A General Framework for Types in Graph Rewriting*

Barbara König (koenigb@in.tum.de)

Fakultät für Informatik, Technische Universität München

Abstract. A general framework for typing graph rewriting systems is presented: the idea is to statically derive a type graph from a given graph. In contrast to the original graph, the type graph is invariant under reduction, but still contains meaningful behaviour information. We present conditions, a type system for graph rewriting should satisfy, and a methodology for proving these conditions. In two case studies it is shown how to incorporate existing type systems (for the polyadic π-calculus and for a concurrent object-oriented calculus) into the general framework.

1 Introduction

In the past, many formalisms for the specification of concurrent and distributed systems have emerged. Some of them are aimed at providing an encompassing theory: a very general framework in which to describe and reason about interconnected processes. Examples are action calculi [18], rewriting logic [16] and graph rewriting [3] (for a comparison see [4]). They all contain a method of building terms (or graphs) from basic elements and a method of deriving reduction rules describing the dynamic behaviour of these terms in an operational way.

A general theory is useful, if concepts appearing in instances of a theory can be generalised, yielding guidelines and relieving us of the burden to prove universal concepts for every single special case. An example for such a generalisation is the work presented for action calculi in [15] where a method for deriving a labelled transition semantics from a set of reaction rules is presented. We concentrate on graph rewriting (more specifically hypergraph rewriting) and attempt to generalise the concept of type systems, where, in this context, a type may be a rather complex structure.

Compared to action calculi[1] and rewriting logic, graph rewriting differs in a significant way in that connections between components are described explicitly (by connecting them by edges) rather than implicitly (by referring to the same channel name). We claim that this feature—together with the fact that it is easy to add an additional layer containing annotations and constraints to a graph—can simplify the design of a type system and therefore the static analysis of a graph rewriting system.

* Research supported by SFB 342 (subproject A3) of the DFG.

[1] Here we mean action calculi in their standard string notation. There is also a graph notation for action calculi, see e.g. [7].

After introducing our model of graph rewriting and a method for annotating graphs, we will present a general framework for type systems where both—the expression to be typed and the type itself—are hypergraphs and will show how to reduce the proof obligations for instantiations of the framework. We are interested in the following properties: correctness of a type system (if an expression has a certain type, then we can conclude that this expression has certain properties), the subject reduction property (types are invariant under reduction) and compositionality (the type of an expression can always be derived from the types of its subexpressions). Parts of the proofs of these properties can already be conducted for the general case.

We will then show that our framework is realistic by instantiating it to two well-known type systems: a type system avoiding run-time errors in the polyadic π-calculus [17] and a type system avoiding "message not understood"-errors in a concurrent object-oriented setting. A third example enforcing a security policy for untrustworthy applets is included in the full version [11].

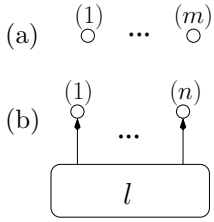
2 Hypergraph Rewriting and Hypergraph Annotation

We first define some basic notions concerning hypergraphs (see also [6]) and a method for inductively constructing hypergraphs.

Definition 1. (Hypergraph) *Let L be a fixed set of labels. A hypergraph $H = (V_H, E_H, s_H, l_H, \chi_H)$ consists of a set of nodes V_H, a set of edges E_H, a connection mapping $s_H : E_H \to V_H^*$, an edge labelling $l_H : E_H \to L$ and a string $\chi_H \in V_H^*$ of external nodes. A hypergraph morphism $\phi : H \to H'$ (consisting of $\phi_V : V_H \to V_{H'}$ and $\phi_E : E_H \to E_{H'}$) satisfies*[2] *$\phi_V(s_H(e)) = s_{H'}(\phi_E(e))$ and $l_H(e) = l_{H'}(\phi_E(e))$. A strong morphism (denoted by the arrow $\twoheadrightarrow$) additionally preserves the external nodes, i.e. $\phi_V(\chi_H) = \chi_{H'}$. 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 $ar(e) = |s_H(e)|$. 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 Figure (a) to the right, external nodes are labelled (1), (2), ... in their respective order).
The hypergraph $H = [l]_n$ contains exactly one edge e with label l where $s_H(e) = \chi_H$, $ar(e) = n$ and[3] $V_H = Set(\chi_H)$ (see (b), nodes are ordered from left to right).



