

Description and Verification of Mobile Processes with Graph Rewriting Techniques

Barbara König

Lehrstuhl für Informatik II
der Technischen Universität München

**Description and Verification of Mobile Processes
with Graph Rewriting Techniques**

Barbara König

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen
Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. R. Bayer, Ph.D.

Prüfer der Dissertation: 1. Univ.-Prof. Dr. Dr.h.c. J. Eickel

2. Univ.-Prof. T. Nipkow, Ph.D.

Die Dissertation wurde am 17.12.1998 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik am 30.4.1999 angenommen.

Abstract

The aim of this thesis is to describe the semantics of a process calculus by means of hypergraph rewriting, creating a specification mechanism combining modularity of process calculi and locality of graph transformation. Verification of processes is addressed by presenting two methods: barbed congruence for relating processes displaying the same behaviour and generic type systems, forming a central part of this work.

Based on existing work in graph rewriting we compare various methods of hypergraph construction—the set-based and the categorical approach (Ehrig), graph expressions (Courcelle) and a name-based notation—and concentrate on the decomposition of a hypergraph into factors.

The hypergraph-based process calculus features higher-order communication and mobility of port addresses while its graph syntax allows an intuitive graphical representation. We demonstrate the expressiveness of our calculus by encodings of the λ -calculus and the π -calculus.

Verification of processes is supported by a generic type system, forming a framework which can be instantiated in order to check a property (e.g. absence of deadlocks, confluence, privacy). The type system satisfies the subject reduction property, has principal types and allows automated type inference.

The rather complex type of a process is again a graph, which enables us to use techniques of graph construction introduced earlier. The key idea is, that there exists a graph morphism from each process into its type. Labelling a type with lattice or monoid elements (e.g. the number of messages attached to a port) allows us to encode properties of a process into its type and to employ the information for verification.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Prof. Jürgen Eickel for giving me the opportunity to conduct this research at his chair, for introducing me to graph grammars and for his guidance and help.

I am also indebted to Prof. Tobias Nipkow for acting as a referee of this thesis.

Special thanks go to Boris Reichel and Christine Röckl for many helpful discussions and new ideas, for reading drafts of this thesis and for their encouragement. I want to thank my present and former colleagues for a good working atmosphere, for our discussions during lunch and for many other things. They are (in alphabetical order) Bernhard Bauer, Alfons Brandl, Matthias Göbel, Franz Haßmann, Riitta Höllerer, Marianne Hargitay, Aurel Huber, Frank Lonczewski, Christiana Maschmeyer, Peter Müller, Christoph Pleier, Arnd Poetzsch-Heffter, Siegfried Schreiber, Werner Schreiber, Olga Trinkler-Yesaulenko and Hans Wittenner.

Furthermore I would like to thank my fellow scholarship holders for the exchange of ideas, their support and encouragement. My work would not have been possible without the financial and organizational support of the graduate college “Kooperation und Ressourcenmanagement in verteilten Systemen”. Special thanks go to the speakers of the graduate college Prof. P.P. Spies and Prof. Wilfried Brauer.

Finally I want to thank Alexander and my parents for their encouragement and their help during the whole time.

Contents

1	Introduction	5
1.1	Describing Mobile Processes with Hypergraphs	5
1.2	Generic Type Systems	6
1.3	Summary of this Work	7
1.4	Dependency Graph of the Chapters and Sections	9
1.5	Remarks on Notation	9
2	Methods of Hypergraph Construction	12
2.1	Set-Based Approach	13
2.1.1	Basic Definitions	13
2.1.2	Quotient Graphs	19
2.1.3	Hypergraph Construction with Context Graphs	20
2.2	Categorical Approach	22
2.2.1	Introduction to Category Theory	22
2.2.2	Graph Construction	26
2.2.3	Connection with the Double-Pushout Approach	32
2.2.4	Co-Limits, Factorizations and Contexts	33
2.3	Graph Expressions	36
2.4	Name-based Notation	41
2.5	Comparison of Hypergraph Construction Methods	47
3	Process Calculi: An Overview	49
3.1	Introduction to CCS	50
3.2	Reduction Semantics and the Chemical Abstract Machine	51
3.3	Mobility in the π -Calculus	52
3.4	Higher-Order Communication—Static and Dynamic Binding	54
3.5	Bisimulation and Proof Techniques	54
3.6	Comparison of Calculi and Full Abstraction	56
3.7	Synchronicity and Asynchronicity	57
3.8	Graph Calculi	57
3.9	Type Systems	57
3.10	Other Process Calculi	58
4	The SPIDER Calculus: Syntax and Semantics	59
4.1	Syntax	59
4.2	Graphical Representation	60

4.3	Semantics	61
4.4	Example: Client-Server-Interaction	65
5	Extended Example: Mail Delivery System	68
5.1	Informal Description	68
5.2	Routers and Routing Information	69
5.3	Local Administrators	73
5.4	Possible Extensions	75
6	Bisimulation and Proof Techniques	76
6.1	Barbed Congruence	76
6.2	Garbage Collection	79
6.3	Normal Bisimulation	80
6.4	Example: Simulating Replication	84
7	Comparison with other Formalisms	87
7.1	How to Compare Calculi	87
7.2	The λ -Calculus	90
7.2.1	Informal Comparison	90
7.2.2	Encoding the λ -Calculus into SPIDER, Version I	92
7.2.3	Encoding the λ -Calculus into SPIDER, Version II	94
7.3	SPIDER with Graph Expressions	97
7.4	SPIDER in the Name-based Notation	99
7.5	The π -Calculus	103
8	Generating Type Systems	107
8.1	Motivation	109
8.2	Lattices, Monoids and Type Graphs	115
8.2.1	Lattice-Ordered Commutative Monoids	116
8.2.2	Type Graphs and Type Functors	118
8.3	A Type System Based on Lattices	124
8.3.1	The Type System	124
8.3.2	Subject Reduction Property	126
8.3.3	Type Inference Algorithm	129
8.3.4	Verification with the Type System	133
8.3.5	Composing Type Systems	134
8.3.6	Examples	134
8.4	A Type System Based on Monoids	138
8.4.1	The Type System	138
8.4.2	Subject Reduction Property	141
8.4.3	Type Inference	143
8.4.4	Verification with the Type System	143
8.4.5	Composing Type Systems	144
8.4.6	Compatible Type Functors	145
8.4.7	Examples	147
8.4.8	Confluence	154
8.4.9	Avoiding Deadlocks	156

8.5	Transformation of Type Systems	159
8.5.1	Conversion Lattice \rightarrow Monoid	159
8.5.2	Conversion Monoid \rightarrow Lattice	159
8.6	Comparison of Type Systems: SPIDER \leftrightarrow π -Calculus	160
8.6.1	SPIDER \rightarrow π -Calculus	161
8.6.2	π -Calculus \rightarrow SPIDER	162
9	Conclusion	163
A	Proofs	165
A.1	Methods of Hypergraph Construction	165
A.2	Generating Type Systems	179
A.2.1	A Type System Based on Lattices	179
A.2.2	A Type System Based on Monoids	193
A.2.3	Transformation of Type Systems	197
A.2.4	Comparison of Type Systems: SPIDER \leftrightarrow π -Calculus . . .	202

Chapter 1

Introduction

1.1 Describing Mobile Processes with Hypergraphs

The aim of this work is to introduce a specification method (called SPIDER) for mobile processes, which is based on hypergraphs. It combines features of higher-order process algebras [San92, Tho95] and of graph rewriting. Graph rewriting is an intuitive formalism well-suited for the specification of complex structures and their dynamic behaviour [Hab92, Laf90, KLG93, JR90]. Replacing a connected subgraph by another graph is a strictly local operation without any influence on the rest of the graph. Yet graph rewriting often lacks modularity since it is difficult to decompose a set of rewrite rules into modules.

Process algebras and other calculi on the other hand are strong in modularity issues. Interacting components are found by common names, rather than by adjacency. (The concept of identifying interacting components by adjacency seems to be also natural and can be found in the λ -calculus [Bar84] and in Boudol's γ -calculus [Bou89].)

We attempt to preserve, in our calculus, both features: *locality* and *modularity*.

SPIDER adopts basic concepts of the λ -calculus (such as abstraction and application), while the string notation is replaced by hierarchical hypergraphs with expressions attached to hyperedges. This reflects a natural perception of a distributed system, where interconnected components are part of a multi-dimensional structure. No global port or channel names are needed.

The two versions of representing interconnected structure, which we will call *graph-based* and *name-based*, both have their merits. It is part of this work to show that both notations can be converted into one another, so that it is possible to exploit the advantages of both.

While name-based notation is very common, attempts to describe processes by graphs are less widespread, so it is necessary to explain, in more detail, the reasons for this notation. We will now summarize the main features of hypergraph representation for processes:

Visualization: One obvious strength of graph representation is, of course, visualization. There are several approaches to simplify programming by

allowing the programmer to specify and view components in a graphical way. Especially in concurrent programming graphical representation arises very naturally. Although it would be, in my opinion, a mistake to simply equate the terms “graph-based” and “graphical”, most types of graphs come with an intuitive and suggestive graphical depiction.

Various Representations: As already stated above, there are several methods of graph construction and graph rewriting: the *set-based approach*, the *categorical approach*, *graph expressions*, and a *name-based notation*, which is used in most process calculi. We will see that these approaches have different advantages but can easily be converted into one another.

Therefore the various graph representations give us the possibility to view a structure from different angles and to choose whatever representation is appropriate.

Fusing of External Ports: There is one feature of graph notation which cannot be simulated in process calculi where fusing of ports is described by name substitution (another case is the Fusion Calculus [PV98], where fusion of ports is realized by an equivalence relation). In these calculi a process cannot connect two external ports to which it is attached, a concept which is entirely natural and desirable in many cases. We will show later how this can easily be achieved in graph notation.

Concepts from Graph Theory: We will show in this work how concepts adopted from graph theory play an important role in the analysis of reactive systems. Concepts like adjacency, morphism and isomorphism can be used to analyze and verify programs.

Imposing Additional Structure: Since, in the set-theoretic representation of graphs, each component (process, message, port) is an object of its own (having a unique name), it is easy to impose additional structure, i.e. to introduce labels that were not conceived at the beginning. This is especially important for our type system where a graph is annotated with additional information in order to check certain properties.

Since perfect solutions seem to be rare, there is also a price one has to pay for the benefits mentioned above. Our graph notation introduces a certain overhead since the interface of a process has to be specified explicitly on each hierarchy level.

1.2 Generic Type Systems

It is the aim of the second part of this thesis to propose a general framework for the generation of type systems checking invariant properties of processes. We will propose a generic type system for our graph-based calculus. Specialized type systems can then be generated by instantiating the original system. We

show the subject reduction property, the absence of runtime errors for well-typed processes and the existence of principal types and of a type inference algorithm.

This part of our work is based on the observation that different type systems for process calculi [PS93, NS97, Aba97, Kob97] have similarities and could be integrated into one single system.

Types will also be represented by graphs and the main idea is that there exists a graph morphism from each process into its type. So if a property is preserved by inverse graph morphisms, (e.g. the absence of circles, necessary for deadlock prevention) and if this property is valid for the type, it is also valid for the process and—because of the subject reduction property—for all its successors. If a property is preserved by inverse graph morphisms it is, of course, not necessarily preserved by graph morphisms, so not every process satisfying a certain condition is actually well-typed.

In order to make the type system more powerful, each type graph can be labelled with lattice elements, which can be used to derive more information about processes. It is, however, necessary to define, how these lattice elements behave under morphisms. This is described by a type functor mapping graph morphisms onto join-morphisms in lattices. An extension of the type system will use monoids instead of lattices.

Since our formalism relies on graphs and morphisms, it is sensible to describe graph rewriting respectively graph construction by means of category theory, as opposed to more constructive descriptions. This will simplify the definition of the type system, the treatment of lattice respectively monoid elements and the following proofs.

