Acta Informatica (2005) 42(4/5): 349–388
DOI 10.1007/s00236-005-0180-4

ORIGINAL ARTICLE

Barbara König

A general framework for types in graph rewriting

Received: 6 January 2004 / Revised: 10 October 2004 / Published online: 14 October 2005

Abstract We investigate a general framework which can be instantiated in order to obtain type systems for graph rewriting, allowing us to statically infer behavioural properties of a graph. We describe conditions such as the subject reduction property and compositionality that should be satisfied by such a framework. We present a methodology for proving these conditions, specifically we prove that it is sufficient to show properties that are local to graph transformation rules. In order to show the applicability of this framework, we describe in several case studies how to integrate existing type systems (for the π-calculus and the λ-calculus) and a system for typing acyclic graphs.

1 Introduction

Today's software systems are becoming increasingly more complex, concurrent and dynamic in nature. Examples for such systems are communication protocols, mobile processes in dynamically evolving networks and pointer structures on the heap generated by a program. This development requires the design of new verification and analysis methods which are specifically suited for dynamically evolving systems. From current work on verification environments [2, 7, 19, 46] for programming languages such as C or Java a trend emerges: It seems that a mix of several techniques will be the preferred method for software verification in the future.

This is a completely revised and extended version of a paper of which an earlier version has appeared in FSTTCS '00.

B. König (✉)
Institut für Formale Methoden der Informatik, Universität Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany,
E-mail: koenigba@fmi.uni-stuttgart.de

One promising technique for this mix are certainly type systems, which allow us to inductively infer behavioural information from a system description. There is a rich theory of type systems going far beyond types for variables and methods as used in imperative programming languages. Functional languages [37] have an especially elaborate type theory, but there has also been much work on process calculi such as the π-calculus [32, 36, 41] and the ambient calculus [6]. More general theories of types for process calculi are described in [20, 21]. Type systems are usually characterized by compositionality and modularity, i.e., the type of a term can be derived from the types of its subterms. Very often type systems also allow efficient type inference and can thus be used for fast debugging and quickly deriving basic properties of a system. This makes type systems, in combination with other methods, very useful tools for program analysis and verification.

In order to describe a general framework for type systems of concurrent and dynamically evolving systems, it is first necessary to determine a modelling language for such systems. Our work is based on graph transformation systems [44], which are a simple and intuitive, but at the same time rigorous and general, formalism, in which dynamically evolving systems and interacting processes can be specified. We will specifically work with hypergraphs where an arbitrarily long sequence of nodes can be attached to an edge. As will be shown later, all necessary ingredients of a type system can be naturally integrated into graph transformation systems. We feel that graph transformation systems are a good choice, since rewriting on graph-like structures has also emerged as the basis of a different encompassing theory, namely that of deriving labelled transitions and bisimulation congruences for a set of reaction rules [13, 23, 34].

In this setting two important questions have to be answered: What does the type of a graph look like? What are the essential properties of a type system that should be present in the general framework? It turns out that the former question is not very hard to answer: The type of a graph is again a graph. There are two reasons for this choice: First, complex types for functional languages or process calculi are often represented by infinite regular trees, i.e., trees with only finitely many subtrees. Such a regular tree can be finitely represented by a graph in a straightforward way. Second, in graph rewriting there already exists the concept of type graph [8], which is a graph over which all productions have to be typed. Otherwise, however, a type graph, which has to be fixed *a priori* and which can not be used for system analysis, is quite different from the types presented in this paper.

The latter question concerns the essential properties of a type system. Here we have identified the following properties:

- *correctness*
  If a system has a certain type, then we can conclude that this system has certain properties.
- *type invariance*, also called *subject reduction property*
  Types are invariant under reduction. Or put differently: The type describes an invariant property of the system under consideration.
- *compositionality*
  The type of a system can always be derived from the types of its subsystems.

– *subtypes and principal types*
  Types are ordered with respect to how much information they contain, i.e. with respect to their precision, and every system has a smallest type with respect to that order.
– *type inference*
  There exists a method for computing the (principal) type of a system.

While the first three properties seem to be fairly standard for type systems, the last two properties are certainly debatable since they are not satisfied by every type system. In our type systems, however, these two properties hold, since they greatly simplify working with type systems and since type inference gives us an effective method for system analysis. The subtype relation for graphs can be naturally defined by graph morphisms. Furthermore we add an extra layer of lattice annotations to type graphs in order to be able to express more specific system properties. This can be compared to frameworks for dataflow analysis or abstract interpretation [39] which are also parameterized over lattices.

Before we can define type systems for graphs, the following problem has to be solved: It is not entirely clear what it means to define a type inductively on the structure of a graph. Graphs are usually not defined inductively, they are defined as tuples containing a set of edges and a set of nodes. But there exists also an algebraic view on graphs which regards graphs as objects that can be composed and decomposed using operators on graphs [5, 15]. For our purposes it seems most convenient to describe graph construction using an explicit "construction plan" that can be applied to different graphs as long as they have the same external interface. This construction plan describes how graphs are attached independently of their internal structure by attaching nodes in their interfaces.

Our design decisions in developing a general framework were the following: The framework should be natural and not overly complex, but it should still be possible to embed existing well-known type systems, such as type systems for the λ-calculus and the π-calculus. It seemed also natural in this setting to integrate type inference into the typing rules and thus create type systems from which the algorithm for computing principal types can be immediately derived. The central aim for setting up this framework was to provide a proof methodology for showing that the essential properties of a type system, such as type invariance, are indeed satisfied. Specifically, it is sufficient to show the subject reduction property locally for graph transformation rules, the subject reduction property for global transformations then follows automatically.

This paper mainly focusses on the definition and justification of the type framework, the introduction of the proof methodology and the embedding of other type systems. An example for a useful application of this general framework (typing of (un)trustworthy applets) can be found in [25].

During the development of this framework we found the following mathematical concepts to be useful: colimits for graph construction, lattices and join-morphisms in order to specify type annotations, functors for having a clear correspondence between graph morphisms and the transformation of type annotations and closure operators. As a consequence the paper uses some concepts from category theory [4, 35]. These appear mainly in some of the proofs, especially in the proofs of Propositions 1 and 3. The rest of the paper should be understandable without any knowledge of category theory.

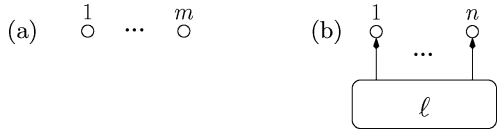

Fig. 1 (**a**) A discrete hypergraph and (**b**) a single hyperedge

2 Hypergraph rewriting and hypergraph annotation

We first define some basic notions concerning hypergraphs (see also [18]) and a method for inductively constructing hypergraphs. This concept has previously been introduced in [24, 30].

We are using the following notation. If $\tilde{s} \in A^*$ is a string of elements of the set A, then $Set(\tilde{s})$ denotes the set underlying $\tilde{s}$ and $\lfloor \tilde{s} \rfloor_i$ stands for the i-th element of $\tilde{s}$. If $f\colon A \to B$ is a function, then $f(\tilde{s})$ denotes the pointwise application of f to every element of A, resulting in a string of B^*.

Definition 1 (Hypergraph) Let L be a fixed set of labels. A *hypergraph* $H = (V_H, E_H, c_H, l_H, \chi_H)$ consists of a set of nodes V_H, a set of edges E_H, a connection mapping $c_H\colon E_H \to V_H^*$, an edge labelling $l_H\colon E_H \to L$ and a string $\chi_H \in V_H^*$ of external nodes or interface nodes. A *hypergraph morphism* (or simply morphism) $\varphi\colon H \to H'$ (consisting of mappings $\varphi_V\colon V_H \to V_{H'}$ and $\varphi_E\colon E_H \to E_{H'}$) maps nodes to nodes and edges to edges, preserving connections and labelling, i.e., $\varphi_V(c_H(e)) = c_{H'}(\varphi_E(e))$ and $l_H(e) = l_{H'}(\varphi_E(e))$.

A *strong morphism* (denoted by the arrow $\twoheadrightarrow$) additionally preserves external nodes, i.e. $\varphi_V(\chi_H) = \chi_{H'}$. We write $H \cong H'$ (H is isomorphic to H') if there is a bijective strong morphism from H to H'.

The arity of a hypergraph H is defined as $ar(H) = |\chi_H|$ while the arity of an edge e of H is $ar(e) = |c_H(e)|$. External nodes are the interface of a hypergraph towards its environment and are used to attach hypergraphs.

Notation: We call a hypergraph *discrete*, if its edge set is empty. By $\mathbf{m}$ we denote a discrete graph of arity $m \in \mathbb{N}$ with m nodes where every node is external (see Fig. 1(a), external nodes are labelled $1, 2, \ldots$ in their respective order).

The hypergraph $H = [\ell]_n$ contains exactly one edge e with label ℓ where $c_H(e) = \chi_H$, $ar(e) = n$ and $V_H = Set(\chi_H)$ (see Fig. 1(b)). We draw hyperedges in such a way that the sequence $c_H(e)$ is enumerated from left to right.

The next step is to define a method (first introduced in [24]) for the annotation of hypergraphs with lattice elements and to describe how these annotations are transformed under morphisms. We use annotated hypergraphs as types where the annotations can be considered as extra typing information, therefore we sometimes use the terms *annotated hypergraph* and *graph type* as synonyms.

Definition 2 (Annotated Hypergraphs) Let $\mathcal{A}$ be a mapping assigning a complete lattice $\mathcal{A}(H) = (I, \leq)$ to every hypergraph and a function $\mathcal{A}_\varphi\colon \mathcal{A}(H) \to \mathcal{A}(H')$ to every morphism $\varphi\colon H \to H'$. We assume that $\mathcal{A}$ satisfies:

$$\mathcal{A}_\varphi \circ \mathcal{A}_\psi = \mathcal{A}_{\varphi\circ\psi} \qquad \mathcal{A}_{id_H} = id_{\mathcal{A}(H)}$$
$$\mathcal{A}_\varphi(\bigvee A) = \bigvee\{\mathcal{A}_\varphi(a) \mid a \in A\} \qquad \mathcal{A}_\varphi(\bot) = \bot$$

where $\vee$ is the join-operation, A is a subset of the lattice $\mathcal{A}(H)$ and $\bot$ is its bottom element.

If $a \in \mathcal{A}(H)$, then H together with the lattice element a is called an annotated hypergraph and is denoted by $H[a]$. Furthermore $\varphi\colon H[a] \to_{\mathcal{A}} H'[a']$ is called an $\mathcal{A}$-morphism if $\varphi\colon H \to H'$ is a hypergraph morphism and $\mathcal{A}_\varphi(a) \leq a'$. Furthermore $H[a]$ and $H'[a']$ are called isomorphic (written $H[a] \cong H'[a']$) if there exists a strong bijective $\mathcal{A}$-morphism φ with $\mathcal{A}_\varphi(a) = a'$ between them.

Annotations can be used in order to add information to graph types that can not directly be derived from the graph structure or that requires additional manipulation (such as closure operations). A typical example is the case where nodes can be seen as communication ports and the annotations provide information on whether an external port can be used for input, for output or for both (see also Sect. 4.2).

Example 1 We consider the following annotation mapping $\mathcal{A}$ as an example. Let $(L, \leq)$ be an arbitrary lattice. We define $\mathcal{A}(H)$ to be the set of all mappings from V_H into L (which yields a lattice with pointwise order). So let $a\colon V_H \to L$ be an element of $\mathcal{A}(H)$ and let $\varphi\colon H \to H'$, $v' \in V_H$. We define: $\mathcal{A}_\varphi(a) = a'$ where $a'\colon V_{H'} \to L$ with $a'(v') = \bigvee_{\varphi(v)=v'} a(v)$. That is, we take the join of the lattice annotations of all nodes that are mapped to v'.

From the point of view of category theory, $\mathcal{A}$ is a functor from the category of hypergraphs and hypergraph morphisms into the category of lattices and join-morphisms (i.e. functions preserving the join operation of the lattice). The obvious (forgetful) functor from the category of annotated hypergraphs into the category of hypergraphs without annotations is an opfibration, which can be obtained by the Grothendieck construction [4]. This property, however, is of no consequence for the rest of this paper.

As in type systems types are usually defined inductively over the structure of a term, we want to define graph types inductively over the structure of a graph. However, it is not immediately clear how to decompose a graph. In order to have a clear concept for the composition and decomposition of hypergraphs, we now introduce a method for attaching (annotated) hypergraphs with a construction plan consisting of discrete graph morphisms.

Definition 3 (Hypergraph Construction) Let $H_1[a_1], \ldots, H_n[a_n]$ be annotated hypergraphs and let $\zeta_i : \mathbf{m_i} \to D$, $i \in \{1, \ldots, n\}$ be hypergraph morphisms where $ar(H_i) = m_i$ and D is discrete. Furthermore let $\varphi_i : \mathbf{m_i} \twoheadrightarrow H_i$ be the unique strong morphisms from $\mathbf{m_i}$ into H_i.

For this construction we assume that the node and edge sets of $H_1, \ldots, H_n$ and D are pairwise disjoint. Furthermore let $\approx$ be the smallest equivalence on their nodes satisfying $\zeta_i(v) \approx \varphi_i(v)$ for $i \in \{1, \ldots, n\}$, $v \in V_{\mathbf{m_i}}$. The nodes of the constructed graph are the equivalence classes of $\approx$. We define a hypergraph H as

$$\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) = \left(\left(V_D \cup \bigcup_{i=1}^{n} V_{H_i}\right)/\!\approx, \bigcup_{i=1}^{n} E_{H_i}, c_H, l_H, \chi_H\right)$$

where $c_H(e) = [v_1]_\approx \ldots [v_k]_\approx$ if $e \in E_{H_i}$ and $c_{H_i}(e) = v_1 \ldots v_k$. Furthermore $l_H(e) = l_{H_i}(e)$ if $e \in E_{H_i}$. We also define $\chi_H = [v_1]_\approx \ldots [v_k]_\approx$ if $\chi_D = v_1 \ldots v_k$.

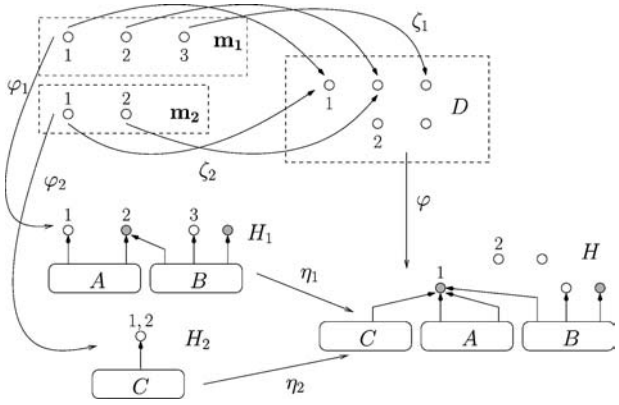

Fig. 2 An example for hypergraph construction

If $n = 0$, the result of the construction is D itself.

We construct embeddings $\varphi\colon D \twoheadrightarrow H$ and $\eta_i\colon H_i \to H$ by mapping every node to its equivalence class and every edge to itself. Then the construction of annotated graphs can be defined as follows:

$$\textcircled{D}_{i=1}^{n}(H_i[a_i], \zeta_i) = \left(\textcircled{D}_{i=1}^{n}(H_i, \zeta_i)\right)\left[\bigvee_{i=1}^{n} \mathcal{A}_{\eta_i}(a_i)\right].$$

In other words: we join all graphs $D, H_1, \ldots, H_n$ and fuse exactly the nodes which are the image of one and the same node in the $\mathbf{m_i}$, the image of χ_D becomes the new sequence of external nodes. The new lattice annotation is the supremum of all annotations a_i, transferred to the new graph H.

Example 2 We present an example for graph construction, where we combine two hypergraphs H_1, H_2 using the discrete morphisms $\zeta_1\colon \mathbf{3} \to D$ and $\zeta_2\colon \mathbf{2} \to D$ depicted in Fig. 2 below (ignore the grey nodes for the moment). The resulting hypergraph is H. The following points are noteworthy:

- The first external node of $\mathbf{m_1}$ and the first external node of $\mathbf{m_2}$ are mapped to the same node in D, which means that the respective nodes of H_1 and H_2 are to be fused in H.
- The hypergraph H_2 contains duplicates in its sequence of external nodes. This causes all nodes that are to be fused with either the first or the second node of H_2 to be fused themselves, which occurs to the two nodes attached to the A-edge.
- The discrete graph D contains an internal and an external node which are not in the range of the ζ_i. This indicates that they are still present in the resulting graph H, but not attached to any edge, i.e., they are isolated.

To continue this example we assume an annotation mapping as in Example 1 where $L = \{true, false\}$ with $false < true$. If in H_1 and H_2 we colour all nodes that are labelled *true* in grey, then, in the annotation mapping for H, exactly the nodes that are the image of at least one grey node will be again grey.

