Towards an Embedding of Graph Transformation in Intuitionistic Linear Logic

Paolo Torrini & Reiko Heckel
Department of Computer Science, University of Leicester
{pt95,reiko}@mcs.le.ac.uk

Linear logics have been shown to be able to embed both rewriting-based approaches and process calculi in a single, declarative framework. In this paper we are exploring the embedding of double-pushout graph transformations into quantified linear logic, leading to a Curry-Howard style isomorphism between graphs / transformations and formulas / proof terms. With linear implication representing rules and reachability of graphs, and the tensor modelling parallel composition of graphs / transformations, we obtain a language able to encode graph transformation systems and their computations as well as reason about their properties.

1 Introduction

Graphs are among the simplest and most universal models for a variety of systems, not just in computer science, but throughout engineering and life sciences. When systems evolve, we are generally interested in the way they change, to predict, support, or react to evolution. Graph transformations combine the idea of graphs, as a universal modelling paradigm, with a rule-based approach to specify the evolution of systems. But while graph transformations are specified at the level of visual rules, these specifications are still operational rather than declarative. To reason about their properties at a realisation-independent level, a logics-based representation is desirable.

Intuitionistic linear logic (ILL) allows us to reason on concurrent processes at a level of abstraction which can vary from statements about individual steps to the overall effect of a longer computation. Unlike operational formalisms, linear logics are not bound to any particular programming or modelling paradigm and thus have a potential for integrating and comparing different such paradigms through embeddings [Gir87, Abr93].

In this paper we propose an embedding of graph transformation systems (GTS) based on the double-pushout approach (DPO) [EEPT06] in a variant of quantified intuitionistic linear logic (QILL). The DPO approach is arguably the most mature of the mathematically-founded approaches to graph transformation, with a rich theory of concurrency comparable to (and inspired by) those of place-transition Petri nets and term rewriting systems.

What makes ILL well applicable to GTS is the handling of resources and the way this allows for expressing creation/deletion of graph components. However, meeting the DPO conditions (identification condition and dangling edge condition) is not straightforward, and complicates the quantification aspect. Our logic translation of DPO GTS uses a non-standard quantifier, operationally defined, that allows us to associate names to variables, and therefore to represent graph morphisms in terms of substitution, compatibly with the DPO application conditions.

Preliminary Report. Final version to appear in:
ICE 2009

Types for restriction have been investigated in [Pit01, CC04] in the wider context of nominal logic, where names can be treated as bindable atoms. This approach involves a quite sophisticated set-theoretical foundation. We will rather use non-quantifiable constants, treated as linear resources, to which individual variables may explicitly refer.

QILL is based on linear λ-calculus [BBPH93, CP02, Pfe02], and can be obtained by adding to ILL the typed quantifiers $\hat{\exists}, \forall$. The resource-bound existential $\hat{\exists}$ has a separating character (quite different from that of intensional quantifiers in [Pym02]) — a resource is associated to each variable. It also has a freshness character, as implied by one of the introduction rule hypothesis. We use it to type the λ-calculus translation of restriction as we have it in our algebraic account of DPO. Standard universal quantification $\forall$ is used to represent DPO rule interfaces, translated by λ-abstraction.

Summarising, the QILL encoding of graph expressions is based on predicates for edges and variables for node names, the neutral element type $\mathbf{1}$ for the empty graph, the tensor product $\otimes$ for parallel composition and $\hat{\exists}$ for node name restriction. Transformation rules can be encoded by using linear implication $\multimap$ to express transformation, and $\forall$ to represent interfaces. Linear implication can be further associated to reachability. The unlimitedness constructor $!$ can be used to express the potentially unbounded applicability of transformation rules.

Our translation relies on a preliminary algebraic presentation of DPO-GTS in terms of an SHR-style formalism [FHL+06], which gives us syntactic notions of graph expression and transformation rule. We use a constructive approach — translating algebraic expressions to linear λ-calculus, so that component identity is represented in the proof-terms, whereas typing information and connectivity is represented in the logic formula. We can obtain a Curry-Howard style isomorphism between graph expressions and a subset of typing derivations, and between graphs and a subset of logic formulas (graph formulas) modulo linear equivalence. This can be extended to a mapping from GTS runs into typing derivations, and from reachable graphs into logic formulas. This approach offers the possibility of applying goal-directed proof-methods to the verification of well-formedness and reachability properties in GTS. Goal-directed methods can be useful insofar as they allow to focus proof attempts on the property to be verified, rather than exploring systematically the problem state space [MS07, DSB06].

2 Basic concepts and intuition

We introduce the main ideas behind the approach we are working on, before getting further into technical details.

2.1 Hypergraphs and their Transformations

Graph transformations can be defined on a variety of graph structures, including simple edge or node labelled graphs, attributed or typed graphs, etc. In this paper we prefer typed hypergraphs, their n-ary hyperedges to be presented as predicates in the logic.

A hypergraph (V, E, s) consists of a set V of vertices, a set E of hyperedges and a function $\mathsf{s} : E \to V^*$ assigning each edge a sequence of vertices in V. A morphism of hypergraphs is a pair of functions $\phi_V : V_1 \to V_2$ and $\phi_E : E_1 \to E_2$ that preserve the assignments of nodes, that is, $\phi_V^* \circ \mathsf{s}_1 = \mathsf{s}_2 \circ \phi_E$.

Typed hypergraphs are defined in analogy to typed graphs. Fixing a type hypergraph $TG = (\mathcal{V}, \mathcal{E}, \mathsf{ar})$ we establish sets of node types $\mathcal{V}$ and edge types $\mathcal{E}$ as well as defining the arity $\mathsf{ar}(a)$ of each edge type $a \in \mathcal{E}$ as a sequence of node types. A TG-typed hypergraph is a pair $(HG, type)$ of a hypergraph HG and a morphism $type : HG \to TG$. A TG-typed hypergraph morphism $f : (HG_1, type_1) \to (HG_2, type_2)$ is a hypergraph morphism $f : HG_1 \to HG_2$ such that $type_2 \circ f = type_1$.

A *graph transformation rule* is a span of injective hypergraph morphisms $s = (L \xleftarrow{l} K \xrightarrow{r} R)$, called a *rule span*. A hypergraph transformation system (GTS) $\mathcal{G} = \langle TG, P, \pi, G_0 \rangle$ consists of a type hypergraph TG, a set P of rule names, a function mapping each rule name p to a rule span $\pi(p)$, and an initial TG-typed hypergraph G_0.

A *direct transformation* $G \overset{p,m}{\Longrightarrow} H$ is given by a *double-pushout (DPO) diagram* as shown below, where (1), (2) are pushouts and top and bottom are rule spans. If we are not interested in the match and/or rule of the transformation we will write $G \overset{p}{\Longrightarrow} H$ or just $G \Longrightarrow H$.

For a GTS $\mathcal{G} = \langle TG, P, \pi, G_0 \rangle$, a derivation $G_0 \Longrightarrow G_n$ in $\mathcal{G}$ is a sequence of direct transformations $G_0 \overset{r_1}{\Longrightarrow} G_1 \overset{r_2}{\Longrightarrow} \cdots \overset{r_n}{\Longrightarrow} G_n$ using the rules in $\mathcal{G}$. The set of all hypergraphs reachable from G_0 via derivations in $\mathcal{G}$ is denoted by $\mathcal{R}_{\mathcal{G}}$.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] \\
G & D \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}$$

Intuitively, the left-hand side L contains the structures that must be present for an application of the rule, the right-hand side R those that are present afterwards, and the gluing graph K specifies the "gluing items", i.e., the objects which are read during application, but are not consumed.