We see the following advantages in describing both processes and types by graphs:

- since both have the same structure, the inference of properties of a process from its type is rather intuitive
- it is more convenient to add additional labels or structures (e.g. arrows between arbitrary nodes) to a type represented by a graph than to a type represented by a term
- recursive types, essential for typing processes, can be described by cycles in a graph (see also [RV97, Yos96])

1.3 Summary of this Work

We give abstracts for the chapters to come:

2 Methods of Hypergraph Construction We investigate various methods of hypergraph representation and construction, the first being the *set-based notation*. This is simply the representation of graphs by means of sets of nodes and edges. We will define the notions of *morphism*, of *isomorphism*, of *factorization* and of *graph context*.

For some proofs and applications, set-based notation turns out to be too constructive in nature. We will thus introduce the *categorical approach* (also called *algebraic approach*) which regards hypergraphs as the objects of a category. New hypergraphs are constructed using co-limits.

Another equivalent method is the representation of hypergraphs by *graph expressions*. The algebra of hypergraphs is described by sorts, function symbols and a set of equations. The operations on graphs used in this algebra turn out to be useful for the compact description of graphs.

In order to bridge the gap between graph-based and name-based description of processes, we will describe a way of presenting graphs based on node names. This notation is useful for the translation of process calculi (e.g. the π -calculus) into our calculus.

While all methods of describing graphs do already exist, we add extensions specific to our problems and give translations from one notations into the other.

3 Process Calculi: An Overview We give an overview over existing process calculi (such as CCS, the π -calculus, $\text{HO}\pi$, CHOCS, CHAM) and explain how they differ from or have inspired the graph-based calculus SPIDER.

4 The SPIDER Calculus: Syntax and Semantics We present the syntax and operational semantics of the SPIDER calculus. Every SPIDER expression is essentially a hierarchical hypergraph. The semantic rules are grouped into rules of structural congruence and reduction rules. The main reduction rules are the *replication* of a process and *message reception*.

SPIDER was designed as an asynchronous calculus containing two forms of mobility: *mobility of port addresses* and *mobility of entire processes*. That is we can send port addresses as well as processes as the content of a message, which leads to dynamic reconfigurations of the process graph.

5 Extended Example: Mail Delivery System This chapter is intended for demonstrating the features of SPIDER and for familiarizing the reader with the new notation.

In order to demonstrate the usefulness of mobility we model a mail delivery system with routers and client processes and show how to realize registering of new clients and how to enrich messages with routing information.

6 Bisimulation and Proof Techniques It is often desirable to regard processes modulo a bisimulation equivalence. This equivalence abstracts from the internal structure of a process and focuses on the interaction of a process with its environment.

We demonstrate how to define bisimulation equivalence in SPIDER and show how to adopt known proof techniques to SPIDER. Bisimulation is known to be undecidable, but there is a collection of methods, called bisimulations up-to, making proofs possible in some cases.

7 Comparison with other Formalisms We show that SPIDER is able to simulate well-known calculi, such as the λ -calculus or the π -calculus. In order to do so we first define what it means for a calculus to be simulated by another.

Furthermore we show two alternative versions of the SPIDER calculus, one based on the algebraic approach and the other on the name-based notation, the latter being an intermediate step in the encoding of the π -calculus into SPIDER.

8 Generating Type Systems This chapter is concerned with the type system for SPIDER, as described above. We present a new kind of type system based on the following idea: *types are graphs and if a process can be typed, then there exists a morphism from the graph representing the process, into its type.*

First, we introduce the theory of lattice-ordered commutative monoids, of which lattices are only a special case. Then we give a type system based on lattices, i.e. type graphs are labelled with lattice elements.

We extend this type system by using lattice-ordered commutative monoids as label sets, which enables us to actually count. This type system is more powerful in some ways but introduces additional complications. So, in order to keep everything manageable, we only give this type system for a restricted calculus without higher-order communication.

For both kinds of type systems we can prove the subject reduction property (types do not change during reduction), the existence of principal types and automatic type inference. Both type systems are generic, i.e. they can be instantiated in order to show specific properties of a process.

We end this chapter by comparing the two type systems with each other and with standard type systems for the π -calculus and show in which cases they coincide.

1.4 Dependency Graph of the Chapters and Sections

We now give a graph describing the dependency of the chapters and sections of this work (see figure 1.1). If there is a path from x to y , this means that section/chapter y relies on definitions, propositions or other important parts of section/chapter x .

1.5 Remarks on Notation

Now we introduce some notations which will be used throughout this work:

Strings: Let A be any alphabet. A^* denotes the set of all strings of elements of A . If $s \in A^*$ then $Set(s) \subseteq A$ denotes the set of all elements in s . $|s|$ denotes the length of s , while $s_1 \circ s_2$ denotes the concatenation of strings s_1 and s_2 . ε represents the empty string.

Equivalence Relations: Let R be an equivalence relation on the set X , i.e. R is reflexive, transitive and symmetric. $[x]_R$ with $x \in X$ denotes the equivalence class of x , i.e.

$$[x]_R := \{y \mid y \in X, x R y\}$$

The *quotient* X/R is the set of all equivalence classes.

Chapter 2

Methods of Hypergraph Construction

Hypergraphs form the foundation of the syntax of the SPIDER calculus. They are a generalization of directed graphs, where each edge is associated with an arbitrarily long string of nodes. Another feature of our hypergraph model is the existence of a special string of nodes, called “external nodes”. These external nodes are useful for assembling hypergraphs.

Unlike in string rewriting where there seems to be one standard approach to describe and modify strings, there are several equivalent concepts in graph rewriting. We will consider in more detail the following four approaches:

The Set-Based Approach: a hypergraph consists of edge and node sets and of mappings connecting edges with nodes.

The Categorical Approach: we will show how hypergraphs and hypergraph morphisms can be regarded as a category.

Graph Expressions: hypergraphs can also be represented by terms in an algebra. We give a signature and a set of equations.

The Name-Based Notation: we will relate hypergraphs to the name-based notation of structures used in most process algebras.

These concepts have different strengths and weaknesses and, depending on the problem we are working on, it is often appropriate to switch to another model or to use several approaches together. Examples will appear later in this work.

A very important concept in graph theory is that of isomorphism, i.e. we do not want to distinguish graphs having the same internal structure, but different internal names.

We will furthermore focus on methods of hypergraph construction, i.e. we will show how to concatenate hypergraphs. Having established what graph construction means, the definition of graph rewriting is obvious: in a hypergraph which is constructed of several subgraphs, one of them the left-hand side of a rewrite rule, we just replace the left-hand side by the right-hand side and assemble the new graph in the same way as the old one.

We then compare the methods of decomposing or factorizing hypergraphs in the four approaches and show that they are all equivalent. That is we can translate one method into another and obtain results which are the same up to isomorphism. Furthermore we will show that there are basic graphs, consisting of one edge only, which form the prime elements of a factorization, i.e. a factorization into basic graphs is unique up to isomorphism.

2.1 Set-Based Approach

This is the most obvious approach to describe graphs and graph construction (for a slightly different presentation see [Hab92]). Its advantages are its direct and constructive nature. However, proofs in this notation tend to get rather clumsy quite soon. Furthermore we do not want to distinguish graphs of the same structure, i.e. isomorphic graphs. In this approach we have to explicitly construct equivalence classes of graphs, whereas in other approaches the concept of isomorphism arises more naturally.

2.1.1 Basic Definitions

A hypergraph consists of nodes and edges just as directed graphs. The difference to ordinary graphs consists in the method of connecting edges with nodes. Every edge possesses a string of nodes of arbitrary length. If we only use strings of length two, we obtain directed graphs.

Another feature of hypergraphs is a string of external nodes, which are used to attach hypergraphs to one another. Their purpose will be clarified in the rest of this chapter.

Definition 2.1.1 (Hypergraphs) 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 $G = (V, E, s, z, l)$ whose components have the following meaning:

- V is a set of *nodes*
- E is a set of *edges* or *hyperedges* which is disjoint from V
- $s : E \rightarrow V^*$ maps each hyperedge to a string of *source nodes*
- $z : E \rightarrow Z$ assigns a sort to every hyperedge
- $l : E \rightarrow L$ labels the hyperedges

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

The class of all simple hypergraphs with labels L and sorts Z is called $\mathcal{G}(Z, L)$, while the corresponding class of hypergraphs is denoted by $\mathcal{H}(Z, L)$.

□

Similar definitions for hypergraphs are given in [BC87, Hab92]. In [Hab92] the author distinguishes source and target nodes of a hyperedges. Furthermore there are two strings of external nodes: the begin-nodes and the end-nodes. We follow the argumentation in [BC87] which says that this notation can normally be converted into the one in definition 2.1.1 by concatenating the source and target strings as well as the begin and end strings.

One reason for using the notation in [Hab92] would be to demand that no end node is used as a source node and that no begin node is used as a target node. This restriction is useful in some cases but has the grave disadvantage that not every subgraph of a restricted hypergraph satisfies this condition.

Edge sorts are important to distinguish edges representing processes, messages or variables. The label, on the other hand, contains the process description, the message content or the name variable.

Notes and Abbreviations:

Components of a Hypergraph: The components of a hypergraph H are denoted by $V_H, E_H, s_H, z_H, l_H, \chi_H$. The set of all external nodes is called $EXT_H := Set(\chi_H)$.

$E_H^z := \{e \in E_H \mid z_H(e) = z\}$ denotes the set of all edges of sort z .

Source Nodes: Let $e \in E_H$. We define $SOURCE(e) := Set(s_H(e))$

Cardinality: The cardinality of a hyperedge $e \in E_H$ is defined by: $card(e) := |s_H(e)|$. The cardinality of a hypergraph H is $card(H) := |\chi_H|$.

Internal Nodes: All the nodes in $V_H \setminus EXT_H$ are called *internal* and an internal node $v \in V_H$ that is not in the range of s_H is called *isolated*.

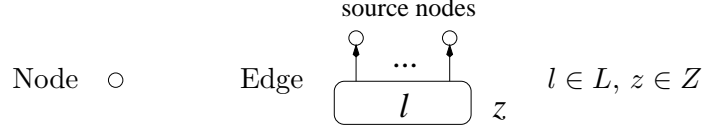
Redefining External Nodes: Let $H := G[\chi]$ be a hypergraph. We define $H[\chi'] := G[\chi']$ if $\chi' \in V_G^*$.

Removing Edges: Let $E' \subseteq E_G$.

We define $G|_{E'} := (V_G, E', s_G|_{E'}, z_G|_{E'}, l_G|_{E'})$. And if $H = G[\chi]$ we define $H|_{E'} := (G|_{E'})[\chi]$.

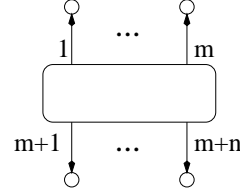
As mentioned in the introduction, graphs have the great advantage of possessing an appealing visual representation, which makes it easier to grasp some concepts.

We will use the following graphical notation for hypergraph nodes and edges:

Graphical Notation:

External nodes will be labelled $(1), (2), (3), \dots$

Sometimes graphs are easier to draw if we attach nodes above *and* below edges. They are ordered from left to right, nodes above precede nodes below:



In the rest of this work we will often map hypergraphs onto other hypergraphs, while at the same time preserving their structure. Mappings preserving the structure of an object are a widespread concept in mathematics and are ordinarily called *morphisms* or *homomorphisms*.

Definition 2.1.2 (Hypergraph Morphism) Let $R \subseteq L \times L$ be a reflexive and transitive relation on the set of labels.