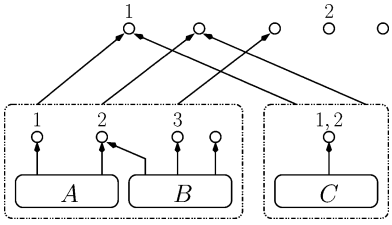

Fig. 3 An alternative notation for graph construction

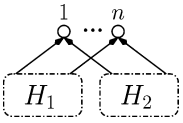

Fig. 4 A multi-hole graph context representing $H_1 \square H_2$

We also use another, more intuitive notation for graph construction, which we call *multi-hole graph context*. Let $\zeta_i : \mathbf{m_i} \to D$, $i \in \{1, \ldots, n\}$. Then we depict $\textcircled{D}_{i=1}^{n}(H_i, \zeta_i)$ by drawing the hypergraph $(V_D, \{e_1, \ldots, e_n\}, c_H, l_H, \chi_D)$ where $c_H(e_i) = \zeta_i(\chi_{\mathbf{m_i}})$ and $l_H(e_i) = H_i$ (see Fig. 3). Note that the edges of this hypergraph containing the H_i are depicted by using dashed lines. If there is an edge with a dashed line labelled with an edge $[\ell]_n$ we preferably draw it with a solid line and label it with ℓ (see e.g. Fig. 12).

Example 3 Using the notion of multi-hole graph context we can draw $\textcircled{n}_{i=1}^{2}(H_i, \zeta_i)$ where $\zeta_1, \zeta_2 : \mathbf{n} \twoheadrightarrow \mathbf{n}$ as in Fig. 4. Here we fuse the external nodes of H_1 and H_2 in their respective order and denote the resulting graph by $H_1 \square H_2$.

It can be shown that this form of graph construction is as expressive as other operators or operations on graphs, such as the graph expressions of Bauderon and Courcelle [5] or hyperedge replacement [18]. Furthermore every graph has a unique normal form which corresponds to its decomposition into hyperedges.

In terms of category theory, $\textcircled{D}_{i=1}^{n}(H_i[a_i], \zeta_i)$ is the colimit of the ζ_i and the φ_i regarded as $\mathcal{A}$-morphisms (D and the $\mathbf{m_i}$ are annotated with the bottom element $\bot$). The properties of the annotation mapping, given in Definition 2, are needed to show that $\textcircled{D}_{i=1}^{n}(H_i[a_i], \zeta_i)$ is in fact a colimit.

Proposition 1 (Hypergraph Construction as a Colimit) *Let $H_1[a_1], \ldots, H_n[a_n]$ be annotated hypergraphs with $m_i = ar(H_i)$, let $\zeta_i : \mathbf{m_i}[\bot] \to_{\mathcal{A}} D[\bot]$ be discrete morphisms and let $\varphi_i : \mathbf{m_i}[\bot] \twoheadrightarrow_{\mathcal{A}} H_i[a_i]$ be the unique strong morphisms.*

Then $H[a] = \textcircled{D}_{i=1}^{n}(H_i[a_i], \zeta_i)$ (with morphisms η_i, φ of Definition 3) is the colimit of the ζ_i and the φ_i in the category of annotated hypergraphs and $\mathcal{A}$-morphisms (see Fig. 5).

Proof The proof is straightforward and is thus omitted. □

From the fact that graph construction can be described by a colimit, we can immediately derive another interesting fact, which will be needed later on.

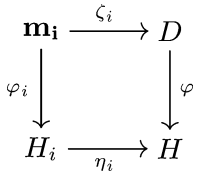

Fig. 5 Hypergraph construction as a colimit

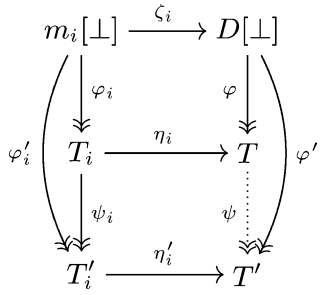

Fig. 6 Proof of Proposition 2

Proposition 2 *Let $\psi_i : T_i \twoheadrightarrow_{\mathcal{A}} T_i'$, $i \in \{1, \ldots, n\}$ be strong $\mathcal{A}$-morphisms. Then there exists a strong $\mathcal{A}$-morphism $\psi : \textcircled{D}_{i=1}^{n}(T_i, \zeta_i) \to \textcircled{D}_{i=1}^{n}(T_i', \zeta_i)$.*

Proof We set $T = \textcircled{D}_{i=1}^{n}(T_i, \zeta_i)$ and $T' = \textcircled{D}_{i=1}^{n}(T_i', \zeta_i)$. Furthermore let

$$\begin{aligned}
&\zeta_i : \mathbf{m_i}[\bot] \to_{\mathcal{A}} D[\bot], \quad \varphi_i : \mathbf{m_i}[\bot] \to_{\mathcal{A}} T_i, \\
&\varphi : D[\bot] \twoheadrightarrow_{\mathcal{A}} T, \quad \eta_i : T_i \to_{\mathcal{A}} T, \\
&\varphi_i' : \mathbf{m_i}[\bot] \to_{\mathcal{A}} T_i, \quad \varphi' : D[\bot] \twoheadrightarrow_{\mathcal{A}} T', \quad \eta_i' : T_i' \to T'
\end{aligned}$$

be the morphisms of the two colimits. Since the φ_i' are the unique strong morphisms, it holds that $\varphi_i' = \psi_i \circ \varphi_i$. Therefore $(\eta_i' \circ \psi_i) \circ \varphi_i = \eta_i' \circ \varphi_i' = \varphi' \circ \zeta_i$, i.e., the diagram consisting of the morphisms $\varphi_i, \zeta_i, \eta_i' \circ \psi_i$ and φ' commutes. The universal property of the colimit implies the existence of a morphism $\psi : T \to_{\mathcal{A}} T'$. (This situation is depicted in Fig. 6). Furthermore $\varphi' = \psi \circ \varphi$, and therefore ψ is also a strong morphism. □

Having established the notion of graph construction it is now straightforward to define hypergraph rewriting.

Definition 4 (Hypergraph Rewriting) Let $\mathcal{R}$ be a set of pairs (L, R) (called rewriting rules), where the left-hand side L and the right-hand side R are both hypergraphs of the same arity. Then $\to_{\mathcal{R}}$ is the smallest relation generated by the pairs of $\mathcal{R}$ and closed under hypergraph construction.

Our approach is equivalent to rewriting in the double-pushout approach [12] using discrete interface graphs only (for more details concerning this claim see [29]). Note that every rewriting rule with a non-discrete interface can in principle be simulated by a rule with a discrete interface by deleting and recreating the edges

in the interface. This is equivalent in all contexts in which concurrency aspects are not taken into account.

We need one more concept: A linear mapping which is an inductively defined transformation, mapping hypergraphs to hypergraphs and adding annotation.

Definition 5 (Linear Mapping) A function from hypergraphs to hypergraphs is called arity-preserving if it preserves arity and isomorphism classes of hypergraphs.

Let t be an arity-preserving function that maps hypergraphs of the form $[\ell]_n$ to annotated hypergraphs. Then t can be extended to arbitrary hypergraphs by defining $t(\textcircled{D}_{i=1}^{n}([l_i]_{n_i}, \zeta_i)) = \textcircled{D}_{i=1}^{n}(t([l_i]_{n_i}), \zeta_i)$ and is then called a *linear mapping*.

A linear mapping t is always well-defined and satisfies

$$t(\textcircled{D}_{i=1}^{n}(H_i, \zeta_i)) \cong \textcircled{D}_{i=1}^{n}(t(H_i), \zeta_i)$$

for arbitrary hypergraphs H_i. Note that the construction operator on the left-hand side of the equation operates on ordinary hypergraphs, while the one on the right-hand side operates on annotated hypergraphs.

3 Type systems for graph rewriting

3.1 Basic requirements for type systems

Having introduced all underlying notions we now specify the requirements for type systems. The basic assumption is that the type of a hypergraph is an annotated hypergraph. As will be seen later this is a convenient requirement that allows us to regard a graph as an abstract representation of the behaviour of a graph transformation system.

We assume that there is a fixed set $\mathcal{R}$ of rewriting rules, an annotation mapping $\mathcal{A}$, a predicate X on hypergraphs (representing the property we want to check), a property Y on graph types and a relation $\triangleright$ with the following meaning: if $H \triangleright T$ where H is a hypergraph and T a graph type (annotated with respect to $\mathcal{A}$), then we say that H has type T. It is required that H and T have the same arity.

We demand that $\triangleright$ satisfies the following conditions: first, a type should contain information concerning the properties of a hypergraph, i.e., if a hypergraph has a type and Y holds for this type, then we can be sure that the property X holds for the hypergraph.

$$\textbf{Correctness:} \qquad H \triangleright T \ \wedge\ Y(T) \ \Rightarrow\ X(H) \tag{1}$$

During reduction, the type stays invariant.

$$\textbf{Subject reduction property:} \qquad H \triangleright T \ \wedge\ H \to_{\mathcal{R}} H' \ \Rightarrow\ H' \triangleright T \tag{2}$$

From (1) and (2) we can conclude that $H \triangleright T$, $Y(T)$ and $H \to_{\mathcal{R}}^{*} H'$ imply $X(H')$, that is X holds during the entire reduction.

The strong $\mathcal{A}$-morphisms introduced in Definition 2 impose a preorder on graph types. It should always be possible to weaken the type with respect to that preorder. Since intuitively a graph type represents structure that may be present, adding new parts or fusing components results in a less precise type.

Weakening: (3)

$$H \triangleright T \ \wedge \ T \twoheadrightarrow_{\mathcal{A}} T' \ \Rightarrow \ H \triangleright T'$$

We also demand that the type system is compositional, i.e., a graph has a type if and only if this type can be obtained by typing its subgraphs and combining these types. It will be necessary to take into account effects caused by non-context-free rewriting rules. Hence it is not possible to obtain the type of an expression simply by combining the types of the subgraphs in exactly the same way the expression is constructed and so we introduce a partial arity-preserving mapping ρ doing some post-processing. Since ρ might be partial, its application to a graph type is not necessarily defined. So it might be the case in the first line below that $\textcircled{D}_{i=1}^{n}(H_i, \zeta_i)$ does not have a type.

Compositionality: (4)

$$\forall i : H_i \triangleright T_i \Rightarrow \textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright \overbrace{\rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i))}^{T}$$

whenever T is defined

$$\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright T \Rightarrow \exists T_1, \ldots, T_n : ((\forall i : H_i \triangleright T_i) \ \wedge \ \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \twoheadrightarrow_{\mathcal{A}} T)$$

A last condition—the existence of minimal types—may not be strictly needed for type systems, but type systems satisfying this condition are much easier to handle.

Existence of minimal types: (5)

$$H \text{ typable } \Rightarrow \ \exists T : (H \triangleright T \wedge \forall T' : (H \triangleright T' \iff T \twoheadrightarrow_{\mathcal{A}} T'))$$

3.2 Introducing a more specific type system

Let us now assume that types are computed from graphs in the following way: there is a linear mapping t, such that $H \triangleright \rho(t(H))$, if $\rho(t(H))$ is defined, and all other types of H are derived by the weakening rule, i.e., $\rho(t(H))$ is the minimal type of H.

The meaning of the mappings t and ρ can be explained as follows: t is a transformation *local* to edges, abstracting from irrelevant details and adding annotation information to a graph. The mapping ρ on the other hand, is a *global* (closure) operation, merging or removing parts of a graph in order to anticipate future reductions and thus ensuring the subject reduction property.

The motivation to use closure operators came from the area of abstract interpretation [9, 10] and from unification as it is used in type inference algorithms [43]. Furthermore the type system was tailored in such a way as to be able to integrate well-known existing type systems (see Sect. 4). Presenting the type system

using the mappings t and ρ is convenient, since we obtain a formalism already close to type inference algorithms, but there exists a rule-based formulation as well (see below). It might be the case that in order to extend the framework and integrate more complex type systems, it is necessary to rely on the rule-based formulation and design a separate type inference algorithm.

In this setting it is sufficient to prove some simpler conditions, especially the proof of (2) can be conducted locally.

Theorem 1 *Let $\mathcal{A}$ be a fixed annotation mapping, let ρ be an arity-preserving mapping as above, let t be a linear mapping, let X and Y be predicates on hypergraphs respectively graph types and let $H \triangleright T$ if and only if $\rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T$. Let us further assume that ρ and Y satisfy*[1]

$$T \twoheadrightarrow_{\mathcal{A}} T' \wedge Y(T') \Rightarrow Y(T) \tag{6}$$

$$\rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \cong \rho(\textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i)) \tag{7}$$

$$T \twoheadrightarrow_{\mathcal{A}} T' \Rightarrow \rho(T) \twoheadrightarrow_{\mathcal{A}} \rho(T') \tag{8}$$

Then the relation $\triangleright$ satisfies conditions (1)–(5) if and only if it satisfies

$$Y(\rho(t(H))) \Rightarrow X(H) \tag{9}$$

$$(L, R) \in \mathcal{R} \Rightarrow \rho(t(R)) \twoheadrightarrow_{\mathcal{A}} \rho(t(L)) \tag{10}$$

Proof We first show that (9) and (10) imply (1)–(5)

(1) Let $H \triangleright T$ and $Y(T)$. From the definition of $\triangleright$ it follows that $\rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T$ and (6) implies that $Y(\rho(t(H)))$ is satisfied. With (9) we conclude that $X(H)$ holds.

(2) Let $H \triangleright T$ and $H \to_{\mathcal{R}} H'$. From the definition of $\triangleright$ it follows that $\rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T$.
The relation $\to_{\mathcal{R}}$ is defined via the closure of $\mathcal{R}$ under hypergraph construction, i.e., $H \cong \textcircled{D}_{i=1}^{n}(H_i, \zeta_i)$, $H' \cong \textcircled{D}_{i=1}^{n}(H'_i, \zeta_i)$ and there is a rule $(L, R) \in \mathcal{R}$ such that $H_j \cong L$ and $H'_j \cong R$. For all other i with $i \neq j$ it holds that $H_i \cong H'_i$.
Since ρ and t preserve isomorphism classes, it holds that $\rho(t(H_i)) \cong \rho(t(H'_i))$ and with (10) it follows that $\rho(t(R)) \twoheadrightarrow_{\mathcal{A}} \rho(t(L))$. From Proposition 2 it follows that

$$\textcircled{D}_{i=1}^{n}(\rho(t(H'_i)), \zeta_i) \twoheadrightarrow_{\mathcal{A}} \textcircled{D}_{i=1}^{n}(\rho(t(H_i)), \zeta_i).$$

Finally, Conditions (8) and (7) imply that

$$\begin{aligned}\rho(t(H')) &\cong \rho(\textcircled{D}_{i=1}^{n}(\rho(t(H'_i)), \zeta_i)) \twoheadrightarrow_{\mathcal{A}} \rho(\textcircled{D}_{i=1}^{n}(\rho(t(H_i)), \zeta_i)) \\ &\cong \rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T.\end{aligned}$$

Hence $\rho(t(H')) \twoheadrightarrow_{\mathcal{A}} T$ and this implies $H' \triangleright T$.

(3) Let $H \triangleright T$ and $T \twoheadrightarrow_{\mathcal{A}} T'$. From the definition of $\triangleright$ it follows that $\rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T \twoheadrightarrow_{\mathcal{A}} T'$ and therefore $H \triangleright T'$.

[1] In an equation of the form $T \cong T'$ we assume that T is defined if and only if T' is defined. And in a condition of the form $T \twoheadrightarrow_{\mathcal{A}} T'$ we assume that T is defined if T' is defined.

(4) We show that both parts of the condition are satisfied:
- We assume that there are graph types T_i such that $H_i \triangleright T_i$. It follows that $\rho(t(H_i)) \twoheadrightarrow_{\mathcal{A}} T_i$. Since $\mathcal{A}$-morphisms are preserved by graph construction (see Proposition 2) and by the operation ρ (see Condition (8)) we conclude with Proposition 2 that

$$\begin{aligned}\rho(t(\textcircled{D}_{i=1}^{n}(H_i, \zeta_i))) &\cong \rho(\textcircled{D}_{i=1}^{n}(\rho(t(H_i)), \zeta_i)) \\ &\twoheadrightarrow_{\mathcal{A}} \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)).\end{aligned}$$

  And therefore $\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i))$ whenever the type graph is defined.
- Let $\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright T$. From the definition of $\triangleright$ and with (7) it follows that

$$\rho(\textcircled{D}_{i=1}^{n}(\rho(t(H_i)), \zeta_i)) \twoheadrightarrow_{\mathcal{A}} T.$$

  We set $T_i = \rho(t(H_i))$ and hence $H_i \triangleright T_i$ is satisfied.

(5) Let H be typable, i.e. $T = \rho(t(H))$ is defined. We show that T is the minimal type. If $H \triangleright T'$ for any graph type T' it follows from the definition of $\triangleright$ that $T = \rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T'$. If, on the other hand, $T \cong \rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T'$, it follows immediately with (3) that $H \triangleright T'$.

We will now show that (1)–(5) imply (9) and (10).

- (9) We assume that $Y(\rho(t(H)))$ holds. Since $H \triangleright \rho(t(H))$, Condition (1) implies that $X(H)$ holds.
- (10) let $(L, R) \in \mathcal{R}$, that is $L \to_{\mathcal{R}} R$ and $L \triangleright \rho(t(L))$. Condition (2) implies that $R \triangleright \rho(t(L))$. And from the definition of $\triangleright$ it follows that $\rho(t(R)) \twoheadrightarrow_{\mathcal{A}} \rho(t(L))$. □

Notes: If a graph rewriting rule satisfies Condition (10), we sometimes also say that this rule can be typed.

The fact that local type invariance implies global type invariance is one of the main features of this framework. Examples for rules satisfying Condition (10) can be found in later sections in Figs. 10 and 20.