Operationally speaking, the transformation is performed in two steps. First, we delete all the elements in G that are in the image of $L \setminus l(K)$ leading to the left-hand side pushout (1) and the intermediate graph D. Then, a copy of $L \setminus l(K)$ is added to D, leading to the derived graph H via the pushout (2).

It is important to point out that the first step (deletion) is only defined if a built-in application condition, the so-called gluing condition, is satisfied by the match m. This condition, which characterises the existence of pushout (1) above, is usually presented in two parts.

Identification condition: Elements of L that are meant to be deleted are not shared with any other elements, i.e., for all $x \in L \setminus l(K)$, $m(x) = m(y)$ implies $x = y$.

Dangling condition: Nodes that are to be deleted must not be connected to edges in G, unless they already occur in L, i.e., for all $v \in V_G$ such that $v \in m_V(L_V)$, if there exists $e \in E_G$ such that $\mathsf{s}(e) = v_1 \ldots v \ldots v_n$, then $e \in m_E(L_E)$.

The first condition guarantees two intuitively separate properties of the approach: First, nodes and edges that are deleted by the rule are treated as resources, i.e., m is injective on $L \setminus l(K)$. Second, there must not be conflicts between deletion and preservation, i.e., $m(L \setminus l(K))$ and $m(l(K)$ are disjoint.

The second condition ensures that after the deletion of nodes, the remaining structure is still a graph and does not contain edges short of a node. It is the first condition which makes linear logic so attractive for graph transformation. Crucially, it is also reflected in the notion of concurrency of the approach, where items that are deleted cannot be shared between concurrent transformations.

There is a second, more declarative interpretation of the DPO diagram as defining a rewrite relation over graphs. Two graphs G, H are in this relation $G \overset{p}{\Longrightarrow} H$ iff there exists a morphism $d : K \to D$ from the interface graph of the rule such that G is the pushout object of square (1) and H that of square (2) in the diagram above. In our algebraic presentation we will adopt this more declarative view.

As terms are often considered up to renaming of variables, it is common to abstract from the identity of nodes and hyperedges considering hypergraphs up to isomorphism. However, in order to be able to compose graphs by gluing them along common nodes, these have to be identifiable. Such potential gluing points are therefore kept as the *interface* of a hypergraph, a set of nodes I embedded into HG by a morphism $i : I \to HG$.

An abstract hypergraph $i : I \to [HG]$ is then given by the isomorphism class $\{i' : I \to HG' \mid \exists$ isomorphism $j : HG \to HG'$ such that $j \circ i = i'\}$.

If we restrict ourselves to rules with interfaces that are discrete (i.e., containing only nodes, but no edges.), a rule can be represented as a pair of hypergraphs with a shared interface I, i.e., $\Lambda I.L \Longrightarrow R$, such that the set of nodes I is a subgraph of both L, R. This restriction does not affect expressivity in describing individual transformations because edges can be deleted and recreated, but it reduces the level concurrency. In particular, concurrent transformation steps can no longer share edges because only items that are preserved by both rules can be accessed concurrently.

2.2 Linear logic

In terms of sequent calculus, ILL can be obtained from intuitionistic logic by restricting the application of standard structural rules *weakening* and *contraction*, thus making it possible to interpret premises as limited resources. Standard logic reasoning about unlimited resources can be recovered via the unlimitedness operator !. It is possible to interpret ILL formulas as partial states and express transitions in terms of consequence relation [CS06]. Tensor product ($\otimes$) can be used to represent parallel composition, as an assembling operation, additive conjunction (&) to represent non-deterministic choice as a composition of alternatives, and linear implication ($\multimap$) to express reachability.

ILL has an algebraic interpretation based on quantales and a categorical one based on symmetric monoidal closed categories [BBPH93], it has interpretations into Petri-nets, and for its $\vee$-free fragment, it has a comparatively natural Kripke-style semantics based on a ternary relation [IH01] — in common with relevant logics. ILL can be extended with quantifiers and enriched by adding proof terms, thus obtaining *linear λ-calculus* [BBPH93, Pfe02], where linear λ-abstraction and linear application require that the abstraction/application term is used only once. We are going to rely on an operational semantics in terms of natural deduction rules, following [Pfe02].

Derivation, represented as a sequent, can be formalised in terms of natural deduction, based on introduction/elimination rules closely related to the constructor/destructor duality in recursive datatypes [TS00]. Natural deduction rules fit well with forward application — from premises to conclusions. Proof normalisation guarantees modularity, meaning that detours in proofs can be avoided, i.e. one does not need to introduce a constructor thereafter to eliminate it. Proof normalisation shows that introducing a constructor brings nothing more than what it is taken away by eliminating it.

In order to check whether a sequent represents a derivation, or in our case, to check whether a graph expression is derivable from its context — either in the sense of well-formedness or in that of reachability — it can be useful to build proofs backward from the goal sequents. This is the idea behind sequent calculus. Sequent calculus rules fit in well with backward application and thus with goal-directed proof-search. Cut elimination is proof normalisation for sequent calculus, and it has essentially the same meaning [TS00]. Backward reasoning can also be used to synthesise proof-terms from their specified types. The possibility to integrate the constructive aspect (correctness by construction) with the synthesis one (correctness by design) is indeed one of the aspects that has made Curry-Howard style formalisms an important research topic in system verification [MS07].

2.3 GTS in QILL

We can represent GTS by relying on a constructive presentation of QILL, interpreting the application of derived rules as steps in constructing graph expressions and in transforming them. We look at the relationship between the structural congruence defined over the algebraic expressions ($\equiv$) and linear equivalence defined in the logic ($\hat{=}$). We are also interested in defining a bisimulation relation between the derivation of graph transformations in the logic and well-formed transformations in the algebraic formalism.

Graphs can be represented by formulas of form $\hat{\exists}\overline{x:A}.L_1\,(\overline{x}_1)\otimes\ldots\otimes L_k\,(\overline{x}_k)$ where $\overline{x:A}$ is a sequence $x_1 : A_1,\ldots,x_j : A_j$ of typed variables and $\overline{x}_1,\ldots,\overline{x}_k \subseteq \overline{x}$. A DPO rule (we consider rules with interfaces made only of nodes) can be represented as $\forall\overline{x:A}.\alpha \multimap \beta$ where α,β are graph expressions.

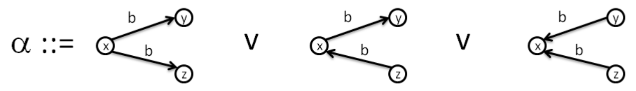
A sequent $G_0,P_1,\ldots,P_k \Vdash G_1$ can express that graph G_1 is reachable from the initial graph G_0 by applying rules

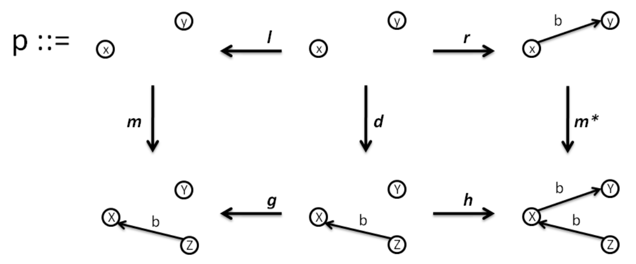
$$P_1 = \forall\overline{x}_1.\alpha_1 \multimap \beta_1,\ \ldots,\ P_k = \forall\overline{x}_k.\alpha_k \multimap \beta_k$$

abstracting away from the application order, each occurrence resulting into a transformation step. A sequent $G_0,!P_1,\ldots,!P_k \Vdash G_1$ can express that G_1 is reachable from G_0 by the same rules, regardless of whether or how many times they must be applied.