The next step is to define a method (first introduced in [10]) for the annotation of hypergraphs with lattice elements and to describe how these annotations

[2] The application of ϕ_V to a string of nodes is defined pointwise.
[3] $Set(\tilde{s})$ is the set of all elements of a string $\tilde{s}$

change under morphisms. We use annotated hypergraphs as types where the annotations can be considered as extra typing information, therefore we use the terms *annotated hypergraph* and *type graph* as synonyms.

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

$$\mathcal{A}_\phi \circ \mathcal{A}_\psi = \mathcal{A}_{\phi\circ\psi} \quad \mathcal{A}_{id_H} = id_{\mathcal{A}(H)} \quad \mathcal{A}_\phi(a \vee b) = \mathcal{A}_\phi(a) \vee \mathcal{A}_\phi(b) \quad \mathcal{A}_\phi(\bot) = \bot$$

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

If $a \in \mathcal{A}(H)$, then $H[a]$ is called an annotated hypergraph. And $\phi : H[a] \to_{\mathcal{A}} H'[a']$ is called an $\mathcal{A}$-morphism if $\phi : H \to H'$ is a hypergraph morphism and $\mathcal{A}_\phi(a) \leq a'$. Furthermore $H[a]$ and $H'[a']$ are called isomorphic if there is a strong bijective $\mathcal{A}$-morphism ϕ with $\mathcal{A}_\phi(a) = a'$ between them.

Example: We consider the following annotation mapping $\mathcal{A}$: let $(\{false, true\}, \leq)$ be the boolean lattice where $false < true$. We define $\mathcal{A}(H)$ to be the set of all mappings from V_H into $\{false, true\}$ (which yields a lattice with pointwise order). By choosing an element of $\mathcal{A}(H)$ we fix a subset of the nodes. So let $a : V_H \to \{false, true\}$ be an element of $\mathcal{A}(H)$ and let $\phi : H \to H'$, $v' \in V_H$. We define: $\mathcal{A}_\phi(a) = a'$ where $a'(v') = \bigvee_{\phi(v)=v'} a(v)$. That is, if a node v with annotation *true* is mapped to a node v' by ϕ, the annotation of v' will also be *true*.

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).

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 $\zeta_i : \mathbf{m_i} \to D$, $1 \leq i \leq n$ be hypergraph morphisms where $ar(H_i) = m_i$ and D is discrete. Furthermore let $\phi_i : \mathbf{m_i} \twoheadrightarrow H_i$ be the unique strong morphisms.*

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 $\zeta_i(v) \approx \phi_i(v)$ if $1 \leq i \leq n$, $v \in V_{\mathbf{m_i}}$. The nodes of the constructed graph are the equivalence classes of $\approx$. We define

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

where $s_H(e) = [v_1]_\approx \ldots [v_k]_\approx$ if $e \in E_{H_i}$ and $s_{H_i}(e) = v_1 \ldots v_k$. Furthermore $l_H(e) = l_{H_i}(e)$ if $e \in E_{H_i}$. And we define $\chi_H = [v_1]_\approx \ldots [v_k]_\approx$ if $\chi_D = v_1 \ldots v_k$.

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

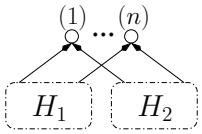
We construct embeddings $\phi : D \twoheadrightarrow H$ and $\eta_i : H_i \to H$ by mapping every node to its equivalence class and every edge to itself. Then the construction of

annotated graphs can be defined as follows:

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

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 $\mathbf{m_i}$, χ_D becomes the new sequence of external nodes. Lattice annotations are joined if the annotated nodes are merged. In terms of category theory, $\textcircled{D}^{n}_{i=1}(H_i[a_i], \zeta_i)$ 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 $\bot$). We do not mention this fact in the rest of the paper, but it is used extensively in the proofs (for the proofs and several examples see the full version [11]).

We also use another, more intuitive notation for graph construction. Let $\zeta_i : \mathbf{m_i} \to D$, $1 \leq i \leq n$. Then we depict $\textcircled{D}^{n}_{i=1}(H_i, \zeta_i)$ by drawing the hypergraph $(V_D, \{e_1, \ldots, e_n\}, s_H, l_H, \chi_D)$ where $s_H(e_i) = \zeta_i(\chi_{\mathbf{m_i}})$ and $l_H(e_i) = H_i$.