It is a direct consequence of Condition (7) above that the operator ρ is idempotent, i.e. $\rho(\rho(T)) \cong \rho(T)$. Regard the identity graph construction with $n = 1$ and $\zeta_1 : \mathbf{n} \twoheadrightarrow \mathbf{n}$ where $n = ar(T)$.

This restriction to a more specialised type system is partly motivated by Condition (10) describing local type invariance and partly by the fact that it allows a classical rule-based formulation. By definition $H \triangleright T$ holds if and only if $\rho(t(H)) \twoheadrightarrow_{\mathcal{A}} T$. If we assume that Conditions (6)–(10) are satisfied, this holds if and only if $H \triangleright T$ can be derived using the following typing rules.

$$[\ell]_m \triangleright \rho(t([\ell]_m))$$

$$\frac{\forall i : H_i \triangleright T_i}{\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i))} \qquad \frac{H \triangleright T,\ T \twoheadrightarrow_{\mathcal{A}} T'}{H \triangleright T'}$$

We can now summarize all components needed for a type framework in our setting.

Definition 6 (Type Framework) A *type framework* is a tuple $(\mathcal{A}, t, \rho, X, Y)$ where

- $\mathcal{A}$ is an annotation mapping.
- t is a linear mapping mapping hypergraphs to annotated hypergraphs.
- ρ is a partial arity-preserving mapping from annotated hypergraphs to annotated hypergraphs.
- X is a predicate on hypergraphs.
- Y is a predicate on annotated hypergraphs.

and Conditions (6)–(9) are satisfied.

A type framework is called *type framework for a set of rewriting rules* $\mathcal{R}$ if additionally Condition (10) holds for all rules in $\mathcal{R}$.

3.3 Deriving properties of the closure operator ρ

In order to come to terms with a global operator such as ρ defined above, we present two ways to derive ρ such that at least some of the required properties hold automatically. The first method is described in Sect. 3.3.1 and is based on universal properties. It will be employed in Sects. 4.2 and 4.3 in order to define type frameworks related to type systems of the π-calculus and the λ-calculus. The second method, described in Sect. 3.3.2 reduces a graph to its external interface. We will refer to it again in Sect. 4.1, where a type framework for typing acyclic graphs is developed.

In many ways, the operator ρ is related to closure operators on lattices, which are used in abstract interpretation [11].

Definition 7 (Closure operator on lattices) Let $(L, \leq)$ be a lattice consisting of a set L and a partial order $\leq$. A mapping $\rho\colon L \to L$ is called *upper closure operator* if it satisfies the following three conditions:

- ρ is monotone.
- $\forall l \in L : l \leq \rho(l)$, i.e., ρ is extensive.
- $\forall l \in L : \rho(\rho(l)) = \rho(l)$, i.e., ρ is idempotent.

The monotonicity of ρ is analogous to Condition (8). Furthermore one can easily derive that $\rho(\bigvee_{i=1}^{n} \ell_i) = \rho(\bigvee_{i=1}^{n} \rho(\ell_i))$, where $\ell_1, \ldots, \ell_n \in L$ and $\bigvee$ stands for the least upper bound. This is strongly related to Condition (7).

It is not in all cases convenient to demand the equivalent of extensivity, i.e., there must not necessarily be a morphism $T \twoheadrightarrow_{\mathcal{A}} \rho(T)$. But sometimes the operation ρ can be characterised by a universal property with the intuitive notion that $\rho(T)$ is the "smallest" graph type (with respect to the preorder $\twoheadrightarrow_{\mathcal{A}}$) for which $T \twoheadrightarrow_{\mathcal{A}} \rho(T)$ and a property C hold (see Proposition 3).

3.3.1 Characterizing closure operators by a universal property

Following these ideas, the morphism $T \twoheadrightarrow_{\mathcal{A}} \rho(T)$ can also be seen as the initial element in a comma category $(T \downarrow F)$ where F is the obvious functor

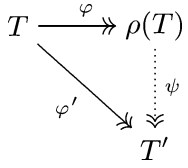

Fig. 7 Characterisation of $\rho(T)$ by a universal property

from the category of hypergraphs satisfying C (with all strong morphisms between them) into the category of all hypergraphs with strong morphisms. In other words, the category $(T \downarrow F)$ has as objects all graph morphisms of the form $\varphi' \colon T \twoheadrightarrow_{\mathcal{A}} T'$, where $C(T')$ is satisfied. And for two such objects $\varphi' \colon T \twoheadrightarrow_{\mathcal{A}} T'$, $\varphi'' \colon T \to T''$ an arrow with source φ' and target φ'' is a morphism $\psi : T' \twoheadrightarrow_{\mathcal{A}} T''$ such that $\psi \circ \varphi' = \varphi''$, i.e., the diagram commutes.

All this can be defined without making use of comma categories in the following way:

Proposition 3 *Let C be a property on graph types. We characterize $\rho(T)$ in the following way: $\rho(T)$ satisfies C, there is a morphism $\varphi \colon T \twoheadrightarrow_{\mathcal{A}} \rho(T)$ and for every other morphism $\varphi' \colon T \twoheadrightarrow_{\mathcal{A}} T'$ where $C(T')$ holds, there is a unique morphism $\psi : \rho(T) \twoheadrightarrow_{\mathcal{A}} T'$ such that $\psi \circ \varphi = \varphi'$ (see Fig. 7). Furthermore we demand that*

(i) If there exists a morphism $\varphi \colon T \twoheadrightarrow_{\mathcal{A}} T'$ such that $C(T')$ holds, then $\rho(T)$ is defined.
(ii) Every discrete graph D satisfies condition C.

Then if $\rho(T)$ is defined, it is unique up to isomorphism. Furthermore ρ satisfies Conditions (7) and (8).

Proof For the purpose of this proof let $\mathcal{H}_{\mathcal{A}}$ be the category of hypergraphs annotated by $\mathcal{A}$, $\mathcal{H}'_{\mathcal{A}}$ the category of annotated hypergraphs which can be folded, i.e., the application of ρ is defined, and let $\mathcal{H}^C_{\mathcal{A}}$ the category of annotated hypergraphs satisfying C. It obviously holds that $\mathcal{H}^C_{\mathcal{A}}$ is a subcategory of $\mathcal{H}'_{\mathcal{A}}$, and $\mathcal{H}'_{\mathcal{A}}$ is a subcategory of $\mathcal{H}_{\mathcal{A}}$. Now let E be the (inclusion) functor which embeds $\mathcal{H}^C_{\mathcal{A}}$ straightforwardly into $\mathcal{H}'_{\mathcal{A}}$.

Because of the universal property through which ρ is defined, it follows that $\rho \colon \mathcal{H}'_{\mathcal{A}} \to \mathcal{H}^C_{\mathcal{A}}$ can be extended to the left adjoint of E (compare [35], IV.1., Theorem 2) and is therefore a functor. This implies that whenever there is a morphism $T \twoheadrightarrow_{\mathcal{A}} T'$, then there exists also a morphism $\rho(T) \twoheadrightarrow_{\mathcal{A}} \rho(T')$, whenever $\rho(T)$ is defined. Furthermore $\rho(T)$ must be defined whenever $\rho(T')$ is defined, according to Condition (i) in Proposition 3. Combined, this implies that Condition (8) holds.

Left adjoints preserve colimits ([35], V.5), i.e., if for two categories A, X, the functor $F : A \to X$ is a left adjoint, it holds that $Col_X F(D) \cong F(Col_A D)$ where $Col_A D$ is the colimit of the diagram D in the category A.

In our case $A = \mathcal{H}'_{\mathcal{A}}$ and $X = \mathcal{H}_{\mathcal{A}}$. Translating the preservation result into this setting implies that the colimit of a diagram $\zeta_i : \mathbf{m_i} \to D, \varphi_i : \mathbf{m_i} \twoheadrightarrow T_i$ in the category $\mathcal{H}'_{\mathcal{A}}$ followed by the application of ρ gives a result isomorphic to the colimit of the diagram $\zeta_i : \mathbf{m_i} \to D, \varphi'_i : \mathbf{m_i} \twoheadrightarrow \rho(T_i)$ in the category $\mathcal{H}^C_{\mathcal{A}}$.

Again φ_i and φ'_i are the unique strong morphisms. By diagram-chasing we can easily show that the latter colimit is isomorphic to the colimit of the ζ_i, φ'_i in the category $\mathcal{H}'_{\mathcal{A}}$, followed by the application of ρ. Since ρ does not modify discrete graphs (Condition (ii)), this immediately implies Condition (7) if both sides of the equation are defined.

If the right-hand side of the equation is defined, then we can conclude with Proposition 2 that there are morphisms

$$\textcircled{D}_{i=1}^{n}(T_i, \zeta_i) \twoheadrightarrow_{\mathcal{A}} \textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i) \twoheadrightarrow_{\mathcal{A}} \rho(\textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i)),$$

where the last graph satisfies condition C. Hence Condition (i) of Proposition 3 implies that the left-hand side of the equation is defined as well.

If the left-hand side is defined, then we can immediately infer that the right-hand side is defined, since colimits are preserved by adjoints. □

Note that this proposition can be shown conveniently using well-known results from category theory. A proof without using category theory is also possible, but considerably longer (see [26]).

3.3.2 Reduction to external nodes

A different, and often useful way to define ρ is to reduce the type graph together with its annotation to its external nodes. This means restricting the type information to the interface. We will use this definition again in Sect. 4.1.

Definition 8 Let $\mathcal{A}$ be an annotation mapping, let $\varphi\colon G \to H$ be a graph morphism and let $b \in \mathcal{A}(H)$. We define $red_\varphi(b) = \bigvee\{a \mid \mathcal{A}_\varphi(a) \leq b\}$.

Let $T = H[b]$ be an $\mathcal{A}$-annotated hypergraph of arity n. We define the *reduction to external nodes* $\rho(T)$ of T to be the discrete annotated graph $X[red_\varphi(b)]$ where X consists of the external nodes of H, i.e.,

$$V_X = Set(\chi_H), \quad E_X = \varnothing, \quad \chi_X = \chi_H$$

and $\varphi\colon X \twoheadrightarrow H$ is the unique strong morphism from X into H.

All reductions satisfy Condition (8), but we can only give a partial answer concerning Condition (7).

Proposition 4 *Let ρ be defined as in Definition 8. Then ρ satisfies Condition (8) and furthermore there is a strong $\mathcal{A}$-morphism*

$$\iota\colon \rho(\textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i)) \twoheadrightarrow_{\mathcal{A}} \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)),$$

which is a strong isomorphism on the underlying (discrete) graphs.

Proof We will first show that, for a given graph type $T = H[b]$, there is a strong injective $\mathcal{A}$-morphism $\rho(T) \twoheadrightarrow_{\mathcal{A}} T$. Let $X[a] = \rho(T)$. First, there obviously is a strong injective morphism $X \twoheadrightarrow H$. It is left to show that $\mathcal{A}_\varphi(red_\varphi(b)) \leq b$. This holds since $\mathcal{A}_\varphi(\bigvee\{a \mid \mathcal{A}_\varphi(a) \leq b\}) = \bigvee\{\mathcal{A}_\varphi(a) \mid \mathcal{A}_\varphi(a) \leq b\} \leq b$.

Let us now consider Condition (8). Let $\psi : T \twoheadrightarrow_{\mathcal{A}} T'$ be a strong $\mathcal{A}$-morphism where $T = H[b]$, $T' = H'[b']$. Furthermore let $\rho(T) = X[a]$, $\rho(T') = X'[a']$

and let $\varphi\colon \rho(T) \twoheadrightarrow_{\mathcal{A}} T$, $\varphi'\colon \rho(T') \twoheadrightarrow_{\mathcal{A}} T'$ be the corresponding morphisms. If we restrict ψ to external nodes we obtain a morphism $\psi'\colon X \to X'$ such that that $\psi \circ \varphi = \varphi' \circ \psi'$.

It is left to show that $\mathcal{A}_{\psi'}(a) \leq a'$. It holds that $a = \bigvee\{c \mid \mathcal{A}_{\varphi}(c) \leq b\}$ and hence $\mathcal{A}_{\psi'}(a) = \bigvee\{\mathcal{A}_{\psi'}(c) \mid \mathcal{A}_{\varphi}(c) \leq b\}$ and $a' = \bigvee\{c' \mid \mathcal{A}_{\varphi'}(c') \leq b'\}$. In order to show the inequality, we will show that every element in the first set is contained in the second set. We take a lattice element $\mathcal{A}_{\psi'}(c)$ such that $\mathcal{A}_{\varphi}(c) \leq b$. It follows that $\mathcal{A}_{\varphi'}(\mathcal{A}_{\psi'}(c)) = \mathcal{A}_{\varphi'\circ\psi'}(c) = \mathcal{A}_{\psi\circ\varphi}(c) = \mathcal{A}_{\psi}(\mathcal{A}_{\varphi}(c)) \leq \mathcal{A}_{\psi}(b) \leq b'$, since ψ is an $\mathcal{A}$-morphism.

We now examine Condition (7). Since there are morphisms $\rho(T_i) \twoheadrightarrow_{\mathcal{A}} T_i$, the existence of a strong morphism

$$\textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i) \twoheadrightarrow_{\mathcal{A}} \textcircled{D}_{i=1}^{n}(T_i, \zeta_i)$$

follows from Proposition 2. Hence the existence of a strong morphism

$$\iota\colon \rho(\textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i)) \twoheadrightarrow_{\mathcal{A}} \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i))$$

follows from Condition (8). It is straightforward to see that ι is an isomorphism on the underlying graphs. □

Example 4

- The reduction to external nodes satisfies Condition (7), if the lattice $\mathcal{A}(H)$ for a hypergraph H consists of all mappings of the form $a\colon V_H^n \to L$ where L is an arbitrary fixed lattice (see also the example after Definition 2).
- Now, as a counterexample, consider the family of lattices

  $$\mathcal{A}(H) = (\{V' \mid d(H) \subseteq V' \subseteq V_H\}, \subseteq),$$

  where $d(H) = \{v \in V_H \mid \exists e \in E_H\, \exists i, j\colon (i \neq j \wedge \lfloor c_H(e)\rfloor_i = v = \lfloor c_H(e)\rfloor_j)\}$, i.e., $d(H)$ is the set of all nodes of H that appear more than once in a sequence of nodes attached to an edge e. For a morphism $\varphi\colon G \to H$ and $V \in \mathcal{A}(G)$ we define $\mathcal{A}_{\varphi}(V) = \varphi(V) \cup d(H)$, which gives us an annotation mapping.
  Now let $H = [\ell]_2$ be a 2-ary edge with an arbitrary label ℓ, $\chi_H = v_1v_2$ and annotation $V' = \varnothing$. We consider the unique morphism $\zeta\colon \mathbf{2} \to \mathbf{1}$. That is, the effect of graph construction with ζ is to glue the two nodes of a 2-ary graph together.
  It holds that $\rho(H[V']) = \mathbf{2}[\varnothing]$ and thus $\rho(\textcircled{D}(\rho(H[V']), \zeta)) = \mathbf{1}[\varnothing]$. On the other hand $\rho(\textcircled{D}(H[V'], \zeta)) = \rho(\mathbf{1}[\{w\}]) = \mathbf{1}[\{w\}]$, where w is the single node of the discrete graph $\mathbf{1}$.

We present another closure operator which actually satisfies Condition (7) and which will be used in Sect. 4.1.

Proposition 5 *Let $\mathcal{A}(H)$ be the lattice consisting of all transitive irreflexive relations $R \subseteq V_H \times V_H$ with $\subseteq$ as partial order. In order to obtain a proper lattice we add $\top$ as top element.*

For a morphism $\varphi\colon G \to H$ and $R \subseteq (V_G \times V_G) \cup \{\top\}$, we define

$$\mathcal{A}_{\varphi}(R) = \begin{cases} TC(\varphi(R)) & \text{if } R \neq \top \text{ and } TC(\varphi(R)) \text{ is irreflexive} \\ \top & \text{otherwise} \end{cases}$$

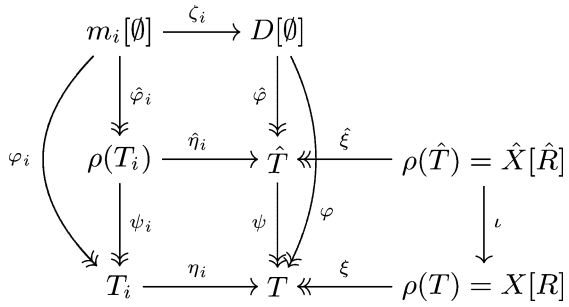

Fig. 8 Figure for the proof of Proposition 5

where TC stands for the transitive closure operation. Then $\mathcal{A}$ is an annotation mapping and the reduction to external nodes as defined in Definition 8 satisfies Condition (7).

Proof It is straightforward to check that $\mathcal{A}$ is a valid annotation mapping.

First, let us observe that the effect of reduction to the external nodes can in this case be described as follows: If $\varphi\colon X \twoheadrightarrow_{\mathcal{A}} H$ where X consists of the external nodes of H and $R \in \mathcal{A}(H)$ with $R \neq \top$, then

$$red_\varphi(R) = \{(w_1, w_2) \in V_X \times V_X \mid (\varphi(w_1), \varphi(w_2)) \in R\}.$$

It is easy to check that this relation is transitive and irreflexive.