The parallel applicability of rules $\forall\overline{x}_1.\alpha_1 \multimap \beta_1, \forall\overline{x}_2.\alpha_2 \multimap \beta_2$ can be represented as applicability of $\forall\overline{x}_1,\overline{x}_2.\alpha_1\otimes\alpha_2 \multimap \beta_1\otimes\beta_2$.

Logic formulas can also be used also to represent abstractly graphs that have specific properties — such as matching certain patterns. Additive conjunction (&) can then be used to express choice, and additive disjunction ($\vee$) to express non-deterministic outcome — as from quantale-based interpretations of ILL [Abr93]. The formula $G_1\&G_2$ can represent a graph that can match two alternative patterns — hence representing a potential situation of conflict in rule application. The formula $G_1 \vee G_2$ represents a graph that may have been obtained in two different ways — hence a situation of non-determinism.





Negative constraints can be expressed using the intuitionistic-style negation $\neg$. The formula $\neg\alpha$ expresses the fact that α must never be reached — in the sense that reaching it implies the system contains an error. Whereas, in a weaker sense, the system satisfies the constraint if α does not follow from the specification. To make an example (see figure below), given

$$\alpha =_{df} \hat{\exists} xyz : A.(b(x,y)\otimes b(x,z)) \vee (b(x,y)\otimes b(z,x)) \vee (b(y,x)\otimes b(z,x))$$

the formula $\neg\alpha$ can be used to express a negative constraint stating that in the system there must be no element of type A which is bound with two distinct ones (graphically represented in the upper part of the picture). The transformation rule (in the same picture) can be represented with $\forall xy : A.\mathbf{1} \multimap b(x,y)$, and the initial graph with $\hat{\exists} xyz : A.b(z,x)$. These two formulas specify our system. The graph transformation determined by the application of the rule to the initial graph (single possibility) can be expressed in terms of logic consequence as follows

$$p =_{df} \hat{\exists} xyz : A.\ b(z,x), \forall x_1 x_2 : A.\ \mathbf{1} \multimap b(x_1,x_2) \Vdash \hat{\exists} xyz : A.\ b(z,x)\otimes b(x,y)$$

If we add the constraint $\neg\alpha$ to the premises, a contradiction follows, since α already follows from the specification. In general, a system including a negative constraint is consistent only if there is no reachable state which violates it.

3 An algebraic presentation of DPO transformation of hypergraphs

Let V be an infinite set of typed node names $n_1, n_2, \ldots$ typed in $\mathcal{V}$, and E an infinite set of typed edge names $e_1, e_2, \ldots$ typed in $\mathcal{E}$, as before. In the following, we assume typing to be implicit — each element associated to its type as aforementioned. When making type explicit, we use indexed A and B for node types, and indexed L for edge types. We introduce a notion of *ground constituent*

$$G = \ e(n_1,\ldots,n_k) \mid \mathsf{Nil} \mid \ G_1 \parallel G_2$$

where $e(n_1,\ldots,n_k)$ is an edge component (with $type(e) = L_e(A_1,\ldots,A_k)$), Nil for the empty graph and $G_1 \parallel G_2$ for the parallel composition of components G_1 and G_2. A *ground expression* is a pair $X \vDash G$ where G is a ground constituent and $X \subseteq V$ contains all the nodes in G. We say that a ground constituent is *normal* when either it is Nil or it does not contain Nil.

3.1 Graph expressions

We introduce a notion of *constituent*

$$C = e(n_1, \dots, n_k) \mid \mathsf{Nil} \mid C_1 \parallel C_2 \mid \nu n.C$$

that extend ground constituents by including restriction (ν), used to preserve distinction between nodes while allowing for renaming. We say that a constituent is *normal* whenever it has form $\nu\overline{n}.G$, where $\nu\overline{n}. = \nu n_1 \dots . \nu n_j.$ is the the prefix, and G is a normal ground constituent.

Given a constituent C, the *ground components* of C are the nodes and the edge components that occur in C. $fn(C)$ are the free nodes (unrestricted), $bn(C)$ are the bound nodes (restricted), and the set of all nodes is $n(C) =_{df} fn(C) \cup bn(C)$. We denote by $cn(C)$ the connected nodes of C, i.e. those which occur in ground components of C. $ibn(C) =_{df} bn(C)/cn(C)$ are the isolated bound nodes of C. C is a *closed* constituent when it has no free nodes.

A *graph expression* is a pair $E = X \vDash C$ where $X \subseteq V$ and C a constituent such that $fn(C) \subseteq X$. We call X the *interface* of E, or the free nodes of E. The nodes of E are $n(E) =_{df} X \cup bn(C)$. The isolated free nodes are $ifn(E) =_{df} X/fn(C)$. The isolated nodes of E are $i(E) =_{df} ifn(E) \cup ibn(C)$. In general, $X = fn(E) =_{df} ifn(E) \cup fn(C)$, and $n(E) = i(E) \cup cn(C)$. We say that the graph expression is ground (normal) whenever C is ground (normal), i.e. with $bn(C) = \emptyset$, *weakly closed* whenever C is closed, i.e. whenever $fn(C) = \emptyset$. We say that E is *closed* whenever $X = \emptyset$. For simplicity, we are going to identify closed graph expressions with their constituents.

Structural congruence between two graph expressions, written $X \vDash C \equiv Y \vDash C'$, holds iff $X = Y$ and $C \equiv C'$, where $\equiv$ is defined over constituents according to the following axioms.

- The parallel operator $\parallel$ is associative and commutative, with Nil as neutral element.
- $\nu n.\ C \equiv \nu m.\ C[m/n]$, if m does not occur free in C.
  $\nu n.\nu m.C \equiv \nu m.\nu n.C$
  $\nu n.(C_1 \parallel C_2) \equiv C_1 \parallel (\nu n.C_2)$ if n does not occur free in C_1

We do not require $\nu n.C \equiv C$ for n not occurring free in C — we will also say that we do not require ν to satisfy η-equivalence. We are going to use this feature in order to keep isolated nodes into account.

For $E = X \vDash C$, we denote by $ec(E)$ the edge components of C, and by $gc(E) = n(E) \cup ec(E)$ the set of the ground components of E. It is not difficult to see the following.

Obs. 1 Two graph expressions E_1, E_2 are congruent if and only if $fn(E_1) = fn(E_2)$ and there is a renaming σ of $bn(E_1)$ such that $gc(E_1)\sigma = gc(E_2)$.

One can also see quite easily that every graph expression is congruent to a normal one, and that normal expressions that are congruent are the same up to reordering of prefix elements and ground components.

An abstract hypergraph in the sense of section 2.1 is represented by an equivalence class of graph expressions up to structural congruence. Intuitively, the free names correspond to nodes in the interface while bound names represent internal nodes.

We will often refer to these equivalence classes as graphs, while reserving the term hypergraph for the real thing. We say that a graph expression represents a graph (is a *representative* of the graph) when it belongs to the equivalence class. A graph is (weakly) closed whenever it is represented by a (weakly) closed graph expression. Clearly, every closed graph has a closed normal representative.

3.2 Graphs and transformation rules

For C_1, C_2 closed graph expression, $C_1 \Longrightarrow C_2$ denotes the transformation that goes from C_1 to C_2. In order to represent transformation rules we need to handle the matching of free nodes. We then introduce variables $x, y, \ldots$ ranging over nodes, substitution of nodes for free variables $C[m/x]$ (where m does not become bound), variable binding (by Λ) and application.

