Hypergraphs with Binding

Anna Matsui*
Johns Hopkins University, USA
amatsui1@jh.edu

Koko Muroya†
Ochanomizu University, Japan
kmuroya@is.ocha.ac.jp

Sands' notion of improvement asserts that one program has the same behaviour and requires less execution time compared to another program. Aiming at a robust and automatable methodology of proving improvement, we develop rewriting systems that use combinatorially-structured graphs, are mathematically solid, and are as expressive as second-order term rewriting. We introduce term-rewriting notions of variable binding, substitution and contexts, to Double Pushout rewriting of hypergraphs. Variable binding is represented simply by extra incidence, with the help of criteria to ensure well-formed binding. We demonstrate our rewriting systems by showing two-way simulation results for Hamana's second-order computational systems, and Alvarez-Picallo et al.'s string-diagram rewriting in a monoidal closed category.

1 Introduction

Behaviour of programs can be compared using multiple measures. A fundamental measure focuses on observable behaviour, such as returned values. Morris' notion of *observational equivalence* [14] asserts equivalence of programs with this measure. Another measure is cost of program execution. Sands' notion of *improvement* [17] combines these two measures, and asserts that one program has the same behaviour and requires less execution time compared to another program. By establishing improvement in this sense, one can for example certify compiler optimisation as correct and also effective.

The combinatorial, not inductive, structure of graphs was shown to be crucial to achieve a robust proof methodology of improvement [5]. The key idea of the methodology is to directly trace parts of a program, even if these parts are not inductively well-defined sub-programs. Ill-defined sub-programs can be represented by combinatorially well-defined subgraphs. The methodology is robust, to the extent that it can accommodate functional programs with general store.

The drawback of the proof methodology is a lack of solid formalisation of semantics. It relies on an ad-hoc rewriting system of hypergraphs, and hence it is not suitable for automation. There was an attempt to transfer the proof methodology to the established setting of second-order term rewriting. It was successful in the sense that it achieved automation of proofs [15, 8]. However, it comes at the cost of combinatorial representation and robustness. It uses inductively-defined terms instead of combinatorially-structured graphs, and it cannot immediately apply to programs with store.

Our long-term goal is to achieve a proof methodology with both robustness, which crucially exploits the combinatorial structure of graphs, and automation, which requires a solid formalisation of rewriting. This work lays a foundation, by developing graph-rewriting systems that are mathematically solid, and as expressive as second-order term rewriting. These systems would enable us to transfer the automation results [15, 8] from term-rewriting systems to graph-rewriting systems.

*Supported by ARO W911NF-20-1-0082
†Supported by JSPS, KAKENHI Project No. 22K17850, Japan

Submitted to:
SOS 2007

	[5]	[15, 8]	this work
combinatorial representation of programs	✓	×	✓
solid formalisation of rewriting	×	✓	✓
proof methodology of improvement	✓	✓	×
—robustness of proof methodology	✓	×	−
—automation of proof methodology	×	✓	−

Table 1: Towards a robust and automatable proof methodology of improvement

Table 1 summarises comparison between this work and previous results [5, 15, 8]. We here focus on the foundation of a proof methodology, namely graph-rewriting systems, and delegate development of an actual proof methodology to future work.

We employ the established categorical framework of Double Pushout rewriting [16], and introduce to the framework the term-rewriting notions that underpin the automation results. We here emphasise our desire to work with combinatorially-structured graphs, which were crucial for the proof methodology to achieve robustness [5].

We focus on the term-rewriting notions of variable binding, substitution, and context. To combinatorially represent variable binding, we equip hypergraphs with extra incidence, dubbed *binding connections*. We formulate substitution as Double Pushout rewriting that replaces the hyperedges labelled with metavariables. We formalise contexts as equivalence classes of hypergraph morphisms.

The simple definition of binding connections, as mere extra incidence, enables Double Pushout rewriting off the shelf. Technically, we can obtain an adhesive category of hypergraphs equipped with binding connections (*hypergraphs with binding* in short). However, the simple definition is too expressive to represent second-order terms. It allows some ill-formed variable binding. To recover the right expressivity, we identify *safety criteria* on hypergraphs with binding. Safety criteria are formulated in low-level terms, with the intention that they can be automatically checked.

Our contributions can be summarised as follows.