Example: we can draw $\textcircled{\mathbf{n}}^{2}_{i=1}(H_i, \zeta_i)$ where $\zeta_1, \zeta_2 : \mathbf{n} \twoheadrightarrow \mathbf{n}$ as in the picture above (note that the edges have dashed lines). Here we fuse the external nodes of H_1 and H_2 in their respective order and denote the resulting graph by $H_1 \square H_2$. If there is an edge with a dashed line labelled with an edge $[l]_n$ we rather draw it with a solid line and label it with l (see e.g. the second figure in section 4.1).

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.*

In our approach we generate the same transition system as in the double-pushout approach to graph rewriting described in [2] (for details see [13]).

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 $[l]_n$ to annotated hypergraphs. Then t can be extended to arbitrary hypergraphs by defining $t(\textcircled{D}^{n}_{i=1}([l_i]_{n_i}, \zeta_i)) = \textcircled{D}^{n}_{i=1}(t([l_i]_{n_i}), \zeta_i)$ and is then called a linear mapping.

3 Static Analysis and Type Systems for Graph Rewriting

Having introduced all underlying notions we now specify the requirements for type systems. We assume that there is a fixed set $\mathcal{R}$ of rewrite rules, an annotation mapping $\mathcal{A}$, a predicate X on hypergraphs (representing the property we

want to check) and a relation $\triangleright$ with the following meaning: if $H \triangleright T$ where H is a hypergraph and T a type graph (annotated wrt. to $\mathcal{A}$), then 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, then we can be sure that the property X holds.

$$H \triangleright T \;\Rightarrow\; X(H) \quad \textbf{(correctness)} \tag{1}$$

During reduction, the type stays invariant.

$$H \triangleright T \wedge H \to_{\mathcal{R}} H' \;\Rightarrow\; H' \triangleright T \quad \textbf{(subject reduction property)} \tag{2}$$

From (1) and (2) we can conclude that $H \triangleright T$ 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 type graphs. It should always be possible to weaken the type with respect to that preorder.

$$H \triangleright T \wedge T \twoheadrightarrow_{\mathcal{A}} T' \;\Rightarrow\; H \triangleright T' \quad \textbf{(weakening)} \tag{3}$$

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. We can not sensibly demand that the type of an expression is obtained by combining the types of the subgraphs in exactly the same way the expression is constructed, so we introduce a partial arity-preserving mapping f doing some post-processing.

$$\begin{aligned} &\forall\, i\colon H_i \triangleright T_i \;\Rightarrow\; \textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright f(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \quad \textbf{(compositionality)} \\ &\textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright T \;\Rightarrow\; \exists\, T_i\colon (H_i \triangleright T_i \text{ and } f(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \twoheadrightarrow_{\mathcal{A}} T) \end{aligned} \tag{4}$$

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.

$$H \text{ typable} \;\Rightarrow\; \exists\, T\colon (H \triangleright T \wedge (H \triangleright T' \iff T \twoheadrightarrow_{\mathcal{A}} T')) \quad \textbf{(minimal types)} \tag{5}$$

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

The meaning of the mappings t and f 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 f on the other hand, is a *global* operation, merging or removing parts of a graph in order to anticipate future reductions and thus ensure the subject reduction property. In the example in section 4.1 f "folds" a graph into itself, hence the letter f. In order to obtain

compositionality, it is required that f can be applied arbitrarily often at any stage of type inference, without losing information (see condition (6) of Theorem 1).

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 f be an arity-preserving mapping as above, let t be a linear mapping, let X be a predicate on hypergraphs and let $H \triangleright T$ if and only if $f(t(H)) \twoheadrightarrow_{\mathcal{A}} T$. Let us further assume that f satisfies*[4]

$$f(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \cong f(\textcircled{D}_{i=1}^{n}(f(T_i), \zeta_i)) \quad (6) \qquad T \twoheadrightarrow_{\mathcal{A}} T' \Rightarrow f(T) \twoheadrightarrow_{\mathcal{A}} f(T') \quad (7)$$

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

$$f(t(H)) \text{ defined} \Rightarrow X(H) \quad (8) \qquad (L, R) \in \mathcal{R} \Rightarrow f(t(R)) \twoheadrightarrow_{\mathcal{A}} f(t(L)) \quad (9)$$

The operation f can often be characterised by a universal property with the intuitive notion that $f(T)$ is the "smallest" type graph (wrt. the preorder $\twoheadrightarrow_{\mathcal{A}}$) for which $T \twoheadrightarrow_{\mathcal{A}} f(T)$ and a property C hold.

