Generating Type Systems for Process Graphs

Barbara König

Fakultät für Informatik, Technische Universität München
koenigb@in.tum.de

Abstract. We introduce a hypergraph-based process calculus with a generic type system. That is, a type system checking an invariant property of processes can be generated by instantiating the original type system. We demonstrate the key ideas behind the type system, namely that there exists a hypergraph morphism from each process graph into its type, and show how it can be used for the analysis of processes. Our examples are input/output-capabilities, secrecy conditions and avoiding vicious circles occurring in deadlocks.
In order to specify the syntax and semantics of the process calculus and the type system, we introduce a method of hypergraph construction using concepts from category theory.

1 Introduction

In this work we propose a framework for the generation of type systems checking invariant properties of processes. We introduce a graph-based, asynchronous process calculus, similar to the polyadic π-calculus [13, 15], and give a generic type system for this calculus. Specialized type systems can then be generated by instantiating the original system.

Type systems are a valuable tool for the static analysis of parallel processes. Applications range from checking the use of channels [19, 11], confirming confluence [17], avoiding deadlocks [10] and ascertaining security properties [1, 4]. In all these cases, types are considered as partial descriptions of process behaviour, staying invariant during reduction. Furthermore a method for inferring properties of a process out of its behaviour description is required. Generally types are computable and in some type systems there is a most general or principal type for every typable process, from which all other types of the process can be derived. The flip side to type systems is the fact that some correct processes may not be typable.

Examining the type systems mentioned above, one can observe that they share similarities concerning the structure of types and typing rules. Our idea is to present a framework making a first step towards the integration of different type systems. The generic type system presented in this paper satisfies the subject reduction property, and we can guarantee absence of runtime errors for well-typed processes, the existence of principal types and of a type inference algorithm.

In order to type communicating processes, recursive types are essential. They can be represented in several ways: as expressions with a recursion operator μ (e.g. in [21]), as infinite trees [19] or as graphs [20, 23] (for different representations of recursive types for the λ-calculus see [22]). We chose graph representation for types as well as for processes. This enables us to establish a close correspondence between processes and types: there is a graph morphism from each process into its type. Thus, if a type graph satisfies a property which is closed under inverse graph morphisms (e.g. absence of circles, necessary for deadlock prevention), it is also valid for the process and—because of the subject reduction property—for all its successors.

Since in a general type system like ours the properties of a process which are to be analyzed are not fixed a priori, a close relationship between processes and their types is essential. Describing both processes and types by graphs seems a convenient method for allowing easy inference of process properties. This allows a systematic approach to obtaining correctness proofs for generated type systems. It is not clear to us how the same effect could be achieved by a string representation of processes.

There are several papers describing various ways of representing processes by graphs [14, 16, 9, 18]. Our method is closest to [9], but differs in several aspects, the most prominent being that we employ hierarchical hypergraphs.

Pure graph structure (or hypergraph structure in our case) is ordinarily not sufficient to capture relevant properties of a process. We therefore enrich our types by annotating them with lattice elements, e.g. describing input/output-capabilities of ports or channels. Every set of mappings assigning lattice elements to nodes or edges of a graph is a lattice itself, and thus it is sufficient to assign only one lattice element to every graph. It is necessary to define, how these lattice elements behave under morphisms. This is described by a functor mapping graph morphisms to join-morphisms in lattices. It is not the only case where we make use of category theory. It also allows us to give an elegant definition of graph construction (related to [6]) in terms of co-limits.

The annotation of graphs with lattice elements is another argument in favour of the use of graphs. It is more convenient to add additional labels or structures to a type represented as a graph than to a type represented by a term. This point will become clearer in section 5 where we will assign lattice elements to pairs of nodes.

2 Categorical Hypergraph Construction

We work with a variant of graphs: so-called hypergraphs [7, 2], where each edge has several (ordered) source nodes. There are two kinds of labels: edge sorts and and edge labels.

Definition 1. (Hypergraph) *Let Z be a fixed set of edge sorts and let L be a fixed set of labels. A* simple hypergraph *G is a tuple $G = (V, E, s, z, l)$ where V is a set of* nodes*, E is a set of* edges *disjoint from V, $s\colon E \to V^*$ maps each edge*

to a string of source nodes*, $z\colon E \to Z$ assigns a sort to each edge and $l\colon E \to L$ assigns a label to each edge.*

A hypergraph *or* multi-pointed hypergraph *$H = G[\chi]$ is composed of a simple hypergraph $G = (V, E, s, z, l)$ and a string $\chi \in V^*$. χ is called the string of* external nodes*. EXT_H is the set of all external nodes in H.*

The components of a hypergraph H are denoted by $V_H, E_H, s_H, z_H, l_H, \chi_H$, while the components of a simple hypergraph G are denoted by V_G, E_G, s_G, z_G, l_G. Furthermore we define the arity of edges and hypergraphs as follows: $ar(e) := |s_H(e)|$, if $e \in E_H$, and $ar(G[\chi]) := |\chi|$.

External nodes are the interface of a hypergraph with its environment and are used to attach hypergraphs. In the process calculus, which will be presented in section 3, we have two edge sorts, dividing the edge set into processes and messages, while the label of a process specifies its behaviour. In the rest of this paper we use both terms graph and hypergraph interchangeably

The following definition of hypergraph morphism is quite straightforward. A morphism is expected to preserve graph structure, as well as edge sorts and labels:

Definition 2. (Hypergraph Morphism) *Let G, G' be two simple hypergraphs.*

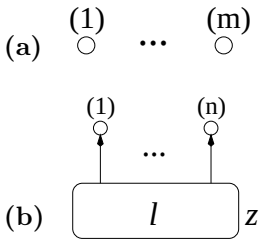
A hypergraph morphism *$\phi\colon G \to G'$ consists of two mappings $\phi_E\colon E_G \to E_{G'}$, $\phi_V\colon V_G \to V_{G'}$ satisfying for all $e \in E_G$:*

$$\phi_V(s_G(e)) = s_{G'}(\phi_E(e)) \quad z_G(e) = z_{G'}(\phi_E(e)) \quad l_G(e) = l_{G'}(\phi_E(e))$$