We set $\hat{T} = \textcircled{D}_{i=1}^{n}(\rho(T_i), \zeta_i)$, $T = \textcircled{D}_{i=1}^{n}(T_i, \zeta_i)$, $X[R] = \rho(T)$ and $\hat{X}[\hat{R}] = \rho(\hat{T})$. According to Proposition 4 there exists a strong morphism $\iota\colon \hat{X}[\hat{R}] \twoheadrightarrow_{\mathcal{A}} X[R]$, which is an isomorphism on the underlying graphs $\hat{X}$, X and which satisfies $\mathcal{A}_\iota(\hat{R}) \subseteq R$. It is left to show that $\mathcal{A}_\iota(\hat{R}) \supseteq R$.

The situation is equivalent to the ones in the proofs of Proposition 2 and 4 and is depicted in Fig. 8 as a commuting diagram.

We assume that $R \neq \top$ and let (w_1, w_2) be a pair of nodes that is contained in R. It follows that $(\xi(w_1), \xi(w_2))$, where both nodes $\xi(w_1)$ and $\xi(w_2)$ are external, is contained in the annotation of T. The annotation of T has the form $TC(\bigcup_{i=1}^{n} TC(\eta_i(R_i))) = TC(\bigcup_{i=1}^{n} \eta_i(R_i))$, where R_i is the annotation of T_i for $i \in \{1, \ldots, n\}$. Since $R \neq \top$, we can assume that $R_i \neq \top$ for all i.

In other words, there is a sequence of pairs $(v_1, i_1), \ldots, (v_k, i_k)$ where $i_j \in \{1, \ldots, n\}$ and v_j is a node of T_{i_j} such that (i) $\eta_{i_1}(v_1) = \xi(w_1)$ and $\eta_{i_k}(v_k) = \xi(w_2)$; (ii) either $i_j = i_{j+1}$ and $(v_j, v_{j+1}) \in R_{i_j}$ or $v_j \approx v_{j+1}$ where $\approx$ is the equivalence defined in Definition 3.

Since two different nodes can only be related by the $\approx$-relation if both are external, and furthermore the relations R_i are transitive, we can assume without loss of generality that all nodes $v_1, \ldots, v_k$ are external. Hence there exists a sequence of pairs $(u_1, i_1), \ldots, (u_k, i_k)$ where u_j is a node of $\rho(T_{i_j})$ such that (iii) $\eta_{i_1}(\psi_{i_1}(u_1)) = \psi(\hat{\eta}_{i_1}(u_1)) = \xi(w_1)$ and $\eta_{i_k}(\psi_{i_k}(u_k)) = \psi(\hat{\eta}_{i_k}(u_k)) = \xi(w_2)$; (iv) either $i_j = i_{j+1}$ and (u_j, u_{j+1}) is contained in the annotation of $\rho(T_{i_j})$ or $u_j \approx u_{j+1}$.

This implies that $(\hat{\eta}_{i_1}(u_1), \hat{\eta}_{i_k}(u_k))$, where both nodes are external in $\hat{T}$, is contained in the annotation of $\hat{T}$ and there is a pair (y_1, y_2) in $\hat{R}$ such that $\hat{\xi}(y_1) =$

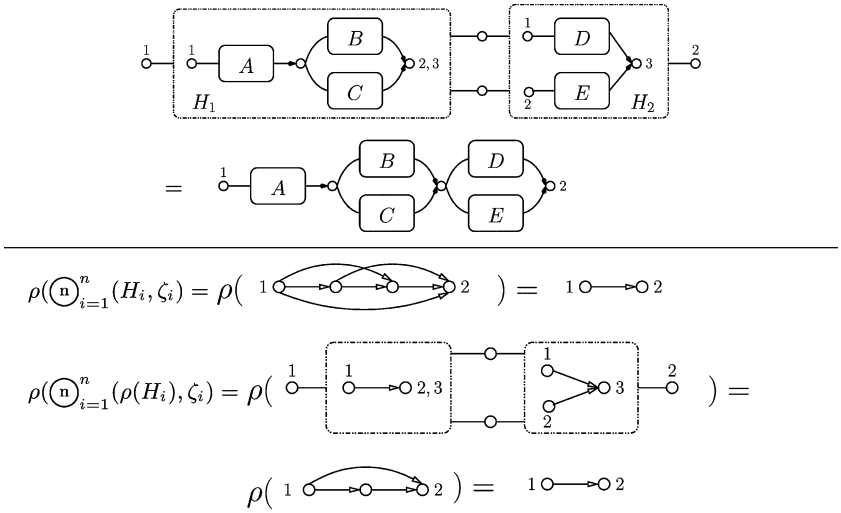

Fig. 9 Example: Condition (7) holds for annotations as defined in Proposition 5

$\hat{\eta}_{i_1}(u_1)$ and $\hat{\xi}(y_2) = \hat{\eta}_{i_k}(u_k)$. It holds that $\xi(\iota(y_1)) = \psi(\hat{\xi}(y_1)) = \psi(\hat{\eta}_{i_1}(u_1)) = \xi(w_1)$ and similarly $\xi(\iota(y_1)) = \xi(w_2)$. Since ξ is injective it holds that $\iota(y_1) = w_1$ and $\iota(y_2) = w_2$. Hence (w_1, w_2) is contained in $\mathcal{A}_\iota(\hat{R})$.

If, on the other hand, $R = \top$, then we obtain $\hat{R} = \top$ by a similar reasoning. □

Example 5 Figure 9 shows an instance graph construction and shows that Condition (7) is satisfied for this example. A pair (v_1, v_2) contained in the relation is represented by an unlabelled arrow.

Naturally, the closure operators presented in this section, have to be computed, respectively implemented if we want to mechanize this general framework. While computing the restriction to external nodes (see Definition 8) seems to be rather straightforward, it is less clear how to give an algorithm for computing the closure defined by a universal property (see Definition 3). Examples, however, suggest that in most practical cases it is quite straightforward to derive a method for computing the closure.

4 Case studies

4.1 Typing acyclic graphs

As a first simple example we present a type system for typing acyclic graphs. Acyclicity is an important concept one wants to verify in the verification of dynamically changing data structures such as trees or lists. It is a common property to be checked in shape analysis [45]. Such a type system might be also useful for detecting deadlocks caused by cyclic dependencies in the resource graph.

In this section we assume that all edges are binary, i.e., for every edge e in a graph H it holds that $ar(e) = 2$. There should be no problem to define a similar

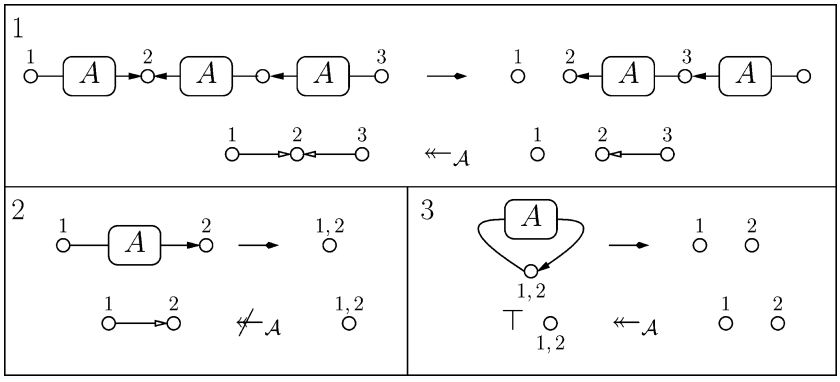

Fig. 10 Examples of typable and non-typable rules

type system for hypergraphs, provided one has fixed some notion of acyclicity for hypergraphs.

The type system consists of the following components:

- An annotation mapping $\mathcal{A}$, which assigns to a graph H the set of all transitive irreflexive relations over the nodes of H as described in Proposition 5.
- A linear mapping t, which assigns to every binary edge $[\ell]_2$ the graph type $\mathbf{2}[\{(v_1, v_2)\}]$ where $\chi_{\mathbf{2}} = v_1 v_2$
- A closure operator ρ which corresponds to the reduction of a graph type to its external nodes, according to Definition 8.
- A predicate X on hypergraphs where $X(H) = \mathit{true}$ whenever H is acyclic.
- And finally a predicate Y on graph types where $Y(T) = \mathit{true}$ whenever the annotation of T is different from $\top$.

From Propositions 4 and 5 we can derive that this type system satisfies Conditions (7) and (8). We can easily check that Conditions (6) and (9) are satisfied as well and that $(\mathcal{A}, t, \rho, X, Y)$ is indeed a type framework according to Definition 6.

Example 6 In order to give an example we consider the three rules in Fig. 10. The types of the left-hand and right-hand sides are given as well. Rule 1 is a typical example of a rule that can be typed, in this case the right-hand side does not introduce additional connectedness compared to the left-hand side. Rule 2 can not be typed since there is no strong morphism from a graph consisting of one node only into the discrete graph **2**. Note also that the application of this rule might destroy the acyclicity of a graph; for instance, if there is another edge in parallel with the A-edge, it is turned into a loop by applying this production. Rule 3 however satisfies Condition (2), specifically the annotation of its left-hand side is $\top$, the largest element in the lattice. Intuitively, any graph to which this rule can be applied, is already cyclic and hence no harm can be done by applying this rule.

Motivated by this example we can now state a sort of completeness result: For every rule r that can not be typed, there is an acyclic graph H such that the application of r to H at an appropriate position creates a cycle.

Proposition 6 *Let $r = (L, R)$ be a rewriting rule which can not be typed and the annotation of $\rho(t(L))$ is not equal to $\top$. Then there exists an acyclic graph H such that r can be applied to H resulting in a graph H' with a cycle.*

Proof Let $X_L[a_L] = \rho(t(L))$ and $X_R[a_R] = \rho(t(R))$ with $a_L \neq \top$ and there is no strong $\mathcal{A}$-morphism $X_R[a_R] \twoheadrightarrow_{\mathcal{A}} X_L[a_L]$. This fact can be due to the following two reasons:

- There is no strong morphism on the underlying graphs. That is, there are indices i, j with $i \neq j$ such that $\lfloor \chi_R \rfloor_i = \lfloor \chi_R \rfloor_j$, but $\lfloor \chi_L \rfloor_i \neq \lfloor \chi_L \rfloor_j$.
  We construct a graph H by adding an edge to the left-hand side L. If the pair $(\lfloor \chi_{X_L} \rfloor_i, \lfloor \chi_{X_L} \rfloor_j)$ is contained in a_L we add a binary edge with source node $\lfloor \chi_L \rfloor_i$ and target node $\lfloor \chi_L \rfloor_j$. Otherwise we add a binary edge going in the opposite direction. Since $a_L \neq \top$ and hence L is acyclic, the resulting graph will still be acyclic.
  Application of the rewriting rule r will turn this additional edge into a loop.
- There is a strong morphism $\varphi\colon X_R \twoheadrightarrow X_L$ on the underlying graphs, but $\mathcal{A}_\varphi(a_R) \not\leq a_L$. We distinguish two cases:
  - If $a_R = \top$, then the right-hand side R already contains a cycle. Setting $H = L$, we can replace H by R, thus obtaining a cycle.
  - If $a_R \neq \top$, then there must be a pair $(\lfloor \chi_{X_R} \rfloor_i, \lfloor \chi_{X_R} \rfloor_j) \in a_R$ such that $(\lfloor \chi_{X_L} \rfloor_i, \lfloor \chi_{X_L} \rfloor_j) \notin a_L$. Again we add a binary edge with source $\lfloor \chi_L \rfloor_j$ and target $\lfloor \chi_L \rfloor_i$ to L, obtaining a graph H. Since there is no path from $\lfloor \chi_L \rfloor_i$ to $\lfloor \chi_L \rfloor_j$ in L, adding this edge does not introduce a cycle. We replace L in H by R and obtain H'. The graph H' contains an edge from $\lfloor \chi_{H'} \rfloor_j$ to $\lfloor \chi_{H'} \rfloor_i$ (the new edge) and a path from $\lfloor \chi_{H'} \rfloor_i$ to $\lfloor \chi_{H'} \rfloor_j$ (which is contained in the right-hand side), and hence a cycle.

Variants: A simple variant of this type system would be to check the acyclicity of certain edges with specific edge labels only. This can be easily achieved by modifying the linear mapping t.

4.2 A type system for the π-calculus

In the next two sections we will show that it is possible to embed existing type systems into the framework. We will do so by presenting two type systems, one for the π-calculus and the other for the λ-calculus, and by showing what form the calculi and their type systems take in the graph rewriting framework.

We first give a short introduction to the asynchronous polyadic π-calculus [36] without choice and matching, already introduced in [27]. We assume that $\mathcal{N}$ is a fixed set of names, $c \in \mathcal{N}$ and $\tilde{a}, \tilde{x} \in \mathcal{N}^*$. The syntax of a process can be described as follows :

$$
\begin{array}{lll}
p ::= & \mathbf{0} & \text{(nil process)} \\
\mid & (\nu c)p & \text{(restriction)} \\
\mid & \overline{c}\langle \tilde{a} \rangle & \text{(output)} \\
\mid & c(\tilde{x}).p & \text{(input)} \\
\mid & p_1 | p_2 & \text{(parallel composition)} \\
\mid & !p & \text{(replication)} \\
\mid & \mathit{wrong} & \text{(error)}
\end{array}
$$

Table 1 Operational semantics of the π-calculus

Rules of Structural Congruence:

$$p_1|p_2 \equiv p_2|p_1 \qquad p_1|(p_2|p_3) \equiv (p_1|p_2)|p_3 \qquad (\nu c)(\nu b)p \equiv (\nu b)(\nu c)p$$
$$(\nu c)\mathbf{0} \equiv \mathbf{0} \qquad ((\nu c)p_1)|p_2 \equiv (\nu c)(p_1|p_2) \text{ if } c \notin fn(p_2)$$
$$p|\mathbf{0} \equiv p \qquad !p \equiv !p|p$$
$$!\,wrong \equiv wrong \qquad wrong|p \equiv wrong \qquad (\nu c)wrong \equiv wrong$$

Reduction Rules:

$$c(\tilde{x}).p \mid \overline{c}\langle\tilde{a}\rangle \to p\{\tilde{a}/\tilde{x}\} \quad \text{if } |\tilde{a}| = |\tilde{x}|$$
$$c(\tilde{x}).p \mid \overline{c}\langle\tilde{a}\rangle \to wrong \quad \text{if } |\tilde{a}| \neq |\tilde{x}|$$
$$\frac{p \to p'}{p|q \to p'|q} \qquad \frac{p \to p'}{(\nu c)p \to (\nu c)p'} \qquad \frac{q \equiv p,\ p \to p',\ p' \equiv q'}{q \to q'}$$

We can now define the operational semantics of the π-calculus: Structural congruence $\equiv$ is the smallest congruence closed under renaming of bound names (α-conversion) and under the rules given in Table 1. The rules generating the reduction relation $\to$ are also listed in Table 1. By $p\{\tilde{a}/\tilde{x}\}$ we denote the substitution of the names $\lfloor\tilde{x}\rfloor_i$ by $\lfloor\tilde{a}\rfloor_i$ in p (with possible α-conversion in order to avoid capture). Furthermore $fn(p)$ denotes the set of free names of a process p.

A process reduces to *wrong* whenever an arity mismatch or "bad redex" occurs. The variant of the type system that we will study ensures that a typable process will never contain such a bad redex. There are other, more complex, type sytems for the π-calculus checking more complicated properties, such as [32, 41] among others.

Our next step is to present a graph rewriting semantics for this variant of the π-calculus (see also [27]). Other ways of encoding the π-calculus into graph rewriting can be found in [16, 17, 48].

We will then apply the theory presented in Sect. 3, introduce a type system avoiding runtime errors produced by mismatching arities and show that it satisfies the conditions of Theorem 1. Afterwards we show that a graph has a type if and only if the corresponding π-calculus process has a type in a standard type system with infinite regular trees.

Definition 9 (Process Graphs) A *process graph* P is inductively defined as follows: P is a hypergraph with a duplicate-free string of external nodes. Furthermore each edge e is either labelled $(k, n)Q$ where Q is again a process graph, $0 \leq n \leq ar(Q)$ and $1 \leq k \leq ar(e) = ar(Q) - n$ (e is a process waiting for a message with n ports arriving at its k-th node), with $!Q$ where $ar(Q) = ar(e)$ (e is a process able to replicate itself) or with the constant M (e is a message sent to its last node).

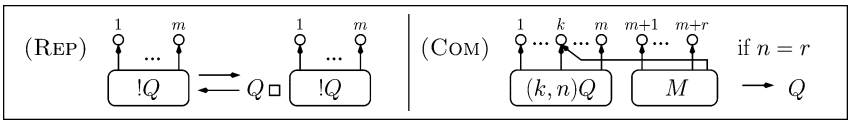

Fig. 11 Operational semantics for process graphs

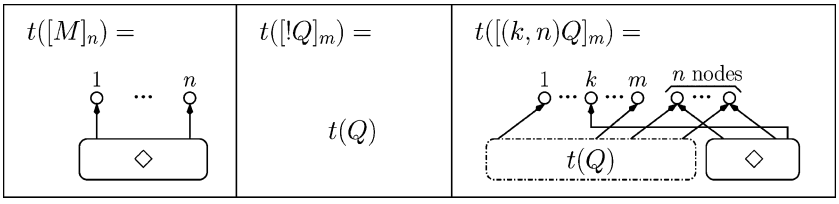

Fig. 12 The linear mapping for process graphs

The reduction relation is generated by the rules (Rep) (replication) and (Com) (reception of a message by a process) in Fig. 11 and is closed under isomorphism and graph construction. In the definition of replication we reuse the operator □ defined earlier in Example 3.

