

A functional language with graphs as first-class data

Jin Sano, Kazunori Ueda

Graphs are 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, as opposed to a purely functional programming style. We proposed a new purely functional language, λ_{GT} , that handles graphs as immutable, first-class data structures with a pattern matching mechanism based on Graph Transformation. Since graphs can be more complex than trees and require non-trivial formalism, the implementation of the language is also more complicated than ordinary functional languages. λ_{GT} is even more advanced than the ordinary graph transformation systems. 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 code.

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[20], doubly linked lists, leaf linked trees, and threaded trees, etc, can be handled succinctly.

Graphs as first-class data. Not pointers/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 [6] [21].

Syntax-driven semantics. To establish the semantics of λ_{GT} , we incorporated that of HyperLMNtal [31] [25] to 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 labeling function, the graphs in HyperLMNtal can be constructed compositionally from sub-graphs. These syntax and semantics follow the style of π -calculus [23] 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 uses infinite descent. However, in this paper, we focus on the language and do not mention much about the type system [26].

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

Immutability. Graphs are immutable in this lan-

* グラフを第一級に扱う関数型言語

This is an unrefereed paper. Copyrights belong to the Authors.

Jin Sano, Kazunori Ueda, 早稲田大学基幹理工学研究科 情報理工・情報通信専攻, Dept. of Computer Science and Communications Engineering, Graduate School of Fundamental Science and Engineering, Waseda University.

guage. We do not rely on destructive rewriting but employ immutable composing and decomposing with pattern matching.

Contributions

λ_{GT} can handle graphs. Since graphs can be more complex than trees, it requires non-trivial formalism. Thus its implementation is also more complicated than the ordinary functional languages.

λ_{GT} is even more advanced than the ordinary graph transformation systems. Graph transformation systems rewrite one global graph with rewriting rules. On the other hand, in λ_{GT} , graphs are local values that can be decomposed by pattern matching with possibly multiple wildcards, passed as inputs of functions, and composed to construct larger graphs from the subgraphs. Therefore, how to trivial to implement the language is not trivial.

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[14] code, which is strikingly concise^{†1}.

Structure of the paper

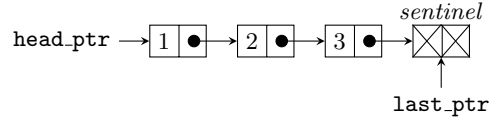
The rest of this paper is organized as follows. Section 2 introduces λ_{GT} informally. Section 3 gives the formal syntax and the operational semantics of λ_{GT} . In Section 4, we give more detailed explanation about the examples we have introduced informally in Section 2. Section 5 describes why we need such an interpreter. Section 6 explains the implementation. Section 7 discusses related work and indicates our expected future work.

2 Informal introduction to λ_{GT}

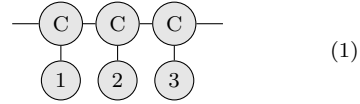
In this section, we introduce λ_{GT} *informally*. We will give the formal syntax and semantics of the language later in Section 3.

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

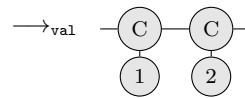
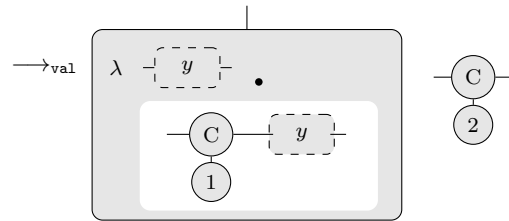
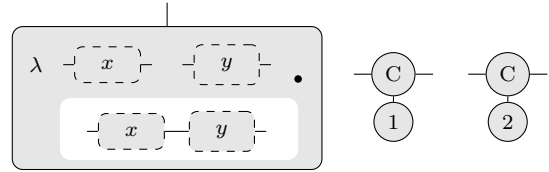
that points to the address of the last node (sentinel node) of the list. Adding a new element to the list can be done with the destructive assignment to the sentinel node with a new number and the address to the newly created sentinel node. We also need to update `last_ptr` to point to the new sentinel 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.



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.



^{†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/>.

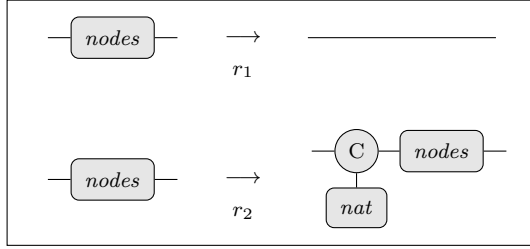
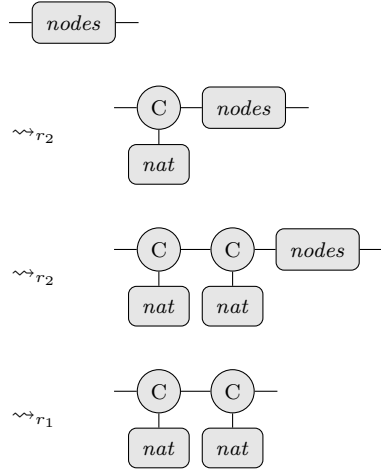


Fig. 1 Production rules for a difference list

Although we will not discuss in detail in this paper, we have also proposed a new type system for the language in [26], which is extended from the typing framework for graph transformation based languages [8] [9] [32]. 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 1.