Proposition 1. *Let C be a property on type graphs such that $f(T)$ can be characterised in the following way: $f(T)$ satisfies C, there is a morphism $\phi : T \twoheadrightarrow_{\mathcal{A}} f(T)$ and for every other morphism $\phi' : T \to_{\mathcal{A}} T'$ where $C(T')$ holds, there is a unique morphism $\psi : f(T) \to_{\mathcal{A}} T'$ such that $\psi \circ \phi = \phi'$. Furthermore we demand that if there exists a morphism $\phi : T \to_{\mathcal{A}} T'$ such that $C(T')$ holds, then $f(T)$ is defined.*

Then if $f(T)$ is defined, it is unique up to isomorphism. Furthermore f satisfies conditions (6) and (7).

4 Case Studies

4.1 A Type System for the Polyadic π-Calculus

We present a graph rewriting semantics for the asynchronous polyadic π-calculus [17] without choice and matching, already introduced in [12]. Different ways of encoding the π-calculus into graph rewriting can be found in [21, 5, 4].

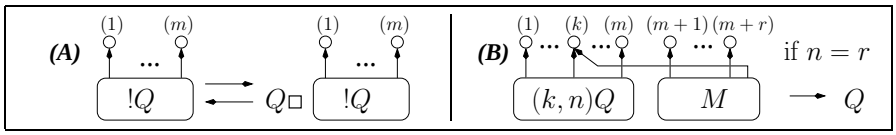
We apply the theory presented in section 3, introduce a type system avoiding runtime errors produced by mismatching arities and show that it satisfies the conditions of Theorem 1. Afterwards we show that a graph has a type if and only if the corresponding π-calculus process has a type in a standard type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

Definition 6. (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 with $(k, n)Q$ where Q is again a process*

[4] In an equation $T \cong T'$ 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.

graph, $1 \leq n \leq ar(Q)$ *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 which can replicate itself) or with the constant M (e is a message sent to its last node).*

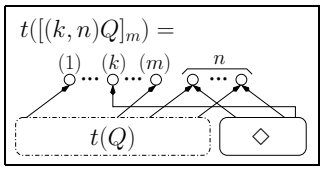
The reduction relation is generated by the rules in (A) (replication) and by rule (B) (reception of a message by a process) and is closed under isomorphism and graph construction.



A process graph may contain a bad redex, if it contains a subgraph corresponding to the left-hand side of rule (B) with $n \neq r$, so 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 f. (Note that in this case, the type graphs are trivially annotated by $\bot$, and so we omit the annotation mapping.)

The linear t mapping is defined on the hyperedges as follows: $t([M]_n) = [\diamond]_n$ ($\diamond$ is a new edge label), $t([!Q]_m) = t(Q)$ and $t([(k,n)Q]_m)$ is defined as in the image to the right (in the notation explained after Definition 3). It is only defined if $n + m = ar(Q)$.



The mapping f is defined as in Proposition 1 where C is defined as follows[5]

$$C(T) \iff \forall\, e_1, e_2 \in E_T\colon (\lfloor s_T(e_1) \rfloor_{ar(e_1)} = \lfloor s_T(e_2) \rfloor_{ar(e_2)} \;\Rightarrow\; e_1 = e_2)$$

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 f 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 $f(t(H))$.

Proposition 2. *The trivial annotation mapping* $\mathcal{A}$ *(where every lattice consists of a single element* $\bot$*), the mappings f and t and the predicate X defined above satisfy conditions (6)–(9) of Theorem 1. Thus if* $P \triangleright T$*, then P will never produce a bad redex during reduction.*

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.

Definition 7. (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:*

[5] $\lfloor s \rfloor_i$ extracts the i-th element of a string s.