A process graph contains a bad redex if it contains a subgraph corresponding to the left-hand side of rule (Com) with $n \neq r$. We define the predicate X as follows: $X(P)$ if and only if P does not contain a bad redex.

We now propose a type system for process graphs by defining the mappings t and ρ. (Note that in this case, the graph types are trivially annotated by $\bot$ and so we omit the annotation mapping.)

The linear t mapping is defined on the hyperedges as follows: $t([M]_n) = [\diamond]_n$ where $\diamond$ is a new edge label, $t([!Q]_m) = t(Q)$ (see Fig. 12) and $t([(k,n)Q]_m)$ is defined in Fig. 12 (in the notation explained in Example 2). It holds that $n + m = ar(Q)$.

The mapping ρ is defined according to Proposition 3 where C is as follows:

$$C(T) \iff$$
$$\forall e_1, e_2 \in E_T : (\lfloor c_T(e_1) \rfloor_{ar(e_1)} = \lfloor c_T(e_2) \rfloor_{ar(e_2)} \Rightarrow e_1 = e_2).$$

The linear mapping t extracts the communication structure from a process graph, i.e. an edge of the form $[\diamond]_n$ indicates that its nodes (except the last) might be sent or received via its last node. Then ρ makes sure that the arity of the arriving message matches the expected arity and that nodes that might get fused during reduction are already fused in $\rho(t(H))$. In order to achieve this, the mapping ρ iteratively merges all edges having a common last node. Whenever two such edges have a different arity, the closure is undefined.

The condition that we want to check is simply that $\rho(t(H))$ is defined. Thus we set $Y(T) = \mathit{true}$ for every graph type T.

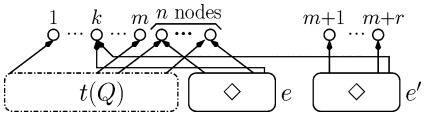

Fig. 13 A drawing of $t(Red)$, where Red is a bad redex

Proposition 7 *Let $\mathcal{A}$ be the trivial annotation mapping (where every lattice consists of a single element $\bot$) and let the mappings ρ and t and the predicates X and Y be defined as above. Then $(\mathcal{A}, t, \rho, X, Y)$ is a type framework for the rules* (REP) *and* (COM)*, i.e., Conditions (6)–(10) of Theorem 1 are satisfied.*

Thus if $P \triangleright T$, P will never produce a bad redex during reduction.

Proof We show that $\mathcal{A}, \rho, t, X, Y$ satisfy the conditions of Theorem 1.

(6) This obviously holds since $Y(T) = true$ for every graph T.

(7), (8) We have to show that the conditions of Proposition 3 are satisfied.
The universal property holds by definition and it is left to show that $\rho(T)$ is defined whenever there is a morphism $\varphi : T \to T'$ with $C(T')$. We are regarding equivalence relations $\sim$ on edges and nodes such that factoring through $\sim$ is well-defined. Specifically, it must be the case that $e \sim e'$ implies $\lfloor c_T(e) \rfloor_i \sim \lfloor c_T(e') \rfloor_i$ for all indices i.
We show the existence of a smallest such equivalence which additionally satisfies that $\lfloor c_T(e) \rfloor_{ar(e)} \sim \lfloor c_T(e') \rfloor_{ar(e')}$ implies $e \sim e'$. It can be shown that factoring through this equivalence gives us a graph isomorphic to $\rho(T)$.
It is easy to show that the intersection of two equivalences satisfying the above conditions is again an equivalence satisfying these conditions. So, in order to show that a smallest relation exists, it is enough to prove the existence of any such relation. The relation $\sim$ defined by $e \sim' e' \iff \varphi(e) = \varphi(e')$ and $v \sim' v' \iff \varphi(v) = \varphi(v')$ is such a relation. This implies that $\rho(T)$ is defined.

(9) Let $Y(\rho(t(H)))$ hold, which means that $\rho(t(H))$ is defined. Let us assume that H contains a bad redex Red, which implies that $t(H)$ contains $t(Red)$ which is depicted in Fig. 13 where $n \neq r$.
Furthermore $\rho(t(H))$ is defined only if $\rho(t(Red))$ is defined. We show that $\rho(t(Red))$ is undefined. The two edges to the right of $t(Red)$ are denoted by e respectively e'. If there were a morphism $\psi : t(Red) \twoheadrightarrow_{\mathcal{A}} \rho(t(Red))$, then it would hold that $\psi(e) = \psi(e')$, because of Condition C. This, however, is a contradiction, since $ar(e) \neq ar(e')$ and arities are preserved by morphisms.

(10) We show the local subject reduction property for both rules (in the case of (REP) for both directions).
(REP) We first consider the direction from right to left, i.e., $L = Q \square [!Q]_m$ and $R = [!Q]_m$. We assume that $\rho(t(Q \square [!Q]_m))$ is defined and since $\rho(t(Q \square [!Q]_m)) \cong \rho(\rho(t(Q)) \square t([!Q]_m))$ we know that $\rho(t(Q))$ is also defined and there exists a strong $\mathcal{A}$-morphism from $\rho(t(Q))$ into $\rho(t(Q \square [!Q]_m))$.
Concerning the direction from left to right, i.e. $L = [!Q]_m$ and $R = Q \square [!Q]_m$: we assume that $\rho(t(L)) \cong \rho(t(Q))$ is defined and therefore $\rho(t(Q)) \square \rho(t(Q)) \cong \rho(t(Q)) \square \rho(t([!Q]_m))$ is defined. Since there is an $\mathcal{A}$-morphism from this graph into $\rho(t(Q)) \cong \rho(t(L))$, it follows with

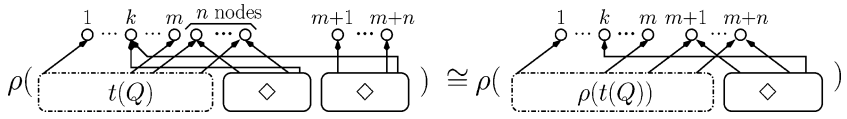

Fig. 14 Transformation of $\rho(t(L))$

Proposition 3, that

$$\rho(t(R)) \cong \rho(\rho(t(Q)) \square \rho(t([!Q]_m)))$$

is defined and that $\rho(t(R)) \twoheadrightarrow_{\mathcal{A}} \rho(\rho(t(L))) \cong \rho(t(L))$.

(COM) In this case $\rho(t(R)) \cong \rho(t(Q))$ and $t(L)$ is the graph type depicted above in Fig. 13 with $r = n$. With Condition (7) and the graph construction operation we can transform $\rho(t(L))$ as shown in Fig. 14.
It follows immediately that there is a strong morphism from $\rho(t(R)) \cong \rho(t(Q))$ into $\rho(t(L))$. □

We now compare our type system to a standard type system of the π-calculus. An encoding of process graphs into the asynchronous π-calculus can be defined as follows (see also [27]).

Definition 10 (Encoding) Let P be a process graph, let $\mathcal{N}$ be the name set of the π-calculus and let $\tilde{t} \in \mathcal{N}^*$ such that $|\tilde{t}| = ar(P)$. We define $\Theta_{\tilde{t}}(P)$ inductively as follows:

$$\Theta_{a_1 \ldots a_{n+1}}([M]_{n+1}) = \overline{a_{n+1}}\langle a_1, \ldots, a_n \rangle \qquad \Theta_{\tilde{t}}([!Q]_m) = !\,\Theta_{\tilde{t}}(Q)$$
$$\Theta_{a_1 \ldots a_m}([(k,n)Q]_m) = a_k(x_1, \ldots, x_n).\Theta_{a_1 \ldots a_m x_1 \ldots x_n}(Q)$$
$$\Theta_{\tilde{t}}(\textcircled{D}_{i=1}^{n}(P_i, \zeta_i)) = (\nu\, \mu(V_D \backslash Set(\chi_D)))(\Theta_{\mu(\zeta_1(\chi_{\mathbf{m_1}}))}(P_1) \mid \ldots \mid \Theta_{\mu(\zeta_n(\chi_{\mathbf{m_n}}))}(P_n))$$

where $\zeta_i : \mathbf{m_i} \to D$, $i \in \{1, \ldots, n\}$ and $\mu : V_D \to \mathcal{N}$ is a mapping such that μ restricted to $V_D \backslash Set(\chi_D)$ is injective, $\mu(V_D \backslash Set(\chi_D)) \cap \mu(Set(\chi_D)) = \varnothing$ and $\mu(\chi_D) = \tilde{t}$. Furthermore the names $x_1, \ldots, x_n \in \mathcal{N}$ are fresh.

The encoding of a discrete graph is included in the last case, if we set $n = 0$ and assume that the empty parallel composition represents the nil process $\mathbf{0}$.

An operational correspondence can be stated as follows:

Proposition 8 *Let p be an arbitrary expression in the asynchronous polyadic π-calculus without summation. Then there exists a process graph P and a duplicate-free string $\tilde{t} \in \mathcal{N}^*$ such that $\Theta_{\tilde{t}}(P) \equiv p$. Furthermore for process graphs P, P' and for every duplicate-free string $\tilde{t} \in \mathcal{N}^*$ with $|\tilde{t}| = ar(P) = ar(P')$ it is true that:*

- *$P \cong P'$ implies $\Theta_{\tilde{t}}(P) \equiv \Theta_{\tilde{t}}(P')$.*
- *$P \to^* P'$ implies $\Theta_{\tilde{t}}(P) \to^* \Theta_{\tilde{t}}(P)$.*
- *$\Theta_{\tilde{t}}(P) \to^* p \not\equiv$ wrong implies that $P \to^* Q$ and $\Theta_{\tilde{t}}(Q) \equiv p$ for some process graph Q.*
- *$\Theta_{\tilde{t}}(P) \to^*$ wrong iff $P \to^* P'$ for a process graph P' containing a bad redex.*

Table 2 Typing rules for the π-calculus

$$\Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \mid q} \qquad \frac{\Gamma \vdash p}{\Gamma \vdash !\, p} \qquad \frac{\Gamma, a : t \vdash p}{\Gamma \vdash (\nu a)\, p}$$

$$\frac{\Gamma(a) = [t_1, \ldots, t_m] \quad \Gamma, x_1 : t_1, \ldots, x_m : t_m \vdash p}{\Gamma \vdash a(x_1, \ldots, x_m).p} \qquad \frac{\Gamma(a) = [\Gamma(a_1) \ldots, \Gamma(a_m)]}{\Gamma \vdash \overline{a}\langle a_1, \ldots, a_m \rangle}$$

We now compare our type system with a standard type system of the π-calculus: a *type tree* is a regular tree, i.a., a potentially infinite ordered tree with only finitely many non-isomorphic subtrees. A type tree is represented by the tuple $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are again type trees, the children of the root. A type assignment $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ assigns names to type trees where $\Gamma(x_i) = t_i$. By $\Gamma, x : t$ we denote a type environment Γ with an additional assignment of the type t to the name x. If an assignment to x is already contained in Γ, then this assignment is overwritten by $x : t$.

The rules of the type system given in Table 2 are simplified versions of the ones from [40], obtained by removing the subtyping annotations.

We will now show that if a process graph has a type, then its encoding has a type in the π-calculus type system and vice versa. In order to express this we first describe the unfolding of a graph type into type trees, which we model using so-called consistent mappings.

Definition 11 (Consistent Mapping) Let T be a graph type and let σ be a mapping from V_T into the set of type trees. The mapping σ is called *consistent*, if it satisfies for every edge $e \in E_T$: $c_T(e) = v_1 \ldots v_n v \;\Rightarrow\; \sigma(v) = [\sigma(v_1), \ldots, \sigma(v_n)]$.

For consistent mappings we can show the following useful lemmas:

Lemma 1 *Every graph type of the form $\rho(t(P))$ has a consistent mapping.*

Proof We know that $\rho(t(P))$ satisfies condition C. So we can define: For any node v of T for which there is no edge $e \in V_T$ with $\lfloor c_T(e) \rfloor_{ar(e)} = v$, let $\sigma(v)$ be an arbitrary type tree. For all other nodes v there is a unique edge e with $\lfloor c_T(e) \rfloor_{ar(e)} = v$ and we set $\sigma(v) = [\sigma(v_1), \ldots, \sigma(v_n)]$ if $c_T(e) = v_1 \ldots v_n v$. The definition has a fixed-point, which is then our mapping σ. □

Lemma 2 *Let $\varphi : T \to T'$ be a graph morphism and let σ' be a mapping which is consistent for T'. We define $\sigma(v) = \sigma'(\varphi(v))$ for every $v \in V_T$. Then σ is consistent for T.*

Proof Let $e \in E_T$ with $c_T(e) = v_1 \ldots v_n v$. Then

$$\begin{aligned}
\sigma(v) &= \sigma'(\varphi(v)) = \sigma'(\varphi(\lfloor c_T(e) \rfloor_{n+1})) = \sigma'(\lfloor c_{T'}(\varphi(e)) \rfloor_{n+1}) \\
&= [\sigma'(\lfloor c_{T'}(\varphi(e)) \rfloor_1), \ldots, \sigma'(\lfloor c_{T'}(\varphi(e)) \rfloor_n)] \\
&= [\sigma'(\varphi(\lfloor c_T(e) \rfloor_1)), \ldots, \sigma'(\varphi(\lfloor c_T(e) \rfloor_n))] \\
&= [\sigma(\lfloor c_T(e) \rfloor_1), \ldots, \sigma(\lfloor c_T(e) \rfloor_n)] \\
&= [\sigma(v_1), \ldots, \sigma(v_n)].
\end{aligned}$$

□

Lemma 3 *Let T be a graph type which has a consistent mapping σ. Then it holds that $\rho(T)$ is defined and there is a consistent mapping σ' for $\rho(T)$ such that $\sigma'(\lfloor \chi_{\rho(T)} \rfloor_i) = \sigma(\lfloor \chi_T \rfloor_i)$.*

Proof We define an equivalence $\sim$ on the nodes and edges of T in the following way: $v_1 \sim v_2 \iff \sigma(v_1) = \sigma(v_2)$ and $e_1 \sim e_2 \iff \sigma(\lfloor c_T(e_1) \rfloor_{ar(e_1)}) = \sigma(\lfloor c_T(e_2) \rfloor_{ar(e_2)})$. Factoring T by $\sim$ yields a well-defined hypergraph $\hat{T} = T/\sim$ into which T can be mapped by a strong graph morphism. Furthermore it can be shown that $T/\sim$ has a consistent mapping $\sigma_\sim$, defined on the equivalence classes by $\sigma_\sim([v]_\sim) = \sigma(v)$.

If, for two edges $e = [e_1]_\sim, e' = [e_2]_\sim$ of $\hat{T}$ it holds that $\lfloor c_{\hat{T}}(e) \rfloor_{ar(e)} = \lfloor c_{\hat{T}}(e') \rfloor_{ar(e')}$, then $\sigma_\sim(\lfloor c_{\hat{T}}(e) \rfloor_{ar(e)}) = \sigma_\sim(\lfloor c_{\hat{T}}(e') \rfloor_{ar(e')})$, from which it follows that $\sigma(\lfloor c_T(e_1) \rfloor_{ar(e_1)}) = \sigma(\lfloor c_T(e_2) \rfloor_{ar(e_2)})$ and hence $e_1 \sim e_2$ and $e = e'$. Thus $\hat{T}$ satisfies Condition C.

Hence $\rho(T)$ is defined and there exist morphisms $\varphi : T \to \rho(T)$ and $\psi : \rho(T) \twoheadrightarrow_{\mathcal{A}} \hat{T}$ such that $\psi(\varphi(e)) = [e]_\sim$ and $\psi(\varphi(v)) = [v]_\sim$ for an edge e and a node v of T. We define a consistent mapping σ' for $\rho(T)$ as follows: $\sigma'(v') = \sigma_\sim(\psi(v'))$. Then it follows from Lemma 2 that σ' is consistent and furthermore $\sigma(\lfloor \chi_T \rfloor_i) = \sigma_\sim([\lfloor \chi_T \rfloor_i]_\sim) = \sigma_\sim(\lfloor \chi_{\hat{T}} \rfloor_i) = \sigma_\sim(\psi(\lfloor \chi_{\rho(T)} \rfloor_i)) = \sigma'(\lfloor \chi_{\rho(T)} \rfloor_i)$. □

Having shown these three lemmas we are now ready for the main result of this section. We show the correspondence between the original π-calculus type system and the type system for process graphs. Note that in the following proof we will often abbreviate a type environment of the form $x_1 : t_1, \ldots, x_n : t_n$ by $x_i : t_i$.

Proposition 9 *Let P a process graph with $P \triangleright T$. Let $n = ar(T)$ and let σ be a consistent mapping for T. Then it holds for every duplicate-free string $\tilde{t}$ of length n that*

$$\lfloor \tilde{t} \rfloor_1 : \sigma(\lfloor \chi_T \rfloor_1), \ldots, \lfloor \tilde{t} \rfloor_n : \sigma(\lfloor \chi_T \rfloor_n) \vdash \Theta_{\tilde{t}}(P).$$

Now let $\Gamma \vdash \Theta_{\tilde{t}}(P)$. Then there exists a graph type T such that $P \triangleright T$ and a consistent mapping σ such that for every $i \in \{1, \ldots, |\tilde{t}|\}$ it holds that $\sigma(\lfloor \chi_T \rfloor_i) = \Gamma(\lfloor \tilde{t} \rfloor_i)$.