- We introduce hypergraphs with binding, and show that they form an adhesive category, in which Double Pushout rewriting is immediately possible.
- We identify safety criteria on hypergraphs with binding, and also on rewrite rules. Safe rewrite rules are shown to preserve safety of hypergraphs with binding.
- We formalise substitution as Double Pushout rewriting, and contexts as equivalence classes of hypergraph morphisms. Contexts are shown to be preserved by rule application that involves substitution.
- To demonstrate expressivity of our rewriting systems of (safe) hypergraphs with binding, we establish two-way simulation results for: second-order computational systems [7], and string-diagram rewriting in a monoidal closed category [1].

2 Preliminaries

We briefly recall the standard definitions concerning hypergraphs, interfaces, DPO rewriting, and higher-order terms. For a set S, let S^* denote an ordered list of elements of S. Let $PO(A \leftarrow B \rightarrow C)$ denote the

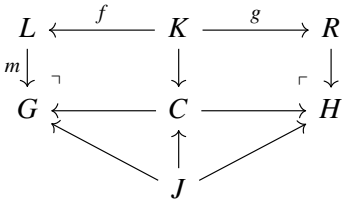
pushout of the morphisms $A \leftarrow B \rightarrow C$, and let $PC(D \leftarrow A \leftarrow B)$ denote a pushout complement, i.e. an object C such that $D = PO(A \leftarrow B \rightarrow C)$. It needs not always exist or be unique, in general.

Definition 2.1 (Hypergraphs)**.** *A (directed)* ***hypergraph*** *is a tuple $G = (V, E, s : E \rightarrow V^*, t : E \rightarrow V^*)$ where V and E are finite sets of nodes and hyperedges, s maps each hyperedge to a list of source nodes and t maps each hyperedge to a list of target nodes. The* ***arity*** *of a hyperedge is the length of the source list, and the* ***coarity*** *of a hyperedge is the length of the target list.*

Let Σ be a signature consisting of labels x with assigned arities n and coarities m. A Σ-labelled hypergraph is a hypergraph equipped with a labelling function $l : E \rightarrow \Sigma$ respecting these arities. The category of such hypergraphs and structure-preserving morphisms is denoted $\mathbf{Hyp}_\Sigma$.

Rewriting on hypergraphs is usually given via Double Pushout rewriting with Interfaces:

Definition 2.2. *A* ***rewrite rule*** *is a span $L \leftarrow K \rightarrow R$ in $\mathbf{Hyp}_\Sigma$. A* ***rewrite system*** *$\mathscr{R}$ is a finite set of rewrite rules. Given two hypergraphs with interfaces, $G \leftarrow J$ and $H \leftarrow J$, we say that G rewrites into H if there exists a rewrite rule $L \xleftarrow{f} K \xrightarrow{g} R$, a morphism $L \xrightarrow{m} G$ (called a* ***match****) and a hypergraph with interface $C \leftarrow J$, such that the squares below are pushouts and the whole diagram commutes:*



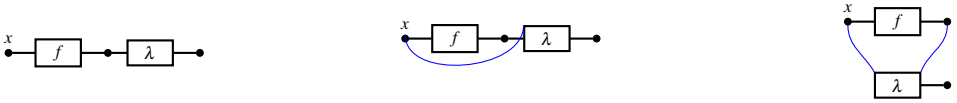
For higher-order term rewriting, in addition to the countably infinite set X of variables, we consider a countably infinite set of **metavariables** $\mathscr{M}$. Each $M \in \mathscr{M}$ has an arity $m \in \mathbb{N}$, denoted by $M : m$.

Definition 2.3. *A* ***binding signature*** *Σ consists of a set Σ of function symbols with an arity function $a : \Sigma \rightarrow \mathbb{N}^*$. A function symbol of arity $\langle n_1, \ldots, n_k \rangle$, denoted by $f : \langle n_1, \ldots, n_k \rangle$, has k arguments and binds n_i variables in the i-th argument $(1 \leq i \leq k)$.*

3 Hypergraphs with Binding

3.1 Binding

A hypergraph with binding is a hypergraph with distinguished connections between hyperedges and vertices. Consider an example from λ-calculus, which is going to resurface later: a hypergraph with binding representation of $\lambda x.f(x)$ for some f. If we tried to represent it as a regular hypergraph, we might get something like the picture on the left:



However, this sequential representation does not capture the fact that x is bound, so what it actually represents is $\lambda(f(x))$. To show the binding relation, we add an extra connection, thus obtaining a hypergraph with binding in the middle. To show that, intuitively, the part $f(x)$ in $\lambda x.f(x)$ lies one layer more on the "inside" than the edge λ, we usually draw it like in the picture on the right.