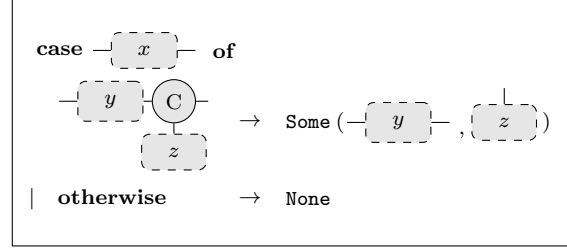
Informally, we can say graph has a type if we can obtain the graph from the type applying production rules zero or more times. The following example shows that we can obtain a difference list with two elements using the production rules.



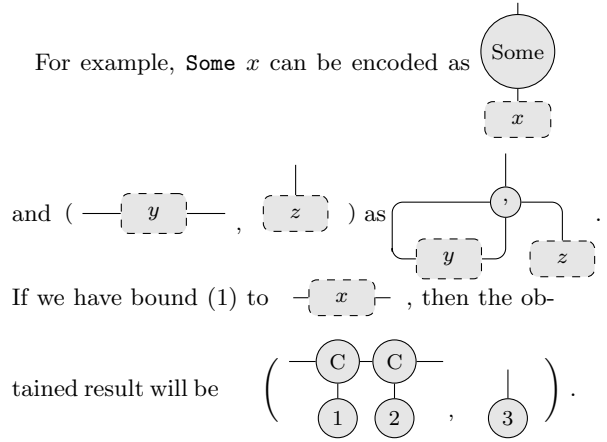
Here, \rightsquigarrow_r is a HyperLMNtal reduction using the rule r , where the formal semantics is described in [26] [25]. In [26], we have shown that the function has a type that takes two difference lists and returns a difference list.

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 tra-

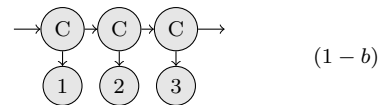
ditional functional languages with Algebraic Data Types. With Algebraic Data Types, normally, only the root of a tree can be matched. Taking the last element of a list needs iterating from the head of the list. On the other hand, in λ_{GT} , we can pop the last element in one step.



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.



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.



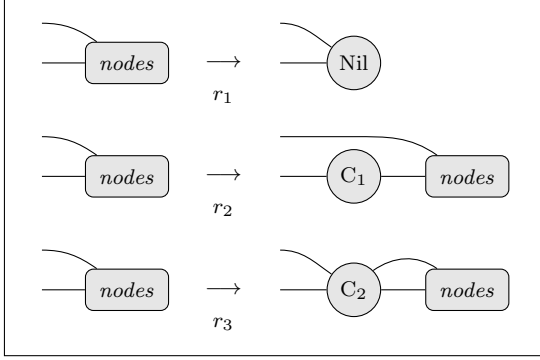
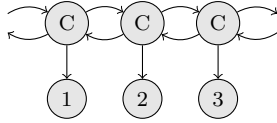


Fig. 2 Production rules for 2-level skip list

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.



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 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 5, using the production rules in Figure 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 6. Figure 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 7.

Here, if $\oplus 1$ is a function that returns the successor of a given number, and we have bound

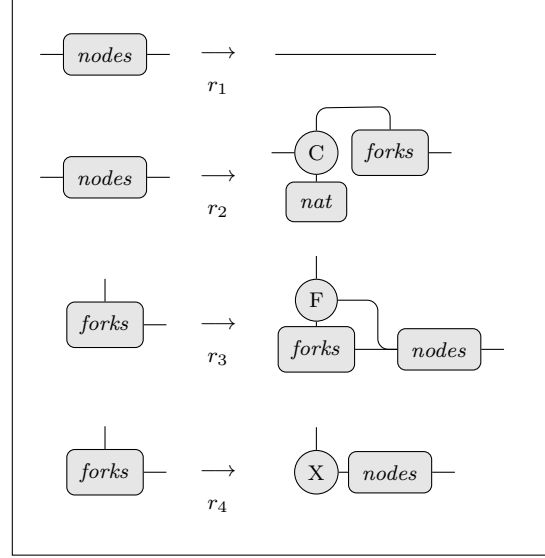
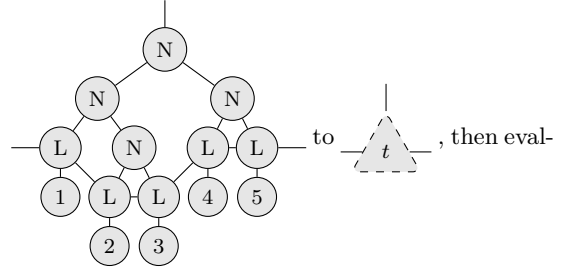
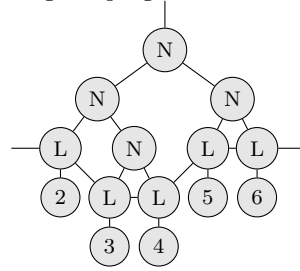


Fig. 3 Production rules for an arbitrary-level skip list



uating the program will result in



3 The syntax and semantics of λ_{GT}

Throughout the paper, we use the following syntactic conventions.