Let $G, G' \in \mathcal{G}(Z, L)$ be two simple hypergraphs. A *hypergraph morphism* $\phi: G \rightarrow G'$ (wrt. R) consists of two mappings $\phi_E: E_G \rightarrow E_{G'}$, $\phi_V: V_G \rightarrow V_{G'}$ satisfying for all $e \in E_G$:

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

A morphism ϕ is called a

$$\begin{array}{lll} \text{monomorphism} & \iff & \phi_V, \phi_E \text{ are injective} \\ \text{epimorphism} & \iff & \phi_V, \phi_E \text{ are surjective} \\ \text{isomorphism} & \iff & \phi_V, \phi_E \text{ are bijective} \end{array}$$

We write $\phi: G[\chi] \rightarrow G'[\chi']$ if $\phi_V(\chi) = \chi'$. In this case ϕ is called a *strong morphism*. We define $\text{card}(\phi) := |\chi|$. \square

From now on we omit the subscripts in ϕ_E and ϕ_V if they can be derived from the context. If we do not indicate otherwise, the relation R of a morphism is always the identity.

Notes: Let $\phi: G \rightarrow G'$ and $\psi: G' \rightarrow G''$ be two morphisms wrt. R . Then the morphism $\psi \circ \phi$ consisting of $\psi_V \circ \phi_V$ and $\psi_E \circ \phi_E$ (where \circ is the composition operator on mappings) is also a morphism wrt. R . Note that this is only true because of the transitivity of R .

Furthermore the reflexivity of R yields an identity morphism $\text{id}_G: G \rightarrow G$ for every simple hypergraph G , consisting of id_{V_G} and id_{E_G} .

There are several notions of isomorphism, simple hypergraphs as well as multi-pointed hypergraphs can be isomorphic. We also define isomorphism of families of morphisms.

Definition 2.1.3 (Isomorphism) Let G, G' be simple hypergraphs. G is *isomorphic* to G' ($G \cong_R G'$) if there is an isomorphism $\phi : G \rightarrow G'$ (wrt. R). We will write $G \cong G'$ if R is the identity on L .

Two hypergraphs $G[\chi], G'[\chi']$ are called *isomorphic*, if there exists a strong isomorphism $\phi : G[\chi] \rightarrow G'[\chi']$ (wrt. R).

Let $(\phi_i)_{i \in \{1, \dots, n\}}, (\psi_i)_{i \in \{1, \dots, n\}}$ be two families of morphisms where $\phi_i : G_i \rightarrow G$ and $\psi' : G'_i \rightarrow G$. They are called *isomorphic* wrt. R

$$(\phi_i)_{i \in \{1, \dots, n\}} \cong_R (\psi_i)_{i \in \{1, \dots, n\}}$$

if there exists an isomorphism $\phi : G \rightarrow G'$ (wrt. R) such that $\forall i \in \{1, \dots, n\} : \phi \circ \phi_i = \psi_i$. \square

It is our aim to work with isomorphism classes of graphs, i.e. with *abstract graphs* rather than with *concrete graphs*, as far as possible.

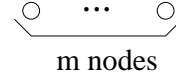
Some Special Graphs:

We call a hypergraph *discrete*, if its edge set is empty.

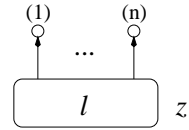
\mathbf{m} with $m \in \mathbb{N}$ denotes a discrete graph of cardinality m with m nodes where every node is external.



$\overline{\mathbf{m}}$ denotes a discrete graph with m nodes and with an empty string of external nodes.



$H := z_n(l)$ is the hypergraph with exactly one edge e , where $l_H(e) = l$, $z_H(e) = z$, $s_H(e) = \chi_H$, $|\chi_H| = n$, $V_H = EXT_H$. it is also called *basic graph*.



The following type of morphism will be needed later in this work, especially in chapter 8.

Definition 2.1.4 (Nice Graph Morphisms)

A strong morphism $\phi : H \rightarrow H'$ is called *nice* if $\phi(V_H \setminus EXT_H) \subseteq V_{H'} \setminus EXT_{H'}$ and $\phi|_{EXT_H}$ is injective. \square

We now concentrate on the main purpose of this chapter and show how to compose and decompose hypergraphs. We start by defining an *embedding* (see also [BC87, Ehr79]) which is a special kind of morphism. The main idea behind an embedding is the following: if there is an embedding $\eta : H \rightarrow H'$ then H can be considered as a “part” or *factor* of H' . Note that only the images of

external nodes of H' are allowed to be in contact with the rest of the graph. η is injective on any other part of H' except the external nodes.

Definition 2.1.5 (Embedding) Let $G[\chi], G'[\chi'] \in \mathcal{H}(Z, L)$ be two hypergraphs. A hypergraph morphism $\eta : G \rightarrow G'$ is called an *embedding* of $G[\chi]$ into $G'[\chi']$ (written $\eta : G[\chi] \hookrightarrow G'[\chi']$) if

- η is edge-injective, i.e. η_E is injective
- If $v, v' \in V_G$ with $v \neq v'$, then $\eta(v) = \eta(v')$ implies $v, v' \in \text{Set}(\chi')$
- Let $v \in V_G$ and let $\eta(v) \in \text{SOURCE}(e)$ with $e \in E_{G'} \setminus \eta(E_G)$ or $\eta(v) \in \text{Set}(\chi')$. This implies $v \in \text{Set}(\chi)$

If there is an embedding $\eta : H \rightarrow H'$ we call H a *factor* of H' . \square

It turns out that the composition $\eta' \circ \eta$ of two embeddings $\eta : H \hookrightarrow H'$, $\eta' : H' \hookrightarrow H''$ is again an embedding.

A hypergraph can consist of several factors which are expected to form a kind of partition of the graph. These factors only overlap in the images of their external nodes.

Definition 2.1.6 (Factorization) Let $\eta_i : G_i \rightarrow G$, $i \in \{1, \dots, n\}$ be morphisms and let $\chi \in V_G^*$, $\chi_i \in V_{G_i}^*$. Furthermore let $v_i \in V_{G_i}$, $v_j \in V_{G_j}$, $e_i \in E_{G_i}$, $e_j \in E_{G_j}$ and

$$\eta_i(e_i) = \eta_j(e_j) \Rightarrow i = j, e_i = e_j \quad (2.1)$$

$$\begin{aligned} \eta_i(v_i) = \eta_j(v_j) &\Rightarrow (i = j, v_i = v_j \\ &\vee v_i \in \text{Set}(\chi_i), v_j \in \text{Set}(\chi_j)) \end{aligned} \quad (2.2)$$

$$\eta_i(v_i) \in \text{Set}(\chi) \Rightarrow v_i \in \text{Set}(\chi_i) \quad (2.3)$$

$$E_G = \bigcup_{i=1}^n E_{G_i} \quad (2.4)$$

It is straightforward to show that $\eta_i : G_i[\chi_i] \hookrightarrow G[\chi]$ are embeddings. $(\eta_i)_{i \in \{1, \dots, n\}}$ is called a *factorization*. \square

Any set of embeddings, whose ranges overlap only in the images of the external nodes, can be extended to a factorization.

Proposition 2.1.7 (Extending Embeddings to a Factorization) Let $\eta_i : H_i \hookrightarrow H$, $i \in \{1, \dots, n\}$ be embeddings such that the sets $\eta_i(E_{H_i})$ are pairwise disjoint and $\eta_i(V_{H_i} \setminus \text{EXT}_{H_i}) \cap \bigcup_{j=1, j \neq i}^n V_{H_j} = \emptyset$ for every $i \in \{1, \dots, n\}$.

Then there exists an embedding $\eta_{n+1} : H_{n+1} \hookrightarrow H$ such that $\eta_i : H_i \rightarrow H$, $i \in \{1, \dots, n+1\}$ is a factorization.

Proof: We define a hypergraph H_{n+1} with:

$$\begin{aligned} E_{H_{n+1}} &:= E_H \setminus \bigcup_{i=1}^n \eta_i(E_{H_i}) \\ V_{H_{n+1}} &:= \bigcup_{e \in E_{H_{n+1}}} \text{SOURCE}(e) \\ s_{H_{n+1}} &:= s_H|_{E_{H_{n+1}}} \\ l_{H_{n+1}} &:= l_H|_{E_{H_{n+1}}} \end{aligned}$$

And choose any $\chi_{H_{n+1}}$ such that $\text{Set}(\chi_{H_{n+1}}) = V_{H_{n+1}}$. Furthermore let $\eta_{n+1}(e) := e$ if $e \in E_{H_{n+1}}$ and $\eta_{n+1}(v) := v$ if $v \in V_{H_{n+1}}$.

It is straightforward to check that $(\eta_i)_{i \in \{1, \dots, n+1\}}$ is a factorization.

Note that η_{n+1} is, by no means, unique. \square

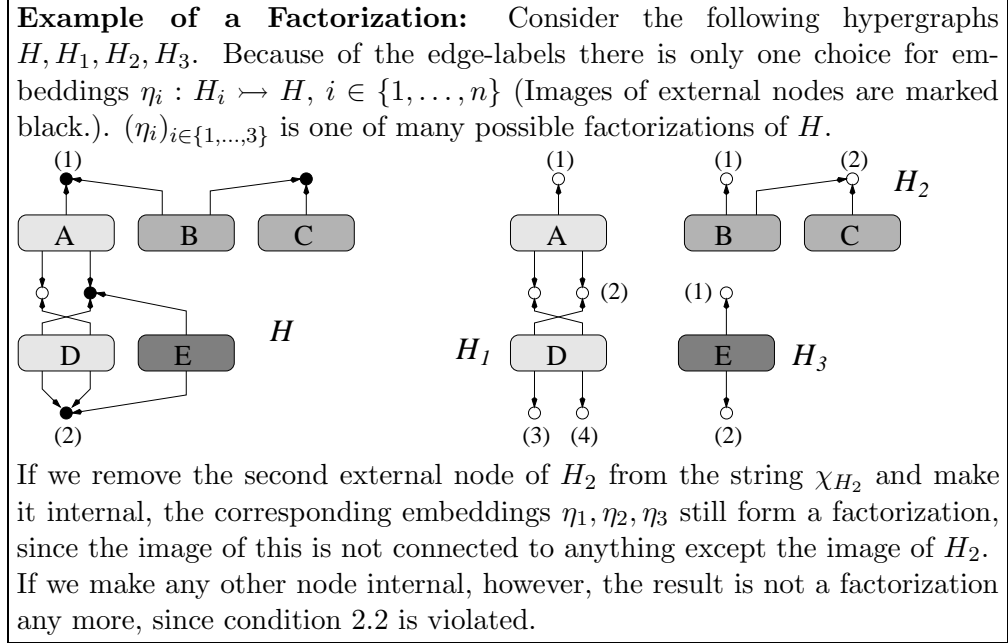


Figure 2.1: Example of a Factorization

Note: For the sake of simplicity, we will assume in all the examples in this sections, that all edges have one single, common edge sort. From now on, the edge sort is represented by the form of the line surrounding the edge (e.g. dashed or dotted). We see in the next section that edges with edge sort *var* (edges representing variables) are surrounded by a line with long dashes.

It is quite obvious that the factorization of a hypergraph is by no means unique. We will show later in proposition 2.2.20 that there are basic hypergraphs corresponding to primes, and that a factorization into basic hypergraphs is actually unique.

2.1.2 Quotient Graphs

One way to construct a new graph (a *quotient graph*) out of other graphs is to define an equivalence relation on the nodes and hyperedges of the graphs and to collapse them according to the equivalence. That is, we join the graphs and merge all edges and nodes related by the equivalence.

Our definitions are slightly complicated by the fact that the node and edge sets of the hypergraphs need not be disjoint. Therefore we have to keep track of the origin of each node or hyperedge and pair them with a natural number, indicating the original graph.

We could omit this if we would demand that all node and hyperedge sets are disjoint, an assumption we do not want to rely on.