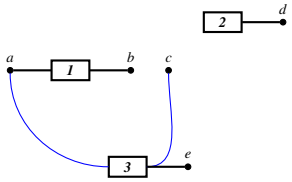
Thus, to extend from a hypergraph to a hypergraph with binding, in addition to the regular list of sources, each edge can also have a list of lists of vertices that are bound to the edge in each source.

Together, this data is given by two functions: $s : E \to V^*$ as before, and $b : E \to (V^*)^*$, where for each edge $e \in E$ the length of the lists $s(e)$ and $b(e)$ is the same. This can be encoded more compactly as one function $(s,b) : E \to (V \times V^*)^*$, sending an edge e to a list of pairs of vertices and a (possibly empty) list of (bound) vertices.

Definition 3.1 (Hypergraph with binding)**.** *A **hypergraph with binding** is a tuple $(E,V,t : E \to V^*, \hat{b} = (s,b) : E \to (V \times V^*)^*)$, where: E, V are the sets of edges and vertices, respectively, and $t : E \to V^*$, $(s,b) : E \to (V \times V^*)^*$ are the target, source, and binding functions, as described above.*

Example 3.2. *Example of a hypergraph with binding consisting of 3 edges, $\{1,2,3\}$, one of them having a binding connection, and 5 vertices, $\{a,b,c,d,e\}$:*

$$\begin{array}{lll} t(1) = \{b\} & t(2) = \{d\} & t(3) = \{e\} \\ s(1) = \{a\} & s(2) = \emptyset & s(3) = \{c\} \\ b(1) = \{\emptyset\} & b(2) = \emptyset & b(3) = \{\{a\}\} \end{array}$$



We show that **bHyp** is adhesive, which is sufficient for DPO-I rewriting to be well-defined [11].

Proposition 3.3. *The category **bHyp** of hypergraphs with binding and their morphisms is adhesive.*

We can also define interfaces to mark input and output vertices. Let $out_b(v)$ be the number of pairs (e,i) for which $s_b(e)_i = v$, i.e. v is a source with associated bound vertices, thus $out_b(v) \leq out(v)$.[1] Denote by $in_b(v)$ the number of triples (e,i,j) for which $b_i(e)_j = v$.

Definition 3.4 (Hypergraph with Binding with Interface)**.** *A hypergraph with binding with an **interface** is a cospan $I_G \to G \leftarrow O_G$ with $G \in$ **bHyp**, and I_G, O_G discrete, such that $I_G + O_G \to G$ maps into those vertices v of G for which one of the following holds: $in(v) = 0$ and $in_b(v) = 0 \iff v \in I_G$; and $out(v) = 0$ (and this implies $out_b(v) = 0$) $\iff v \in O_G$.*

Like the category $\mathbf{Hyp}_\Sigma$ of labelled hypergraphs, we obtain the category $\mathbf{bHyp}_\Sigma$ of labelled hypergraphs with binding for some binding signature Σ.[2]

3.2 Safety

The definition of hypergraphs with binding has no restrictions on what kind of hypergraphs we consider. This generality provides a key advantage: the framework can simulate realistic systems that may contain binding violations, enabling the detection and analysis of such errors through explicit safety checks.

We define safe hypergraphs in terms of *traces*, or *scopes*, of bound vertices: given a hypergraph with binding G with $e \in E_G$ such that $\exists i : b_i(e) \neq \emptyset$, we define the "body" of a sub-hypergraph bound to e as the "trace" of bound vertices. In Example 3.2, it is the path $\{a,1,b\}$ bound to the edge 3.

Construction 3.5 (i-th "body" of e - G_e^i)**.** *Let $e \in E_G$ be an edge with bound arguments. For a source index i such that $b_i(e) \neq \emptyset$, we define a sub-hypergraph G_e^i of G called the **i-th body of** e, spanned from the following rules:*

- *bound vertices are a part of the body: $\forall v \in b_i(e), v \in G_e^i$.*
- *edges with a source in the body are also part of the body (until we reach the bound source vertex $s_i(e)$): $\forall e', s(e') \cap G_e^i \neq \emptyset$, and $s_i(e) \notin s(e') \cap G_e^i, e' \in G_e^i$.*

[1] in fact, $out_b(v) + out_f(v) = out(v)$, with the definition of $out_f(v)$ being analogous to s_f, s_b.