We write $\phi : G[\chi] \to G'[\chi']$ if $\phi : G \to G'$ is a hypergraph morphism. If[1] *$\phi_V(\chi) = \chi'$, ϕ is called a* strong morphism *and we write $\phi : G[\chi] \twoheadrightarrow G'[\chi']$*

$G[\chi]$ and $G'[\chi']$ are called strongly isomorphic *($G[\chi] \cong G'[\chi']$) if there exists a bijective strong morphism from one graph into the other.*

Notation:
We call a hypergraph *discrete*, if its edge set is empty. **m** denotes a discrete graph of arity $m \in \mathbb{N}$ with m nodes where every node is external (see (a), external nodes are labelled (1), (2), ... in their respective order).
$H := z_n(l)$ is the hypergraph with exactly one edge e with sort z and label l where $s_H(e) = \chi_H$, $|\chi_H| = n$, $V_H = EXT_H$ (see (b), nodes are ordered from left to right).



For the definition of the process calculus and its type system we need some basic concepts from category theory, namely categories, functors and co-limits. Detailed definitions can be found in [5].

Since we want to associate hypergraphs with lattice elements, we need a functor between the following two categories:

[1] Each morphism can be extended to strings of nodes in a canonical way, i.e. $\phi_V(v_1 \dots v_n) = \phi_V(v_1) \dots \phi_V(v_n)$

The category of hypergraphs (with hypergraph morphisms): The class of all simple hypergraphs (multi-pointed hypergraphs) forms a category together with the (strong) hypergraph morphisms.

The category of lattices (with join-morphisms): Let $(I_1, \leq_1)$, $(I_2, \leq_2)$ be two lattices with bottom elements $\bot_1$ respectively $\bot_2$. For two elements $a_1, b_1 \in I_1$ ($a_2, b_2 \in I_2$) let $a_1 \vee_1 b_1$ ($a_2 \vee_2 b_2$) be the *least upper bound* or *join* of the two elements.

A mapping $t : I_1 \to I_2$ is called a *join-morphism* iff $t(a_1 \vee_1 b_1) = t(a_1) \vee_2 t(b_1)$ and $t(\bot_1) = \bot_2$

Type graphs are hypergraphs $G[\chi]$ which are associated with a lattice element. A type functor F maps every type graph to a lattice $F(G)$ from which this associated lattice element can be taken. The concept of hypergraph morphisms can be extended to type graph morphisms, from which we demand, that they not only preserve graph structure but also the order in the corresponding lattices.

Definition 3. (Type Functors and Type Graphs) *A functor F from the category of simple hypergraphs into the category of lattices is called a* type functor*.*

$T = G[\chi, a]$ where $G[\chi]$ is a hypergraph and $a \in F(G)$ is called a type graph *wrt. F. The class of all type graphs wrt. F is denoted by $\mathcal{T}_F$.*

We write $\phi : G[\chi, a] \xrightarrow{F} G'[\chi', a']$ if $\phi : G \to G'$ is a hypergraph morphism and $F(\phi)(a) \leq a'$. ϕ is called type graph morphism*.*

We say ϕ is a strong type graph morphism *if additionally $\phi_V(\chi) = \chi'$ and it is denoted by $\phi : G[\chi, a] \overset{F}{\twoheadrightarrow} G'[\chi', a']$.*

Two type graphs $G[\chi, a]$, $G'[\chi', a']$ are called isomorphic (wrt. F) *if there exists a strong isomorphism $\phi : G[\chi] \twoheadrightarrow G'[\chi']$ such that $F(\phi)(a) = a'$. In this case we write $G[\chi, a] \cong_F G'[\chi', a']$.*

Note: If $H = G[\chi]$ we define $H[a] := G[\chi, a]$.

Example: We consider the following type functor F: let $(I, \leq)$ be an arbitrary lattice and let $k \in \mathbb{N}$. For any simple hypergraph G we define $F(G)$ as the set of all mappings from $(V_G)^k$ (cartesian product) into I (which yields a lattice with pointwise order).

Let $a : V_G^k \to I$, $\phi : G \to G'$, $s' \in V_{T'}^k$. We define:

$$F(\phi)(a) := a' \text{ where } a'(s') := \bigvee_{\phi_V(s) = s'} a(s)$$

It is straightforward to check that F is indeed a type functor.

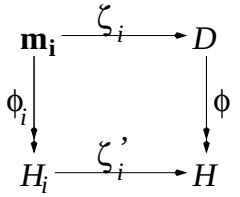
We now introduce a mechanism for the construction of hypergraphs. Compared to string concatenation it is not so obvious how to build larger graphs out of smaller ones. We describe a construction plan with morphisms mapping discrete graphs into discrete graphs. This construction plan is then applied to hypergraphs by a co-limit construction. Our method is related to the double-pushout approach for graph rewriting described in [6].

If we define how to transform and sum up lattice elements, we can assemble type graphs in the same way.

Definition 4. (Construction of Hypergraphs and Type Graphs)
Let $H_1, \ldots, H_n$ be hypergraphs and let $\zeta_i : \mathbf{m_i} \to D$, $i \in \{1, \ldots, n\}$ be hypergraph morphisms where $ar(H_i) = m_i \in \mathbb{N}$ and D is a discrete graph. There is always a unique strong morphism $\phi_i : \mathbf{m_i} \twoheadrightarrow H_i$ for every $i \in \{1, \ldots, n\}$.

