theta_x^{-1}\theta_\lambda^{-1} : \tau\,(\overrightarrow{X}).$$

Case $\tau\,(\overrightarrow{X}) = \mathcal{T}$ (The length of $\rightsquigarrow_P^*$ is zero):

> Follows by Ty-Ctx or Ty-Arrow depending on whether the $\tau\,(\overrightarrow{X})$ is replaced with the graph context in θ_x^{-1} or the λ-abstraction atom in θ_λ^{-1}.

Case $\tau\,(\overrightarrow{X}) \rightsquigarrow_P^* \mathcal{T}' \rightsquigarrow_P \mathcal{T}$ (The length of $\rightsquigarrow_P^*$ is $n > 0$):

> Suppose the production rule applied to reduce from $\mathcal{T}'$ to $\mathcal{T}$ was $\alpha\,(\overrightarrow{Y}) \longrightarrow \mathcal{T}''$. Using (R1), (R2), and (R3), we can obtain (new) $\tau\,(\overrightarrow{X}) \rightsquigarrow_P^* \mathcal{T}' \rightsquigarrow_P \mathcal{T}$ which satisfies $\mathcal{T} = \mathcal{T}'[\mathcal{T}''/\alpha\,(\overrightarrow{Y})]$. By induction hypothesis, we can obtain the derivation tree of
>
> $$(\Gamma, P) \vdash \mathcal{T}'\theta_x^{-1}\theta_\lambda^{-1} : \tau\,(\overrightarrow{X}). \tag{B.1}$$
>
> Since $\mathcal{T}'$ contains $\alpha\,(\overrightarrow{X})$, there exists a derivation of
>
> $$(\Gamma', P) \vdash \alpha\theta_x^{-1}\theta_\lambda^{-1} : \alpha\,(\overrightarrow{X}). \tag{B.2}$$
>
> in the tree. Since
>
> $$(\Gamma', P) \vdash \mathcal{T}''\theta_x^{-1}\theta_\lambda^{-1} : \alpha\,(\overrightarrow{X}). \tag{B.3}$$
>
> holds immediately by Ty-Prod, we can replace the derivation tree of (B.2) with that of (B.3) in that of (B.1), which will result in the derivation tree of the desired typing relation.

□

B.2 Theorem 5.2 (decomposing graph with the last applied production rule)

We omit $(\varnothing, P) \vdash$ for brevity.

Proof of Theorem 4.3.2. We prove by induction on the derivation of $G : \alpha(\overrightarrow{Y})$ after the last application of Ty-Prod.

By Lemma 4.3.1, there exists the last Ty-Prod and only Ty-Cong and Ty-Alpha are used later on the derivation of $G : \alpha(\overrightarrow{Y})$.

Case Ty-Prod:

Trivial from the definition of Ty-Prod.

Case Ty-Cong:

The theorem holds on $G : \alpha(\overrightarrow{Y})$ by induction hypothesis. Therefore, it holds on $G' \equiv G$.

Case Ty-Alpha:

By induction hypothesis, we can assume for $G : \alpha(\overrightarrow{Y})$, there exists $\overrightarrow{G_j}^j$ such that $G \equiv \mathcal{T}'[\overrightarrow{G_j/\tau_j\,(\overrightarrow{X_j})}^j]$ where

- $\mathcal{T}' = \mathcal{T}\overrightarrow{\langle Y_i/X_i\rangle}^i$,
- $\tau_j\,(\overrightarrow{X_j})$ are all the type atoms appearing in $\mathcal{T}'$, and
- $\overrightarrow{G_j : \tau_j\,(\overrightarrow{X_j})}^j$.

For $G\langle Z/Y\rangle : \alpha(\overrightarrow{Y})\langle Z/Y\rangle$, we can obtain

- $\mathcal{T}'' = \mathcal{T}\overrightarrow{\langle Z_i/X_i\rangle}^i$, where $Z_i = Y_i\langle Z/Y\rangle$.

The type atom $\tau_j\,(\overrightarrow{Z_j})$ appearing in $\mathcal{T}''$, corresponding to the atom $\tau_j\,(\overrightarrow{X_j})$ in $\mathcal{T}'$, may have substituted its links. Thus, we need to denote it as $\tau_j(\overrightarrow{X_j})\theta_j$ where θ_j is a hyperlink substitution which satisfies $\tau_j(\overrightarrow{X_j})\theta_j = \tau_j\,(\overrightarrow{Z_j})$. Since $\overrightarrow{G_j : \tau_j\,(\overrightarrow{X_j})}^j$ holds by the induction hypothesis, we can show that $\overrightarrow{G_j\theta_j : \tau_j\,(\overrightarrow{X_j})\theta_j}^j$ holds using Ty-Alpha. Therefore, we can obtain $\overrightarrow{G_j\theta_j}^j$ that satisfies the conditions.

□