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

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

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

An operational correspondence can be stated as follows:

Proposition 3. *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* $\Theta_{\tilde{t}}(P) \equiv \Theta_{\tilde{t}}(P')$ $\qquad$ $P \to^* P'$ *implies* $\Theta_{\tilde{t}}(P) \to^* \Theta_{\tilde{t}}(P)$
- $\Theta_{\tilde{t}}(P) \to^* p \neq$ *wrong implies that* $P \to^* Q$ *and* $\Theta_{\tilde{t}}(Q) \equiv p$ *for some process graph* Q.
- $\Theta_{\tilde{t}}(P) \to^*$ *wrong if and only if* $P \to^* P'$ *for some process graph* P' *containing a bad redex*

We now compare our type system with a standard type system of the π-calculus: a *type tree* is 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 $\Gamma(x_i) = t_i$. The rules of the type system are simplified versions of the ones from [19], obtained by removing the subtyping annotations.

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

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

We 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 type graph into type trees.

Proposition 4. *Let* T *be a type graph 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* $e \in E_T$: $s_T(e) = v_1 \ldots v_n v \;\Rightarrow\; \sigma(v) = [\sigma(v_1),\ldots,\sigma(v_n)]$. *Every type graph of the form* $f(t(P))$ *has such a consistent mapping.*

Let $P \triangleright T$ *with* $n = ar(T)$ *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 type graph T such that $P \triangleright T$ and a consistent mapping σ such that for every $1 \leq i \leq |\tilde{t}|$ it holds that $\sigma(\lfloor \chi_T \rfloor_i) = \Gamma(\lfloor \tilde{t} \rfloor_i)$.

4.2 Concurrent Object-Oriented Programming

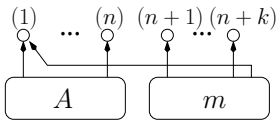
We now show how to model a concurrent object-oriented system by graph rewriting and then present a type system. In our model, several objects may compete in order to receive a message, and several messages might be waiting at the same object. Typically, type systems in object-oriented programming are there to ensure that an object that receives a message is able to process it.

Definition 8. (Concurrent object-oriented rewrite system) *Let $(\mathcal{C}, <:)$ be a lattice of classes with a top class*[6] *$\top$ and a bottom class $\bot$. We denote classes by the letters $A, B, C, \ldots$. Furthermore let $\mathcal{M}$ be a set of method names. The function $ar : \mathcal{C} \cup \mathcal{M} \to \mathbb{N}\backslash\{0\}$ assigns an arity to every class or method name.*

An object graph G is a hypergraph with a duplicate-free string of external nodes, labelled with elements of $\mathcal{C}\backslash\{\bot\} \cup \mathcal{M}$ where for every edge e it holds that $ar(e) = ar(l_G(e))$. A concurrent object-oriented rewrite system (specifying the semantics) consists of a set of rules $\mathcal{R}$ satisfying the following conditions:

- *the left-hand side of a rule always has the form shown in Figure (C) below (where $A \in \mathcal{C}\backslash\{\bot\}$, $ar(A) = n$, $m \in \mathcal{M}$, $ar(m) = k + 1$).*

  The right-hand side is again an object graph of arity $n + k$. If a left-hand side $R_{A,m}$ exists, we say that A understands m.

  (C) $= R_{A,m}$

- *If $A <: B$, $A \neq \bot$ and B understands m, then A also understands m.*
- *For all $m \in \mathcal{M}$, either $\{A \mid A \text{ understands } m\}$ is empty or it contains a greatest element.*

An object graph G contains a "message not understood"-error if G contains a subgraph $R_{A,m}$, but A does not understand m.

Thus the predicate X for this section is defined as follows: $X(G)$ if and only if G does not contain a "message not understood"-error.

In contrast to the previous section, we now use annotated type graphs: the annotation mapping $\mathcal{A}$ assigns a lattice $(\{a : V_H \to \mathcal{C} \times \mathcal{C}\}, \leq))$ to every hypergraph H. The partial order is defined as follows: $a_1 \leq a_2 \iff \forall v\colon (a_1(v) = (A_1, B_2) \wedge a_2(v) = (A_2, B_2) \Rightarrow A_1 <: A_2 \wedge B_1 :> B_2)$, i.e. we have covariance in the first and contravariance in the second position. If a node v is labelled (A, B), this has the following intuitive meaning: we can accept at least as many messages as an object of class A on this node *and* we can send at most as many messages as an object of class B can accept.