Let H (with morphisms $\phi : D \to H$, $\zeta_i' : H_i \to H$) be the co-limit of $\zeta_1, \ldots, \zeta_n, \phi_1, \ldots, \phi_n$ such that ϕ is a strong morphism. We define:

$$\bigotimes_{i=1}^{n}(H_i, \zeta_i) := H$$



Let $T_i = H_i[a_i]$, $i \in \{1, \ldots, n\}$ be type graphs and let F be a fixed type functor. The construction of type graphs wrt. F is defined in the following way:

$$\bigotimes_{i=1}^{n}(T_i, \zeta_i) := \left(\bigotimes_{i=1}^{n}(H_i, \zeta_i)\right)[a] \text{ where } a := \bigvee_{i=1}^{n} F(\zeta_i')(a_i)$$

Generally, co-limits do not necessarily exist, but they always exist in our case. The co-limit is unique up to isomorphism (i.e. bijective morphisms), but *not* unique up to strong isomorphism. Therefore we demand above that the morphism from D into the co-limit is a strong morphism and thereby determine the string of external nodes of the result.

In order to clarify the intuition behind graph construction we give the following two examples:

Example 1: As stated above, the morphisms ζ_i can be regarded as a construction plan for assembling hypergraphs. The example in figure 1 will illustrate this: we describe how to construct H below out of smaller hypergraphs H_1, H_2, H_3. In this case $H \cong \bigotimes_{i=1}^{3}(H_i, \zeta_i)$ (see the graphical description of $\zeta_1, \zeta_2, \zeta_3$ below).
Example 2: Let H_1, H_2 be hypergraphs with $ar(H_1) = ar(H_2) = n$. Then $H_1 \square H_2 := \bigotimes_{i=1}^{2}(H_i, \zeta_i)$ where $\zeta_1, \zeta_2 : \mathbf{n} \twoheadrightarrow \mathbf{n}$ are the unique strong morphisms from $\mathbf{n}$ into $\mathbf{n}$. That is $H_1 \square H_2$ is constructed out of H_1, H_2 by fusing corresponding external nodes.

Every hypergraph can be decomposed into hyperedges and has the following normal form:

Proposition 1. (Graph construction out of hyperedges) *Let H be a hypergraph. Then there exists a natural number n, sorts z_i, labels l_i and morphisms $\zeta_i : \mathbf{m_i} \to D$ (where $i \in \{1, \ldots, n\}$ and D is a discrete hypergraph) such that*

$$H \cong \bigotimes_{i=1}^{n}((z_i)_{m_i}(l_i), \zeta_i)$$

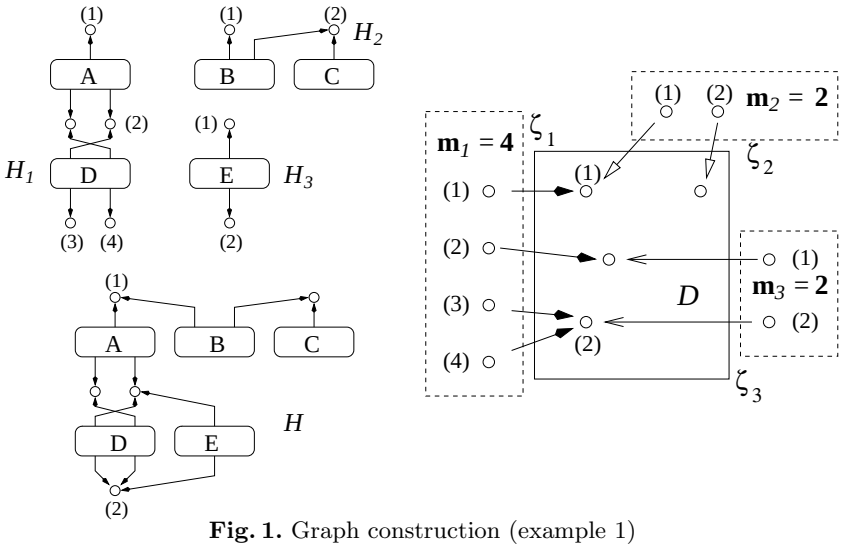


Fig. 1. Graph construction (example 1)

3 Process Graphs

We are now ready to introduce the calculus: an expression in our process calculus is a hierarchical hypergraph. Edges are representing either processes or messages (since we present an asynchronous calculus we distinguish processes and messages) and nodes are representing ports. In the rest of this paper we use the names "port" and "node" interchangeably.

Definition 5. (Process Graph) *A* process graph P *is inductively defined as follows:* P *is a hypergraph with edge sorts* $Z = \{proc, mess\}$*. Edges with* $z_P(e) = proc$ *are called processes and edges with* $z_P(e) = mess$ *are called messages.*

Processes are either labelled $!Q$ *(Replication) or* $\lambda_k^{(n)}.Q$ *(the process receives a message with* $n+1$ *ports—one of it the send-port—on its* k*-th port and then behaves like* Q*) where* Q *is again a process graph. Messages have at least arity* 1 *and remain unlabelled (or are labelled with dummies).*

By definition, a message is sent to its last port[2]: $send_P(e) := \lfloor s_P(e) \rfloor_{ar(e)}$ if $z_P(e) = mess$. Process graphs have an intuitive graphical representation which will be introduced step by step.

The most important form of reduction in our calculus is the reception of a message by a process, which means the replacement of a redex, consisting of process and message, by the hypergraph inside the process.

Redex: Let $P_1 := proc_m(l)$, $P_2 := mess_{n+1}$ and let $1 \leq k \leq m$. Furthermore let

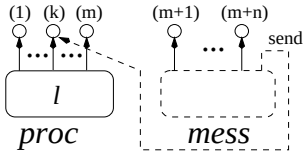
$$\zeta_1 : \mathbf{m} \to \mathbf{m} + \mathbf{n} \text{ with } \zeta_1(\chi_{\mathbf{m}}) := \lfloor \chi_{\mathbf{m}+\mathbf{n}} \rfloor_{1\ldots m}$$

[2] We define the following operator on strings: if $s = a_1 \ldots a_n$ is a string, we define $\lfloor s \rfloor_{i_1 \ldots i_n} := a_{i_1} \ldots a_{i_n}$.

$$\zeta_2 : \mathbf{n+1} \to \mathbf{m+n} \text{ with } \zeta_2(\chi_{\mathbf{n+1}}) := \lfloor \chi_{\mathbf{m+n}} \rfloor_{m+1 \ldots m+n\, k}$$

We define $Red_{k,m,n}(l) := \bigotimes_{i=1}^{2}(P_i, \zeta_i)$.