Proof We prove the two main parts of this proposition, starting with the first part. Let $P \triangleright T$, which implies that there is a morphism $\varphi : \rho(t(P)) \twoheadrightarrow T$. Furthermore let σ' be a mapping which is consistent for T.

Now we can define a mapping σ on the nodes of $\rho(t(P))$ with $\sigma(v) = \sigma'(\varphi(v))$. From Lemma 2 it follows that σ is also consistent and furthermore σ and σ' coincide on the external nodes. So it is sufficient to show our claim for $T = \rho(t(P))$.

We will do so by induction on P but with a stronger induction hypothesis: let $\tilde{t}$ be a string of names (possibly with duplicates), σ be a consistent mapping for $T = \rho(t(P))$ such that $\lfloor \tilde{t} \rfloor_i = \lfloor \tilde{t} \rfloor_j$ implies $\sigma(\lfloor \chi_T \rfloor_i) = \sigma(\lfloor \chi_T \rfloor_j)$ (we will say that σ and $\tilde{t}$ are *compatible*). Then it holds that $\lfloor \tilde{t} \rfloor_1 : \sigma(\lfloor \chi_T \rfloor_1), \ldots, \lfloor \tilde{t} \rfloor_n : \sigma(\lfloor \chi_T \rfloor_n) \vdash \Theta_{\tilde{t}}(P)$.

From this we can derive the original claim, since $\tilde{t}$ is duplicate-free by definition. We will only prove the case of graph construction and omit the other (easier) cases.

– $P = \textcircled{D}_{i=1}^{n}(P_i, \zeta_i)$. We set $T_i = \rho(t(P_i))$ and in this case it holds that $T = \rho(t(P)) \cong \rho(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i))$.
Let $\overline{T} = \textcircled{D}_{i=1}^{n}(T_i, \zeta_i)$, let $\eta_i : T_i \to \overline{T}$ and $\varphi : D \to \overline{T}$ be the standard embeddings generated by graph construction and let $\psi : \overline{T} \twoheadrightarrow T$ be the morphism which exists according to the definition of ρ.
We define mappings σ_i on the T_i by setting $\sigma_i(v) = \sigma(\psi(\eta_i(v)))$. According to Lemma 2, the σ_i are consistent.
We now prove a property concerning σ and μ: Let $\mu : V_D \to N$ be the function defined in the encoding (Definition 10). We show that $\mu(v) = \mu(v')$ implies $\sigma(\psi(\varphi(v))) = \sigma(\psi(\varphi(v')))$. If $\mu(v) = \mu(v')$, then either $v = v'$ and the claim holds obviously, or there are indices i, j such that $v = \lfloor \chi_D \rfloor_i$, $v' = \lfloor \chi_D \rfloor_j$ and $\lfloor \tilde{t} \rfloor_i = \lfloor \mu(\chi_D) \rfloor_i = \mu(v) = \mu(v') = \lfloor \mu(\chi_D) \rfloor_j = \lfloor \tilde{t} \rfloor_j$. Then it follows with the fact that σ and $\tilde{t}$ are compatible, that $\sigma(\psi(\varphi(v))) = \sigma(\psi(\varphi(\lfloor \chi_D \rfloor_i))) = \sigma(\lfloor \chi_T \rfloor_i) = \sigma(\lfloor \chi_T \rfloor_j) = \sigma(\psi(\varphi(\lfloor \chi_D \rfloor_j))) = \sigma(\psi(\varphi(v')))$.
We show that σ_i and $\mu(\zeta_i(\chi_{\mathbf{m_i}}))$ are compatible. Let $\lfloor \mu(\zeta_i(\chi_{\mathbf{m_i}})) \rfloor_j = \lfloor \mu(\zeta_i(\chi_{\mathbf{m_i}})) \rfloor_k$. This implies that $\mu(\zeta_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_j)) = \mu(\zeta_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_k))$ and it follows with the property shown above that

$$\begin{aligned}\sigma_i(\lfloor \chi_{T_i} \rfloor_j) &= \sigma(\psi(\eta_i(\lfloor \chi_{T_i} \rfloor_j))) = \sigma(\psi(\varphi(\zeta_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_j)))) \\ &= \sigma(\psi(\varphi(\zeta_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_k)))) = \sigma(\psi(\eta_i(\lfloor \chi_{T_i} \rfloor_k))) = \sigma_i(\lfloor \chi_{T_i} \rfloor_k).\end{aligned}$$

Then the induction hypothesis implies that

$$\Gamma_i = \lfloor \mu(\zeta_i(\chi_{\mathbf{m_i}})) \rfloor_j : \sigma_i(\lfloor \chi_{T_i} \rfloor_j) \vdash \Theta_{\mu(\zeta_i(\chi_{\mathbf{m_i}}))}(P_i).$$

Now let Δ be a type assignment, containing all assignments of the form $\mu(v) : \sigma(\psi(\varphi(v)))$ for all $v \in V_D$. From $(\mu(v) = \mu(v') \Rightarrow \sigma(\psi(\varphi(v))) = \sigma(\psi(\varphi(v'))))$ shown above, it follows that Δ is well-defined and since $\sigma_i(\lfloor \chi_{T_i} \rfloor_j) = \sigma(\psi(\varphi(\zeta_i(\chi_{\mathbf{m_i}}))))$, it follows that Δ can be obtained from any Γ_i by adding extra type assignments. Thus it follows by weakening that $\Delta \vdash \Theta_{\mu(\zeta_i(\chi_{\mathbf{m_i}}))}(P_i)$. With the rule for parallel composition it follows that

$$\Delta \vdash \Theta_{\mu(\zeta_1(\chi_{\mathbf{m_1}}))}(P_1) \mid \ldots \mid \Theta_{\mu(\zeta_n(\chi_{\mathbf{m_n}}))}(P_n).$$

Now let Γ be a type assignment containing all assignments of the form $\mu(v) : \sigma(\psi(\varphi(v)))$ for $v \in Set(\chi_D)$. The restriction rule of the π-calculus type system implies that

$$\Gamma \vdash (\nu\, \mu(V_D \backslash Set(\chi_D)))(\Theta_{\mu(\zeta_i(\chi_{\mathbf{m_i}}))}(P_1) \mid \ldots \mid \Theta_{\mu(\zeta_i(\chi_{\mathbf{m_i}}))}(P_n)),$$

that is $\Gamma \vdash \Theta_{\tilde{t}}(P)$. It is left to show that $\Gamma = \lfloor \tilde{t} \rfloor_i : \sigma(\lfloor \chi_T \rfloor_i)$. This is quite straightforward, since $\mu(v), v \in Set(\chi_D)$ is exactly $\lfloor \tilde{t} \rfloor_i$ if $v = \lfloor \chi_D \rfloor_i$, and in this case $\sigma(\psi(\varphi(v))) = \sigma(\psi(\varphi(\lfloor \chi_D \rfloor_i))) = \sigma(\lfloor \chi_T \rfloor_i)$.

Now let $\Gamma \vdash \Theta_{\tilde{t}}(P)$, where $\tilde{t}$ is a duplicate-free sequence of names. We show that $T = \rho(t(P))$ is defined and that there is a mapping σ consistent with T, such that $\sigma(\lfloor \chi_T \rfloor_i) = \Gamma(\lfloor \tilde{t} \rfloor_i)$. We proceed by induction on P. Again we prove only one case.

– $P = \textcircled{D}_{i=1}^{n}(P_i, \zeta_i)$ which implies that

$$\Theta_{\tilde{t}}(P) = (\nu\ \mu(V_D \backslash Set(\chi_D)))(\Theta_{\mu(\zeta_1(\chi_{\mathbf{m_1}}))}(P_1) \mid \ldots \mid \Theta_{\mu(\zeta_n(\chi_{\mathbf{m_n}}))}(P_n))$$

and $\mu : V_D \to N$ and $\mu(\chi_D) = \tilde{t}$.
It follows from the typing rules of the π-calculus that there is a type assignment Δ, which is exactly Γ enriched by type assignments of the form $\mu(v) : t_v$ for all $v \in V_D \backslash Set(\chi_D)$ and that $\Delta \vdash \Theta_{\mu(\zeta_i(\chi_{\mathbf{m_i}}))}(P_i)$.
The induction hypothesis thus implies that all $T_i = \rho(t(P_i))$ are defined and that there are mappings σ_i consistent with T_i such that $\sigma_i(\lfloor \chi_{T_i} \rfloor_j) = \Delta(\lfloor \mu(\zeta_i(\chi_{\mathbf{m_i}})) \rfloor_j)$.
Now we define $\hat{T} = \textcircled{D}_{i=1}^{n}(T_i, \zeta_i)$. We know that $T = \rho(t(P))$ is defined if and only if $\rho(\hat{T})$ is defined. In order to show this, we define a consistent mapping $\hat{\sigma}$ for $\hat{T}$. We assume that $\eta_i : T_i \to \hat{T}$, $\varphi : D \twoheadrightarrow \hat{T}$ are the standard embeddings generated by the graph construction and we define $\hat{\sigma}$ as a consistent mapping for $\hat{T}$ in the following way:

$$\hat{\sigma}(\hat{v}) = \begin{cases} \sigma_i(v) & \text{if } \hat{v} = \eta_i(v) \\ \Delta(\mu(v)) = t_v & \text{if } \hat{v} = \varphi(v) \end{cases}$$

We first have to show that $\hat{\sigma}$ is well-defined: let $\varphi_i : \mathbf{m_i} \twoheadrightarrow T_i$ be the unique strong morphisms from $\mathbf{m_i}$ into T_i. When we restrict all morphisms to the node sets, then the $(\eta_i)_V$ and φ_V are still the colimit of the $(\varphi_i)_V$ and the $(\zeta_i)_V$, since colimits in the category of hypergraphs are taken componentwise. We show that $\Delta \circ \mu \circ (\zeta_i)_V = \sigma_i \circ (\varphi_i)_V$:

$$\Delta(\mu(\zeta_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_j))) = \sigma_i(\lfloor \chi_{T_i} \rfloor_j) = \sigma_i(\varphi_i(\lfloor \chi_{\mathbf{m_i}} \rfloor_j)).$$

Because of the colimit property, there must be a unique mapping $\hat{\sigma}$ from the nodes of $\hat{T}$ into the set of type trees such that $\hat{\sigma} \circ (\eta_i)_V = \sigma_i$ and $\hat{\sigma} \circ \varphi_V = \Delta \circ \mu$. This is exactly the mapping $\hat{\sigma}$ defined above.
It is left to show that $\hat{\sigma}$ is consistent for $\hat{T}$: Let e be one of the edges of $\hat{T}$. It follows that there must be a $e' \in E_{T_i}$ such that $e = \eta_i(e')$. If $c_{\hat{T}}(e) = v_1 \ldots v_{n+1}$, it follows that $v_j = \eta_i(\lfloor c_{T_i}(e') \rfloor_j)$. Thus

$$\begin{aligned} \hat{\sigma}(v_{n+1}) &= \hat{\sigma}(\eta_i(\lfloor c_{T_i}(e') \rfloor_{n+1})) = \sigma_i(\lfloor c_{T_i}(e') \rfloor_{n+1}) \\ &= [\sigma_i(\lfloor c_{T_i}(e') \rfloor_1), \ldots, \sigma_i(\lfloor c_{T_i}(e') \rfloor_n)] \\ &= [\hat{\sigma}(\eta_i(\lfloor c_{T_i}(e') \rfloor_1)), \ldots, \hat{\sigma}(\eta_i(\lfloor c_{T_i}(e') \rfloor_n))] \\ &= [\hat{\sigma}(v_1), \ldots, \hat{\sigma}(v_n)]. \end{aligned}$$

Lemma 3 implies that $T = \rho(\hat{T}) = \rho(t(P))$ is defined and that there is a mapping σ consistent with T such that $\sigma(\lfloor \chi_T \rfloor_i) = \hat{\sigma}(\lfloor \chi_{\hat{T}} \rfloor_i)$. Thus it holds that $\sigma(\lfloor \chi_T \rfloor_i) = \hat{\sigma}(\lfloor \chi_{\hat{T}} \rfloor_i) = \hat{\sigma}(\varphi(\lfloor \chi_D \rfloor_i)) = \Delta(\mu(\lfloor \chi_D \rfloor_i)) = \Delta(\lfloor \tilde{t} \rfloor_i)$. □

Example 7 As an example, we regard the process graph P depicted in Fig. 15. It consists of two edges, both able to replicate themselves, where the edge on the left-hand side waits for incoming messages on its first and only node. Each message should be equipped with two nodes, to the first of which another message is sent.

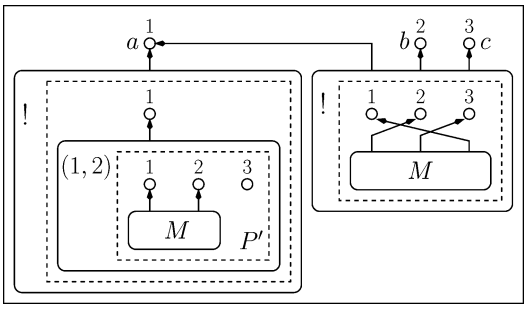

Fig. 15 An example process graph

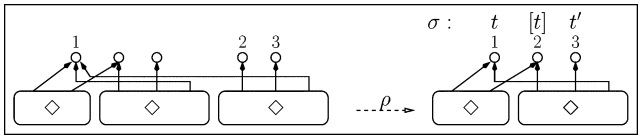

Fig. 16 Typing the example process graph

The edge on the right-hand side produces arbitrarily many messages to be received by the edge on the left-hand side. Note that this process graph has an infinite reduction sequence (not even counting the replication steps) and can evolve into arbitrarily many different processes. If we denote the innermost process graph of arity 3 on the left-hand side edge by P', then $P \to^* P \,\square\, P' \,\square\, \ldots \,\square\, P'$.

If we set $\tilde{t} = abc$, its π-calculus counterpart is

$$\Theta_{\tilde{t}}(P) = !\,a(x, y).\overline{x}\langle a\rangle \mid !\,\overline{a}\langle b, c\rangle = p,$$

which reduces to $!\,a(x, y).\overline{x}\langle a\rangle \mid !\,\overline{a}\langle b, c\rangle \mid \overline{b}\langle c\rangle \mid \ldots \mid \overline{b}\langle c\rangle$. The process p can be typed under the type assignment $\Gamma = a\colon t, b\colon [t], c\colon t'$ where t' is an arbitrary type tree and t is the solution of the fixed-point equation $t = [[t], t']$. Note that the infinity of the tree is not caused by replication, but rather by the fact that the left-hand side process emits its own name as the content of a message.

Now, computing $t(P)$ yields the graph type depicted in Fig. 16, where the edge in the middle is generated by applying t to the process abstraction $[(1, 2)Q]_1$, and the other two edges are generated by $t([M]_2)$ respectively $t([M]_3)$. Computing $\rho(t(P))$ fuses the two rightmost edges. We indicate a consistent mapping σ by mapping the nodes to appropriate type trees. This consistent mapping corresponds exactly to the type assignment Γ given above.

In a way, the graph-based type system is closer to Milner's sort system [36] for the π-calculus than to more recent type systems as presented above. In our setting sorts correspond to nodes and the object sort mapping of Milner's system can be immediately derived from the edge set of the type graph.

Variants: It is fairly simple to define variants of the type system that enable us to derive more specific properties of processes. We can obtain a simple type system typing input/output-behaviour of processes, by using an annotation mapping $\mathcal{A}$ that assigns to every node v an element of the lattice $(\{none, in, out, both\}, \leq)$

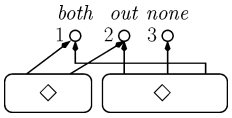

Fig. 17 Typing the example process graph with lattice annotations

where $none < in < both$ and $none < out < both$. The intuitive meaning of labelling a node v with *out* is, that there might be a message with v as target port. Similarly, if a node v is labelled *in*, it means that there might be a process listening at v for a message. The rest of the definition of the annotation mapping $\mathcal{A}$ is analogous to the one in Example 1.

The components of the type system are the following:

- A linear mapping t that is defined in analogy to the original linear mapping apart from the lattice annotation. In the case of $t([M]_n)$ we assign *out* to the last external node and *none* to all other nodes, whereas in the case of $t([(k,n)Q]_m)$ we use the annotation for $t(Q)$ and additionally take the join of the annotation of the k-th external node and *in*.
- The folding operation is again defined according to Proposition 3 with the exact same condition C.

So, from the point of graph structure, we obtain exactly the same graph types, only the annotations differ. There are several meaningful and useful ways to define the predicates X and Y, for instance:

- For a fixed index j define that $X(H)$ holds if there is no message edge e labelled M such that $\lfloor c_H(e) \rfloor_{ar(e)} = \lfloor \chi_H \rfloor_j$, i.e., there is no message attached to the j-th external node. The corresponding predicate Y is $Y(H[a]) = (a(\lfloor \chi_H \rfloor_j) \leq in)$.
- For a fixed index j define that $X(H)$ holds if there is no edge e labelled $(k,n)Q$ such that $\lfloor c_H(e) \rfloor_k = \lfloor \chi_H \rfloor_j$, i.e., there is no process listening at the j-th external node. The corresponding predicate Y is $Y(H[a]) = (a(\lfloor \chi_H \rfloor_j) \leq out)$.