[2] The binding signature as described in the Section 2, together with function symbols allowing multiple outputs.

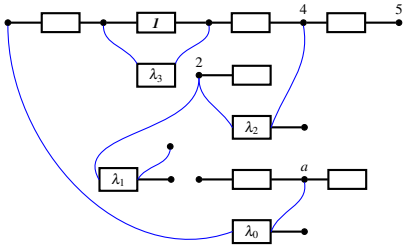
- *targets of edges in the body are also part of the body:* $\forall v$ *s.t.* $\exists e' \in G_e^i : v \in t(e'), v \in G_e^i$.

We define safe hypergraph with binding by gathering all the "scope" conditions together: no leaking, no sharing of scopes, and only variables/vertices[3] can be bound.

Definition 3.6 (Safe part of **bHyp**)**.** ***Safe hypergraphs with binding*** *are those elements of* ***bHyp****, for which the following restrictions hold:*

1. *vertices can be bound to only one edge and only once: if* $v \in b_i(e)$ *for some* $e \in E, i \in \mathbb{N}$*, and* $v \in b_j(e')$ *for some* $e' \in E, j \in \mathbb{N}$*, then* $e = e', i = j$.
2. *no shared body:* $\forall e \neq e'$, $\forall i, j : Edges(G_e^i) \cap Edges(G_{e'}^j) = \emptyset$.
3. *bound vertices cannot be a target: if* $v \in b_i(e)$ *for some* $e \in E, i \in \mathbb{N}$*, then* $v \notin t(e')$ *for* $\forall e' \in E$.
4. *no leaking - outputs of the body, connected to the bound vertex, cannot be shared: if* $y \in G_e^i$ *such that* $\exists$ *directed path* $[e_0, \ldots, e_1]$ *such that* $s(e_0) \cap b_i(e) \neq \emptyset, y \in t(e_1)$ *for some* $e \in G, i \in \mathbb{N}$*, and* $y = s(e)_i$*, then* $\forall e' \in G$, $y \notin s(e')$.
5. *no leaking - outputs of the body, connected to the bound vertex, cannot be in the output interface: if* $y \in G_e^i$ *such that* $\exists$ *directed path* $[e_0, \ldots, e_1]$ *such that* $s(e_0) \cap b_i(e) \neq \emptyset, y \in t(e_1)$ *for some* $e \in G, i \in \mathbb{N}$*, then* $y \notin O_G$.

Example 3.7. *The hypergraph below is clearly unsafe. The vertices and edges with number labels correspond to conditions for safety not being satisfied. Notice that the vertex a is not a problem - as it is a "free" vertex, it is not part of the body* $G^1_{\lambda_0}$ *and can be a target as well.*



3.3 Metavariables and Substitution

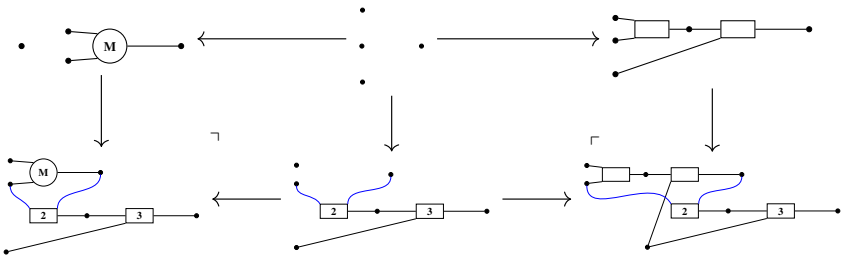
To perform higher-order substitution, first we must define metavariables. As metavariables can have their own arguments, we define them to be a special kind of edges. Those edges labelled with a metavariable, called **metaedges**, have no binding connections. To distinguish visually between edges and metaedges, the latter are drawn as circles.

A substitution operation must take a hypergraph G, a metavariable $M \in G$, a replacement hypergraph T, and yield $G[M/T]$. If M appears n times in G, we must insert n copies of T, sharing the *free* vertices of T across all copies while duplicating its bound internal structure.

Before the formal definition, consider an example.