As it turns out, arbitrary equivalences do not always yield well-defined results. Therefore we will restrict our equivalences:

Definition 2.1.8 (Consistent Equivalence Relations) Let G_0, \dots, G_n be simple hypergraphs, let \approx_V be an equivalence on $\bigcup_{i \in \{1, \dots, n\}} (V_{G_i}, i)$ and let \approx_E be an equivalence on $\bigcup_{i \in \{1, \dots, n\}} (E_{G_i}, i)$.

\approx_V, \approx_E are called *consistent* iff $\forall e_i \in E_{G_i}, e_j \in E_{G_j}, i, j \in \{0, \dots, n\}$

$$(e_i, i) \approx_E (e_j, j) \Rightarrow \text{card}(e_i) = \text{card}(e_j), z_{G_i}(e_i) = z_{G_j}(e_j), \\ l_{G_i}(e_i) = l_{G_j}(e_j) \text{ and } (s_{G_i}(e_i), i) \approx_V (s_{G_j}(e_j), j)$$

where $(s, i) \approx (s', j) \iff |s| = |s'| \wedge \forall k \in \{1, \dots, |s|\} : (s_k, i) \approx (s'_k, j) \quad \square$

Note: We will often write \approx for either \approx_E or \approx_V .

Definition 2.1.9 (Quotient Graph) Let G_0, \dots, G_n be simple hypergraphs and let \approx_V, \approx_E be consistent equivalences as defined above. We will now define the *quotient graph* $G := G_0 \dots G_n / \approx$.

$$G := ((\bigcup_{i=0}^n (V_{G_i}, i)) / \approx, (\bigcup_{i=0}^n (E_{G_i}, i)) / \approx, s_G, z_G, l_G)$$

with

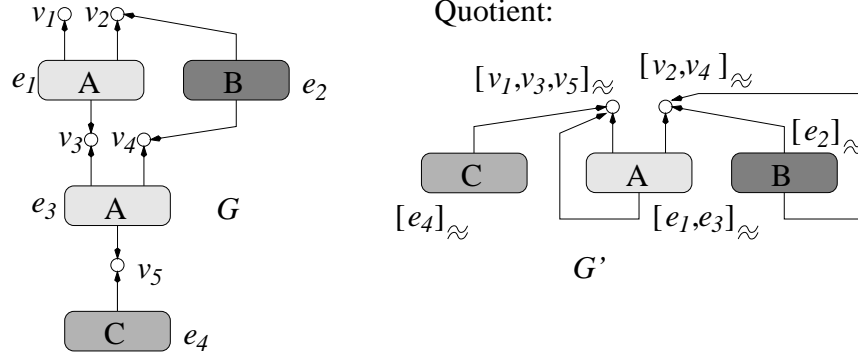
$$s_G([(e_i, i)]_{\approx_E}) := [([s_{G_i}(e_i)]_1, i)]_{\approx_V} \dots [([s_{G_i}(e_i)]_m, i)]_{\approx_V} \text{ if } \text{card}(e_i) = m \\ z_G([(e_i, i)]_{\approx_E}) := z_{G_i}(e_i) \\ l_G([(e_i, i)]_{\approx_E}) := l_{G_i}(e_i)$$

\square

Since \approx_V, \approx_E are consistent, it is ensured that G is well-defined.

We can define *canonical projection morphisms* $p_i : G_i \rightarrow G$ with $(p_i)_V(v) := [(v, i)]_{\approx_V}, (p_i)_E(e) := [(e, i)]_{\approx_E}$.

Example of a Quotient Graph: In the following example we will, for once, not only label the hyperedges, but also give names to the nodes and hyperedges of the graph G .



For G the equivalence $e_1 \approx e_3$, $v_1 \approx v_3 \approx v_5$, $v_2 \approx v_4$ is consistent. Its quotient is formed by merging the equivalent components.

Note: Any surjective morphism $\phi : G \rightarrow G'$ can be characterized—up to isomorphism—by an equivalence \approx on G such that $G' \cong G/\approx$ and ϕ is the projection of G into G' .

2.1.3 Hypergraph Construction with Context Graphs

The process of replacing a hyperedge with a hypergraph of the same cardinality is quite intuitive, but somewhat tedious to define.

There are several possible definition, we use the concept of the quotient graph to define substitution of variables.

Definition 2.1.10 (Substitution and Contexts)

We assume that the set L of edge labels contains an element var . Let $X = \{x_1, \dots, x_n\}$ be a set of variables where each variable $x \in X$ has sort $sort(x) \in \mathbb{N}$.

A *context graph* $C\langle x_1, \dots, x_n \rangle = G[\chi]$ is a hypergraph where, for every $i \in \{1, \dots, n\}$ there is exactly one edge e_i labelled x_i and of edge sort var and where $card(e_i) = sort(x_i)$.

Let

$$H_1 = G_1[\chi_1], \dots, H_n = G_n[\chi_n] \in \mathcal{H}(L \cup X)$$

be hypergraphs with $sort(x_i) = |\chi_i|$.

Let \approx be the smallest equivalence such that for all $i \in \{1, \dots, n\}$

$$(s_G(e_i), 0) \approx (\chi_i, i)$$

We define

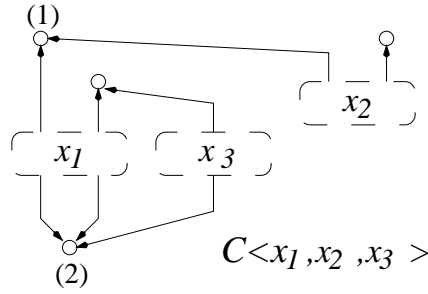
$$C\langle H_1, \dots, H_n \rangle := ((G \setminus \{e_1, \dots, e_n\})G_1 \dots G_n / \approx)[\chi']$$

where $\chi' := p_0(\chi)$ if p_0 is the projection of $G \setminus \{e_1, \dots, e_n\}$ into the quotient graph.

$C\langle x_1, \dots, x_n \rangle$ (or C for short) is called an n -ary context. It is called *discrete* if it only contains edge labels x_1, \dots, x_n . \square

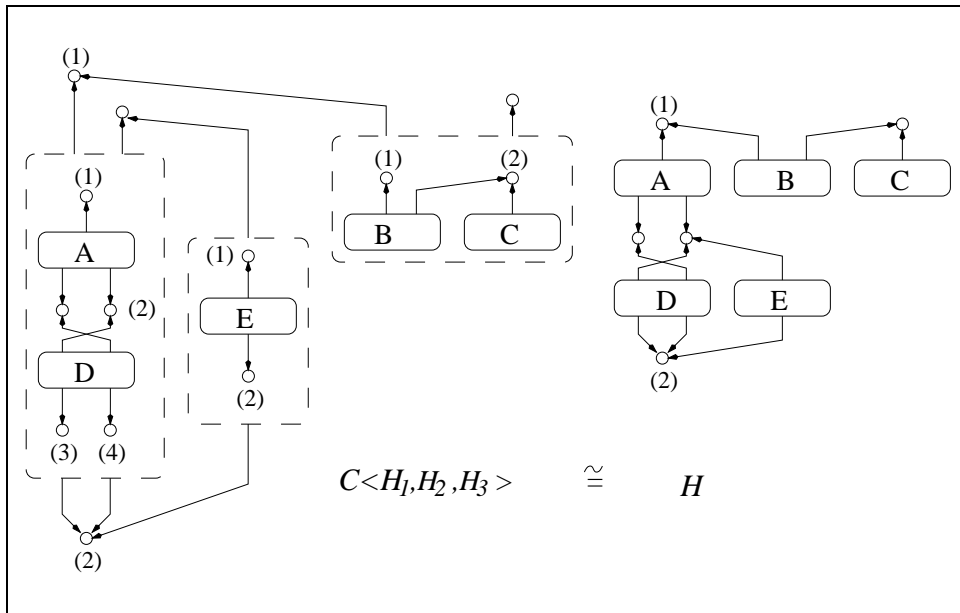
What we are, in fact, doing in the definition above is the following: we remove all edges in G labelled x_i and add an isomorphic copy of G_i for every edge. Afterwards we connect the nodes in χ_i with the source nodes of the deleted edge.

Example Context: For the following context C it holds that $H \cong C\langle H_1, H_2, H_3 \rangle$ (see figure 2.1).



Lines with long dashes indicate that these edges have edge sort *var*.

When we draw a graph of the form $C\langle H_1, \dots, H_n \rangle$ it might be desirable to indicate the structure of the original context C . In order to do so we draw the context C with all its edges and put the graphs H_1, \dots, H_n into the edges originally labelled x_1, \dots, x_n .



Note that this is purely a notational convention and is not identical with hierarchical hypergraphs introduced in chapter 4.

Note: Let $C\langle x_1, \dots, x_n \rangle$ be a context graph with $m_i := \text{sort}(x_i)$. Then

$$C\langle x_1, \dots, x_n \rangle \cong C\langle \text{var}_{m_1}(x_1), \dots, \text{var}_{m_n}(x_n) \rangle$$

2.2 Categorical Approach

The main idea behind category theory is to define objects with respect to their relations with other objects, and not by their internal structure, as we have done in the previous section.

Since this can be done not only in graph theory but also in many other fields of mathematics (e.g. algebra, lattice theory), category theory is a meta-theory capturing the properties these different fields have in common.

Compared to the set-theoretic approach the categorical approach to graph representation leads to non-constructive definitions. Its application lies mainly in proofs since it makes assertions about the existence of morphisms with certain properties. In this way it relieves us from the burden to construct every morphism by hand and to show its properties “manually”.

The treatment of graphs as objects of a category and the double-pushout approach for graph rewriting are introduced in [Ehr79].

We will only use a few concepts from category theory which we review in the following section (see also [Cro93]).

2.2.1 Introduction to Category Theory

The first step is to define what a category actually is. It turns out that it is a very general mathematical structure.

Definition 2.2.1 (Category) A *category* \mathcal{C} consists of the following components:

- A class $\text{obj}(\mathcal{C})$ of entities called *objects*.
- A class $\text{mor}(\mathcal{C})$ of entities called *morphisms*.
- Two operations src and tar assigning a *source* object $\text{src}(\phi)$ and a *target* object $\text{tar}(\phi)$ to every morphism $\phi \in \text{mor}(\mathcal{C})$.
(Notation: $\phi : \text{src}(\phi) \rightarrow \text{tar}(\phi)$)
- For every object A there is a morphism $\text{id}_A : A \rightarrow A$, the *identity morphism*.
- Two morphisms ϕ, ψ are *composable* if $\text{src}(\phi) = \text{tar}(\psi)$. The composition is denoted by $\phi \circ \psi : \text{src}(\psi) \rightarrow \text{tar}(\phi)$.

The morphisms satisfy the following properties:

$$\text{id}_{\text{tar}(\phi)} \circ \phi = \phi \tag{2.5}$$

$$\phi \circ \text{id}_{\text{src}(\phi)} = \phi \tag{2.6}$$

Furthermore composition is associative, i.e. given morphisms ϕ, ψ, ζ with $\text{tar}(\phi) = \text{src}(\psi)$, $\text{tar}(\psi) = \text{src}(\zeta)$

$$(\zeta \circ \psi) \circ \phi = \zeta \circ (\psi \circ \phi) \quad (2.7)$$

□

Note that while often morphisms are expected to be mappings preserving some kind of structure, in this definition we do not demand any such thing. We do not even require them to be functions at all. In practice, however, and especially in our case, the morphisms of a category very often satisfy this property.

Examples: Typical examples for categories are the category *Set*, which contains sets as objects and functions mapping sets onto sets as morphisms, and the category *Rel*, which contains relations instead of functions. In both cases we use the usual composition operator.

It is not difficult to check that also the class $\mathcal{G}(Z, L)$ of simple hypergraphs forms a category with the hypergraph morphisms. $\mathcal{H}(Z, L)$ is a category with the strong morphisms, the embeddings or the nice morphisms.