Given two graph expressions $E_1 = K \vDash L$ and $E_2 = K \vDash R$, sharing the same interface and no free isolated nodes, we represent the transformation rule $\pi(p) = L \xleftarrow{l} K \xrightarrow{r} R$ by the *rule expression* $\Lambda\overline{x}.L \overset{p}{\Longrightarrow} R$ where $\overline{x} = x_1, \ldots, x_k$ is a sequence of variables bijectively associated to the node names in K (rule interface). Essentially, we represent rules by replacing free node sets with sets of bound variables — therefore rule expressions contain neither free variables nor free nodes.

Given a closed graph representative G, a match for $\pi(p)$ in G is determined by a graph homomorphism $d : K \rightarrow n(G)$ which determines the morphisms $m_l : bn(L) \rightarrow n(G)$ and $m_e : ec(L) \rightarrow ec(G)$ (d being the defining components of $m : L \rightarrow G$), as well as $m^*_r : bn(R) \rightarrow n(H)$ and $m^*_e : ec(R) \rightarrow ec(H)$ (d being the defining components of $m^* : R \rightarrow H$).

The dangling edge condition requires that whenever n is in the domain of m_l and edge component c depends on n, then c is in the domain of m_e. The identification condition requires that m_l and m_e are injective, and that the images of d and m_l are disjoint. The injectivity of m^*_r and m^*_e follows, as well as the disjointness of the images of d and m^*_r.

Here the injective components can be represented in terms of inclusion, whereas the interface morphism d can be represented in terms of substitution, i.e. we represent d by $[\overline{n} \xleftarrow{d} \overline{x}] = [x_1/n_1, \ldots x_k/n_k]$, where $\overline{n} = \{n_1, \ldots, n_k\} \subseteq n(G)$. The following operational rule (application schema) represents the application of the transformation rule p at match m (i.e. interface match d)

$$\frac{\pi(p) = \Lambda\overline{x}.L \overset{p}{\Longrightarrow} R \quad G \equiv \nu\overline{n}.L[\overline{n} \xleftarrow{d} \overline{x}] \parallel C \quad H \equiv \nu\overline{n}.R[\overline{n} \xleftarrow{d} \overline{x}] \parallel C}{G \overset{p,d}{\Longrightarrow} H} \; \overset{\langle p,m\rangle}{\Longrightarrow}$$

where G is a closed graph expression — and therefore H is, too.

Obs. 2 The application schema satisfies the DPO conditions.

Let $L' = L[\overline{n} \xleftarrow{d} \overline{x}]$, $R' = R[\overline{n} \xleftarrow{d} \overline{x}]$. The definition and the injectivity of component morphisms m_l, m_e, m^*_r, m^*_e follows from the inclusion of L' and R' as subexpressions in refactorings of G and H, respectively. The disjointness condition is guaranteed by the fact that the variables in $\overline{x}$ are substituted with nodes that are free in L' and R', and therefore cannot be identified with bound nodes in either constituent. The dangling edge condition is guaranteed by the fact that, for each node $n \in bn(L')$, edge components depending on n can only be in $ec(L')$.

4 Linear λ-calculus

A typing expression has form $N :: \alpha$ where N is the proof-term and α is the type (a logic formula). A typing derivation can be represented as a sequent with a context of typed variables on the left and a typing expression on the right of the turnstile. A sequent represents a typing derivation when it can be derived by means of typing rules.

We use two-entry sequents of form $\Gamma;\Delta \vdash N :: \alpha$, where Δ is a multiset of linear ones (linear context), denoted $u,v,\ldots$, $\Gamma = (\Sigma,\Omega)$ is a pair of multisets, Σ is an environment of individual variables, denoted $x,y,\ldots$, Ω is a multiset of unbounded resources (linear context), denoted $p,q,\ldots$, and $\vdash$ represents derivability. We use sequence notation — modulo permutation and associativity, and $\cdot$ for the empty multiset. When we "forget" about proof terms we are left with logic formulas and the consequence relation — then we use $\Vdash$ instead of $\vdash$. We say that $\Phi \subseteq \Delta$ is a multiset of constituents, denoted $c,d,\ldots$, and that $\Theta \subseteq$ is a multiset of primitive node names, denoted $m,n,\ldots$.

The typing expression for a node n name has form $n :: \alpha \downharpoonleft N$, where N is a term of type α. $\alpha \downharpoonleft N$ lifts α to a type of propositional sort, and denotes a naming reference of n to N. It should be possible to define this type in terms of freshness quantification, but here it is not really needed, since names are taken as primitive — hence introduced by axiom (with the condition that the naming term is well typed). Essentially, the rules for $\hat{\exists}$ allow us to hide names, and replace them with variables. We use a notion of syntactic equality $=$ over types, stronger than linear equivalence $\hat{\equiv}$, to express a weak freshness constraint on $\hat{\exists}$ (in the right introduction rule).

We use $\hat{\lambda}$ to denote linear abstraction, in order to distinguish it from the ordinary one (λ), though the difference between the two is actually determined by whether the abstraction variable is linear or unbounded. Similarly, whether λ is typed by $\forall$ depends on whether the abstraction is over an individual variable. We assume standard β-reduction rules [BBPH93, Pfe02].

Each type constructor is associated to a term constructor by right introduction. The natural deduction system is given by the axioms (*Eq*, *Id*, *UId*, *NId*, *FId*) together with the *I* (introduction) and *E* (elimination) rules. The sequent calculus system is given by the same axioms, rules *Copy*, *Cut* and *Cut′*, together with the *R* (right introduction, same as *I*) and *L* (left introduction) rules. We use *let* expressions in order to allow for abstraction with patterns (as needed for some of the rules).

Normalisation should be provable for natural deduction, and cut elimination for sequent calculus (for both *Cut* and *Cut′*). The main difference with the system in [Pfe02] is the quantifiers $\hat{\exists}$, and it does not look like its rules are particularly problematic from the point of view of proving cut elimination (given [Pfe02] is an unpublished report, we cannot make any stronger claim at the moment).

4.1 Proof systems

$\alpha = A \mid L(N_1,\ldots,N_n) \mid \mathbf{1} \mid \alpha_1 \otimes \alpha_2 \mid \alpha_1 \multimap \alpha_2 \mid !\alpha_1 \mid \top \mid \bot \mid \alpha_1 \& \alpha_2 \mid \alpha \to \beta \mid \alpha \vee \beta \mid \forall x : \beta.\alpha \mid \hat{\exists} x : \beta.\alpha \mid \alpha \downharpoonleft N \mid$
$\alpha = \alpha$