Example 3.8. *Let G be a hypergraph with binding with edges "2", "3", and metaedge M. Let T be a hypergraph with two edges, and the interface of 3 inputs and 1 output. Let* $K = \{\bullet\}$ *be shared variables, mapping into the top one of T's inputs and into the "free" input of the edge* 3 *of G. The DPO construction for the substitution application is the following:*

[3] i.e., not full expressions



Definition 3.9. *Let $I_G \to G \leftarrow O_G$ and $I_T \to T \leftarrow O_T$ be hypergraphs with binding with interfaces, $M \in \mathscr{M}_G$ a metavariable. A substitution $\theta : M \mapsto T$, $\theta G := I \to G[M/T] \leftarrow O$ is given by a map $f : I_M \to I_T + O_T \in$ **bHyp**, and a span $I_G \leftarrow K \to I_{F_T}$, with K a discrete hypergraph of shared variables. Then the substitution is constructed via the following DPO in **bHyp**:*

$$\begin{array}{ccccc} n \cdot M \sqcup K & \longleftarrow & n \cdot (I_M) \sqcup K & \longrightarrow & n \hat{\cdot} T \\ \downarrow & & \downarrow & & \downarrow \\ G & \longleftarrow & C & \longrightarrow & G[M/T] \end{array}$$

with $I = PO(I_G \leftarrow K \to I_{F_T})$, $O = O_{n \hat{\cdot} T} \sqcup O_G$, $\mathscr{M} = (\mathscr{M}_\mathscr{G} \setminus \{M\}) \cup \mathscr{M}_\mathscr{T}$.

3.4 Rewriting in Contexts

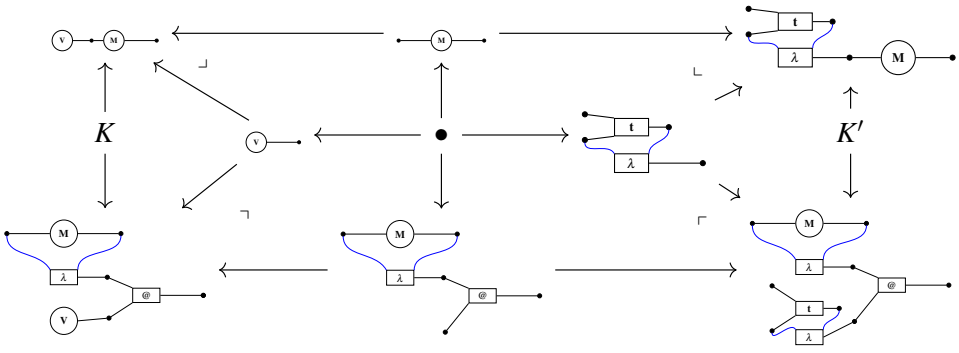
In term rewriting, a context [4] is a "term with a hole"—i.e., a special variable $\bullet$ appearing only once, called a hole. Here we formalise "hypergraphs with a hole" as equivalence classes of pushout squares.

Definition 3.10 (Context)**.** *A **context** $\mathscr{C}$ in **bHyp** given by a monomorphism $f : H \hookrightarrow G$ is an equivalence class of pushout squares defining the pushout complement $PC(G \leftarrow H \hookrightarrow I_H)$, where I_H is an interface of H. We say that $f_1 \sim f_2$ when the top rows of their squares are equivalent modulo $=_{bHyp}$.*

We prove that it is well-defined, and define composition of contexts, context substituion, and a context with its hole filled by a hypergraph T, denoted as $C[T]$.

A **rewrite system** on **bHyp** consists of a set of rewrite rules $\{L \leftarrow K \to R\}$ as before, together with a set of contexts $Ectx = \{\mathscr{C} : [H \hookrightarrow G]\}$ that restricts the matching morphism during rewrite rule application. Generally, we also require L, R to have the same metaedges.

Example 3.11. *Consider a rewrite rule: $(\lambda x.M[x])V \to M[V]$, with a substitution $V \mapsto \lambda x.t$. The top row is $R = M[V] \leftarrow C_R \to M[V/T]$ — a substitution $R \mapsto R[V/T]$ on the right-hand-side of the rule, — the bottom row is $L \leftarrow C_L \to L[V/T]$ — a substitution $L \mapsto L[V/T]$ on the left-hand-side of the rule. The general rewrite rule is the vertical span on the left side of the diagram. The extracted rewrite rule is then the vertical span on the right side of the diagram.*



[4] in some literature it is also called *environment*; not to be confused with type theoretic "context".