Definition 2.2.2 (Isomorphism) Let \mathcal{C} be a category and let $\phi \in \text{mor}(\mathcal{C})$ with $\phi : A \rightarrow B$. ϕ is called *isomorphism* if there exists a morphism $\psi : B \rightarrow A$ such that $\psi \circ \phi = \text{id}_A$, $\phi \circ \psi = \text{id}_B$.

In this case A and B are called *isomorphic*. □

If we regard the category of strong hypergraph morphisms, this concept of isomorphism in category theory coincides with the isomorphism of graphs from definition 2.1.2.

If we step up one level we may decide to regard categories as the objects of a very large category. In this case we need some kind of mapping between categories, taking the place of morphisms. This mapping is called *functor*:

Definition 2.2.3 (Functor) A *functor* $F : \mathcal{C} \rightarrow \mathcal{D}$ between categories \mathcal{C} and \mathcal{D} is specified by:

- An operation mapping objects $A \in \text{obj}(\mathcal{C})$ onto objects $F(A) \in \text{obj}(\mathcal{D})$.
- An operation mapping morphisms $\phi : A \rightarrow B$ in \mathcal{C} onto morphisms $F(\phi) : F(A) \rightarrow F(B)$ in \mathcal{D} . (We will also denote $F(\phi)$ by F_ϕ).

F satisfies the following properties:

$$\begin{aligned} F(\text{id}_A) &= \text{id}_{F(A)} \\ F(\phi \circ \psi) &= F(\phi) \circ F(\psi) \quad \text{if } \phi \text{ and } \psi \text{ are composable} \end{aligned}$$

□

We will now start to define the notion of a co-limit, which is a method of defining new objects out of known objects and morphisms. The new object is determined only by its relation with other objects.

Every co-limit construction starts with its input: a *diagram*.

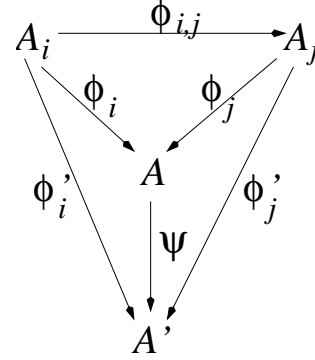
Definition 2.2.4 (Diagram) Let \mathcal{C} be a category. A *diagram* D in \mathcal{C} consists of objects $A_1, \dots, A_n \in \text{obj}(\mathcal{C})$ and of sets of morphisms $M_{ij} \subseteq \text{mor}(\mathcal{C})$, $i, j \in \{1, \dots, n\}$ where M_{ij} contains morphisms of the form $A_i \rightarrow A_j$ in \mathcal{C} .

Let D, D' be two diagrams with objects A_1, \dots, A_n respectively A'_1, \dots, A'_n and morphism sets M_{ij} respectively M'_{ij} , where each morphism $\phi_{ij} \in M_{ij}$ is bijectively mapped onto a corresponding morphism ϕ'_{ij} . D, D' are called *isomorphic* if there are isomorphisms $\psi_i : A'_i \rightarrow A_i$ such that $\psi_j \circ \phi'_{ij} = \phi_{ij} \circ \psi_i$ for every pair ϕ_{ij}, ϕ'_{ij} . \square

Given a diagram consisting of objects A_1, \dots, A_n and morphism sets M_{ij} we attempt to construct a new object A and new morphisms $\phi_i : A_i \rightarrow A$ in the following way:

Definition 2.2.5 (Co-Limit) Let $A_1, \dots, A_n \in \text{obj}(\mathcal{C})$, together with morphism sets $M_{ij} \subseteq \text{mor}(\mathcal{C})$ (where M_{ij} contains morphisms of the form $A_i \rightarrow A_j$) be a diagram D in the category \mathcal{C} . The *co-limit* of the diagram consists of an object A and of morphisms $\phi_i : A_i \rightarrow A$, $i \in \{1, \dots, n\}$ satisfying the following conditions:

- For every $\phi_{ij} \in M_{ij}$: $\phi_j \circ \phi_{ij} = \phi_i$
- For any object A' and morphisms $\phi'_i : A_i \rightarrow A'$ satisfying $\phi'_j \circ \phi_{ij} = \phi'_i$ for every $\phi_{ij} \in M_{ij}$ there is a unique morphism $\psi : A \rightarrow A'$ such that $\psi \circ \phi_i = \phi'_i$ for every $i \in \{1, \dots, n\}$.



We define $\lim D := (\phi_1, \dots, \phi_n)$. \square

The idea behind the second condition is to take the object A that is, somehow, the “smallest” or initial object satisfying the first condition.

If the diagram consists of two morphisms $\phi_1 : B \rightarrow B_1$, $\phi_2 : B \rightarrow B_2$ then its co-limit is called a *pushout*.

It is not always obvious if a co-limit actually exists. But if it exists it is unique up to isomorphism. We can even state that isomorphic diagrams have isomorphic co-limits.

Proposition 2.2.6 (Isomorphism of Co-Limits) *Let D, D' be isomorphic diagrams with objects A_1, \dots, A_n respectively A'_1, \dots, A'_n and isomorphisms $\psi_i : A'_i \rightarrow A_i$. Let $\phi_i : A_i \rightarrow A$ respectively $\phi'_i : A'_i \rightarrow A'$ be their co-limits.*

Then there exists an isomorphism $\psi : A' \rightarrow A$ such that $\phi_i \circ \psi_i = \psi \circ \phi'_i$.

Proof: Since A' is the co-limit of D and there are morphism $\phi_i \circ \psi_i : A'_i \rightarrow A$ with $(\phi_j \circ \psi_j) \circ \phi'_{ij} = \phi_j \circ \phi_{ij} \circ \psi_i = \phi_i \circ \psi_i$.

The properties of a co-limit imply that there exists a morphism $\psi : A' \rightarrow A$ with $\psi \circ \phi'_i = \phi_i \circ \psi_i$.

In the same way we conclude that there exists a morphism $\psi' : A \rightarrow A'$ with $\psi' \circ \phi_i = \phi'_i \circ (\psi_i)^{-1}$ which is equivalent to $\psi' \circ \phi_i \circ \psi_i = \phi'_i$.

By substituting ϕ'_i in the equation $\psi \circ \phi'_i = \phi_i \circ \psi_i$ by $\psi' \circ \phi_i \circ \psi_i$ we obtain

$$\psi \circ \psi' \circ \phi_i \circ \psi_i = \phi_i \circ \psi_i$$

Since ψ_i is an isomorphism this is equivalent to $\psi \circ \psi' \circ \phi_i = \phi_i$. But according to the properties of a co-limit there is only one morphism with such a property, i.e. $\psi \circ \psi' = id_A$.

Similarly we can show that $\psi' \circ \psi = id_{A'}$. Therefore ψ is an isomorphism.

□

We will now give an alternate definition of the quotient graph from section 2.1.2 in terms of category theory.

Proposition 2.2.7 (Quotient Graph) *Let G_0, \dots, G_n be hypergraphs and let \approx_V, \approx_E be equivalence relations as defined in definition 2.1.9.*

Then $G := G_0 \dots G_n / \approx$ is defined up to isomorphism by the following property:

- *There are morphisms $p_i : G_i \rightarrow G$ such that $(e_i, i) \approx_E (e_j, j)$ implies $p_i(e_i) = p_j(e_j)$ and $(v_i, i) \approx_V (v_j, j)$ implies $p_i(v_i) = p_j(v_j)$*
- *For any hypergraph G' with morphisms $p'_i : G_i \rightarrow G'$ satisfying*

$$\begin{aligned} (e_i, i) \approx_E (e_j, j) &\Rightarrow p'_i(e_i) = p'_j(e_j) \\ (v_i, i) \approx_V (v_j, j) &\Rightarrow p'_i(v_i) = p'_j(v_j) \end{aligned}$$

it follows that G is defined and that there is a unique morphism $\phi : G \rightarrow G'$ such that $\phi \circ p_i = p'_i$.

Proof: The existence of projection morphisms p_i is clear (see note after definition 2.1.9).

Let G' be a hypergraph with morphisms $p'_i : G_i \rightarrow G'$ satisfying the conditions above. For $e \in E_G, v \in V_G$ we define $\phi(e) := p'_i(e_i)$ if $e = p_i(e_i)$ and $\phi(v) := p'_i(v_i)$ if $v = p_i(v_i)$.

We will now show that ϕ is well-defined: by definition of the quotient graph and the projections $p_i(e_i) = p_j(e_j)$ is equivalent to $(e_i, i) \approx_E (e_j, j)$. This implies $p'_i(e_i) = p'_j(e_j)$. In the case of nodes the proof is analogous.

With the definition of ϕ it follows immediately that $\phi \circ p_i = p'_i$.

It is left to show that ϕ is unique. Let $\phi : G \rightarrow G'$ be another morphism with these properties. Let $e \in E_G$. e is an entire equivalence class, i.e. $e = [(e_i, i)]_{\approx_E} = p_i(e_i)$ for some i . Therefore $\phi'(e) = \phi'(p_i(e_i)) = p'_i(e_i) = \phi(e)$. For any $v \in V_G$ $\phi'(v) = \phi(v)$ follows in an analogous way. \square

We can now immediately state the following corollary:

Corollary 2.2.8 *Let G_0, \dots, G_n be a hypergraph and let \approx, \approx' be two consistent equivalences on G_0, \dots, G_n such that $\approx' \subseteq \approx$. Let $p_i : G_i \rightarrow G/\approx$ and $p'_i : G_i \rightarrow G/\approx'$ be the projection morphisms.*

Then there exists a morphism $\phi : G/\approx' \rightarrow G/\approx$ such that $\phi \circ p'_i = p_i$ for $i \in \{1, \dots, n\}$.

2.2.2 Graph Construction

We now show how category theory can help to construct new graphs and how factorization can be expressed in this setting.

Our idea is to give a construction plan (similar to the context defined in section 2.1), only we want the construction plan to be a diagram in the category of hypergraph morphisms. We now introduce a construction mechanism even more general than the context from definition 2.1.10. Later, we will demonstrate the connection between these two methods.

Definition 2.2.9 (Graph Construction) Let \mathcal{C} be the category of hypergraph morphisms. Let $\eta_i : G_i[\chi_i] \rightarrow G[\chi]$, $i \in \{1, \dots, n\}$ be embeddings and let $\phi_i : G_i[\chi_i] \rightarrow G'_i[\chi'_i]$, $i \in \{1, \dots, n\}$ be strong morphisms.

Let D be a diagram consisting of the morphisms $\eta_i : G_i \rightarrow G$ and $\phi_i : G_i \rightarrow G'_i$.

Now let $\phi : G \rightarrow G'$, $\eta'_i : G'_i \rightarrow G'$ (and $\phi \circ \eta_i = \eta'_i \circ \phi_i : G_i \rightarrow G'$) with $i \in \{1, \dots, n\}$ be the co-limit of D , if it exists. We define

$$\begin{aligned} \bigotimes_{i=1}^n (\phi_i, \eta_i) &:= G'[\phi(\chi)] & G_i[\chi_i] &\xrightarrow{\eta_i} G[\chi] \\ (\phi_1, \eta_1) \otimes \dots \otimes (\phi_n, \eta_n) &:= G'[\phi(\chi)] & \phi_i \downarrow & \quad \downarrow \phi \\ \lim_{i=1}^n (\phi_i, \eta_i) &:= (\phi, \eta'_1, \dots, \eta'_n) & G'_i[\chi'_i] &\xrightarrow{\eta'_i} G'[\phi(\chi)] \end{aligned}$$

If the $G_i[\chi_i]$ have the form \mathbf{m}_i there is only one (canonical) choice for the strong morphisms ϕ_i and we define:

$$\begin{aligned} \bigotimes_{i=1}^n (G'_i[\chi'_i], \eta_i) &:= \bigotimes_{i=1}^n (\phi_i, \eta_i) \\ (G'_1[\chi'_1], \eta_1) \otimes \dots \otimes (G'_n[\chi'_n], \eta_n) &:= (\phi_1, \eta_1) \otimes \dots \otimes (\phi_n, \eta_n) \\ \lim_{i=1}^n (G'_i[\chi'_i], \eta_i) &:= (\phi, \eta'_1, \dots, \eta'_n) \end{aligned}$$

\square

Since co-limits do not always exist, it is not, at first sight, clear if the graph construction mechanism defined above, will always yield a result. Thus we will now show, how the co-limit can be constructed “manually” and that it always exists. Furthermore since we are interested in factorizations, decomposing an existing graph, we will show that factorization are preserved by the co-limit.