For some syntactic entity E , \vec{E} stands for a sequence E_1, \dots, E_n for some $n (\geq 0)$. The length of the sequence \vec{E} is denoted as $|\vec{E}|$.

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

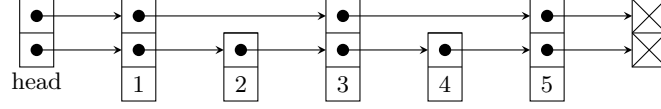


Fig. 4 2-level skip list with heaps and pointers

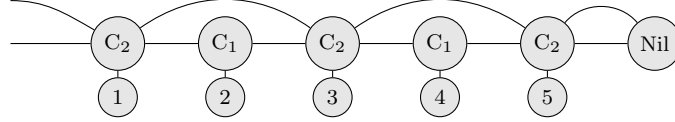


Fig. 5 2-level skip list in λ_{GT}

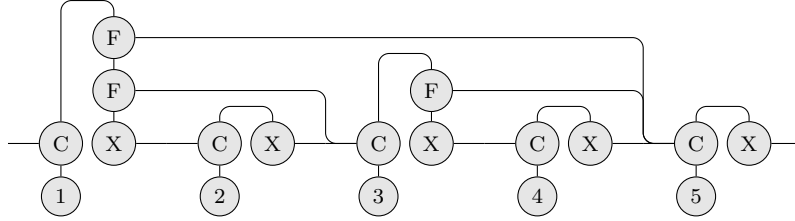


Fig. 6 n -level skip list ($n = 3$)

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.

3.1 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*.
 - Cons, Fork, etc.
- x denotes a *Graph Context Name*.

λ_{GT} is designed to be a *small* language focusing on handling graphs. The syntax of λ_{GT} is given in Figure 8.

Graph Template. $\mathbf{0}$ denotes an empty graph. A *graph context* $x[\vec{X}]$, where \vec{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 \vec{X} . $v(\vec{X})$ is an *atom*. Intuitively, it is a *node* of a data structure with label v and links \vec{X} . (T, T) is a multiset union. $\nu X.T$ hides the link X . We call a link *free link*, if it is not hidden. The set of the free links in T is denoted as $fn(T)$. The links that do not occur free are called *local links*.

Atom name. $C(\vec{X})$ is a constructor atom. For

example, $\text{Nil}(X)$, $\text{Cons}(Y, Z, X)$, etc. The λ -abstraction atoms have the form $(\lambda x[\vec{X}].e)(\vec{Y})$. Intuitively, the atom takes a graph with free links \vec{X} , binds it to the graph context $x[\vec{X}]$, and returns the value obtained by evaluating the expression e with the bound graph context. An atom $X \bowtie Y$, called a *fusion*, fuses the link X and the link Y into a single link.

Expression. T is a graph template we have defined so far. Notice that the corresponding concepts of variables and λ -expressions have already incorporated as graph contexts and λ -abstraction atoms in a graph template. (**case** e_1 **of** $T \rightarrow e_2$ | **otherwise** $\rightarrow e_3$) evaluates e_1 , checks whether this matches the graph template T , and reduces to e_2 or e_3 . $(e_1 e_2)$ is an application.

Value. 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.

Although we omitted in the figures in the previous section, graphs have free links and the links (edges) atoms (nodes) are ordered. This is different from ordinary graphs in graph theory. We believe that the graph in λ_{GT} is a structure that naturally incorporates PL concepts and is easy to handle in