To put it formally,

Definition 3.12 (Rewriting in context)**.** *Let $Ectx$ be a set of contexts, $L \leftarrow K \rightarrow R$ a rewrite rule with L, R having the same metaedges and the same interface. An **application of the rewrite rule** $L \leftarrow K \rightarrow R$ to a hypergraph with binding G is given by: **1,** substitutions $V_i \mapsto T_i$ for metaedges $V_i \in M_L$; **2,** a matching map $L[V/T] \xrightarrow{m} G$ such that $[m] \in Ectx$. The result of the rewrite is then constructed via the usual DPO-I diagram for $L[V/T] \leftarrow K \rightarrow R[V/T]$.*

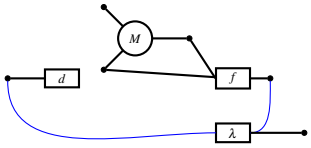
We prove that this construction is well-defined, and so context is preserved under DPOI rewriting.

4 Simulations

4.1 Second-Order Computational Systems

A **second-order CS** ([7, Section 3]) is a tuple $(\Sigma, \mathscr{M}, \mathscr{R})$, consisting of a binding signature Σ of function symbols, a set of metavariables $\mathscr{M}$, and a set $\mathscr{R}$ of rewrite rules. The raw meta-terms are given by the grammar $t ::= x \mid x.t \mid f(t_1, \ldots, t_n) \mid M[t_1, \ldots, t_n]$, while well-formed meta-terms are those derived by the rules that ensure arities and variable scopes are respected (see [7, Figure 1]). Additionally, rewrite rules are of the form $\mathscr{M} \rhd 0 \vdash l \Longrightarrow r \in \mathscr{R}$ such that all metavariables that are in r occur in l. One-step rewrites $\rhd n \vdash s \rightarrow_{\mathscr{C}} t$ are then derived by the rules that instantiate a rewrite rule by replacing metavariables with terms (see [7, Figure 2]).

We construct translation $(-)^\dagger$ from SOCS meta-terms into $\mathbf{bHyp}_{\Sigma \cup \{d\}}$. The additional label '$d$' is necessary to prevent the translation of variables from appearing in the interface. For example, a meta-term $\lambda(x.f(M[z,y],y))$ is translated to the following hypergraph with binding, with 'y' being the shared vertex. Here, the "body" of the edge λ is the part corresponding to $f(M[z,y],y)$, and it is safe.



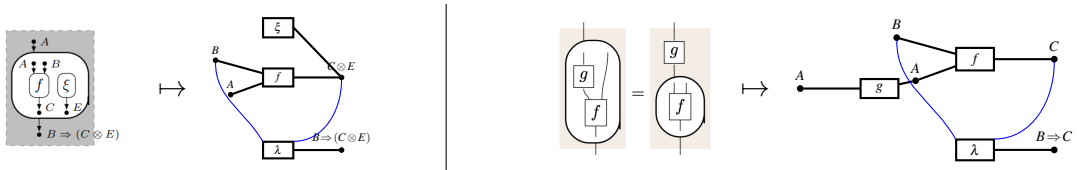
Theorem 4.1. *Let $(\Sigma, \mathscr{M}, \mathscr{R})$ be a second-order CS. For each meta-term t, there exists a hypergraph with binding $t^\dagger$ with the signature $\Sigma \cup \{d\}$. Moreover, the following holds.*

1. *(Safety). If t is well-formed, $t^\dagger$ is safe.*
2. *(One-step rewrite). Each one-step rewrite $\rhd n \vdash s \rightarrow_C t$ translates to a rewrite rule application in $\mathbf{bHyp}_{\Sigma \cup \{d\}}$ via $(-)^\dagger$.*
3. *(Inverse one-step rewrite). For each rewrite rule R in $\mathbf{bHyp}$ $L \leftarrow K \rightarrow R$, $\theta : L \ni V \mapsto T$, $m : L \hookrightarrow G$, there is a rewrite rule application in the second-order CS $(\Sigma, \mathscr{M}, \mathscr{R})$ such that it translates to R via $(-)^\dagger$ modulo $=_{bHyp}$.*

4.2 Rewriting morphisms in a Monoidal Closed Category

Hypernets, as special hierarchical hypergraphs, are introduced as a suitable formalisation of string diagrams for monoidal closed categories (MCC in short) in [1]. We relate hypernets, which do not form an adhesive category, to hypergraphs with binding, and show how to simulate MCC in **bHyp**.

We demonstrate that hypergraphs with binding natively simulate hypernets, absorbing the hierarchical structure into the flat topological binding incidence. We define a translation $(\cdot)^\dagger$ mapping a hierarchical hypergraph into $\mathbf{bHyp}_{\Sigma_E \cup \{\lambda\}}$, where $\lambda : \langle 1 \mid 1 \rangle$ acts as the boundary of the abstraction.