$M = x \mid p \mid u \mid c \mid n \mid \mathsf{nil} \mid N_1 \otimes N_2 \mid \hat{\varepsilon}(N_1|N_2).N_3 \mid \lambda x.N \mid \lambda p.N \mid \hat{\lambda} u.N \mid N_1 \hat{\ } N_2 \mid N_1 N_2 \mid \mathsf{error}^{\alpha}\ M \mid$
$\langle N_1, N_2 \rangle \mid \langle\rangle \mid \mathsf{fst}\ N \mid \mathsf{snd}\ N \mid \mathsf{case}\ N\ \mathsf{of}\ P_1.N_1; P_2.N_2 \mid \mathsf{inr}^{\alpha}\ N \mid \mathsf{inl}^{\alpha}\ N \mid \mathsf{id}_{\alpha}$

$$\mathsf{let}\ P = N_1\ \mathsf{in}\ N_2 \ =_{df}\ (\lambda P.N_2)N_1 \qquad \text{where } P \text{ is a variable pattern}$$

$$\alpha \hat{\equiv} \beta \ =_{df}\ (\alpha \multimap \beta)\&(\beta \multimap \alpha) \qquad \neg\alpha \ =_{df}\ \alpha \multimap \bot \qquad \alpha\#(x,N) \ =_{df}\ (\alpha[N/x])[x/N] = \alpha$$

$$\frac{}{\Gamma; u :: \alpha \vdash u :: \alpha}\ Id \qquad \frac{}{\Gamma, x :: \alpha; \cdot \vdash x :: \alpha}\ UId$$

$$\frac{}{\Gamma, p :: \alpha; \cdot \vdash p :: \alpha}\ FId \qquad \frac{}{\Gamma; \cdot \vdash \mathsf{id}_\alpha :: \alpha = \alpha}\ Eq$$

$$\frac{\Gamma, p :: \alpha; u :: \alpha, \Delta \vdash N :: \gamma}{\Gamma, p :: \alpha; \Delta \vdash \mathsf{let}\ u = \mathsf{copy}(p)\ \mathsf{in}\ N :: \gamma}\ Copy$$

$$\frac{\Gamma; \Delta \vdash N :: \alpha \quad \Gamma; u :: \alpha, \Delta' \vdash M :: \beta}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ N = u\ \mathsf{in}\ M :: \beta}\ Cut \quad \frac{\Gamma; \cdot \vdash N :: \alpha \quad \Gamma, p :: \alpha; \Delta \vdash M :: \beta}{\Gamma; \Delta \vdash \mathsf{let}\ N = p\ \mathsf{in}\ M :: \beta}\ Cut'$$

$$\frac{\Gamma; \Delta_1 \vdash M :: \alpha \quad \Gamma; \Delta_2 \vdash N :: \beta}{\Gamma; \Delta_1, \Delta_2 \vdash M \otimes N :: \alpha \otimes \beta}\ \otimes I/R \qquad \frac{\Gamma; \Delta_1 \vdash M :: \alpha \otimes \beta \quad \Gamma; \Delta_2, u :: \alpha, v :: \beta \vdash N :: \gamma}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ u \otimes v = M\ \mathsf{in}\ N :: \gamma}\ \otimes E$$

$$\frac{\Gamma; \Delta, u :: \alpha, v :: \beta \vdash N :: \gamma}{\Gamma; \Delta, w :: \alpha \otimes \beta \vdash \mathsf{let}\ u \otimes v = w\ \mathsf{in}\ N :: \gamma}\ \otimes L \quad \frac{\Gamma; \Delta_1 \vdash M :: \alpha \quad \Gamma; \Delta_2, u :: \beta \vdash N :: \gamma}{\Gamma; \Delta_1, \Delta_2, v :: \alpha \multimap \beta \vdash \mathsf{let}\ u = v\hat{\ }N\ \mathsf{in}\ N :: \gamma}\ \multimap L$$

$$\frac{\Gamma; \Delta, u :: \alpha \vdash M :: \beta}{\Gamma; \Delta \vdash \hat{\lambda} u : \alpha.\ M :: \alpha \multimap \beta}\ \multimap I/R \quad \frac{\Gamma; \Delta_1 \vdash M :: \alpha \multimap \beta \quad \Gamma; \Delta_2 \vdash N :: \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash M\hat{\ }N :: \beta}\ \multimap E$$

$$\frac{}{\Gamma; \cdot \vdash \mathsf{nil} :: \mathbf{1}}\ \mathbf{1}I/R \qquad \frac{\Gamma; \Delta \vdash M :: \mathbf{1} \quad \Gamma; \Delta' \vdash N :: \alpha}{\Gamma; \Delta, \Delta' \vdash \mathsf{let\ nil} = M\ \mathsf{in}\ N :: \alpha}\ \mathbf{1}E$$

$$\frac{\Gamma; \Delta \vdash M :: \alpha \quad \Gamma; \Delta \vdash N :: \beta}{\Gamma; \Delta \vdash \langle M, N \rangle :: \alpha \& \beta}\ \& I/R \quad \frac{\Gamma; \Delta \vdash N :: \alpha}{\Gamma; \Delta, u :: \mathbf{1} \vdash \mathsf{let\ nil} = u\ \mathsf{in}\ N :: \alpha}\ \mathbf{1}L$$

$$\frac{\Gamma; \Delta \vdash M :: \alpha \& \beta}{\Gamma; \Delta \vdash \mathsf{fst}\ M :: \alpha}\ \& E1 \qquad \frac{\Gamma; \Delta \vdash M :: \alpha \& \beta}{\Gamma; \Delta \vdash \mathsf{snd}\ M :: \beta}\ \& E2$$

$$\frac{\Gamma; \Delta, v :: \alpha \vdash N :: \gamma}{\Gamma; \Delta, u :: \alpha \& \beta \vdash \mathsf{let}\ v = \mathsf{fst}\ u\ \mathsf{in}\ N :: \gamma}\ \& L1 \quad \frac{\Gamma; \Delta, v :: \beta \vdash N :: \gamma}{\Gamma; \Delta, u :: \alpha \& \beta \vdash \mathsf{let}\ v = \mathsf{fst}\ u\ \mathsf{in}\ N :: \gamma}\ \& L2$$

$$\frac{\Gamma; \Delta \vdash \mathsf{inl}^\beta M :: \alpha \vee \beta}{\Gamma; \Delta \vdash M :: \alpha}\ \vee I1/R1 \qquad \frac{\Gamma; \Delta \vdash \mathsf{inr}^\alpha M :: \alpha \vee \beta}{\Gamma; \Delta \vdash M :: \beta}\ \vee I2/R2$$

$$\frac{\Gamma; \Delta \vdash M :: \alpha \vee \beta \quad \Gamma; \Delta', u :: \alpha \vdash N_1 :: \gamma \quad \Delta', v :: \beta \vdash N_2 :: \gamma}{\Gamma; \Delta, \Delta' \vdash \mathsf{case}\ M\ \mathsf{of\ inl}\ u.\ N_1;\ \mathsf{inr}\ v.\ N_2\ :: \gamma}\ \vee E$$

$$\frac{\Gamma; \Delta, u_1 :: \alpha \vdash N_1 :: \gamma \quad \Gamma; \Delta, u_2 :: \beta \vdash N_2 :: \gamma}{\Gamma; \Delta, v :: \alpha \vee \beta \vdash \mathsf{case}\ v\ \mathsf{of\ inl}\ u_1.\ N_1;\ \mathsf{inr}\ u_2.\ N_2\ :: \gamma}\ \vee L$$

$$\frac{}{\Gamma;\Delta \vdash \langle\rangle :: \top}\ \top I/R \qquad \frac{\Gamma;\Delta \vdash M :: \bot}{\Gamma;\Delta,\Delta' \vdash \mathsf{error}^{\alpha}\ M :: \alpha}\ \bot E$$

$$\frac{}{\Gamma;\Delta,u :: \bot \vdash \mathsf{error}^{\alpha}\ M :: \alpha}\ \bot L \quad \frac{\Gamma,p :: \alpha;\Delta \vdash N :: \beta}{\Gamma;\Delta,u :: !\alpha \vdash \mathsf{let}\ p = !u\ \mathsf{in}\ N :: \beta}\ !L$$

$$\frac{\Gamma;\cdot \vdash M :: \alpha}{\Gamma;\cdot \vdash !M :: !\alpha}\ !I/R \quad \frac{\Gamma;\Delta_1 \vdash M :: !\alpha \quad \Gamma,p :: \alpha;\Delta_2 \vdash N :: \beta}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ p = M\ \mathsf{in}\ N :: \beta}\ !E$$