Proposition 2.2.10 (Graph Construction and Quotient Graphs)

Let $\phi_i : G_i[\chi_i] \rightarrow G'_i[\chi'_i]$ be strong morphisms and let $\eta_i : G_i[\chi_i] \rightarrow G[\chi]$ be embeddings where $i \in \{1, \dots, n\}$.

Let \approx be the smallest equivalence on G, G'_1, \dots, G'_n satisfying:

$$\begin{aligned} \forall v_i \in V_{G_i} : (\phi_i(v_i), i) &\approx (\eta_i(v_i), 0) \\ \forall e_i \in E_{G_i} : (\phi_i(e_i), i) &\approx (\eta_i(e_i), 0) \end{aligned}$$

Let $G' := GG'_1 \dots G'_n / \approx$ and let $\phi : G \rightarrow G'$, $\eta'_i : G'_i \rightarrow G'$ be the corresponding projections. Then

$$(\phi, \eta'_1, \dots, \eta'_n) \cong \lim_{i=1}^n (\phi_i, \eta_i)$$

That is, the co-limit is always defined.

Proof: Since \approx is the identity on hyperedges it is obviously consistent and therefore G' is defined.

With proposition 2.2.7 it follows that G' satisfies the conditions of a co-limit of the η_i and ϕ_i and therefore

$$(\phi, \eta'_1, \dots, \eta'_n) \cong \lim_{i=1}^n (\phi_i, \eta_i)$$

□

Now we show that factorizations are preserved by co-limits, i.e. if the η_i are a factorization, the η'_i are a factorization as well.

Proposition 2.2.11 (Graph Construction and Factorizations)

Let $(\phi, \eta'_1, \dots, \eta'_n) := \lim_{i=1}^n (\phi_i, \eta_i)$ with $\eta_i : G_i[\chi_i] \rightarrow G[\chi]$, $\phi_i : G_i[\chi_i] \rightarrow G'_i[\chi'_i]$.

If the η_i are a factorization of $G[\chi]$ then the $\eta'_i : G'_i[\chi'_i] \rightarrow G'[\phi(\chi)]$, $i \in \{1, \dots, n\}$ are a factorization.

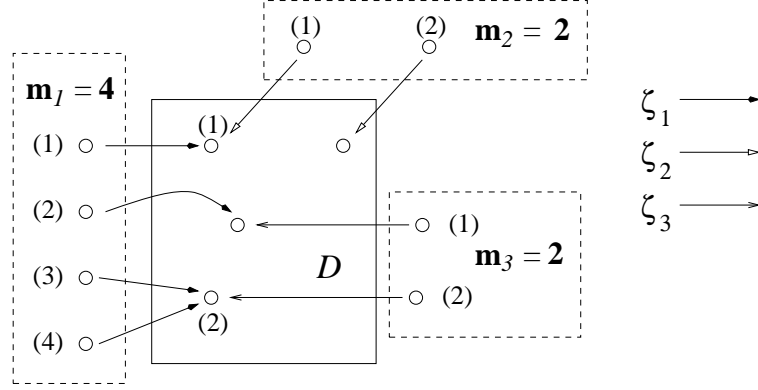
Proof: See appendix A.1.

□

Notation: If there is only one embedding η_1 we will write $\otimes(H_1, \eta_1)$ instead of $\bigotimes_{i=1}^1 (H_1, \eta_1)$.

Example: We give embeddings $\zeta_1, \zeta_2, \zeta_3$ for the factorization example (figure 2.1) in the previous section.

$\zeta_1 : \mathbf{4} \rightarrow D$, $\zeta_2 : \mathbf{2} \rightarrow D$, $\zeta_3 : \mathbf{2} \rightarrow D$ and $H \cong (H_1, \zeta_1) \otimes (H_2, \zeta_2) \otimes (H_3, \zeta_3)$.



The ζ_i can be regarded as some kind of construction plan showing how to connect the graphs H_1, H_2, H_3 . If two nodes are mapped onto the same node in D by the ζ_i , this indicates that the corresponding nodes of the H_i are to be glued together.

We will now define a kind of graph construction we will use very often: Let $H, J \in \mathcal{H}(Z, L)$ with $\text{card}(H) = \text{card}(J) = m$. Furthermore let $\zeta : \mathbf{m} \rightarrow \mathbf{m}$ be a strong morphism. We define $H \square J := (H, \zeta) \otimes (J, \zeta)$. That is we attach H, J at their external nodes.

Now we are going to show that the co-limit used for graph construction can be regarded as some kind of building block which can be used to construct new co-limits. It turns out that merging two matching co-limits yields a new co-limit. Co-limits can be merged in various ways:

Proposition 2.2.12 (Combination of Co-Limits) Let $\eta_i : H_i \rightarrowtail H$, $\phi_i : H_i \rightarrow H'_i$, $\eta'_i : H'_i \rightarrowtail H'$ and $\phi : H \rightarrow H'$ with

$$(\phi, \eta'_1, \dots, \eta'_n) = \lim_{i=1}^n (\phi_i, \eta_i)$$

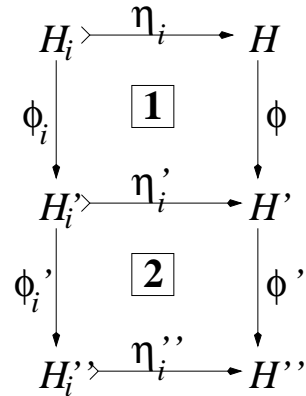
(A) $\boxed{1}, \boxed{2}$ co-limits $\Rightarrow \boxed{1} + \boxed{2}$ co-limit:

Let $\phi'_i : H'_i \rightarrow H''_i$ and let

$$(\phi', \eta''_1, \dots, \eta''_n) := \lim_{i=1}^n (\phi'_i, \eta'_i)$$

Then

$$(\phi' \circ \phi, \eta''_1, \dots, \eta''_n) \cong \lim_{i=1}^n (\phi'_i \circ \phi_i, \eta_i)$$



(B) $\boxed{1}, \boxed{1} + \boxed{2}$ co-limits \Rightarrow $\boxed{2}$ co-limit:

Let $\phi'_i : H'_i \rightarrow H''_i$ and let

$$(\phi'', \eta''_1, \dots, \eta''_n) := \lim_{i=1}^n (\phi'_i \circ \phi_i, \eta_i)$$

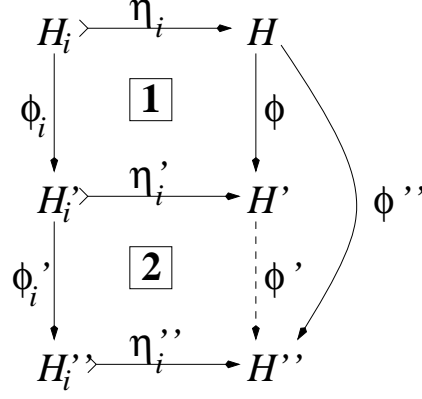
with $\phi'' : H \rightarrow H''$, $\eta''_i : H''_i \rightarrow H''$.

Then there exists a strong morphism

$\phi' : H' \rightarrow H''$ such that $\phi'' = \phi' \circ \phi$

and

$$(\phi', \eta'_1, \dots, \eta'_n) \cong \lim_{i=1}^n (\phi'_i, \eta'_i)$$



Now let $\eta_{ij} : H_{ij} \rightarrow H_i$, $\phi_{ij} : H_{ij} \rightarrow H'_{ij}$, $\eta'_{ij} : H'_{ij} \rightarrow H'_i$ and $\phi_i : H_i \rightarrow H'_i$ with $(\phi_i, \eta'_{i1}, \dots, \eta'_{in_i}) = \lim_{j=1}^{n_i} (\phi_{ij}, \eta_{ij})$ ($i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}$). (Note that in this case $\boxed{1}$ consists of m co-limits.)

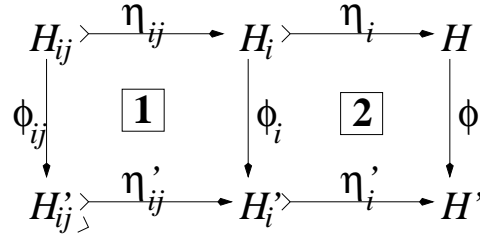
(C) $\boxed{1}, \boxed{2}$ co-limits \Rightarrow $\boxed{1} + \boxed{2}$ co-limit:

Let $\eta_i : H_i \rightarrow H$, $i \in \{1, \dots, m\}$ and let

$$(\phi, \eta'_1, \dots, \eta'_m) := \lim_{i=1}^m (\phi_i, \eta_i)$$

Then

$$(\phi, \eta'_i \circ \eta'_{i1}, \dots, \eta'_i \circ \eta'_{in_i}) \cong \lim_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (\phi_{ij}, \eta_i \circ \eta_{ij})$$



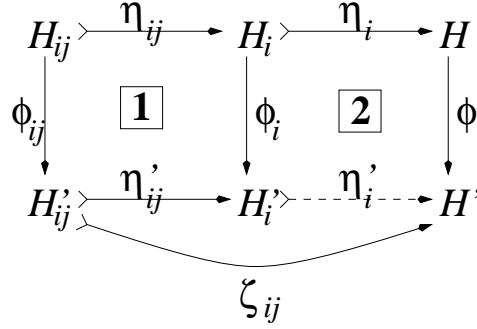
(D) $\boxed{1}, \boxed{1} + \boxed{2}$ co-limits \Rightarrow $\boxed{2}$ co-limit:

Let $\eta_i : H_i \rightarrow H$ and let

$$(\phi, \eta_{11}, \dots, \eta_{mn_m}) := \lim_{i \in \{1, \dots, m\}, 1 \leq j \leq n_i} (\phi_{ij}, \eta_i \circ \eta_{ij})$$

with $\eta_{ij} : H'_{ij} \rightarrow H'_i$ and $\phi : H \rightarrow H'$. Then there exist embeddings $\eta'_i : H'_i \rightarrow H'$ such that $\eta_{ij} = \eta'_i \circ \eta'_{ij}$ and

$$(\phi, \eta'_1, \dots, \eta'_n) := \lim_{i=1}^n (\phi_i, \eta_i)$$



Proof: See appendix A.1 □

Assume that we have several graphs H_1, \dots, H_m , each constructed by the help of a co-limit. These graphs are then combined to form a new graph H . The following question arises: Can we describe H with only one co-limit instead of a hierarchy of co-limits? The following proposition details how this can be done.

Proposition 2.2.13 *Let $\zeta_{ij} : \mathbf{m}_{ij} \rightarrow D_i$ and $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n_i\}$ be embeddings with $m_i = \text{card}(D_i)$ and let H_{ij} be hypergraphs with $m_{ij} = \text{card}(H_{ij})$.*

We define

$$(\phi, \xi_1, \dots, \xi_m) := \lim_{i=1}^m (D_i, \zeta_i) \quad \eta_{ij} := \xi_i \circ \zeta_{ij}$$

It follows that

$$\bigotimes_{i=1}^m \left(\bigotimes_{j=1}^{n_i} (H_{ij}, \zeta_{ij}), \zeta_i \right) \cong \bigotimes_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (H_{ij}, \eta_{ij})$$

Proof: Let $H_i := \bigotimes_{j=1}^{n_i} (H_{ij}, \zeta_{ij})$.