Example 8 For the example process graph in Fig. 15 we obtain the graph type depicted in Fig. 17.

This is a simpler version of type systems checking input/output-capabilities of mobile processes [31, 32, 41] and has already been presented in [24, 30].

4.3 A type system for the λ-calculus

We will now present an encoding of λ-expressions into graphs—similar to the way this is done in term graph rewriting—and a type system for the resulting graphs. We proceed in a way rather similar to the previous section on type systems for the π-calculus. However, the encoding of the expressions and the type systems are of a different nature and thus give us a new perspective concerning graph types.

Table 3 Typing rules of the λ-calculus

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathsf{nat}} \quad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{bool}} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash M : \mathsf{nat},\ \Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash M + N : \mathsf{nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{nat},\ \Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash \mathsf{Eq}(M, N) : \mathsf{bool}} \quad \frac{\Gamma \vdash M : \mathsf{bool},\ \Gamma \vdash N_1 : \tau,\ \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \tau}$$

$$\frac{\Gamma \vdash M : \tau \to \sigma,\ \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} \quad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma}$$

We are using the following variant of the λ-calculus, a similar language is called PCF in [37]. We are using a fixed set X of variables. A λ-expression N is of the following form:

$N ::=$	$\lambda x.N$	(abstraction)
$\mid$	$N_1 N_2$	(application)
$\mid$	x	(variable)
$\mid$	b	(boolean)
$\mid$	n	(natural number)
$\mid$	$M + N$	(plus)
$\mid$	$\mathsf{Eq}(M, N)$	(equality)
$\mid$	$\mathsf{if}\ N\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2$	(if-then-else)

where N, M, N_1, N_2 are λ-expressions, $x \in X$ is a variable, $b \in \{\mathsf{true}, \mathsf{false}\}$ and $n \in \mathbb{N}$. The set of free variables of a λ-expression M is denoted by $free(M)$.

Expressions of the λ-calculus are typed by the typing rules given in Table 3, where we are using recursive types. A *type* τ has the following syntax:

$$\tau ::= \mathsf{bool} \mid \mathsf{nat} \mid \tau_1 \to \tau_2 \mid \mathsf{x} \mid \mu\mathsf{x}.\tau$$

The operator μ binds the variable x and it is used to create recursive types. We are equating a type with its "unfolding", which is always a regular tree, i.e., a (possibly infinite) tree with only finitely many subtrees. This is equivalent to factoring the set of types according to the congruence generated by $\mu\mathsf{x}.\tau = \tau\{\mu\mathsf{x}.\tau/\mathsf{x}\}$ on all types.

A *type environment* Γ is a set of assignments of the form $x : \tau$, where $x \in X$ and τ is a type. Each variable occurs at most once in every type environment. Again, we write $\Gamma(x) = \tau$ whenever $x : \tau$ appears in Γ.

Based on these definitions we can now give the typing rules of the λ-calculus, summarized in Table 3. We assume that all occurring types are closed expressions without free variables.

If a λ-expression M can be typed with type τ under any type environment, this guarantees absence of runtime errors for M, such as the application of a function to parameters of incorrect type. Furthermore, if we use types without recursion, termination for a well-typed λ-expression is guaranteed [37].

In order to encode λ-expressions into hypergraphs, we are using the following types of edges: 3-ary edges labelled λ, denoting function abstraction; 3-ary edges labelled @, denoting function application; unary edges labelled true, false or with a natural numbers $n \in \mathbb{N}$; 3-ary edges labelled Eq or $+$ and 4-ary edges labelled if.

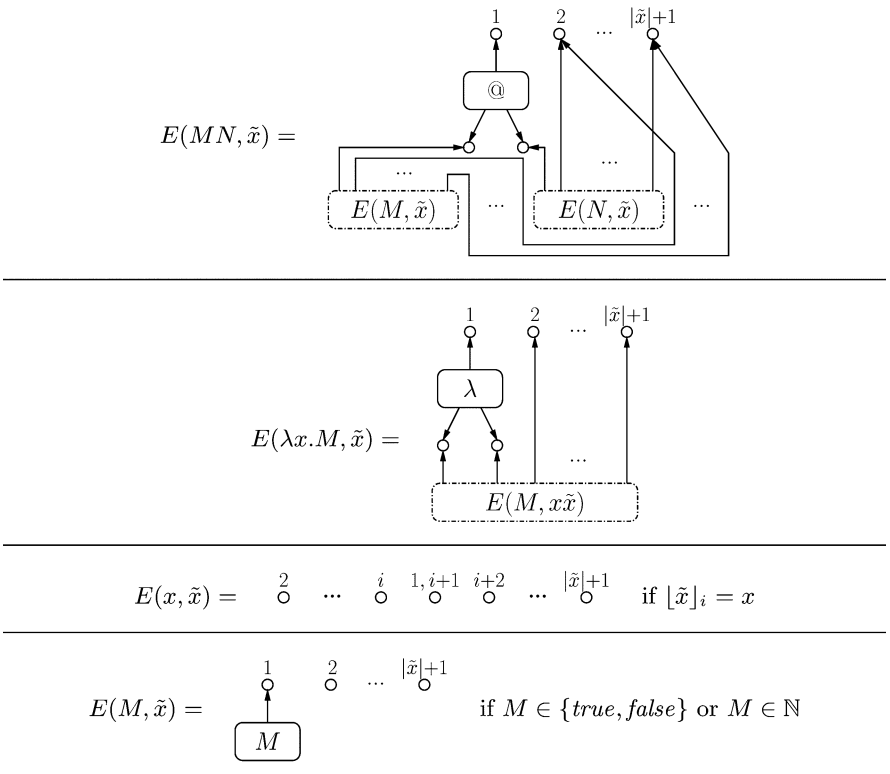

Fig. 18 Encoding λ-expressions as hypergraphs

Variables will not be represented by hyperedges, but by nodes. The encoding itself takes a λ-expression M and a duplicate-free string $\tilde{x} \in X^*$, such that $free(M) \subseteq Set(\tilde{x})$ and returns a graph $E(M, \tilde{x})$ of arity $|\tilde{x}| + 1$, where the first external node represents the expression itself and the other external nodes denote the free variables contained in $\tilde{x}$. We demand that all bound variables are different from each other and different from the free variables.

The encoding is inductively defined as depicted in Fig. 18. The encodings of plus, equality and if-then-else are defined by analogy to that of function application, producing edges labelled $+$, Eq and if. We assume that the tentacle pointing upwards from an edge is attached to the first node of an edge, whereas the positions of the tentacles pointing downwards are numbered from left to right in counter-clockwise order.

Example 9 Encoding the λ-expression $\lambda f.(f((\lambda x.(x+3))5))$ results in the hypergraph depicted on the left of Fig. 19.

This encoding is similar to the encodings used in term graph rewriting [3, 22, 47]. One of the main ideas behind these encodings is sharing: If a subterm occurs more than once in a λ-expression, it is represented only once and is pointed to

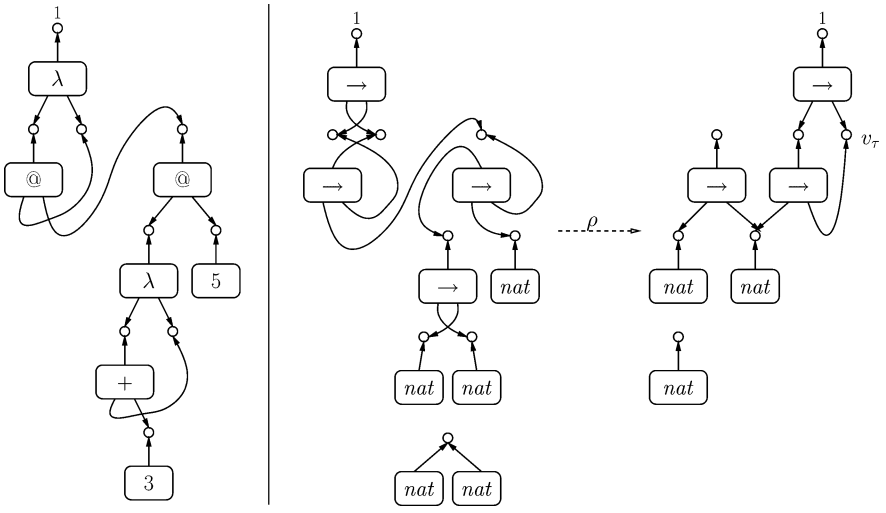

Fig. 19 The λ-graph for $\lambda f.(f\,((\lambda x.(x+3))5))$ (left) and its graph type (right)

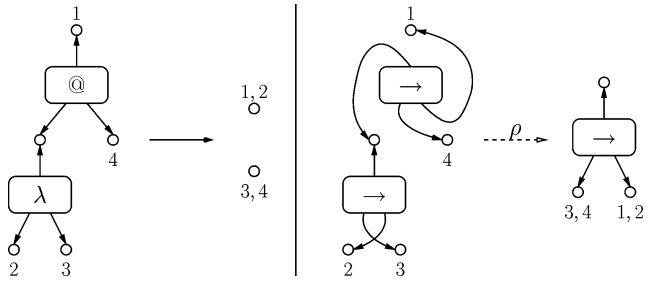

Fig. 20 β-reduction in the linear case and the typing of the left-hand side

from several places. This concept of sharing is not pursued in the encoding given below, but the type system would work just as well in the presence of sharing. Furthermore variables are represented by nodes instead of names.

By contrast with the encoding of the π-calculus we are not using a hierarchical encoding, but encode a λ-expression into a "flat" graph structure. The resulting graphs are called λ-*graphs*. Giving an operational semantics for such term graphs is not entirely trivial, since β-reduction or ensuing reductions may involve the copying of entire subgraphs. Since type systems are the main topic of this section, this issue is beyond the scope of this paper. We only give the rule for the linear case in Fig. 20 (left), where we assume that there is at most one occurrence of any variable in a λ-expression. The edges labelled λ and @ are removed and the remaining nodes are attached in such a way that the substitution is performed (attaching nodes 3 and 4) and that the expression underneath the λ-operator is moved to the top (attaching nodes 1 and 2).

In our graph types, we are remodelling the λ-calculus types defined above. We are using graphs without annotations, having the following edge labels: 3-ary edges labelled $\rightarrow$ and unary edges labelled either *bool* or *nat*.

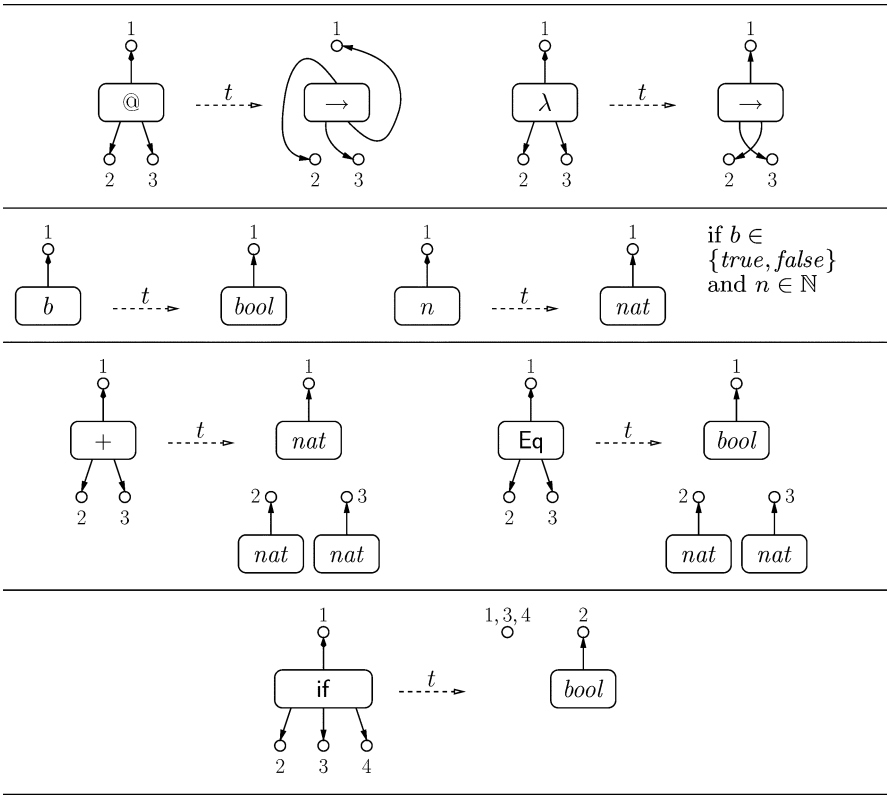

Fig. 21 Definition of the linear mapping t

Recursion is not modelled explicitly, but is represented by cycles in the graph. We are using the linear mapping t depicted in Fig. 21 and a closure operation ρ as in Proposition 3 with condition C defined as follows:

> A graph type T satisfies C, if for every two edges e_1, e_2 with $\lfloor c_T(e_1) \rfloor_1 = \lfloor c_T(e_2) \rfloor_1$ it holds that $e_1 = e_2$.

Again, $\mathcal{A}$ is the trivial annotation mapping, assigning a single-element lattice to every hypergraph. In a way very similar to Proposition 7 it can now be shown that Conditions (7) and (8) are satisfied.

Example 10 We come back to the previous example graph H depicted on the left-hand side of Fig. 19 and compute $\rho(t(H))$. On the right-hand side of Fig. 19 $t(H)$ and $\rho(t(H))$ are depicted. For the corresponding λ-expression it holds that

$$\Gamma \vdash \lambda f.(f((\lambda x.(x + 3))5)) \colon (nat \to \tau) \to \tau$$

for any type environment Γ and any type τ.

If we now regard the graph type in Fig. 19 we can reconstruct the type $(nat \to \tau) \to \tau$ starting from the only external node labelled 1. A node which is not the

first node of an $\rightarrow$-edge, can represent any type. This is true for the node v_τ in $\rho(t(H))$, this is the node representing the type τ. Everything that is not directly reachable from external nodes could in principle be garbage-collected.

Example 11 The rewriting rule (L, R) for the linear λ-calculus in Fig. 20 can be typed. On the right of Fig. 20 we depict $t(L)$ and its transformation into $\rho(t(L))$. The type $\rho(t(R))$ of the right hand side is equal to R itself and there clearly exists a strong morphism $\rho(t(R)) \twoheadrightarrow \rho(t(L))$.

There is a clear correspondence between the original type system for the λ-calculus and the graph-based type system. In analogy to Sect. 4.2 we first define the notion of a consistent mapping.

Definition 12 (Consistent Mapping) Let T be a graph type and let σ be a mapping from V_T into the set of λ-calculus types. The mapping σ is called *consistent*, if it satisfies for every edge $e \in E_T$:

- if $l_T(e) = \rightarrow$ and $c_T(e) = v_1 v_2 v_3$, then it holds that $\sigma(v_1) = \sigma(v_2) \rightarrow \sigma(v_3)$.
- if $l_t(e) = bool$ and $c_T(e) = v$, then it holds that $\sigma(v) = \mathsf{bool}$.
- if $l_t(e) = nat$ and $c_T(e) = v$, then it holds that $\sigma(v) = \mathsf{nat}$.

We will now state a series of lemmas that lead to the main result of this section. The proofs are similar to the proofs of the analogous Lemmas 1, 2 and 3 in Sect. 4.2 and are therefore not given here.

Lemma 4 *Every graph type of the form $\rho(t(P))$ has a consistent mapping.*

Lemma 5 *Let $\varphi : T \rightarrow T'$ be a graph morphism and let σ' be a mapping which is consistent for T'. We define $\sigma(v) = \sigma'(\varphi(v))$ for every $v \in V_T$. Then σ is consistent for T.*

Lemma 6 *Let T be a graph type which has a consistent mapping σ. Then it holds that $\rho(T)$ is defined and there is a consistent mapping σ' for $\rho(T)$ such that $\sigma'(\lfloor \chi_{\rho(T)} \rfloor_i) = \sigma(\lfloor \chi_T \rfloor_i)$.*

It is now possible to prove that a λ-expression can be typed if and only if its corresponding graph can be typed and furthermore the types are related in a specific way.

Proposition 10 *Let M be a λ-expression and let $\tilde{x}$ be a duplicate-free sequence of variables of length n such that free$(M) \subseteq Set(\tilde{x})$.*

Let $E(M, \tilde{x}) \triangleright T$ and let σ be a consistent mapping for T. Then it holds that $\lfloor \tilde{x} \rfloor_1 : \sigma(\lfloor \chi_T \rfloor_2), \ldots, \lfloor \tilde{x} \rfloor_n : \sigma(\lfloor \chi_T \rfloor_{n+1}) \vdash M : \sigma(\lfloor \chi_T \rfloor_1)$.

Now let $\Gamma \vdash M : \tau$ and let $Set(\tilde{x})$ be contained in the variables of Γ. Then there exists a graph type T such that $E(M, \tilde{x}) \triangleright T$ and a consistent mapping σ such that for every $i \in \{1, \ldots, n\}$ it holds that $\sigma(\lfloor \chi_T \rfloor_1) = \tau$ and $\sigma(\lfloor \chi_T \rfloor_{i+1}) = \Gamma(\lfloor \tilde{x} \rfloor_i)$ for $i \in \{1, \ldots, n\}$.