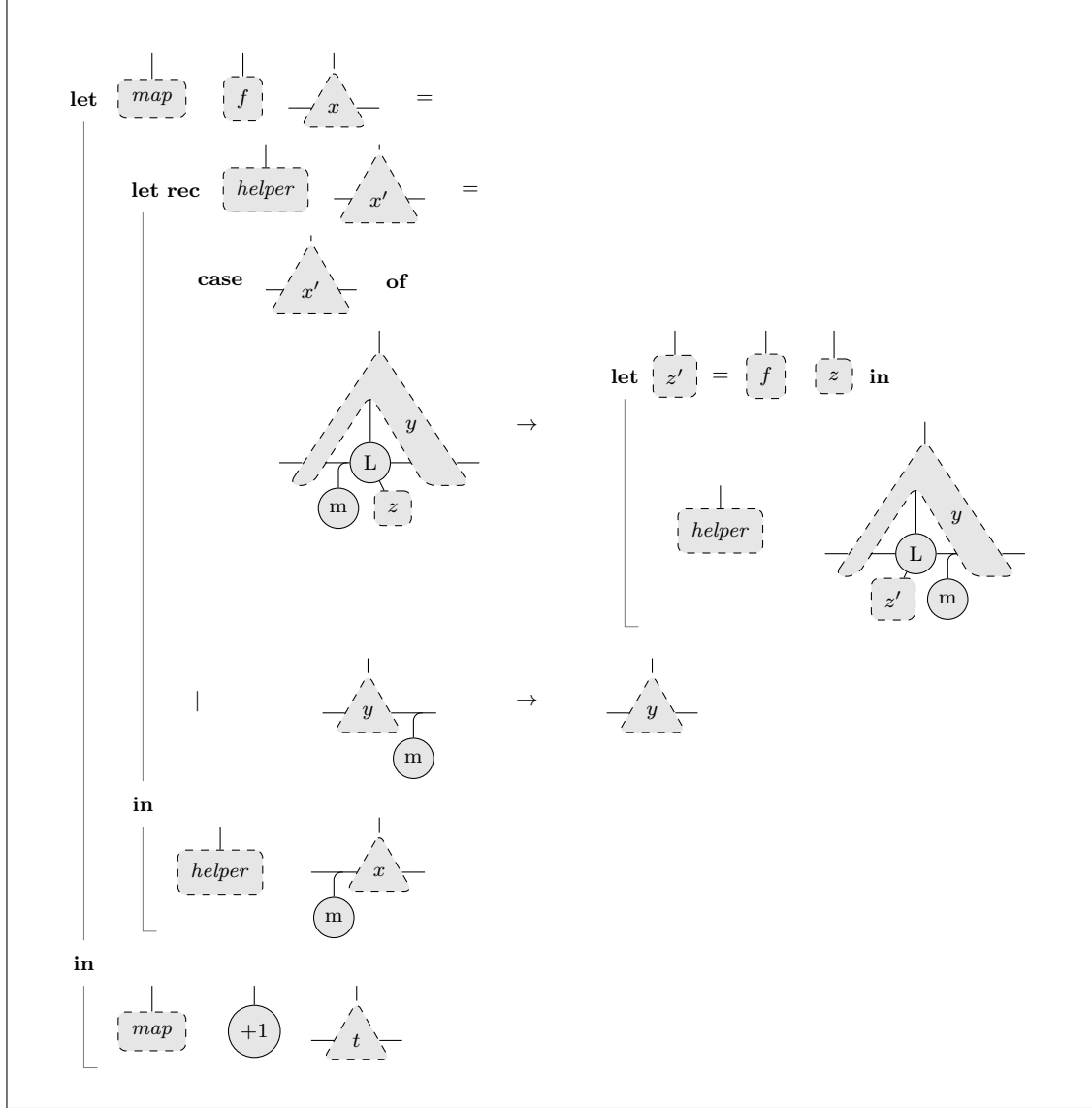


Fig. 7 A map function for leaf linked trees

programming. Also, the graph is composed of sub-graphs, in which we can naturally use structural induction, which is advantageous in verification.

The graphs in λ_{GT} is *undirected*. However, since the edges of an atom are ordered, we can easily encode directed graphs by stipulating that the final argument of atoms is the head of the links. All the examples in this paper can be interpreted as such.

3.2 Structural Congruence

Structural congruence \equiv defines what graphs T

or G are essentially the same.

Definition 3.1 (Structural Congruence). We define the relation \equiv on graphs as the minimal equivalence relation satisfying the rules shown in Figure 9. Here, $G\langle Y/X \rangle$ is a *hyperlink substitution* that replaces all free occurrences of X with Y . Notice if a free occurrence of X occurs in a location where Y would not be free, α -conversion may be required.

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

Expression	$e ::=$	T	Graph
		case e of $T \rightarrow e$ otherwise $\rightarrow e$	Case
		$(e\ e)$	Application
Graph Template	$T ::=$	$\mathbf{0}$	Null
		$x[\vec{X}]$	Graph context
		$v(\vec{X})$	Atom
		(T, T)	Molecule
		$\nu X.T$	Hyperlink creation
Atom Name	$v ::=$	C	Constructor name
		\bowtie	Fusion
		$\lambda x[\vec{X}].e$	Abstraction
Value	$G ::=$	$\mathbf{0} \mid v(\vec{X}) \mid (G, G) \mid \nu X.G$	Graph

Fig. 8 Syntax of λ_{GT}

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 9.

Theorem 3.1 (Symmetry of \bowtie).

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

Proof. See Chapter 3 of [24]. \square

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

Theorem 3.2 (α -conversion of hyperlinks).

Bound link names are α -convertible, i.e.,

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

Proof. See Chapter 3 of [24]. \square

Using structural congruence (syntax-directed approach) instead of graph isomorphism or bisimulation is *not* a common approach in graph transformation formalisms. However, we believe using syntax-directed approach makes the semantics a lot easier than adopting the other approaches, since

(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 $X \notin fn(G_2)$

Fig. 9 Structural congruence on graphs

the λ -calculus and many other computational models derived from the λ -calculus are defined as Structural Operational Semantics [19].

The pair of the name p and the arity $n = |\vec{X}|$ of a graph context $x[\vec{X}]$ is referred to as the *functor*^{†2} of the context and is written as x/n .

Definition 3.2 (Free functors of an expression). We define *free functors* of an expression e , $ff(e)$, in Figure 10. Free functors are not to be confused with free link names.