Since $\text{card}(D_i) = \text{card}(H_i) = m_i$, it follows that there are canonical strong morphisms $\phi_i : \mathbf{m}_i \rightarrow D_i$ and

$$(\phi, \xi_1, \dots, \xi_m) \cong \lim_{i=1}^m (\phi_i, \zeta_i)$$

Let

$$(\psi_i, \zeta'_{i1}, \dots, \zeta'_{in_i}) := \lim_{j=1}^{n_i} (\psi_{ij}, \zeta_{ij})$$

where the $\psi_{ij} : \mathbf{m}_{ij} \rightarrow H_{ij}$ are the canonical strong morphisms. Since the ψ_i are strong, the $\psi_i \circ \phi_i : \mathbf{m}_i \rightarrow H_i$ are also strong morphisms, in fact they are the canonical strong morphisms.

Let

$$(\phi', \xi'_1, \dots, \xi'_m) := \lim_{i=1}^m (\psi_i \circ \phi_i, \zeta_i)$$

This is exactly the co-limit defining H , i.e. $\xi'_i : H_i \rightarrow H$ and $\phi' : D \rightarrow H$.

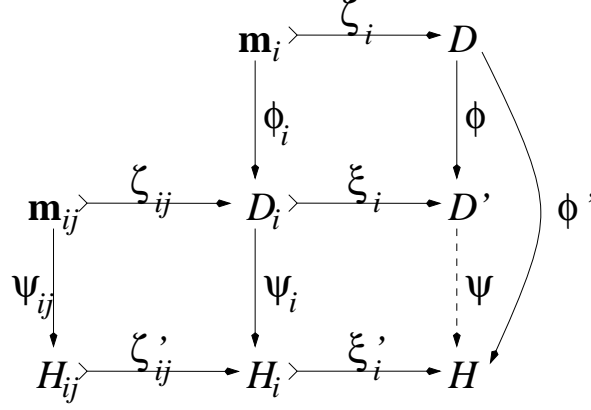
Proposition 2.2.12 **(B)** implies that there exists a strong morphism $\psi : D' \rightarrow H$ such that $\psi \circ \phi = \phi'$ and

$$(\psi_i, \xi'_1, \dots, \xi'_m) = \lim_{i=1}^m (\xi_i, \psi_i)$$

It now follows with proposition 2.2.12 (C) that

$$(\psi, \xi'_1 \circ \zeta'_{11}, \dots, \xi'_m \circ \zeta'_{mn_m}) \cong \lim_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (\psi_{ij}, \xi_i \circ \zeta_{ij})$$

which is equivalent to $H \cong \bigotimes_{i \in \{1, \dots, m\}, 1 \leq j \leq n_i} (H_{ij}, \eta_{ij})$ since $\eta_{ij} = \xi_i \circ \zeta_{ij}$.



□

If $D_i \cong \mathbf{m}_i$ in the proposition above, matters are easier:

Corollary 2.2.14 *Let $\zeta_{ij} : \mathbf{m}_{ij} \rightarrow m_i$ and $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n_i\}$ be embeddings and let H_{ij} be hypergraphs with $m_{ij} = \text{card}(H_{ij})$. It follows that*

$$\bigotimes_{i=1}^m \left(\bigotimes_{j=1}^{n_i} (H_{ij}, \zeta_{ij}), \zeta_i \right) \cong \bigotimes_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (H_{ij}, \zeta_i \circ \zeta_{ij})$$

Proof: It is easy to check that, in this case, that $(\zeta)_{i \in \{1, \dots, m\}} \cong (\zeta)_{i \in \{1, \dots, m\}}$. □

We will now describe the decomposition of a context into smaller contexts, which is the inverse of proposition 2.2.13.

Proposition 2.2.15 *Let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ and let*

$$\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, k\}, \quad l_1, \dots, l_k \in \mathbb{N}$$

Furthermore let $\xi_i : \mathbf{m}_i \rightarrow \mathbf{l}_{\alpha(i)}$, $i \in \{1, \dots, n\}$ such that ξ_i is surjective and for all $j \in \{1, \dots, k\}$, $i_1, i_2 \in \alpha^{-1}(j)$ it follows that

$$v_1 \in V_{\mathbf{m}_{i_1}}, v_2 \in V_{\mathbf{m}_{i_2}}, \quad \xi_{i_1}(v_1) = \xi_{i_2}(v_2) \Rightarrow \zeta_{i_1}(v_1) = \zeta_{i_2}(v_2) \quad (2.8)$$

Then there are embeddings $\eta_j : \mathbf{l}_j \rightarrow D$, $j \in \{1, \dots, k\}$ such that

$$\bigotimes_{j=1}^k \left(\bigotimes_{i \in \alpha^{-1}(j)} (H_i, \xi_i), \eta_j \right) \cong \bigotimes_{i=1}^n (H_i, \zeta_i)$$

Proof: We define $\eta_j(v') := \zeta_i(v)$ if $v' = \xi_i(v)$. The η_j are well-defined, since the ξ_i are surjective and because of (2.8).

Since $\eta_j(\xi_i(v)) = \zeta_i(v)$ it follows that $\eta_j \circ \xi_i = \zeta_i$. With corollary 2.2.14 the proposition follows immediately. \square

Note that there are other possibilities for decomposing a co-limit.

We will give one more lemma, simplifying co-limits, which will prove to be useful later in this work.

Lemma 2.2.16 *Let $H_j \cong \mathbf{m}_j$ where $m_j \in \mathbb{N}$. It follows that*

$$\bigotimes_{i=1}^n (H_i, \zeta_i) \cong \bigotimes_{i=1, i \neq j}^n (H_i, \zeta_i)$$

where $\zeta_j : \mathbf{m}_j \rightarrow D$ and D is a discrete graph.

Proof: We will show that

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

is the co-limit of $\phi_1, \dots, \phi_n, \zeta_1, \dots, \zeta_n$.

Let

$$(\phi, \eta_1, \dots, \eta_{j-1}, \eta_{j+1}, \dots, \eta_n) := \lim_{i=1, i \neq j}^n (\phi_i, \zeta_i)$$

We define $\eta_j : H_j \rightarrow H$ with $\eta_j(\chi_{H_j}) := \phi(\zeta_j(\chi_{\mathbf{m}_j}))$. Obviously $\eta_j \circ \phi_j = \phi \circ \zeta_j$.

Furthermore let $\hat{\phi} : D \rightarrow \hat{H}$, $\hat{\eta}_i : H_i \rightarrow \hat{H}$ such that $\hat{\eta}_j \circ \phi_j = \hat{\phi} \circ \zeta_j$. Since H is the co-limit of ϕ_i, η_i , $i \in \{1, \dots, j-1, j+1, \dots, n\}$ it follows that there exists a unique morphism $\psi : H \rightarrow \hat{H}$ with $\psi \circ \phi = \hat{\phi}$ and $\psi \circ \eta_i = \hat{\eta}_i$ for $i \in \{1, \dots, j-1, j+1, \dots, n\}$.

$$\psi(\eta_j(\chi_{H_j})) = \psi(\phi(\zeta_j(\chi_{\mathbf{m}_j}))) = \hat{\phi}(\zeta_j(\chi_{\mathbf{m}_j})) = \hat{\eta}_j(\phi_j(\chi_{\mathbf{m}_j})) = \hat{\eta}_j(\chi_{H_j})$$

And therefore $\psi \circ \eta_j = \hat{\eta}_j$.

Now H satisfies all conditions of a co-limit of $\phi_1, \dots, \phi_n, \zeta_1, \dots, \zeta_n$ (see definition 2.2.5). \square

2.2.3 Connection with the Double-Pushout Approach

We will now give the categorical view to graph-rewriting. We will introduce the well-known double-pushout approach, which is also often called algebraic approach, and characterize it with the notation defined above. For more details see [Ehr79].

Definition 2.2.17 (Double-Pushout) Let $r = (L, R)$ be a rewrite rule with $\text{card}(L) = \text{card}(R) = n$. There exist strong canonical morphisms $\phi_L : \mathbf{n} \rightarrow L$ and $\phi_R : \mathbf{n} \rightarrow R$.

Furthermore let H, H' be two hypergraphs. We write $H \xrightarrow{r} H'$ if there exists an embedding $\eta : \mathbf{n} \rightarrow K$ such that

$$\begin{array}{ccccc}
 & & \phi_L & & \phi_R \\
 & & \leftarrow & & \rightarrow \\
 L & & \mathbf{n} & & R \\
 \eta_H \downarrow & & \downarrow \eta & & \downarrow \eta_{H'} \\
 H & \xleftarrow{\psi_H} & K & \xrightarrow{\psi_{H'}} & H'
 \end{array}$$

□

Note that the graph K , representing the part of H that is not affected by the rewriting step, is, in most cases, not discrete.

Since in our case, we prefer to work with discrete graphs, this having a closer resemblance to factorization in groups or monoids, we will assume in the following sections that for any expression $\bigotimes_{i=1}^n (H_i, \zeta_i)$ the embeddings $\zeta_i : \mathbf{m}_i \rightarrow D$ map into a discrete graph D .

Therefore we will define graph rewriting in a slightly different way: $H \xrightarrow{r} H'$ if and only if there exist embeddings ζ, ζ' into a discrete graph and a hypergraph K such that

$$H \cong (L, \zeta) \otimes (K, \zeta') \text{ and } H' \cong (R, \zeta) \otimes (K, \zeta')$$

Any set of embeddings into a discrete graph is a factorization which will be called a *discrete factorization*.

Both definitions are equivalent.

2.2.4 Co-Limits, Factorizations and Contexts

We have now introduced three methods for constructing respectively decomposing graphs: factorizations, contexts and co-limits. Now we show how they are related and how they can be converted into one another.

We first compare factorizations with co-limits:

Proposition 2.2.18 (Co-Limit \leftrightarrow Factorization)

Let $\zeta_i : \mathbf{m}_i \rightarrow D, i \in \{1, \dots, n\}$ be a discrete factorization and let H_1, \dots, H_n be hypergraphs with $m_i := \text{card}(H_i)$. We define:

$$(\phi, \eta_1, \dots, \eta_n) := \lim_{i=1}^n (H_i, \zeta_i)$$

Then $(\eta_i)_{i \in \{1, \dots, n\}}$ with $\eta_i : H_i \rightarrow H$ is a factorization of H .

If $\eta_i : H_i \rightarrow H, i \in \{1, \dots, n\}$ is a factorization of H it follows that there exists a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$ and a strong morphism $\phi : D \rightarrow H$ such that

$$(\phi, \eta_1, \dots, \eta_n) \cong \lim_{i=1}^n (H_i, \zeta_i)$$

If the χ_{H_i} are duplicate-free, the discrete factorization $(\zeta_i)_{i \in \{1, \dots, n\}}$ is unique up to isomorphism.

Proof: See appendix A.1. \square

We now demonstrate, as promised above, how a hypergraph can be uniquely decomposed into basic graphs, corresponding to the prime elements of a factorization. Let us first define the notion of basic graphs:

Definition 2.2.19 (Basic Graphs) The class \mathcal{B} of *basic graphs* contains all graphs of the form $z_n(S)$. \square

It is now our task to show that every hypergraph allows a factorization into basic graph and that this factorization is unique up to isomorphism.

Proposition 2.2.20 (Factorization into Basic Graphs) For every hypergraph H there is a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ and basic graphs H_1, \dots, H_n such that $H \cong \bigotimes_{i=1}^n (H_i, \zeta_i)$.

Let

$$\bigotimes_{i=1}^n (H_i, \zeta_i) \cong_R \bigotimes_{i=1}^n (H'_i, \zeta'_i)$$

where $(\zeta_i)_{i \in \{1, \dots, n\}}, (\zeta'_i)_{i \in \{1, \dots, k\}}$ are discrete factorizations and the $H_1, \dots, H_n, H'_1, \dots, H'_k$ are basic graphs.