Graphical representation: We draw messages with dashed lines, thereby distinguishing them from processes.



We favour reduction semantics in the spirit of the Chemical Abstract Machine [3]. Our calculus obeys the rules of structural congruence and reduction in table 1. $\equiv$ is the smallest equivalence which contains hypergraph isomorphism and which satisfies the rules (C-ABSTR), (C-REPL$_1$), (C-CON) and (C-REPL$_2$).

Note that runtime errors may occur if $ar(P) \neq m+n$ in (R-MR) or $ar(P) \neq n$

$$\frac{P_1 \equiv P_2}{proc_n(\lambda_k.P_1) \equiv proc_n(\lambda_k.P_2)}\ \text{(C-ABSTR)} \qquad \frac{P_1 \equiv P_2}{proc_n(!P_1) \equiv proc_n(!P_2)}\ \text{(C-REPL}_1\text{)}$$

$$\frac{P_i \equiv Q_i,\ i \in \{1, \ldots, n\}}{\bigotimes_{i=1}^{n}(P_i, \zeta_i) \equiv \bigotimes_{i=1}^{n}(Q_i, \zeta_i)}\ \text{(C-CON)}$$

$$proc_n(!P) \equiv P \square proc_n(!P)\ \text{(C-REPL}_2\text{)}$$

$$Red_{k,m,n}(\lambda_k^{(n)}.P) \to P\ \text{(R-MR)}$$

$$\frac{Q \equiv P,\ P \to P',\ P' \equiv Q'}{Q \to Q'}\ \text{(R-EQU)} \qquad \frac{P_i \to P_i',\ (i \neq j \ \Rightarrow\ P_j \equiv P_j')}{\bigotimes_{i=1}^{n}(P_i, \zeta_i) \to \bigotimes_{i=1}^{n}(P_i', \zeta_i)}\ \text{(R-CON)}$$

Table 1. Operational semantics of process graphs

in (C-REPL$_2$), i.e. if the left hand and the right hand side of a rule do not have the same arity, or if there is a mismatch in arities for the construction operator $\square$.

Mobility of port addresses is inherent in rule (R-MR): For a process of the form $proc_m(\lambda_k^{(n)}.P)$ the arity of P should be $m+n$ in order not to cause runtime errors. If such a process receives a message with n ports attached to it, the rules cause the first m external ports of P to fuse with the ports of the process, while the rest of the ports fuses with the ports of the message. In this way a process can gain access to new ports which means dynamic restructuring of the entire process graph. This feature is often called mobility [13].

Our calculus, as presented here, is closely related to the asynchronous, polyadic π-calculus without summation (see appendix A). Asynchronous means, in this case, that the continuation of an output prefix is always 0, the nil process.

Example: In figure 2 we give a small example, illustrating message reception in the calculus. Note that messages are drawn with dashed lines, all other edges are representing processes. The dashed arrow leading away from a message indicates the send-port of a message. The arrows leading to the source nodes of an edge are ordered from left to right. Furthermore the external ports inside a process abstraction which are going to be fused with the ports of a message are filled with grey. The corresponding expression in the π-calculus would be

$$(\nu a)(\nu b)(a(a_1 a_2).\overline{a_2}\langle a_1\rangle.0 \mid b(b_1).\overline{a}\langle e_1 b_1\rangle.0 \mid \bar{b}\langle e_2\rangle.0)$$

where e_1, e_2 are the names representing external ports (the only free names). (See also appendix A.)

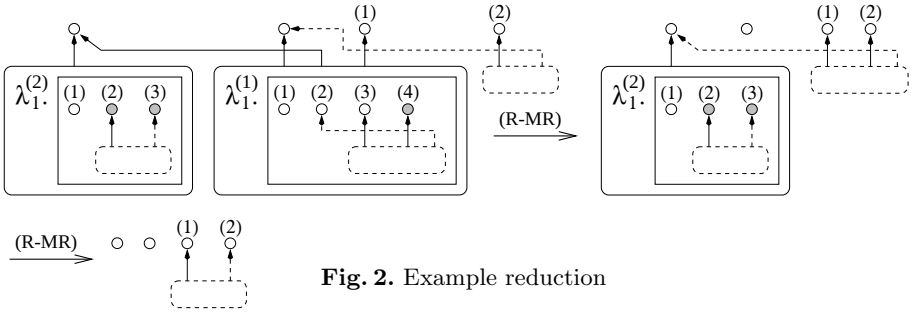


Fig. 2. Example reduction

4 The Type System

We assume that F is a fixed type functor. It is one of two parameters of the type system, we will now specify the second: we need a method for mapping a process graph to a corresponding type graph. It is only necessary to describe this mapping for graphs consisting of one edge only. The extension to arbitrary graphs is straightforward.

Definition 6. (Linear Mapping) *Let L be a function which maps graphs of the form $z_n(l)$ to type graphs in $\mathcal{T}_F$, satisfying $ar(L(z_n(l))) = n$. Furthermore we demand that*

$$P_1 \equiv P_2 \Rightarrow L(P_1) \cong_F L(P_2) \tag{1}$$

$$\exists\, \phi : mess_n[\bot] \overset{F}{\twoheadrightarrow} L(mess_n) \tag{2}$$

Since proposition 1 implies that all hypergraphs can be constructed out of graphs of the form $z_n(l)$ we can expand L to arbitrary hypergraphs in the following way:

$$L(\bigotimes_{i=1}^{n}(H_i, \zeta_i)) := \bigotimes_{i=1}^{n}(L(H_i), \zeta_i)$$

L is well-defined and is called a linear mapping.

Condition (2) in the definition of the linear mapping may seem somewhat out of place. It is however (together with condition (3) below and rule (T-ABSTR)) essential to the proof of the subject reduction property (see proof sketch below). Both conditions ensure that nodes that might get fused during reduction are already fused in the type graph.