^{†2} Synonym of function symbol and function object; not to be confused with functors in category theory.

$$\begin{aligned}
ff(\text{case } e_1 \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) &= \\
ff(e_1) \cup (ff(e_2) \setminus ff(T)) \cup ff(e_3) \\
ff((e_1 \ e_2)) &= ff(e_1) \cup ff(e_2) \\
ff(x[\vec{X}]) &= \{x/\vec{X}\} \\
ff(v(\vec{X})) &= \emptyset \\
ff((\lambda x[\vec{X}].e)(\vec{Y})) &= ff(e) \setminus \{x/\vec{X}\} \\
ff((T_1, T_2)) &= ff(T_1) \cup ff(T_2) \\
ff(\nu X.T) &= ff(T)
\end{aligned}$$

Fig. 10 Free functors of an expression

3.3 Operational semantics of λ_{GT}

3.3.1 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 $x[\vec{X}]$ with a template T in e , written $e[T/x[\vec{X}]]$, as in Figure 11. The definition is standard except that it handles the substitution of the free links of graph contexts in the third rule.

3.3.2 Matching

We say that T matches a graph G if there exists a graph substitution θ such that $G \equiv T\theta$. For example, Figure 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 ([25]).

$$\begin{aligned}
(T_1, T_2)\theta &= (T_1\theta, T_2\theta) \\
(\nu X.T)\theta &= \nu X.T\theta \\
(x[\vec{X}])[T/y[\vec{Y}]] &= \\
&\quad \text{if } x/\vec{X} = y/\vec{Y} \text{ then } T\langle X_1/Y_1 \rangle \dots \langle X_{|\vec{X}|}/Y_{|\vec{Y}|} \rangle \\
&\quad \text{else } x[\vec{X}] \\
(v(\vec{X}))\theta &= v(\vec{X}) \\
((\lambda x[\vec{X}].e)(\vec{Z}))[T/y[\vec{Y}]] &= \\
&\quad \text{if } x/\vec{X} = y/\vec{Y} \text{ then } (\lambda x[\vec{X}].e)(\vec{Z}) \\
&\quad \text{else if } x/\vec{X} \notin ff(e) \text{ then } (\lambda x[\vec{X}].e[T/y[\vec{Y}]])(\vec{Z}) \\
&\quad \text{else } (\lambda z[\vec{X}].e[z[\vec{X}]/x[\vec{X}]](T/y[\vec{Y}]))(\vec{Z}) \\
&\quad \text{where } z/\vec{X} \notin ff(e). \\
(\text{case } e_1 \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3)\theta &= \\
&\quad \text{case } e_1\theta \text{ of } T \rightarrow e_2\theta \mid \text{otherwise} \rightarrow e_3\theta \\
(T_1 \ T_2)\theta &= (T_1\theta \ T_2\theta)
\end{aligned}$$

Fig. 11 Graph Substitution

3.3.3 Reduction

The evaluation strategy of λ_{GT} is call-by-value. In order to define the small-step reduction relation, we extend the syntax with evaluation contexts defined as follows: $E ::= [] \mid (\text{case } E \text{ of } T \rightarrow e \mid \text{otherwise} \rightarrow e) \mid (E \ e) \mid (G \ E) \mid T$. As usual, $E[e]$ stands for E whose hole is filled with e .

We define the reduction relation in Figure 13. Rd-Case1 and Rd-Case2 are for matching graphs we have explained so far. Rd- β is the key relation that applies a value to a function. The definition is standard except that (i) we need to check the correspondence of free links and (ii) we use graph substitution but the normal substitution in λ -calculus. The Rd-App1, Rd-App2, and Rd-Ctx are the same as in the call-by-value λ -calculus..

4 Programs examples in detail

This section describes some of the programs we have introduced informally in Section 2 in detail with the formal semantics in order to discuss the implementation algorithm in later sections.

We introduce some abbreviations before moving on to the examples.

Definition 4.1 (Abbreviations). We introduce the following abbreviation schemes:

1. A nullary atom $v()$ can be simply written as v .
2. Term Notation:
 $\nu X_n.(v_1(\dots, X_n, \dots), v_2(X_1, \dots, X_n))$ can be