[6] This corresponds to the class Object in Java

Furthermore we define $\mathcal{A}_\phi(a)(v') = \bigvee_{\phi(v)=v'} a(v)$ where $\phi : H \to H'$, a is an element of $\mathcal{A}(H)$ and $v' \in V_{H'}$.

We now define the operator f: let $T[a]$ be a type graph of arity n where it holds for all nodes v that $a(v) = (A, B)$ implies $A <: B$ (otherwise f is undefined). Then f reduces the graph to its string of external nodes, i.e $f(T[a]) = \mathbf{n}[b]$ where $b(\lfloor \chi_{\mathbf{n}} \rfloor_i) = a(\lfloor \chi_T \rfloor_i)$.

The linear mapping t determines the type of a class or method. It is necessary to choose a linear mapping that preserves the interface of left-hand and right-hand sides, i.e. we can use any t that satisfies condition (9) and the following two conditions below for $A \in \mathcal{C}\backslash\{\bot\}$ and $m \in \mathcal{M}$:

$$t([A]_n) = [A]_n[a] \text{ where } a(\lfloor \chi_{[A]_n} \rfloor_1) \geq (A, \top)$$
$$t([m]_n) = [m]_n[a] \text{ where } a(\lfloor \chi_{[m]_n} \rfloor_n) \geq (\bot, \max\{B \mid B \text{ understands } m\})$$

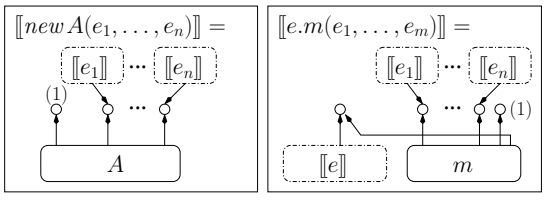
Proposition 5. *The annotation mapping $\mathcal{A}$, the mappings f and t and the predicate X defined above satisfy conditions (6)–(9) of Theorem 1. Thus if $G \triangleright T$, then G will never produce a "message not understood"-error during reduction.*

In this case we do not prove that this type systems corresponds to an object-oriented type system, but rather present a semi-formal argument: we give the syntax and a type system for a small object calculus, and furthermore an encoding into hypergraphs, without really defining the semantics. For the formal semantics of object calculi see [20, 9], among others.

An expression e in the object calculus either has the form $new\ A(e_1, \ldots, e_n)$ where $A \in \mathcal{C}\backslash\{\bot\}$ and $ar(A) = n + 1$ or $e.m(e_1, \ldots, e_n)$ where $m \in \mathcal{M}$ and $ar(m) = n + 2$. The e_i are again expressions. Every class A is assigned an $(ar(A) - 1)$-tuple of classes defining the type of the fields of A ($A : (A_1, \ldots, A_n)$) and every method m with $ar(m) = n + 2$ defined in class B is assigned a type $B.m : C_1, \ldots, C_n \to C$. If a method is overwritten in a subclass it is required to have the same type. A simple type systems looks as follows:

$$\frac{e : A,\ A <: B}{e : B} \qquad \frac{A : (A_1, \ldots, A_n),\ e_i : A_i}{new\ A(e_1, \ldots, e_n) : A} \qquad \frac{e : B,\ B.m : C_1, \ldots, C_n \to C,\ e_i : C_i}{e.m(e_1, \ldots, e_n) : C}$$

Now an encoding $[\![\cdot]\!]$ can be defined as shown in the figure to the right. We introduce the convention that the penultimate node of a message can be used to access the result after the rewriting step.



If $A : (A_1, \ldots, A_n)$ we define t in such a way that the $n + 1$ external nodes of $t([A]_{n+1})$ are annotated by $(A, \top)$, $(\bot, A_1)$, …, $(\bot, A_n)$. And if $B.m : C_1, \ldots, C_n \to C$ (where B is the maximal class which understands method m), we annotate the external nodes of $t([m]_{n+2})$ by $(\bot, C_1)$, …, $(\bot, C_n)$, $(C, \top)$, $(\bot, B)$. Now we can show by induction on the typing rules that if $e : A$, then there exists a type graph $T[a]$ such that $[\![e]\!] \triangleright T[a]$ and $a(\lfloor \chi_T \rfloor_1) = (A, \top)$.