The type system works with arbitrary linear mappings as long as they satisfy conditions (1) and (2). In practice, however, the structure of the graphs created by the linear mappings does not vary much (see also section 5) and the important part in defining the linear mappings is to choose sensible lattice elements.

We have now described the two parameters of the type system: the type functor F and the linear mapping L and the conditions imposed on them. The typing rules in table 2 describe how a type can be assigned to an expression, $P \triangleright G[\chi, a]$ meaning that the process graph P has type $G[\chi, a]$. We demand that in G every port is the send-port of at most one message, i.e. G satisfies:

$$e, e' \in E_G,\ z_G(e) = z_G(e') = mess_G,\ send_G(e) = send_G(e') \ \Rightarrow\ e = e' \quad (3)$$

The main motivation behind the typing rules is to ensure that there exists a morphism $L(P) \overset{F}{\twoheadrightarrow} T$, if $P \triangleright T$, and that the subject reduction property holds. The former is ensured by the morphisms in rules (T-PROC), (T-MESS) and (T-CON) and the latter is mainly ensured by typing rule (T-ABSTR). In (T-ABSTR) we demand the existence of a message in the type graph which implies, with conditions (2) and (3), that, in the type graph, the images of ports attached to any message arriving at the k-th port are already fused with the images of the last n ports of P. (T-CON) checks that all parts of a hypergraph are typed and

$$\frac{l \triangleright G[\chi, a],\ \exists\, \phi : L(proc_n(l)) \overset{F}{\twoheadrightarrow} G[\chi, a]}{proc_n(l) \triangleright G[\chi, a]}\ \text{(T-PROC)}$$

$$\frac{\exists\, \phi : L(mess_n) \overset{F}{\twoheadrightarrow} G[\chi, a]}{mess_n \triangleright G[\chi, a]}\ \text{(T-MESS)} \qquad \frac{P \triangleright G[\chi, a]}{!P \triangleright G[\chi, a]}\ \text{(T-REPL)}$$

$$\frac{P \triangleright G[\chi, a],\ \exists\, \phi : mess_{n+1} \twoheadrightarrow G[\lfloor \chi \rfloor_{m+1 \ldots m+n\, k}],\ |\chi| = m + n}{\lambda_k^{(n)}.P \triangleright G[\lfloor \chi \rfloor_{1 \ldots m}, a]}\ \text{(T-ABSTR)}$$

$$\frac{\exists\, \phi : D \twoheadrightarrow G[\chi],\ \zeta_i : \mathbf{m_i} \to D,\ P_i \triangleright G[\phi(\zeta_i(\chi_{\mathbf{m_i}})), a],\ i \in \{1, \ldots, n\}}{\bigotimes_{i=1}^{n} (P_i, \zeta_i) \triangleright G[\chi, a]}\ \text{(T-CON)}$$

Table 2. Typing Rules

that their types overlap at least in the corresponding external ports (they may overlap in other places as well). (T-REPL) states that we can produce copies of a process without changing its type since all copies are represented by the same part of the type graph, and the join operation in lattices is idempotent.

Condition (3) can not be satisfied if there are messages with a different number of ports sent to the same port. It therefore ensures that the arities of expected and received message match and thus avoids runtime errors.

The type system satisfies the properties listed below.

Proposition 2. (Properties of the Type System)

Subject Reduction Property: $P \triangleright G[\chi, a],\ P \rightarrow^* Q \ \Rightarrow\ Q \triangleright G[\chi, a]$

Runtime Errors: $P \triangleright G[\chi, a]$ *implies that* P *will never cause a runtime error.*

Morphisms: $P \triangleright G[\chi, a] \ \Rightarrow\ \exists\, \phi : L(P) \overset{F}{\twoheadrightarrow} G[\chi, a]$

Principal Types: *If* P *is typable then there exists a* principal *type graph* $P \triangleright G[\chi, a]$ *with*

- $\phi : G[\chi, a] \overset{F}{\twoheadrightarrow} G'[\chi', a']$ *implies* $P \triangleright G'[\chi', a']$
- $P \triangleright G'[\chi', a']$ *implies the existence of* $\phi : G[\chi, a] \overset{F}{\twoheadrightarrow} G'[\chi', a']$*, where* ϕ *is a strong morphism*

And there exists a type inference algorithm constructing the principal type graph for every process graph, if it exists.

In order to expose the intuition behind the type system we show that reduction of a process does not change its type. In order to unravel the typing and to be able to trace it backwards, we need the following non-trivial lemma:

Lemma 1. *Let* $\bigotimes_{i=1}^{n}(P_i, \zeta_i) \triangleright G[\chi, a]$ *with* $\zeta_i : \mathbf{m_i} \rightarrow D$*. Then there is a strong morphism* $\phi : D \twoheadrightarrow G[\chi]$ *such that for all* $i \in \{1, \ldots, n\}$*:* $P_i \triangleright G[\phi(\zeta_i(\chi_{\mathbf{m_i}})), a]$

We now demonstrate how to prove the subject reduction property in the case of (R-MR):

Proof Sketch (Subject Reduction Property): *Let* $Red_{k,m,n}(\lambda_k.Q) \triangleright G[\chi, a]$*. It follows with lemma 1 that*

$$proc_m(\lambda_k.Q) \ \triangleright\ G[\lfloor\chi\rfloor_{1\ldots m}, a], \quad mess_{n+1} \ \triangleright\ G[\lfloor\chi\rfloor_{m+1\ldots m+n\,k}, a]$$

Since $proc_m(\lambda_k.Q)$ *was typed with (T-PROC) and (T-ABSTR) it follows that there exists a* $\chi' \in V_G^*$ *such that*

$$Q \ \triangleright\ G[\lfloor\chi\rfloor_{1\ldots m} \circ \chi', a], \quad mess_{n+1} \ \twoheadrightarrow\ G[\chi' \circ \lfloor\chi\rfloor_k]$$

And since (2) and (T-MESS) imply that

$$mess_{n+1}[\bot] \overset{F}{\twoheadrightarrow} L(mess_{n+1}) \overset{F}{\twoheadrightarrow} G[\lfloor\chi\rfloor_{m+1\ldots m+n\,k}, a]$$

it follows with condition (3) that $\chi' = \lfloor\chi\rfloor_{m+1\ldots m+n}$ *and therefore* $Q \triangleright G[\chi, a]$*.*

We now describe how the type system can be used for verification purposes: we introduce two predicates X and Y where X is a predicate on type graphs and Y is a predicate on process graphs. We want to show that Y is an invariant with the help of X.