$$\frac{\Gamma,p :: \alpha;\Delta \vdash M :: \beta}{\Gamma;\Delta \vdash \lambda p.\ M :: \alpha \to \beta} \to I/R \qquad \frac{\Gamma;\Delta \vdash M :: \alpha \to \beta \quad \Gamma;\cdot \vdash N :: \alpha}{\Gamma;\Delta \vdash MN :: \beta} \to E$$

$$\frac{\Gamma;\cdot \vdash M :: \alpha \quad \Gamma;v :: \beta,\Delta \vdash N :: \gamma}{\Gamma;\Delta,u :: \alpha \to \beta \vdash \mathsf{let}\ v = uM\ \mathsf{in}\ N :: \gamma} \to L \quad \frac{\Gamma;\cdot \vdash M :: \beta \quad \Gamma;\Delta,v :: \alpha[M/x] \vdash N :: \gamma}{\Gamma;\Delta,u :: \forall x : \beta.\alpha \vdash \mathsf{let}\ v = uM\ \mathsf{in}\ N :: \gamma}\ \forall L$$

$$\frac{\Gamma,x :: \beta;\Delta \vdash M :: \alpha}{\Gamma;\Delta \vdash \lambda x.\ M :: \forall x : \beta.\ \alpha}\ \forall I/R \quad \frac{\Gamma;\Delta \vdash M :: \forall x : \beta.\ \alpha \quad \Gamma;\cdot \vdash N :: \beta}{\Gamma;\Delta \vdash MN :: \alpha[N/x]}\ \forall E$$

Resource-bound quantifier

$$\frac{\Gamma;\Delta \vdash M :: \alpha[N/x] \quad \Gamma;\cdot \vdash N :: \beta \quad \Gamma;\Delta' \vdash n :: \beta\lfloor N \quad \Gamma,x :: \beta;\cdot \vdash \mathsf{id}_{\alpha} :: \alpha\#(x,N)}{\Gamma;\Delta,\Delta' \vdash \hat{\varepsilon}(n|N).M :: \hat{\exists} x : \beta.\alpha}\ \hat{\exists} I/R$$

$$\frac{\Gamma;\Delta_1 \vdash M :: \hat{\exists} x : \beta.\ \alpha \quad \Gamma,x :: \beta;\Delta_2,n :: \beta\lfloor x,v :: \alpha \vdash N :: \gamma}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ \hat{\varepsilon}(n|x).v = M\ \mathsf{in}\ N :: \gamma}\ \hat{\exists} E$$

$$\frac{\Gamma,x :: \beta;\Delta,n :: \beta\lfloor x,v :: \alpha \vdash N :: \gamma}{\Gamma;\Delta,w :: \hat{\exists} x : \beta.\alpha \vdash \mathsf{let}\ \hat{\varepsilon}(n|x).v = w\ \mathsf{in}\ N :: \gamma}\ \hat{\exists} L$$

4.2 DPO quantification

We have to deal with names and restriction, and we do this by treating names as individual variables (bindable by a quantifier) that refer to special linear resources — the nodes. We need an existential-like quantifier (i.e. distributing over the tensor) that, in contrast with standard ones, ensures that distinct bound variables in a formula never get instantiated with the same node — we will also say that the binder behaves injectively, i.e. that instantiations of sequences of bound variables are always injective mappings.

The rules for $\hat{\exists}$ ensure that the instantiation of two variables, requiring two constants, can never be derived from the instantiation of one — hence multiple instantiations behave injectively. The standard existential elimination rule is fine, since the instantiating term is required to be a fresh variable. The $\hat{\exists}$ introduction rule adds two constraints to the standard one. First — the instantiating term has an implicit reference to a node of corresponding type, which is used as a resource. Second — looking at $\hat{\exists}I$, we can see that the bound variable can only be instantiated with a term that does not occur free in the quantified formula. In fact, $(\alpha[N/x])[x/N] = \alpha$ whenever N does not occur free in $\hat{\exists}x.\alpha$. This also means that the quantified formula $\hat{\exists}x.\alpha$ is determined by the instance $\alpha[N/x]$ modulo renaming of bound variables. It is not difficult to see the following.

Obs. 3 (1) $\not\Vdash (\hat{\exists}x : \beta.\ \alpha(x,x)) \multimap \hat{\exists}xy : \beta.\ \alpha(x,y)$

the resource associated to x cannot suffice for x and y.

(2) $\not\Vdash \forall x : \beta.\ \beta\lfloor x \otimes \alpha(x,x) \multimap \hat{\exists}y : \beta.\alpha(y,x)$

y and x should be instantiated with the same term — but this is prevented by the freshness condition in $\hat{\exists}$ introduction

(3) $\not\Vdash (\hat{\exists}yx : \beta.\ \alpha_1(x) \otimes \alpha_2(x)) \multimap (\hat{\exists}x : \beta.\alpha_1(x)) \otimes \hat{\exists}x : \beta.\alpha_2(x)$

the two bound variables in the consequence require distinct resources and refer to distinct occurrences

Obs. 4 $\hat{\exists}$ satisfies properties of renaming, exchange and distribution over $\otimes$, i.e.

$\Vdash (\hat{\exists}x : \alpha.\beta(x)) \hat{=} (\hat{\exists}y : \alpha.\beta(y))$
$\Vdash (\hat{\exists}xy : \alpha.\gamma) \hat{=} (\hat{\exists}yx.\gamma)$
$\Vdash (\hat{\exists}x : \alpha.\beta \otimes \gamma(x)) \hat{=} (\beta \otimes \hat{\exists}x : \alpha.\gamma(x)) \qquad (x \text{ not in } \alpha)$

In general $\hat{\exists}$ does not satisfy η-equivalence, i.e. it cannot be proved that α is equivalent to $\hat{\exists}x.\ \alpha$ when x does not occur free in α (neither sense of linear implication holds).

We use $\hat{\varepsilon}$ as syntactic sugar for a restriction-like operator based on the linear interpretation of the existential quantifier [Pfe02], closely associated to the standard intuitionistic one [TS00], i.e.

$$\hat{\varepsilon}(n|N).M :: \hat{\exists}x : \alpha.\beta \ =_{df}\ !N \otimes M \otimes n$$

where neither N nor x occur in α. N depends on the non-linear context only, hence it can be replaced with $!N$. The linearity of $n :: \alpha$ ensures injectivity.

5 Linear encoding of GTS

We define a translation of graph expressions to typing derivations. Intuitively, the translation is based on a quite straightforward mapping of graph expressions into proof terms, with Nil mapped to nil, || to $\otimes$, and ν to $\hat{\varepsilon}$. However, we need to cope with two issues.

Formally, we need to distinguish nodes as ground components (node names) from node occurrences in constituents (node variables). Given $E = X \vDash C$, we can translate a node $n \in X$ with $type(n) = A$ as $n :: A\lfloor x$ (*typed node*), and the occurrences of n in C as $x_n :: A$, where A is an unbounded resource type (therefore equivalent to $!A$). We use indexed letters for free variables, informally assuming either x_n refers to name n, or x_i refers to n_i. For bound variables, the reference is implicit in the term $(\hat{\varepsilon}(n|x).N)$.