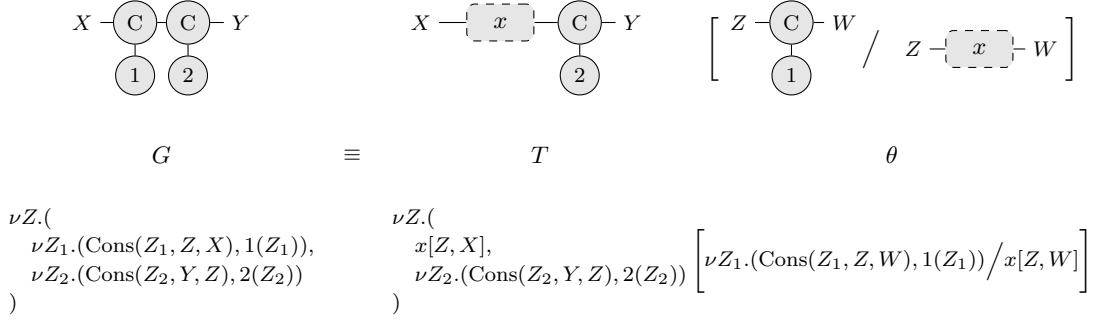


Fig. 12 Example of the graph matching

$\frac{G \equiv T\theta}{(\text{case } G \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) \rightarrow_{\text{val}} e_2\theta} \text{ Rd-Case1}$			Matching succeeded
$\frac{\neg \exists \theta. G \equiv T\theta}{(\text{case } G \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) \rightarrow_{\text{val}} e_3} \text{ Rd-Case2}$			Matching failed
$\frac{fn(G) = \{\vec{X}\}}{((\lambda x[\vec{X}].e)(\vec{Y}) G) \rightarrow_{\text{val}} e[G/x[\vec{X}]]} \text{ Rd-}\beta$			Beta reduction
$\frac{e_1 \rightarrow_{\text{val}} e'_1}{(e_1 e_2) \rightarrow_{\text{val}} (e'_1 e_2)} \text{ Rd-App1}$	$\frac{e \rightarrow_{\text{val}} e'}{(G e) \rightarrow_{\text{val}} (G e')} \text{ Rd-App2}$	$\frac{e \rightarrow_{\text{val}} e'}{E[e] \rightarrow_{\text{val}} E[e']} \text{ Rd-Ctx}$	

Fig. 13 Reduction relation of λ_{GT}

written as $v_1(\dots, v_2(X_1, \dots, X_{n-1}), \dots)$. For

example, a difference list with two elements,

$$\nu Z.(\nu Z_1.(\text{Cons}(Z_1, Z, X), 1(Z_1)), \nu Z_2.(\text{Cons}(Z_2, Y, Z), 2(Z_2)))$$

can be abbreviated as $\text{Cons}(1, \text{Cons}(2, Y), X)$.

3. We abbreviate $\nu X_1 \dots \nu X_n. G$ to $\nu X_1 \dots X_n. G$.
4. Application is left-associative.
5. Parentheses can be omitted if there is no ambiguity.
6. **let** $x[\vec{X}] = e_2$ **in** e_1 reduces in one step to $e_1[e_2/x[\vec{X}]]$.

Here, (1) and (2) also apply to graph contexts.

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

```

let append[Z] =
  (λ x[Y, X].
    (λ y[Y, X].
      ν Z.(x[Z, X], y[Y, Z])
    )(Z))(Z)
in append[Z] Cons(1, Y, X) Cons(2, Y, X)

```

Fig. 14 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 14.

We show the whole process of reduction of this program in Figure 15, whose process is already shown graphically in Section 2.

Firstly, the λ -abstraction atom is bound to the graph context $\text{append}[Z]^{\dagger 3}$. The bound λ -

^{†3} It may appear that the Z of $\text{append}[Z]$ does not

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

Without the term-notation abbreviation, the graph template used in the matching can be written as $\nu W Z.(y[W, X], \text{Cons}(Z, Y, W), z[Z])$. The matching in this template proceeds as the following, which is mostly the same as we have explained in Figure 12 in the previous section.

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

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 $\nu Z.(\text{Cons}(Z, Y, X), 1(Z))$ is bound to $x[Y, X]$. We need a sub-graph 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 sub-graphs.

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.

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

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

Even if a bound fusion atom is used, it may be

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 $\nu Z.(\text{Cons}(Z, Y, X), \text{append}[Z])$). 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.

absorbed and does not appear explicitly in the return value. Consider the program in Figure 17 that cycles the elements by taking the last node of the difference list and reconnecting it to the head. Fusion can be absorbed in the return value in this program.

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

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

The program using the map function for leaf-linked trees in Figure 7 in Section 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.

5 Reference interpreter: Uses and Requirements

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

```

let append[Z] = (λ x[Y, X]. (λ y[Y, X]. x[y[Y], X])(Z))(Z)
  in append[Z] Cons(1, Y, X) Cons(2, Y, X)
→val (λ x[Y, X]. (λ y[Y, X]. x[y[Y], X])(Z))(Z) Cons(1, Y, X) Cons(2, Y, X)
→val (λ y[Y, X]. x[y[Y], X])(Z)[Cons(1, Y, X)/x[Y, X]] Cons(2, Y, X)
= (λ y[Y, X]. Cons(1, y[Y], X))(Z) Cons(2, Y, X)
→val (Cons(1, y[Y], X))(X)[Cons(2, Y, X)/y[Y, X]]
= Cons(1, Cons(2, Y), X)

```

Fig. 15 An example of reduction: append operation on difference lists

```

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]

```

Fig. 16 Pop operation on a difference list

```

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]

```

Fig. 17 Rotate operation on a difference list

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.

6 Implementation

6.1 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 1. This is about half of the lines of a reference interpreter of a graph transformation-based language GP 2 [1], which is about 1,000 lines of Haskell code [2]. 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 use lists to handle graphs.