Proposition 3. (Process Analysis with the Type System) *Let Y be a predicate on process graphs and let X be a predicate on type graphs of the form $G[\chi, a]$. We assume that X, Y satisfy*

$$X(L(P)) \Rightarrow Y(P) \tag{4}$$

$$\phi : G[\chi, a] \overset{F}{\twoheadrightarrow} G'[\chi', a'],\ X(G'[\chi', a']) \Rightarrow X(G[\chi, a]) \tag{5}$$

i.e. X is closed under inverse hypergraph morphisms.

Then $P \triangleright G[\chi, a]$ and $X(G[\chi, a])$ imply $Y(Q)$ for all $P \rightarrow^* Q$.

A full type system is determined by four components: the type functor F, a linear mapping L and the two predicates X, Y.

5 Examples

In the following examples we use a rather restricted linear mapping L with

$$L(proc_n(l)) \cong_F \mathbf{n}[a_p] \quad L(mess_n) \cong_F mess_n[a_m]$$

for lattice elements a_p, a_m, yet to be determined. That is, in this case, type graphs consist of messages only. Furthermore we use the type functor F introduced in the example after definition 3. The simplest version of this type system, where every lattice consists of one element only, corresponds to standard type systems for the π-calculus with recursive types, but without let-polymorphism [21].

In all of the following examples the predicate X is preserved by inverse hypergraph morphisms.

5.1 Input/Output-Capabilities

We want to ensure that some external ports are only used as input ports (i.e. for receiving messages) and that some are only used as output ports (i.e. for sending messages). For F we choose $k = 1$, $I = \{\bot, in, out, both\}$ where $\bot < in < both$ and $\bot < out < both$ and $in \vee out = both$. The linear mapping L is defined in the following way:

$$L(proc_m(l)) = \mathbf{n}[a_p] \text{ where } a_p(\lfloor \chi \rfloor_i) = \begin{cases} in \text{ if } l = \lambda_i^{(n)}.P,\ n \in \mathbb{N} \\ \bot \text{ otherwise} \end{cases}$$

$$L(mess_n) = mess_n[a_m] \text{ where } a_m(\lfloor \chi \rfloor_i) = \begin{cases} out \text{ if } i = n \\ \bot \quad \text{otherwise} \end{cases}$$

Since the only nodes of $proc_n$ and $mess_n$ are external, it is sufficient to define a_p and a_m on the respective string χ of external nodes.

We want to ensure that P will never reduce to a process graph P' where a message is sent to $\lfloor \chi_P \rfloor_i$. The corresponding predicate X is:

$$X(G[\chi, a]) := (a(\lfloor \chi \rfloor_i) \leq in)$$

If we replace *in* by *out* in X we can conclude that P will never reduce to a process graph P' where a process listens at $\lfloor \chi_P \rfloor_i$.

A similar version of this type system is presented in [19].

Typing the example process graph from section 3 (figure 2) yields the principal type in figure 3 (left). This implies that the first external part is not used for any I/O-operations at all, while the second external port is only an output port.

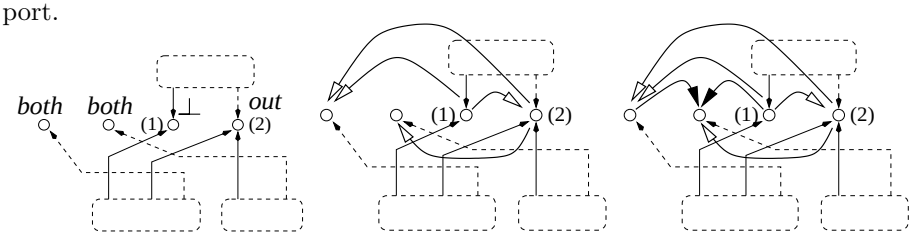


Fig. 3. Types for process graphs (input/output, secrecy, deadlocks)

5.2 Secrecy of External Ports

We assume that the external ports of a process graph can have different levels of secrecy. They might either be public or secret. Both sorts of ports can be used to send or receive message, but it is not allowed to forward a secret port to a receiver listening at a public port.

We choose $k = 2$, $I = \{\mathit{false}, \mathit{true}\}$ where $\{\mathit{false}, \mathit{true}\}$ is the boolean lattice with $\mathit{false} < \mathit{true}$. If a tuple (v_1, v_2) is associated with *true*, this means that the port v_2 is sent to v_1. The mapping L has the following form:

$$L(\mathit{proc}_m(l)) = \mathbf{n}[a_p] \text{ where } a_p(\lfloor \chi \rfloor_i, \lfloor \chi \rfloor_j) = \mathit{false}$$

$$L(\mathit{mess}_n) = \mathit{mess}_n[a_m] \text{ where } a_m(\lfloor \chi \rfloor_i, \lfloor \chi \rfloor_j) = \begin{cases} \mathit{true} \text{ if } i = n, j \neq n \\ \mathit{false} \text{ otherwise} \end{cases}$$

Let P be a process graph and we assume that the sets SEC and PUB form a partition of $\{1, \ldots, ar(P)\}$. If the type of P satisfies

$$X(G[\chi, a]) := (\forall\, i \in PUB, j \in SEC : a(\lfloor \chi \rfloor_i, \lfloor \chi \rfloor_j) = \mathit{false})$$

it follows that no message, with secret ports attached to it, is ever sent to a public port. In this case typing the example process graph yields the principal type in figure 3 (middle), where an arrow from port v_2 to v_1 indicates that $a(v_1, v_2) = \mathit{true}$. The predicate X is not satisfied only in the case where the first external port is secret and the second external port is public. In all other cases, the process graph is well-typed.