Semantically, it is more convenient to take edge components as primitive, rather than edges. In principle, we can introduce a notion of *edge interface* as linear resource, $e :: \forall x_1 : A_1,\ldots,x_k : A_k.L_e(x_1,\ldots x_k)$, translate an edge type $L_e(A_1,\ldots,A_k)$ as $\forall x_1 : A_1,\ldots,x_k : A_k.L_e(x_1,\ldots x_k)$, and a component $e(n_1,\dots,n_k)$ as $c_e = e\ x_1\ \dots x_k$. For all its functional clarity, however, the notion of edge interface is hard to place semantically in graphs. Therefore, we prefer to introduce the notion $c_e :: L(x_1,\dots x_n)$ of *typed edge component* as primitive, which can be translation of the original component under the assumptions $x_1 :: A_1,\dots x_k :: A_k$. Following this approach, component connectivity does not result from the term, rather from the type.

We call *graph formulas* those in the $\mathbf{1},\otimes,\hat{\exists},\downharpoonright$ fragment of the logic containing only primitive graph types (node and edge types). We say that a graph formula γ is in normal form whenever $\gamma = \hat{\exists}(\overline{x : A}).\ \alpha$, where either $\alpha = \mathbf{1}$ or $\alpha = L_1(\overline{x}_1) \otimes \dots \otimes L_k(\overline{x}_k)$, with $\overline{x :: A}$ a sequence of typed variables. The formula is closed if $\overline{x}_i \subseteq \overline{x}$ for each $1 \le i \le k$. A *graph context* is a multiset of typed nodes and typed edge components.

A *graph derivation* is a valid sequent $\Gamma;\Delta \vdash N :: \gamma$, where γ is a graph formula, Δ is a graph context, Γ coincides with the environment Σ, and N represents a normal derivation. A graph derivation uses only axioms and the introduction rules $\mathbf{1}I$, $\otimes I$, $\hat{\exists}I$.

In the following we define $[\![]\!]$ from graph expressions to typing derivations. Only right introduction rules are involved, so the difference between natural deduction and sequent calculus here does not matter. We use the notation $AxiomName\ [\Gamma;;\ Principal\,Formula]$ to abbreviate axiom instances and deduction rules with empty premises, and $RuleName\ [Premise;;\ list\ of\ Premises]$ to abbreviate the application of deduction rules to the given premises. We also define $MainType(\Gamma;\Delta \vdash N :: \alpha) = \alpha$, $MainTerm(\Gamma;\Delta \vdash N :: \alpha) = N$, and $LinearContext(\Gamma;\Delta \vdash N :: \alpha) = \Delta$. We assume Γ to coincide with the environment Σ.

Constituents

$$
\begin{array}{l}
[\![e_i(m,\dots,n) : L_i(A_m,\dots,A_n)]\!] =_{df} Id\ [\Gamma;;\quad c_i :: L_i(x_m,\dots,x_n)] \\
[\![\mathsf{Nil}]\!] =_{df} \mathbf{1}I\ [\Gamma] \\
[\![M \,||\, N]\!] =_{df} \otimes I\ [[\![M]\!];;\quad [\![N]\!]] \\
[\![\nu n : A.N]\!] =_{df} \hat{\exists}I\ [[\![N]\!];; \\
\quad UId\ [\Gamma;;\ x_n :: A];; \\
\qquad\qquad Id\ [\Gamma;;\ n :: A \downharpoonright x_n];; \\
\qquad\qquad \Gamma, y :: A; \cdot \vdash \mathsf{id} : MainType([\![N]\!])[y/x_n]\#(y,x_n)]
\end{array}
$$

Graph interfaces

$$
\begin{array}{l}
[\![n : A]\!] =_{df} Id\ [\Gamma, x :: A;;\quad n :: A \downharpoonright x] \\
[\![\{n : A\}]\!] =_{df} [\![n : A]\!] \\
[\![\{n_1 : A_1\} \cup X]\!] =_{df} \otimes I\ [[\![\{n_1 : A_1\}]\!];;\ [\![X]\!]]
\end{array}
$$

Graph expressions

$$
[\![X \vDash C]\!] =_{df} \otimes I\ [[\![X]\!]_I;;\ [\![C]\!]]
$$

5.1 Properties of the translation

In order to understand the relationship between the domain and the image of $[\![]\!]$, it is useful to consider the following induced mapping, taking graph expressions into QILL formulas ($[\![]\!]^T$), and into multisets of typing expressions corresponding to ground components ($[\![]\!]^C$).

Obs. 5 Let $[\![E]\!]^T = MainType[\![E]\!]$ and $[\![E]\!]^C = LinearContext[\![E]\!]$.

1) $[\![]\!]^T$ results in an extension of the original typing of nodes and edges, based, essentially, on the association of $\otimes$ with $\|$, $\mathbf{1}$ with Nil, and $\hat{\exists}$ with ν, where the free connected nodes are represented as free variables.

2) $[\![E]\!]^C = \Delta$ determines a bijection between Δ and $gc(E)$ — dependant types contain the information about basic graph types and component dependencies, whereas terms preserve component identity.

Prop. 1 There is an isomorphism between graph expressions and graph derivations.

It is not difficult to see that, for each E graph expression, $[\![E]\!] = \Gamma;\Delta \vdash N : \gamma$ defines a graph derivation. N is normal, Γ as required, $[\![E]\!]^T$ gives a graph formula, $[\![E]\!]^C$ a graph context. Vice-versa, for each graph derivation $\delta = \Gamma;\Delta \vdash N : \gamma$, one can define a graph expression E that has the structure of N, such that $[\![E]\!] = \delta$, relying on Obs. 5.

Prop. 2 There is an isomorphism between graphs and graph formulas modulo linear equivalence.

First we can see that, given graph expressions M,N, if $M \equiv N$ then $\Vdash [\![M]\!]^T \hat{=} [\![N]\!]^T$. This follows from the well-known monoidal characterisation of $\otimes$ and from Obs. 4.

On the other hand, whenever γ_1,γ_2 are graph formulas and $\Vdash \gamma_1 \hat{=} \gamma_2$, then for each graph expressions E_1,E_2 such that $\gamma_1 = [\![E_1]\!]^T$, $\gamma_2 = [\![E_2]\!]^T$, it holds $E_1 \equiv E_2$. Every graph formula has a derivation. From the hypothesis, it follows that there is a derivation $\delta_1 = \Gamma;\Delta \vdash N_1 :: \gamma_1$ iff there is a graph derivation $\delta_2 = \Gamma;\Delta \vdash N_2 :: \gamma_2$, and both can be chosen to be graph derivations. Therefore, from Prop. 1, there are graph expressions E_1,E_2 such that $[\![E_1]\!] = \delta_1, [\![E_2]\!] = \delta_2$. From Obs. 5(2), $gc(E_1) = gc(E_2)$. Since γ_1 and γ_2 are equivalent they share the same free variables, and so E_1 and E_2, by Obs. 5(1). Hence $E_1 \equiv E_2$, by Obs. 1.

The above propositions can be understood as stating that there is a Curry-Howard isomorphism between graph expressions and graph derivations on one side, and between graphs and QILL formulas modulo equivalence on the other. They can also be read as stating that our translation of graph expressions is adequate with respect to their congruence.