5 Conclusion and Comparison to Related Work

This is a first tentative approach aimed at developing a general framework for the static analysis of graph rewriting in the context of type systems. It is obvious that there are many type systems which do not fit well into our proposal. But since we are able to capture the essence of two important type systems, we assume to be on the right track.

Types are often used to make the connection of components and the flow of information through a system explicit (see e.g. the type system for the π-calculus, where the type trees indicate which tuple of channels is sent via which channel). Since connections are already explicit in graphs, we can use them both as type and as the expression to be typed. Via morphisms we can establish a clear connection between an expression and its type. Graphs are furthermore useful since we can easily add an extra layer of annotation.

Work that is very close in spirit to ours is [8] by Honda which also presents a general framework for type systems. The underlying model is closer to standard process algebras and the main focus is on the characterisation and classification of type systems.

The idea of composing graphs in such a way that they satisfy a certain property was already presented by Lafont in [14] where it is used to obtain deadlock-free nets.

In graph rewriting there already exists a concept of typed graphs [1], related to ours, but nevertheless different. In that work, a type graph is fixed a priori and there is only one type graph for every set of productions. Graphs are considered valid only if they can be mapped into the type graph by a graph morphism (this is similar to our proposal). In our case, we compute the type graphs a posteriori and it is a crucial point in the design of every type system to distinguish as many graphs as possible by assigning different type graphs to them.

This paper is a continuation of the work presented in [10] where the idea of generic type systems for process graphs (as defined in section 4.1) was introduced, but no proof of the equivalence of our type system to the standard type system for the π-calculus was given. The ideas presented there are now extended to general graph rewriting systems.

Further work will consist in better understanding the underlying mechanism of the type system. An interesting question in this context is the following: given a set of rewrite rules, is it possible to automatically derive mappings f and t satisfying the conditions of Theorem 1?

Acknowledgements: I would like to thank Reiko Heckel and Andrea Corradini for their comments on drafts of this paper, and Tobias Nipkow for his advice. I am also grateful to the anonymous referees for their valuable comments.

References

1. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.

2. H. Ehrig. Introduction to the algebraic theory of graphs. In *Proc. 1st International Workshop on Graph Grammars*, pages 1–69. Springer-Verlag, 1979. LNCS 73.
3. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.
4. F. Gadducci and U. Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoretical Computer Science*, 2000. to appear.
5. Philippa Gardner. Closed action calculi. *Theoretical Computer Science (in association with the conference on Mathematical Foundations in Programming Semantics)*, 1998.
6. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.
7. Masahito Hasegawa. *Models of Sharing Graphs (A Categorical Semantics of Let and Letrec)*. PhD thesis, University of Edingburgh, 1997. available in Springer Distinguished Dissertation Series.
8. Kohei Honda. Composing processes. In *Proc. of POPL'96*, pages 344–357. ACM, 1996.
9. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A core calculus for Java and GJ. In *Proc. of OOPSLA 1999*, 1999.
10. Barbara König. Generating type systems for process graphs. In *Proc. of CONCUR '99*, pages 352–367. Springer-Verlag, 1999. LNCS 1664.
11. Barbara König. A general framework for types in graph rewriting. Technical Report TUM-I0014, Technische Universität München, 2000.
12. Barbara König. A graph rewriting semantics for the polyadic pi-calculus. In *Workshop on Graph Transformation and Visual Modeling Techniques (Geneva, Switzerland), ICALP Workshops 2000*, pages 451–458. Carleton Scientific, 2000.
13. Barbara König. Hypergraph construction and its application to the compositional modelling of concurrency. In *GRATRA 2000: Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, 2000.
14. Yves Lafont. Interaction nets. In *Proc. of POPL '90*, pages 95–108. ACM Press, 1990.
15. James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In *Proc. of CONCUR 2000*, 2000. LNCS 1877.
16. José Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In *Concurrency Theory*, pages 331–372. Springer-Verlag, 1996. LNCS 1119.
17. Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, Heidelberg, 1993.
18. Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.
19. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of LICS '93*, pages 376–385, 1993.
20. David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.
21. Nobuko Yoshida. Graph notation for concurrent combinators. In *Proc. of TPPP '94*. Springer-Verlag, 1994. LNCS 907.