[4] presents a related method for checking the secrecy of ports.

5.3 Avoiding Deadlocks

We attempt to avoid vicious circles of processes and messages waiting for one another and causing a deadlock. Let P be a process graph with a non-empty

edge set such that there is no process graph P' with $P \to P'$ and (C-REPL$_2$) is not applicable, i.e. P is stuck. Then at least one of the following conditions is satisfied:

(1) There is a message waiting or a process listening at an external port. This case is good-natured, since P is only waiting to perform an I/O-operation.

(2) There is an internal port where all edges connected to it are either messages, sent to this port, or processes listening at this port.

(3) There is a vicious circle, i.e. a sequence $v_0, \ldots, v_n = v_0 \in V_P$ such that for every pair v_i, v_{i+1} there is *either* a message q with $send_P(q) = v_i$ and $\lfloor s_P(q) \rfloor_j = v_{i+1}$ for some $j \in \{1, \ldots, ar(q) - 1\}$ *or* a process p with $l_P(p) = \lambda_k^{(n)}.Q$, $\lfloor s_P(p) \rfloor_k = v_i$ and $\lfloor s_P(p) \rfloor_j = v_{i+1}$ for some $j \in \{1, \ldots, ar(p)\}$, $j \neq k$.

Our aim is to avoid circles as described in **(3)**. We set $k := 2$ and I is again the boolean lattice with *false* and *true*. We define:

$$L(proc_m(l)) = \mathbf{n}[a_p] \text{ where } a_p(\lfloor\chi\rfloor_i, \lfloor\chi\rfloor_j) = \begin{cases} \mathit{true} \text{ if } l = \lambda_i^{(n)}.P, j \neq i,\ n \in \mathbb{N} \\ \mathit{false} \text{ otherwise} \end{cases}$$

$$L(mess_n) = mess_n[a_m] \text{ where } a_m(\lfloor\chi\rfloor_i, \lfloor\chi\rfloor_j) = \begin{cases} \mathit{true} \text{ if } i = n, j \neq n \\ \mathit{false} \text{ otherwise} \end{cases}$$

$$X(G[\chi, a]) := (\nexists v_0, \ldots, v_n = v_0 \in V_G : a(v_i, v_{i+1}) = \mathit{true},\ 0 \leq i < n)$$

In this case we can retrieve the principal type of our example process graph rather easily from the type in section 5.2. All we have to do is add the arrows (with filled arrow heads) produced by a_p (see figure 3 (right)).

Since there is no circle of arrows we can conclude that the process graph will (during its reduction) never contain a vicious circle as described in **(3)**.

Similar methods for avoiding deadlocks are presented in [12, 10].

5.4 Composing Type Systems

We asumme that we have two type systems, with functors F_1, F_2, linear mappings L_1, L_2, predicates X_1, X_2 on type graphs and predicates Y_1, Y_2 on process graphs.

We define a functor F with $F(G) := (I_1 \times I_2, \leq)$ if $F_i(G) = (I_i, \leq_i)$, $i = 1, 2$ and $(a_1, a_2) \leq (a_1', a_2') \iff a_1 \leq_1 a_1'$ and $a_2 \leq_2 a_2'$.

Furthermore $F(\phi)((a_1, a_2)) := (F_1(\phi)(a_1), F_2(\phi)(a_2))$

Furthermore let $L(P) := G[\chi, (a_1, a_2)]$ if $L_1(P) \cong_{F_1} G[\chi, a_1]$ and $L_2(P) \cong_{F_2} G[\chi, a_2]$. This requires, of course, that L_1 and L_2 map process graphs to type graphs of the same structure. This is actually not a severe restriction since all practical examples can be defined in such a way that they satisfy this condition (see the examples in this section).

$$\begin{aligned} \text{We define} \quad X_\wedge(G[\chi, (a_1, a_2)]) &:= X_1(G[\chi, a_1]) \wedge X_2(G[\chi, a_2]) \\ X_\vee(G[\chi, (a_1, a_2)]) &:= X_1(G[\chi, a_1]) \vee X_2(G[\chi, a_2]) \end{aligned}$$

Then $F, L, X_\wedge, Y_1 \wedge Y_2$ respectively $F, L, X_\vee, Y_1 \vee Y_2$ denote type systems checking the conjunction respectively disjunction of Y_1 and Y_2. A type system checking a negated property can only be constructed in very special cases.

6 Conclusion and Comparison to Related Work

Several type systems for process calculi have been proposed, each checking different properties of processes, e.g. I/O-capabilities [19], confluence [17], secrecy in security protocols [1] or deadlock-freedom [10]. This paper is an attempt to integrate these approaches and to propose one single generic type system which can be instantiated in order to verify invariant properties of processes. Our type system seems to be especially well-suited for properties related to the geometry of processes and messages represented in the process graph.

In [8] Kohei Honda proposes a general framework for type systems, satisfying the condition of *strict additivity*, i.e. two process connected to each other via several ports can be typed if and only if connections via one port only can be typed. As Honda remarks in his paper, strict additivity is sometimes too strong, e.g. in the case of deadlock-freedom. The type system presented in this paper is *semi-additive*, i.e. the "if and only if" is replaced by "only if". In contrast to [8] our method of instantiation is restricted, but this enables us to prove the subject reduction property for all possible type systems.

There is, of course, a trade-off between generality and the percentage of processes which can be typed: e.g. in the case of I/O-capabilities our type system is less powerful than the type system introduced in [19], which can partly be explained by the very general nature of our type system and partly by the fact that the type system in [19] does not have principal types.

We believe, however, that this type system can serve as a starting point for further research.