Proof We show the first part of the proposition by structural induction on M. It is sufficient to show the claim for $T = \rho(t(E(M, \tilde{x})))$. Otherwise let $T_0 = \rho(t(E(M, \tilde{x})))$ and let σ be a consistent mapping for T. There is a strong morphism $\varphi : T_0 \twoheadrightarrow_{\mathcal{A}} T$. We set $\sigma_0 = \sigma \circ \varphi$ and σ_0 is consistent according to Lemma 5. It follows that $\lfloor \tilde{x} \rfloor_1 : \sigma_0(\lfloor \chi_{T_0} \rfloor_2), \ldots, \lfloor \tilde{x} \rfloor_n : \sigma_0(\lfloor \chi_{T_0} \rfloor_{n+1}) \vdash M : \sigma(\lfloor \chi_{T_0} \rfloor_1)$ and since $\sigma_0(\lfloor \chi_{T_0} \rfloor_i) = \sigma(\lfloor \chi_T \rfloor_i)$, the claim of the proposition follows. For reasons of space we only prove one case.

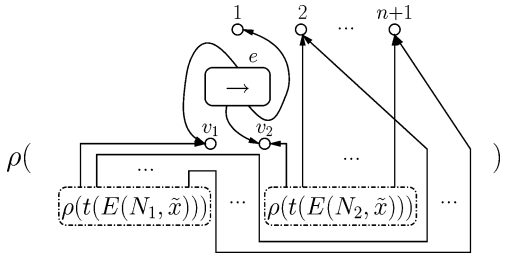

Fig. 22 $\rho(t(E(N_1N_2, \tilde{x})))$

application: Let M be of the form N_1N_2. In this case $T = \rho(t(E(M, \tilde{x})))$ has the form depicted in Fig. 22. We set $T_i = \rho(t(E(N_i, \tilde{x})))$ for $i \in \{1, 2\}$ and denote by T_0 the graph inside the closure operator ρ.
Clearly, there is a strong morphism $\psi\colon T_0 \to T$ and there are morphisms $\eta_i\colon T_i \to T_0$ and since T has a consistent mapping σ, Lemma 5 implies that $\sigma_i = \sigma \circ \psi \circ \eta_i$ is a consistent mapping for T_i. Since $E(N_i, \tilde{x}) \triangleright T_i$, it follows from the induction hypothesis that $\lfloor\tilde{x}\rfloor_1\colon \sigma_i(\lfloor\chi_{T_i}\rfloor_2), \ldots, \lfloor\tilde{x}\rfloor_n\colon \sigma_i(\lfloor\chi_{T_i}\rfloor_{n+1}) \vdash N_i\colon \sigma_i(\lfloor\chi_{T_i}\rfloor_1)$.
Because of the definition of the σ_i it holds that $\sigma_1(\lfloor\chi_{T_1}\rfloor_j) = \sigma_2(\lfloor\chi_{T_2}\rfloor_j) = \sigma(\lfloor\chi_T\rfloor_j)$ for $j \in \{2, \ldots, n+1\}$. Furthermore it holds that

$$\begin{aligned}\sigma_1(\lfloor\chi_{T_1}\rfloor_1) &= \sigma(\psi(\eta_1(\lfloor\chi_{T_i}\rfloor_1))) = \sigma(\psi(v_1))\\ &= \sigma(\psi(v_2)) \to \sigma(\lfloor\chi_T\rfloor_1)\\ &= \sigma(\psi(\eta_2(\lfloor\chi_{T_2}\rfloor_1))) \to \sigma(\lfloor\chi_T\rfloor_1)\\ &= \sigma_2(\lfloor\chi_{T_2}\rfloor_1) \to \sigma(\lfloor\chi_T\rfloor_1),\end{aligned}$$

since there is an edge $\psi(e)$ labelled $\to$ with $c_T(\psi(e)) = \psi(v_1)\,\psi(v_2)\lfloor\chi_T\rfloor_1$.
Then the typing rule for application gives us

$$\lfloor\tilde{x}\rfloor_1\colon \sigma(\lfloor\chi_T\rfloor_2), \ldots, \lfloor\tilde{x}\rfloor_n\colon \sigma(\lfloor\chi_T\rfloor_{n+1}) \vdash N_1N_2\colon \sigma(\lfloor\chi_T\rfloor_1).$$

We now show the second part of the proposition, again by structural induction on M. Again, we show only one of the cases.

application: Let $M = N_1N_2$ with $\Gamma \vdash M\colon \tau$. From the typing rules we can infer that $\Gamma \vdash N_1\colon \tau' \to \tau$ and $\Gamma \vdash N_1\colon \tau'$. Since $Set(\tilde{x})$ is contained in the variables of Γ, we can infer from the induction hypothesis that there exist graph types T_i, $i \in \{1, 2\}$ such that $E(N_i, \tilde{x}) \triangleright T_i$. Furthermore there are consistent mappings σ_i for the T_i such that $\sigma_1(\lfloor\chi_{T_1}\rfloor_1) = \tau' \to \tau$, $\sigma_2(\lfloor\chi_{T_2}\rfloor_1) = \tau'$ and $\sigma_i(\lfloor\chi_{T_i}\rfloor_{j+1}) = \Gamma(\lfloor\tilde{x}\rfloor_j)$ for $j \in \{1, \ldots, n\}$.
From Condition (4), the definition of the encoding $E(N_1N_2, \tilde{x})$ (see Fig. 19) and the definition of the linear mapping (see Fig. 21), we can then infer that $E(N_1N_2, \tilde{x}) \triangleright T$, whenever the graph T, depicted in Fig. 23, is defined.
We denote the graph inside the closure operator ρ by T_0. Clearly, there are injective morphisms $\eta_i\colon T_i \to T_0$. We define a mapping σ_0 for T_0 as follows: We set $\sigma_0(\eta_i(v_i)) = \sigma_i(v_i)$ for every node v_i of T_i. Furthermore we set $\sigma_0(\lfloor\chi_{T_0}\rfloor_1) = \tau$. Because of the properties of the σ_i, the mapping σ_0 is well-defined and consistent.

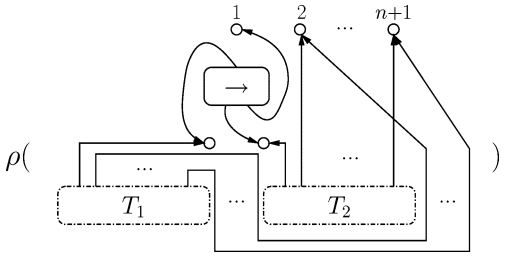

Fig. 23 The graph type T of $E(N_1 N_{,2}\, \tilde{x})$

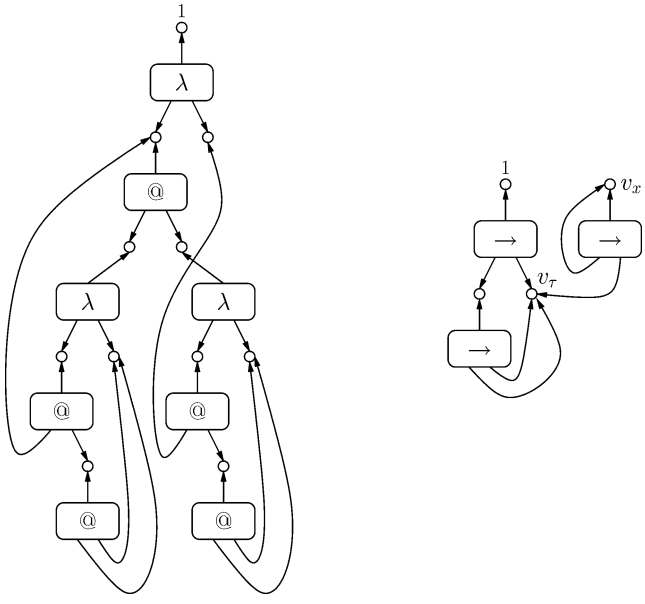

Fig. 24 The λ-graph for the paradoxical combinator Y and its graph type

Lemma 6 implies that $T = \rho(T_0)$ is defined and there is a consistent mapping σ for T such that $\sigma(\lfloor \chi_T \rfloor_i) = \sigma_0(\lfloor \chi_{T_0} \rfloor_i)$ for $i \in \{1, \ldots, n+1\}$. Hence $\sigma(\lfloor \chi_T \rfloor_1) = \sigma_0(\lfloor \chi_{T_0} \rfloor_1) = \tau$ and $\sigma(\lfloor \chi_T \rfloor_{i+1}) = \sigma_0(\lfloor \chi_{T_0} \rfloor_{i+1}) = \sigma_0(\eta_1(\lfloor \chi_{T_1} \rfloor_{i+1})) = \sigma_1(\lfloor \chi_{T_1} \rfloor_{i+1}) = \Gamma(\lfloor \tilde{x} \rfloor_i)$ and that finishes the proof of this case. □

The graph type system presented here has some advantages over the standard type system for the λ-calculus: No type environments are required and type inference, including unification, is already included in the typing procedure.

Example 12 As another example, this time involving recursive types, we regard the paradoxical combinator $Y = \lambda h.((\lambda x.h(xx))(\lambda y.h(yy)))$, which replaces the fixed-point operator and which can be used to "emulate" recursive function calls. It can be typed $\vdash Y : (\tau \to \tau) \to \tau$, but only using recursive types. The λ-graph for Y and its graph type are depicted in Fig. 24. Note that node v_x in the graph type represents the type of x and $\lambda h.h(xx)$, which must be typed by $x : \mu \mathsf{X}.(\mathsf{X} \to \tau) \vdash x : \mu \mathsf{X}.(\mathsf{X} \to \tau)$ respectively $h : \tau \to \tau \vdash \lambda x.h(xx) : \mu \mathsf{X}.(\mathsf{X} \to \tau)$.

Variants: It should not be difficult to show that a λ-expression can be typed without using recursive types if and only if the corresponding λ-graph has a cycle-free graph type.

5 Conclusion

We consider this work to be a first step towards a more general and encompassing theory of types for graphs and dynamically evolving systems, even if there are probably many type systems that do not fit well into this framework.

The need for such type systems manifests itself in work on analysis of graph-like dynamic structures such as work on types for process calculi already mentioned in the introduction, HD-automata [38] for the verification of π-calculus processes and shape analysis [45]. The idea of composing graphs in such a way that they satisfy a certain property was already presented by Lafont in [33] where it is used to obtain deadlock-free nets. Another related work is [14] where shape types are specified using context-free graph grammars and an algorithm for invariance checking is given.

An interesting direction of future work will certainly be the theoretically sound integration of different techniques including type systems in order to obtain powerful and practical analysis techniques for concurrent and distributed systems. In this context we are also interested in a combination of type systems with other techniques for the verification of graph transformation systems [1, 42].

Other avenues for research are suggested by the contents of this paper: First, it seems to be reasonable to search for a technique allowing the automatic derivation of type systems from given graph rewriting rules. Probably some basic requirements such as the lattice annotation and the closure operator ρ have to be fixed, but then it seems feasible to obtain the linear mapping by checking type invariance for the rules and using some sort of fixed-point iteration until a suitable invariant is found.

Second, it is tempting to replace lattices with monoids and thus obtain a type system able to count, i.e., one might be able to derive upper bounds for the number of edges with a certain label present at any given time. However, using monoids instead of lattices introduces many technical difficulties which are not easy to solve. It should be worthwile to try and overcome those difficulties by adapting type systems for the π-calculus working with monoids [28, 32].

Acknowledgements I would like to thank the anonymous referees for their valuable and helpful comments.

References

1. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In Proc. of CONCUR '01, pp. 381–395. Springer-Verlag, LNCS 2154 (2001)
2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In Proc. of POPL '02, pp. 1–3. ACM (2002)
3. Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, R., Plasmeijer, M.J., Sleep, M.R.: Term graph rewriting. In Proc. of PARLE '87, Volume 2, pp. 141–158. Springer, LNCS 259 (1987)
4. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall (1990)

5. Bauderon, M., Courcelle, B.: Graph expressions and graph rewritings. Mathematical Systems Theory, **20**, 83–127 (1987)
6. Cardelli, L., Gordon, A.D.: Types for mobile ambients. In Proceedings of the 26th ACM Symposium on Principles of Programming Languages, pp. 79–92. ACM (1999)
7. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In Proc. of ICSE '03 (25th International Conference on Software Engineering), pp. 385–395. IEEE Computer Society (2003)
8. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26**(3/4), 241–265 (1996)
9. Cousot, P.: Abstract interpretation. ACM Computing Surveys **28**(2) (1996)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of POPL '77 (Los Angeles, California), pp. 238–252. ACM (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In Proc. of POPL '79 (San Antonio, Texas), pp. 269–282. ACM Press (1979)
12. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In Proc. 1st International Workshop on Graph Grammars, pp. 1–69. Springer-Verlag, LNCS 73 (1979)
13. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In Proc. of FOSSACS '04, pp. 151–166. Springer, LNCS 2987 (2004)
14. Fradet, P., Le Métayer, D.: Shape types. In Proc. of POPL '97, pp. 27–39. ACM (1997)
15. Gadducci, F., Heckel, R.: An inductive view of graph transformation. In Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT '97, pp. 223–237. Springer-Verlag, LNCS 1376 (1997)
16. Gadducci, F., Montanari, U.: Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. Theoretical Computer Science **285**(2), 319–358 (2002)
17. Gardner, P.: Closed action calculi. Theoretical Computer Science **228**(1–2), 77–103 (1999)
18. Habel, A.: Hyperedge Replacement: Grammars and Languages. Springer-Verlag, LNCS 643 (1992)
19. Hatcliff, J., Dwyer, M.: Using the Bandera tool set to model-check properties of concurrent Java software. In Proc. of CONCUR 2001, pp. 39–58. Springer, LNCS 2154 (2001)
20. Honda, K.: Composing processes. In Proc. of POPL'96, pp. 344–357. ACM (1996)
21. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. In Proc. of POPL '01, pp. 128–141. ACM (2001)
22. Jeffrey, A.: A fully abstract semantics for concurrent graph reduction. In Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, pp. 128–131 (1994)
23. Jensen, O.H., Milner, R.: Bigraphs and transitions. In Proc. of POPL 2003, pp. 38–49. ACM (2003)
24. König, B.: Generating type systems for process graphs. In Proc. of CONCUR '99, pp. 352–367. Springer-Verlag, LNCS 1664 (1999)
25. König, B.: A general framework for types in graph rewriting. In Proc. of FST TCS '00, pp. 373–384. Springer-Verlag, LNCS 1974 (2000)
26. König, B.: A general framework for types in graph rewriting. Technical Report TUM-I0014, Technische Universität München (2000)
27. König, B.: A graph rewriting semantics for the polyadic π-calculus. In Workshop on Graph Transformation and Visual Modeling Techniques (Geneva, Switzerland), ICALP Workshops '00, pp. 451–458. Carleton Scientific (2000)
28. König, B.: Analysing input/output-capabilities of mobile processes with a generic type system. In Proc. of ICALP '00, pp. 403–414. Springer-Verlag, LNCS 1853 (2000)
29. König, B.: Hypergraph construction and its application to the compositional modelling of concurrency. In GRATRA '00: Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Proceedings available as Report Nr. 2000-2 (Technische Universität Berlin) (2000)
30. König, B.: Hypergraph construction and its application to the static analysis of concurrent systems. Mathematical Structures in Computer Science **12**(2), 149–175 (2002)
31. König, B.: Analysing input/output-capabilities of mobile processes with a generic type system. Journal of Logic and Algebraic Programming **63**(1), 35–58 (2005)
32. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems **21**(5), 914–947 (1999)
33. Lafont, Y.: Interaction nets. In Proc. of POPL '90, pp. 95–108. ACM Press (1990)

34. Leifer, J.J.: Operational congruences for reactive systems. PhD thesis, University of Cambridge Computer Laboratory, September (2001)
35. Mac Lane, S.: Categories for the Working Mathematician. Springer-Verlag (1971)
36. Milner, R.: The polyadic π-calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, Laboratory for Foundations of Computer Science (1991)
37. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
38. Montanari, U., Pistore, M.: π-calculus, structured coalgebras, and minimal hd-automata. In Proc. of MFCS '00, pp. 569–578. Springer-Verlag, LNCS 1893 (2000)
39. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
40. Pierce, B., Sangiorgi, D.: Typing and subtyping for mobile processes. In Proc. of LICS '93, pp. 376–385 (1993)
41. Pierce, B., Sangiorgi, D.: Typing and subtyping for mobile processes. Journal of Mathematical Structures in Computer Science **6**(5), 409–454 (1996)
42. Rensink, A.: Model checking graph grammars. In Proc. of AVOCS '03 (Workshop on Automated Verification of Critical Systems) (2003)
43. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM **12**(1), 23–41 (1965)
44. Rozenberg, G., editor: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations, volume 1. World Scientific (1997)
45. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS (ACM Transactions on Programming Languages and Systems) **24**(3), 217–298 (2002)
46. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In Proc. of ASE '00 (International Conference on Automated Software Engineering), pp. 3–12. IEEE (2000)
47. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. PhD thesis, Oxford University, September (1971)
48. Yoshida, N.: Graph notation for concurrent combinators. In Proc. of TPPP '94. Springer-Verlag, LNCS 907 (1994)