5.2 DPO transformation rules

We can now shift from congruence of graph expressions to reachability in a GTS, extending the translation to deal with graph transformation. We consider transformation up to isomorphism, and therefore we start from the type level, relying on Prop. 2 (i.e. we define directly the map $[\![]\!]^T$ from graph expressions to QILL formulas). We associate transformation to linear implication, and the binding of node variables in rule interfaces to universal quantification.

$$[\![M \Longrightarrow N]\!]^T =_{df} [\![M]\!]^T \multimap [\![N]\!]^T$$
$$[\![\Lambda x : A.N]\!]^T =_{df} \forall x : A.[\![N]\!]^T$$

Transformation rules are meant to be primitive in a GTS, therefore they can be introduced axiomatically (as we have done with nodes and edge components). They have to be introduced as unbounded resources, in order to account for their potentially unlimited applicability. Moreover, consistently with the extension of $[\![]\!]^T$, transformation rules are associated to closed formulas. For $\pi(p) = \Lambda\overline{x}.L \Longrightarrow R$

$$[\![\pi(p)]\!] \;=_{df}\; FId\,[\Gamma;; \quad p :: \forall\overline{x:A_x}.[\![L]\!]^T \multimap [\![R]\!]^T]$$

Reasoning up to isomorphism, it is appropriate to abstract from the derivations associated with the graph expressions in the algebraic definition of the rule, except for their types. A more concrete definition, in an extended language, would be

$$p =_{df} !\Lambda\overline{x}.\hat{\lambda}u.\mathsf{let}\ MainTerm([\![L]\!]) = u\ \mathsf{in}\ MainTerm([\![R]\!])$$

At an intuitive level, in terms of natural deduction, the application of rule p to a graph G involves deriving the matching subgraph L' from $gc(L') \subseteq gc(E)$. The application of p to L' can be expressed in terms of instantiation of the rule interface, corresponding to applications of the $\forall$ elimination rule, and to an application of the instantiated rule to L', corresponding to an application of the $\multimap$ elimination rule, resulting into R'. Finally, the resulting graph H can be derived from $gc(R') \cup (gc(E)/gc(L'))$.

The application of p to a closed graph formula $\alpha_G = \hat{\exists}\overline{y:A_y}.\beta_G$ determined by morphism m relies on the fact that the following rule is derivable, along the lines of the intuitive explanation just given (proof terms omitted)

$$\dfrac{\begin{array}{ll} \Gamma;\Delta \Vdash \forall\overline{x:A_x}.\alpha_L \;\multimap\; \alpha_R & \\ \Gamma;\cdot \Vdash \alpha_G \mathrel{\hat{\equiv}} \alpha_{G'} & \alpha_{G'} = \hat{\exists}\overline{z:A_z}.\alpha_L[\overline{z:A_z} \xleftarrow{d} \overline{x:A_x}] \otimes \alpha_C \\ \Gamma;\cdot \Vdash \alpha_H \mathrel{\hat{\equiv}} \alpha_{H'} & \alpha_{H'} = \hat{\exists}\overline{z:A_z}.\alpha_R[\overline{z:A_z} \xleftarrow{d} \overline{x:A_x}] \otimes \alpha_C \end{array}}{\Gamma;\Delta \Vdash \alpha_G \multimap \alpha_H} \;\overset{\langle p,m\rangle}{\Longrightarrow}$$

where the interface morphism d associated with m is represented by the multiple substitution $[\overline{z:A_z} \xleftarrow{d} \overline{x:A_x}]$, with $\overline{z:A_z} \subseteq \overline{y:A_y}$ Clearly, this rule corresponds quite immediately to the algebraic one.

Obs. 6 The QILL application schema satisfies the DPO conditions.

The proof follows that of Obs. 2 and uses Obs. 4(1,2,3). Injectivity rests on 1 for nodes and on linearity of the consequence relation for edge components, the dangling condition rests on 2, and the identification condition on 3.

It is not difficult to see, along these lines, that an hypergraph transformation system $\mathcal{G} = \langle TG, P, \pi, G_0\rangle$ can be translated to QILL, and that it is possible to prove the completeness side of an adequacy result for QILL with respect to reachability. The following may give an idea of the level of expressiveness.

Obs. 7 Given a linear logic context $\Delta_0 = [\alpha | \alpha = [\![s]\!]^T, s \in gc(G_0)]$, as types of the ground components of G_0, and an unbounded context $\Gamma_P = \Sigma \cup [\rho | \rho = [\![\pi(p)]\!]^T, p \in P]$, as types of the transformation rules, for every graph G reachable in the system, the following logic sequent is provable

$$\Gamma_P; \Delta_0 \Vdash [\![G]\!]^T$$

Given a multiset R of transformations in $\mathcal{G}$, let $\Delta_R = [\tau | \tau = [\![t]\!]^T, t \in R]$. Then, for each graph G which is reachable from G_0 by executing the transformations in R, in some order, the following logic sequent is provable

$$\Sigma; \Delta_0, \Delta_R \Vdash [\![G]\!]^T$$

If G is reachable by executing at least the transformations in R, in some order, the following is provable

$$\Gamma_P; \Delta_0, \Delta_R \Vdash [\![G]\!]^T$$

Soundness of linear implication with respect to reachability is less straightforward — we are still working on it.

6 Conclusion and further work

We have defined a translation of DPO GTS, formulated in algebraic terms with inessential restrictions, into a quantified extension of ILL based on linear λ-calculus. We have introduced a resource-bound existential quantifier with weak freshness features, in order to type restricted node names. On the semantical side, we consider transformation rules with node-only interfaces, taking isolated nodes into consideration.

We have given informal proof sketches of the fact that our translation is sound and complete with respect to well-formedness of graph expressions and their congruence, and it is complete with respect to reachability (both local and global). Related work on the translation of multiset rewriting into ILL has been discussed in [CS06]. We would like to mechanise the logic on a theorem prover, and we are considering Isabelle, for which there is already a theory of ILL [DSB06].

References

[Abr93] S. Abramsky. Computational interpretation of linear logic. *Theoretical Computer Science* 111, 1993.

[BBPH93] N. Benton, G. Bierman, V. de Paiva, M. Hyland. Linear lambda-calculus and categorical models revisited. In Borgër et al. (eds.), *Proceedings of the Sixth Workshop on Computer Science Logic*. Pp. 61–84. Springer Verlag, 1993.

[CC04] L. Caires, L. Cardelli. A Spatial Logic for Concurrency II. *Theoretical Computer Science* 322(3):517–565, 2004.

[CP02] I. Cervesato, F. Pfenning. A linear logical framework. *Information and Computation* 179(1):19–75, 2002.

[CS06] I. Cervesato, A. Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Electron. Notes Theor. Comput. Sci.* 165:145–176, 2006.

[DSB06] L. Dixon, A. Smaill, A. Bundy. Planning as Deductive Synthesis in Intuitionistic Linear Logic. Technical report, University of Edinburgh, 2006.
`http://homepages.inf.ed.ac.uk/ldixon/papers/infrep-06-planill.pdf`

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2006.

[FHL$^+$06] G. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto. Synchronised Hyperedge Replacement as a Model for Service Oriented Computing. In *FMCO'05*. Pp. 22–43. 2006.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50(1):1–102, 1987.

[IH01] K. Ishihara, K. Hiraishi. The completeness of linear logic for Petri net models. *Journal of teh IGPL* 9(4):549–567, 2001.

[MS07] D. Miller, A. Saurin. From proofs to focused proofs: a modular proof of focalization in Linear Logic. In *CSL'07: Computer Science Logic*. Pp. 405–419. 2007.

[Pfe02] F. Pfenning. Linear Logic — 2002 Draft. Technical report, Carnagie Mellon University, 2002.

[Pit01] A. M. Pitts. Nominal Logic: A First Order Theory of Names and Binding. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*. Pp. 219–242. Springer-Verlag, 2001.

[Pym02] D. J. Pym. *The semantics and proof-theory of the logics of bunched implications*. Applied Logic Series. Kluwer, 2002.

[TS00] A. S. Troelstra, H. Schwitchtenberg. *Basic Proof Theory*. Cambridge University Press, 2000.