(a) Translation of a hypernet.

(b) Translation of the slide rule.

Figure 1: Translation examples. Pictures of hypernets are taken from [1, Figure 9(c), Figure 3(c)].

Figure 1a shows an example of the translation. The hierarchical structure, depicted using a rounded square containing a sub-hypergraph of two edges f and ξ, is translated using the λ-labelled edge and binding connections, with the "body" being the *inner* hypergraph. The equations of Monoidal Closed Categories are turned into three hypernet rewrite rules, called β law, η law, and slide rule [1, Figure 3]. Notice that the slide rule becomes trivial (Figure 1b), emphasising that it comes not from a morphism, but additional structure on a category.

Theorem 4.2. *For each hierarchical hypergraph $H = (E_H, V_H, s, t, p, l_E : E \to \Sigma_E + 1)$, there exists a hypergraph with binding $H^\dagger = (E_{H^\dagger}, V_{H^\dagger}, s, t, b)$, with the signature $\Sigma_E \cup \{\lambda\}$ where $\lambda : \langle 1|1 \rangle$. Moreover, the following holds.*

1. *(Safety). If H is a hypernet, $H^\dagger$ is safe.*
2. *(β, η laws). For each hypernet rewrite rule $L \leftarrow K \to R$ that is either a β law or an η law, it translates to a rewrite rule of hypergraphs with binding via $(-)^\dagger$.*
3. *(Inverse rewrite). For any rewrite rule $L \leftarrow K \to R$ of safe hypergraphs with binding, a matching morphism $m : L \hookrightarrow G$ such that the context $\mathscr{C} \ni m$ is convex, there exists a hypernet rewrite rule $\mathscr{L}, \mathscr{R}$ such that $\mathscr{L}^\dagger =_{bHyp} L, \mathscr{R}^\dagger =_{bHyp} R$. Moreover, for the result of the rewrite rule application at the convex matching $i : \mathscr{L} \hookrightarrow \mathscr{G}$, it holds that $\mathscr{G}[\mathscr{L} \mapsto \mathscr{R}]^\dagger =_{bHyp} G[L \mapsto R]$. This construction is unique up to the slide rule application and λ-clauses for vertex labels $A \Rightarrow B$.*

5 Related and Future Work

Hierarchical structures have been studied within Double Pushout rewriting of hypergraphs [3]. A variant was tailored to a certain form of variable binding, namely abstraction of the lambda-calculus [1]. We chose not to directly extend this variant to general variable binding, because it does not yield an adhesive category and hence requires a bespoke set-up of Double Pushout rewriting. We opt for an adhesive category, which can be achieved by a simple mechanism of binding connection.

Contexts have been studied in the graph-rewriting literature, under the name of *application condition* [4]. An application condition of a rewrite rule $L \leftarrow K \to R$ is given by a family of morphisms from L. Its variation, given by morphisms from K, has also been studied [12]. We do not follow this approach, because the rule application in our setting involves substitution; not the rule itself but an arbitrary result of its substitution is applied by Double Pushout. Our notion of context is instead given by an equivalence class of morphisms to a graph to which the rule is applied.

The idea of instantiating a rewrite rule is not new for graph rewriting either. A known form of instantiation copies a subgraph for an arbitrary number of times [10], and another one instantiates expressions that label nodes and edges [9].

Our approach to employ safety criteria to characterise well-formed variable binding resembles the so-called correctness criteria of proof nets, a graphical representation of linear logic proofs [6]. Binding connections are anonymous, which is in contrast to *Nominal string diagrams* [2]. These diagrams do not come with variable binding, so it would be interesting to transfer binding connections to them.

Our primary future direction is to develop a proof methodology of improvement, based on the rewriting systems of hypergraphs with binding. We expect that we could transfer the automatable proof methodology for second-order term rewriting [15, 8], to rewriting of hypergraphs with binding. The methodology relies on the well-known technique of critical pair analysis. There exists a potentially useful algorithm of enumerating critical pairs, for Double Pushout rewriting of certain hypergraphs that correspond to string diagrams [13].

References

[1] Mario Alvarez-Picallo, Dan Ghica, David Sprunger & Fabio Zanasi (2022): *Rewriting for Monoidal Closed Categories*. *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2022.29`.