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

Table 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

Concrete syntax

- Link name starts from a capital letter with an underscore as a prefix.
- ν is written as `nu`.
- λ -abstraction atom $(\lambda x[\vec{X}].e)(\vec{Y})$ is written with `<` and `>` as follows.
`<\ x[_X1, ..., _Xn]. e> (_Y1, ..., _Ym).`
- The graph templates appears in expressions should be surrounded with `{` and `}`.

6.3 Preprocessing graphs

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 **integer** ids. In this paper, 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 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 $y[\text{Cons}(z, Y), X]$, the pattern in the Figure 16, is

transformed into the following.

$$\langle [\text{Cons}(L_{1001}, F_Y, L_{1000})], [y[L_{1000}, F_X], z[L_{1001}]] \rangle \quad (1)$$

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

$$[\text{Cons}(L_0, F_Y, F_X), 1(L_0)] \quad (2)$$

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.

6.4 Matching graphs

The interpreter firstly tries to match all the atoms in the template to the host graph and then match the graph contexts to the rest of the host graph. The interpreter backtracks if the consequent matching fails.

6.4.1 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 18. Free links in the template should 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 $\text{Cons}(L_{1001}, F_Y, L_{1000})$ in the template (1) can be matched to the atom

^{†4} This may be too ad-hoc solution but is simple and does work in our examples.

$\text{Cons}(L_0, F_Y, F_X)$ in the host graph (1) with link environment

$$\{L_{1000} \mapsto F_X, L_{1001} \mapsto L_0\} \quad (3)$$

and we have

$$[1(L_0)] \quad (4)$$

left in the host graph.

6.4.2 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 (4) becomes the following.

$$[1(L_{1001})] \quad (5)$$

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), we have a mapping $\{L_{1000} \mapsto F_X\}$. Thus, we should supply $L_{1000} \mapsto F_X$ and obtain

$$[1(L_{1001}), L_{1000} \bowtie F_X] \quad (6)$$

as the rest sub-graph.

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.

6.4.3 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 not the main scope of our language.

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

1. Find all the atom(s) with link \vec{Y} where $\{\vec{Y}\} \cap \{\vec{X}\} \neq \emptyset$. Add the atoms to G .
2. Update \vec{X} with $\{\vec{Y}\} \setminus \{\vec{X}\}$ and iterate from

```

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})

```

Fig. 18 Link name matching

(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 $x[\vec{X}]$.

For example, $y[L_{1000}, F_X]$ and $z[L_{1001}]$, the graph contexts in (1), can match host graphs $L_{1000} \bowtie F_X$ and $1(L_{1001})$, the sub-graphs in (6), respectively. After the matching, we obtain the following graph substitution.

$$\begin{aligned} & [L_{1000} \bowtie F_X / y[L_{1000}, F_X], \\ & 1(L_{1001}) / z[L_{1001}]] \end{aligned} \quad (7)$$

6.5 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 11 in Section 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 $x[\vec{X}]$ and the template has $x[\vec{Y}]$, we need to update G with substitution $\langle Y_1 / X_1 \rangle \dots \langle Y_{|\vec{Y}|} / X_{|\vec{X}|} \rangle$. 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 (7), we can instantiate the template on the right-hand side of \rightarrow in Figure 16, $[y[F_Y, F_X]]$, which will result in

$$[F_Y \bowtie F_X] \quad (8)$$

Our implementation absorbs fusions as much as possible after graph substitutions. The fusion atom in (8) cannot be absorbed since both its links are

free links. On the other hand, the example in Figure 17 will result in

$$\begin{aligned} & [L_{1000} \bowtie F_X, \\ & \quad \text{Cons}(L_{1001}, F_Y, L_{1000}), \\ & \quad 1(L_{1001}) \\ &] \end{aligned}$$

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

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

6.6 The Evaluator

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

7 Related and Future work

7.1 Related work

7.1.1 Functional languages with graphs

FUnCAL[17] is a functional language that supports graphs as a first-class data structure. This language is based on an existing graph rewriting language, UnCAL[4]. 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 [18] 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[12] 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.

7.1.2 Implementations for Graph Trans-

formation Systems

There are several languages based on graph transformations. For example, AGG [22], GAMMA [3], Structured Gamma [9], GP [16], GP 2 [1], GROOVE [10], GrGen.NET [13], (Hyper) LMNtal [29][31][25], PORGY [7], and PROGRES [27]. However, as far as we know, few published implementations have focused on simplicity.

HyperLMNtal, which is the language we have incorporated, has the compiler [15] and the runtime SLIM [28][11]. 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 [2]. 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.

7.2 Future work

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 [30], and then perform ownership checking [5]. 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.

Acknowledgments Part of this research was supported by Waseda University Research Grant (2022C-421, 2022Q-006).