We will finish by dicussing two existing extensions of this type system which we were not able to present here due to limited space:

- Sometimes labelling processes with lattice elements does not seem to be sufficient. E.g. if we want to type confluent processes [17], typing involves the counting of processes and messages adjacent to a certain port. (In this case we have to demand that there is at most one process listening and at most one message waiting at a certain port.) Counting is also necessary if we want to design a type system checking **(2)** in the conditions for deadlocks or if we attempt to introduce linear types as in [11].
  An extension of our type system is based on latttice-ordered monoids rather than on lattices.
- It is not very difficult to extend the process calculus to a calculus where higher-order communication is possible, i.e. where entire processes can be sent as the content of a message. The corresponding extension of the type system is not very hard.
  We have to add environments, describing the type of the variables in a process graph. Furthermore it is necessary to slightly change the linear mapping L and rule (T-ABSTR).

Another future area of research is the use of the results of the type system in order to establish bisimilarity of processes (as in [1]).

References

1. Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, pages 611–638. Springer-Verlag, 1997.
2. Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20:83–127, 1987.
3. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
4. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *CONCUR '98*, pages 84–98. Springer-Verlag, 1998. LNCS 1466.
5. Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.
6. 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.
7. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.
8. Kohei Honda. Composing processes. In *Proc. of POPL'96*, pages 344–357. ACM Press, 1996.
9. Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *POPL '94*, pages 348–360. ACM Press, 1994.
10. Naoki Kobayashi. A partially deadlock-free typed process calculus. In *LICS '97*, pages 128–139. IEEE, Computer Society Press, 1997.
11. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL '96*, pages 358–371. ACM SIGACT/SIGPLAN, 1996.
12. Yves Lafont. Interaction nets. In *POPL '90*, pages 95–108. ACM Press, 1990.
13. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I. Tech. Rep. ECS-LFCS-89-85, University of Edinburgh, Laboratory for Foundations of Computer Science, 1989.
14. Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 1201–1242. Elsevier, 1990.
15. Robin Milner. The polyadic π-calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, Laboratory for Foundations of Computer Science, 1991.
16. Robin Milner. Pi-nets: a graphical form of pi-calculus. In *European Symposium on Programming*, pages 26–42. Springer-Verlag, 1994. LNCS 788.
17. Uwe Nestmann and Martin Steffen. Typing confluence. In *ERCIM Workshop on Formal Methods in Industrial Critical Systems*, pages 77–101, 1997.
18. Joachim Parrow. Interaction diagrams. *Nordic Journal of Computing*, 2:407–443, 1995.
19. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS '93*, pages 376–385, 1993.
20. Antonio Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Euro-Par '97*. Springer-Verlag, 1997.
21. David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. ECS-LFCS-96-345.
22. P. Urzyczyn. Positive recursive type assignment. In J. Wiedermann and P. Hájek, editors, *Mathematical Foundations of Computer Science 1995*. Springer-Verlag, 1995. LNCS 969.
23. Nobuko Yoshida. Graph types for monadic mobile processes. In *FST/TCS '96*, pages 371–386. Springer-Verlag, 1996.

A Process graphs and the asynchronous polyadic π-calculus

In order to show how process graphs are related to the π-calculus we give an encoding, transforming a subset of all process graphs to expressions in the asynchronous polyadic π-calculus.

Definition 7. (Encoding) *Let P be a process graph such that χ_P is duplicate-free, i.e. $\lfloor\chi_P\rfloor_i = \lfloor\chi_P\rfloor_j$ implies $i = j$. And we assume that the same condition is satisfied for all process graphs occurring inside of P. Let $\mathcal{N}$ be the name set of the π-calculus and let $t \in \mathcal{N}^*$ such that $|t| = ar(P)$. We define $\Theta_t(P)$ inductively as follows:*

Message: $\Theta_{a_1 \dots a_n}(mess_n) := \overline{a_n}\langle a_1 \dots a_{n-1}\rangle.0$
Replication: $\Theta_t(proc_m(!P)) :=! \Theta_t(P)$
Process Abstraction: $\Theta_{a_1 \dots a_m}(proc_m(\lambda_k^{(n)}(P))) := a_k(x_1 \dots x_n).\Theta_{a_1 \dots a_m x_1 \dots x_n}(P)$
where $x_1, \dots, x_n \in \mathcal{N}$ are fresh names.
Process Graph Construction:
$\Theta_t(\bigotimes_{i=1}^n (P_i, \zeta_i)) := (\nu\, \mu(V_D \backslash EXT_D))(\Theta_{\mu(\zeta_1(\chi_{\mathbf{m_1}}))}(P_1) \mid \dots \mid \Theta_{\mu(\zeta_n(\chi_{\mathbf{m_n}}))}(P_n))$
where $\zeta_i : \mathbf{m_i} \to D$, $i \in \{1, \dots, n\}$ and $\mu : V_D \to \mathcal{N}$ is an arbitrary mapping such that μ restricted to $V_D \backslash EXT_D$ is injective and $\mu(\chi_D) = t$.
If $n = 0$ (i.e. if the process graph is identical to D) we set $\Theta_t(\bigotimes_{i=1}^0 (P_i, \zeta_i)) := 0$.

The set of all process graphs satisfying the condition in the definition above is closed under reduction and corresponds exactly to the asynchronous part of the polyadic π-calculus without summation. (We rely on the syntax and semantics given for its synchronous version in [19], omitting sort annotations.)

Proposition 4. *Let p be an arbitrary expression in the asynchronous polyadic π-calculus without summation. Then there exists a process graph P (satisfying the condition in definition 7) and a duplicate-free string $t \in \mathcal{N}^*$ such that $\Theta_t(P) \equiv p$.*

Furthermore for process graphs P, P' satisfying the condition in definition 7 and for every duplicate-free string $t \in \mathcal{N}^*$ with $|t| = ar(P) = ar(P')$ it is true that:

- *$P \equiv P'$ implies $\Theta_t(P) \equiv \Theta_t(P')$* — *$P \to P'$ implies $\Theta_t(P) \to \Theta_t(P)$*
- *$\Theta_t(P) \to p$ with $p \neq wrong$ implies that there exists a process graph Q with $P \to Q$ and $\Theta_t(Q) \equiv p$*

The proposition implies that one calculus can match the reductions of the other step by step. The main difference of the calculi lies in their interface towards the environment. How these interfaces are converted into one another is described by the string t.