[2] Samuel Balco & Alexander Kurz (2023): *Completeness of Nominal PROPs*. *Logical Methods in Computer Science* 19. Available at `https://api.semanticscholar.org/CorpusID:215828153`.

[3] Frank Drewes, Berthold Hoffmann & Detlef Plump (2002): *Hierarchical Graph Transformation*. *J. Comput. Syst. Sci.* 64(2), pp. 249–283, doi:10.1006/JCSS.2001.1790. Available at `https://doi.org/10.1006/jcss.2001.1790`.

[4] H. Ehrig & A. Habel (1986): *Graph Grammars with Application Conditions*, pp. 87–100. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-95486-3_7. Available at `https://doi.org/10.1007/978-3-642-95486-3_7`.

[5] Dan R. Ghica, Koko Muroya & Todd Waugh Ambridge (2025): *A robust graph-based approach to observational equivalence*. *Log. Methods Comput. Sci.* 21(2), doi:10.46298/LMCS-21(2:8)2025. Available at `https://doi.org/10.46298/lmcs-21(2:8)2025`.

[6] Jean-Yves Girard (1987): *Linear logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[7] Makoto Hamana (2022): *Complete algebraic semantics for second-order rewriting systems based on abstract syntax with variable binding*. *Mathematical Structures in Computer Science*. Available at `http://solweb.mydns.jp/hamana/Papers/acs-mscs.pdf`.

[8] Makoto Hamana & Kento Emoto (2026): *ReCheck: Automated Contextual Improvement Verifier for Functional Calculi across User-Defined Operational Semantics*. In Sebastian Junges & Guy Katz, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II*, Lecture Notes in Computer Science, Springer, pp. 215–234, doi:10.1007/978-3-032-22749-2_11. Available at `https://doi.org/10.1007/978-3-032-22749-2_11`.

[9] Ivaylo Hristakiev (2018): *Confluence analysis for a graph programming language*. Ph.D. thesis, University of York, UK. Available at `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.745783`.

[10] Aleks Kissinger, Alex Merry & Matvey Soloviev (2012): *Pattern Graph Rewrite Systems*. In Benedikt Löwe & Glynn Winskel, editors: *Proceedings 8th International Workshop on Developments in Computational Models, DCM 2012, Cambridge, United Kingdom, 17 June 2012*, EPTCS, pp. 54–66, doi:10.4204/EPTCS.143.5. Available at `https://doi.org/10.4204/EPTCS.143.5`.

[11] Stephen Lack & Paweł Sobociński (2004): *Adhesive Categories*. In Igor Walukiewicz, editor: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 273–288.

[12] Michael Löwe (2019): *Double-Pushout Rewriting in Context - Rule Composition and Parallel Independence*. In Esther Guerra & Fernando Orejas, editors: *Graph Transformation - 12th International Conference, ICGT 2019, Held as Part of STAF 2019, Eindhoven, The Netherlands, July 15-16, 2019, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 21–37, doi:10.1007/978-3-030-23611-3_2. Available at `https://doi.org/10.1007/978-3-030-23611-3_2`.

[13] A. Matsui, I. Obi, G. Sabbagh, L. Torres, D. Kessler, J. F. Meleiro & K. Muroya (2025): *A Critical Pair Enumeration Algorithm for String Diagram Rewriting*. In: *Proc. of ACT'25, to appear.*

[14] James Hiram Morris Jr (1969): *Lambda-calculus models of programming languages.* Ph.D. thesis, Massachusetts Institute of Technology.

[15] Koko Muroya & Makoto Hamana (2024): *Term Evaluation Systems with Refinements: First-Order, Second-Order, and Contextual Improvement*. In Jeremy Gibbons & Dale Miller, editors: *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15-17, 2024, Proceedings*, *Lecture Notes in Computer Science* 14659, Springer, pp. 31–61, doi:10.1007/978-981-97-2300-3_3. Available at `https://doi.org/10.1007/978-981-97-2300-3_3`.

[16] Grzegorz Rozenberg (1997): *Handbook of Graph Grammars and Computing by Graph Transformation*. WORLD SCIENTIFIC, doi:10.1142/3303. arXiv:https://www.worldscientific.com/doi/pdf/10.1142/3303.

[17] David Sands (1996): *Total Correctness by Local Improvement in the Transformation of Functional Programs*. *ACM Trans. Program. Lang. Syst.* 18(2), p. 175–234, doi:10.1145/227699.227716.