References

- [1] Bak, C.: *GP 2: efficient implementation of a graph programming language*, PhD Thesis, Department of Computer Science, The University of York, 2015.
- [2] Bak, C., Faulkner, G., Plump, D., and Runciman, C.: A Reference Interpreter for the Graph

- Programming Language GP 2, *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015*, Rensink, A. and Zambon, E.(eds.), EPTCS, Vol. 181, 2015, pp. 48–64.
- [3] Banâtre, J.-P. and Le Métayer, D.: Programming by Multiset Transformation, *Commun. ACM*, Vol. 36, No. 1(1993), pp. 98–111.
- [4] Buneman, P., Fernandez, M., and Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion, *The VLDB Journal*, Vol. 9(2000).
- [5] Dietl, W. and Müller, P.: Universes: Lightweight Ownership for JML., *Journal of Object Technology*, Vol. 4(2005), pp. 5–32.
- [6] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*, Springer, 2006.
- [7] Fernández, M., Kirchner, H., Mackie, I., and Pinaud, B.: Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY, *Language, Life, Limits*, Beckmann, A., Csuhaj-Varjú, E., and Meer, K.(eds.), Cham, Springer International Publishing, 2014, pp. 183–193.
- [8] Fradet, P. and Métayer, D. L.: Shape types, *Proc. POPL’97*, ACM, 1997, pp. 27–39.
- [9] Fradet, P. and Métayer, D. L.: Structured Gamma, *Science of Computer Programming*, Vol. 31, No. 2(1998), pp. 263–289.
- [10] Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M.: Modelling and analysis using GROOVE, *STTT*, Vol. 14, No. 1(2012), pp. 15–40.
- [11] Gocho, M., Hori, T., and Ueda, K.: Evolution of the LMNTal Runtime to a Parallel Model Checker, *Computer Software*, Vol. 28, No. 4(2011), pp. 4.137–4.157.
- [12] Hamana, M.: Initial Algebra Semantics for Cyclic Sharing Tree Structures, *Log. Methods Comput. Sci.*, Vol. 6, No. 3(2010).
- [13] Jakumeit, E., Buchwald, S., and Kroll, M.: GrGen.NET: The expressive, convenient and fast graph rewrite system, *International Journal on Software Tools for Technology Transfer*, Vol. 12, No. 3(2010), pp. 263–271.
- [14] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J.: The OCaml system release 4.14, *INRIA*, Vol. 3(2022).
- [15] LMNTal: <https://github.com/lmntal/lmntal-compiler>.
- [16] Manning, G. and Plump, D.: The GP programming system, *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.
- [17] Matsuda, K. and Asada, K.: A Functional Reformulation of UnCAL Graph-Transformations: Or, Graph Transformation as Graph Reduction, *Proc. POPL’97*, ACM, 2017, pp. 71–82.
- [18] Oliveira, B. C. and Cook, W. R.: Functional Programming with Structured Graphs, *SIGPLAN Not.*, Vol. 47, No. 9(2012), pp. 77–88.
- [19] Plotkin, G.: A Structural Approach to Operational Semantics, *J. Log. Algebr. Program.*, Vol. 60-61(2004), pp. 17–139.
- [20] Pugh, W.: Skip lists: A probabilistic alternative to balanced trees, *Commun. ACM*, Vol. 33, No. 6(1990), pp. 668–676.
- [21] Rozenberg, G.: *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1997.
- [22] Runge, O., Ermel, C., and Taentzer, G.: AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations, *Applications of Graph Transformations with Industrial Relevance*, Schürr, A., Varró, D., and Varró, G.(eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2012, pp. 81–88.
- [23] Sangiorgi, D. and Walker, D.: *The Pi-Calculus: A Theory of Mobile Processes*, Cambridge University Press, USA, 2001.
- [24] Sano, J.: Implementing G-Machine in HyperLMNTal, Bachelor’s thesis, Waseda University, 2021. <https://arxiv.org/abs/2103.14698>.
- [25] Sano, J. and Ueda, K.: Syntax-driven and compositional syntax and semantics of Hypergraph Transformation System, *Proc. 32nd JSSST Annual Conference (JSSST 2021)*, 2021.
- [26] Sano, J., Yamamoto, N., and Ueda, K.: Type checking data structures more complex than trees, Presented at the 141th IPSJ Special Interest Group on Programming, Yamaguchi, Japan, 2022.
- [27] Schürr, A., Winter, A. J., and Zündorf, A.: *The PROGRES Approach: Language and Environment*, World Scientific, 1997, chapter 13, pp. 487–550.
- [28] SLIM: <https://github.com/lmntal/slim>.
- [29] Ueda, K.: LMNTal as a hierarchical logic programming language, *Theoretical Computer Science*, Vol. 410, No. 46(2009), pp. 4784–4800.
- [30] Ueda, K.: Towards a Substrate Framework of Computation, *Concurrent Objects and Beyond*, LNCS, Vol. 8665, Springer, 2014, pp. 341–366.
- [31] Ueda, K. and Ogawa, S.: HyperLMNTal: An Extension of a Hierarchical Graph Rewriting Model, *KI - Künstliche Intelligenz*, Vol. 26, No. 1(2012), pp. 27–36.
- [32] Yamamoto, N. and Ueda, K.: Engineering Grammar-based Type Checking for Graph Rewriting Languages, *Proc. 12th Int. Workshop on Graph Computation Models (GCM 2021)*, 2021.