Then $n = k$ and there exists a permutation $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ such that

$$H_i \cong_R H'_{\alpha(i)}$$

for every i and

$$(\zeta_i)_{i \in \{1, \dots, n\}} \cong (\zeta'_{\alpha(i)})_{i \in \{1, \dots, n\}}$$

Proof: Since every hypergraph H has a factorization $\eta_i : H_i \rightarrow H$, $i \in \{1, \dots, n\}$, where all H_i are basic graphs, it follows with proposition 2.2.18 that there exist embeddings $\zeta_i : \mathbf{m}_i \rightarrow D$ such that $H \cong \bigotimes_{i=1}^n (H_i, \zeta_i)$.

Now we assume that there are two factorizations

$$(\zeta_i)_{i \in \{1, \dots, n\}} \text{ and } (\zeta'_{\alpha(i)})_{i \in \{1, \dots, n\}}$$

Let $(\phi, \eta_1, \dots, \eta_n) := \lim_{i=1}^n (H_i, \zeta_i)$, $(\phi', \eta'_1, \dots, \eta'_k) := \lim_{i=1}^k (H'_i, \zeta'_i)$.

Then $(\eta_i)_{i \in \{1, \dots, n\}}$ and $(\eta'_i)_{i \in \{1, \dots, k\}}$ are both factorizations of H into basic graphs. Since E_H is the disjoint union of the $\eta_i(E_{H_i})$ respectively $\eta'_i(E_{H'_i})$ and both H_i and H'_i consist of only one edge it follows that $n = k$.

Let e_i be the only edge of H_i and let e'_i be the only edge of H'_i .

Furthermore let $E_H = \{e^1, \dots, e^n\}$ such that the only edge e_i of H_i is mapped onto e^i , i.e. $\eta_i(e_i) = e^i$. If $n_i := \text{card}(e^i)$, $z_i := z_H(e_i)$ and $l_i := l_H(e^i)$ it follows that $H_i \cong (z_i)_{n_i}(l_i)$.

We define a permutation $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ with

$$j = \alpha(i) \iff \eta'_j(e'_j) = e^i$$

It follows that

$$\text{card}(e'_{\alpha(i)}) = \text{card}(\eta'_{\alpha(i)}(e'_{\alpha(i)})) = \text{card}(e^i) = \text{card}(\eta_i(e_i)) = \text{card}(e_i)$$

In the same way we can show that

$$\begin{aligned} l_{H'_{\alpha(i)}}(e'_{\alpha(i)}) & R \quad l_{H_i}(e_i) = l_i \\ z_{H'_{\alpha(i)}}(e'_{\alpha(i)}) & = z_{H_i}(e_i) = z_i \end{aligned}$$

This implies $H_i \cong_R H_{\alpha(i)}$ and that there are isomorphisms $\phi_i : H_i \rightarrow H'_{\alpha(i)}$ with $\phi_i(e_i) = e'_{\alpha(i)}$ and $\eta_i = \eta'_{\alpha(i)} \circ \phi_i$.

Therefore

$$(\eta_i)_{i \in \{1, \dots, n\}} \cong_R (\eta'_{\alpha(i)})_{i \in \{1, \dots, n\}}$$

and since the χ_{H_i} are duplicate-free it follows with proposition 2.2.18 that

$$(\zeta_i)_{i \in \{1, \dots, n\}} \cong (\zeta'_{\alpha(i)})_{i \in \{1, \dots, n\}}$$

□

This proposition strongly resembles the factorization of a number into primes or the representation of a vector in terms of base vectors.

Graph construction with co-limits and with contexts is essentially the same. Both methods can be converted into one another:

Proposition 2.2.21 (Co-Limit \leftrightarrow Context) *Let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ be a discrete factorization and let*

$$C\langle x_1, \dots, x_n \rangle := \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

For all hypergraphs H_1, \dots, H_n with $m_i = \text{card}(H_i)$ it follows that

$$C\langle H_1, \dots, H_n \rangle \cong \bigotimes_{i=1}^n (H_i, \zeta_i) \tag{2.9}$$

Let $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ be contexts with holes of cardinality m_1, \dots, m_n . If both satisfy (2.9) for all H_1, \dots, H_n with $m_i = \text{card}(H_i)$ it follows that

$$C\langle x_1, \dots, x_n \rangle \cong C'\langle x_1, \dots, x_n \rangle$$

And for all contexts $C\langle x_1, \dots, x_n \rangle$ with holes of cardinality m_1, \dots, m_n there is a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, m\}$ such that

$$C\langle x_1, \dots, x_n \rangle \cong \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

Proof: See appendix A.1. \square

Since graph construction by context or by co-limit is basically the same we can define construction of hypergraph morphisms (as an extension of hypergraph construction) as follows:

Definition 2.2.22 Let $\phi_i : H_i \rightarrow J_i$, $i \in \{1, \dots, n\}$ be strong morphisms and let C be a context with holes of cardinality $\text{card}(H_1), \dots, \text{card}(H_n)$. Then there exists, according to proposition 2.2.12, **(B)** a strong morphism from $C\langle H_1, \dots, H_n \rangle$ into $C\langle J_1, \dots, J_n \rangle$. This morphism is denoted by $C\langle \phi_1, \dots, \phi_n \rangle$. \square

2.3 Graph Expressions

This section is based on material from [BC87] where graph expressions are introduced.

The idea is to represent hypergraphs by terms and to define isomorphism by means of algebraic laws. This approach leads us into the well-known field of signatures, algebras, terms and algebraic equations. Graph rewriting corresponds to the substitution of subterms.

But there is a price we have to pay: the loss of locality. Given a graph expression we cannot, at first sight, decide which edges and nodes are adjacent. As a consequence, deciding whether there is a morphism from one expression into another, based only on the graph expression, seems to be very difficult. Furthermore changing the labelling function of a graph involves taking apart the entire expression.

First of all we describe the following signature:

Definition 2.3.1 (Graph Expression) Let Z be a fixed set of edge sorts and let L be a fixed set of labels.

A *graph expression* is a term composed of elements of the following signature: for every $n \in \mathbb{N}$ there is a sort n . We define the following constant and function symbols:

Constants:	0 of sort 0
	1 of sort 1
	$(z, l)_n$ of sort n for all $n \in \mathbb{N}$, $l \in L$
Function	Sum: $\oplus_{n,m}$ of sort $m, n \rightarrow m + n$
Symbols:	Node Fusion: $\theta_{\delta,n}$ of sort $n \rightarrow n$ for all $n \in \mathbb{N}$ and all equivalence relations δ on $\{1, \dots, n\}$
	Redefinition of External Nodes: $\sigma_{\alpha,n,p}$ of sort $n \rightarrow p$ for all $n, p \in \mathbb{N}$ and all mappings $\alpha : \{1, \dots, p\} \rightarrow \{1, \dots, n\}$

Furthermore expressions may contain variables with sorts in \mathbb{N} . \square

In a well-formed graph expression the sort of the function symbols can be computed if the sorts of all variables and constants are known. Therefore we

will omit sort indices in the future and write \oplus , θ_δ and σ_α instead of $\oplus_{n,m}$, $\theta_{\delta,n}$ and $\sigma_{\alpha,n,p}$.

We will now define the operations corresponding to the function symbols:

Definition 2.3.2 (Graph Operations)

Sum: Let G_1, G_2 be two simple hypergraphs. (We will assume that their node and edge sets are disjoint. Otherwise we take isomorphic copies of G_1 and G_2). We define:

$$G_1 \oplus G_2 := (V_{G_1} \cup V_{G_2}, E_{G_1} \cup E_{G_2}, s_{G_1} \cup s_{G_2}, z_{G_1} \cup z_{G_2}, l_{G_1} \cup l_{G_2})$$

If $H_i = G_i[\chi_i]$, $i \in \{1, 2\}$ we define:

$$H_1 \oplus H_2 := (G_1 \oplus G_2)[\chi_1 \circ \chi_2]$$

(Note that the operation \oplus is commutative on simple hypergraphs but not on multi-pointed hypergraphs.)

Node Fusion: Let $H = G[\chi]$ be a hypergraph with $n = \text{card}(H)$ and let δ be an equivalence on $\{1, \dots, n\}$. We define:

$$\theta_\delta(H) := (G/\approx)[p(\chi)]$$

where \approx_V is the equivalence on V_H generated by

$$\{([\chi]_i, [\chi]_j) \mid (i, j) \in \delta\}$$

and \approx_E is the identity on E_H . Furthermore p is the canonical projection of G into G/\approx .

Redefinition of External Nodes: Let $H = G[\chi]$.

Furthermore let $\alpha : \{1, \dots, p\} \rightarrow \{1, \dots, n\}$. We define:

$$\sigma_\alpha(H) := G[[\chi]_{\alpha(1)} \dots [\chi]_{\alpha(p)}]$$

□

It is now not difficult to convert a graph expression into a hypergraph:

Definition 2.3.3 (Graph Expressions \rightarrow Hypergraphs) We will define a mapping val_e that assigns a hypergraph in $\mathcal{H}(L \cup X)$ to every graph expression. It is defined inductively as follows:

$$\text{val}_e(0) := \mathbf{0}, \text{val}_e(1) := \mathbf{1}, \text{val}_e((z, l)_n) := z_n(l), \text{val}_e(x) := \text{var}_{\text{sort}(x)}(x)$$

Let u_1, u_2 be graph expressions. We define:

$$\begin{aligned} \text{val}_e(u_1 \oplus u_2) &:= \text{val}_e(u_1) \oplus \text{val}_e(u_2) \\ \text{val}_e(\theta_\delta(u_1)) &:= \theta_\delta(\text{val}_e(u_1)) \\ \text{val}_e(\sigma_\alpha(u_1)) &:= \sigma_\alpha(\text{val}_e(u_1)) \end{aligned}$$

□

Example: We regard the hypergraphs H_1, H_2, H_3 in figure 2.1. They can be described by graph expressions in the following way:

$$\begin{aligned} H_1 &\cong \text{val}_e(\sigma_{\alpha_1}(\theta_{\delta_1}(A_3 \oplus D_4))) \\ H_2 &\cong \text{val}_e(\sigma_{\alpha_2}(\theta_{\delta_2}(B_2 \oplus C_1))) \\ H_3 &\cong \text{val}_e(E_2) \end{aligned}$$

where δ_1 is generated by $\{(2, 5), (3, 4)\}$ and $\alpha_1 : \{1, \dots, 4\} \rightarrow \{1, \dots, 7\}$ with $\alpha_1(1) = 1, \alpha_1(2) = 3, \alpha_1(3) = 6, \alpha_1(4) = 7$ ($\alpha_1(2) = 4$ is also a valid choice).

δ_2 is generated by $\{(2, 3)\}$ and $\alpha_2 : \{1, 2\} \rightarrow \{1, 2, 3\}$ with $\alpha_2(1) = 1, \alpha_2(2) = 2$ (or $\alpha_2(2) = 3$).

We will now give a set of equations (which first appeared in [BC87]) and will find out that hypergraphs are in fact its initial model. That is, the algebraic equations are correct and complete.

Proposition 2.3.4 (Algebraic Properties of Graph Operations) *The following equation schemes generate the equivalence on graph expressions. They are correct and complete: $u \simeq_R v$ if and only if $\text{val}_e(u) \cong_R \text{val}_e(v)$.*

Let u, v, w be graph expressions.

$$(z, l)_n \simeq_R (z, l')_n \quad (2.10)$$

where $z \in Z, l, l' \in L$ with $l R l'$.

$$u \oplus (v \oplus w) \simeq_R (u \oplus v) \oplus w \quad (2.11)$$

$$\sigma_\beta(\sigma_\alpha(u)) \simeq_R \sigma_{\alpha \circ \beta}(u) \quad (2.12)$$

$$\sigma_{id}(u) \simeq_R u \quad (2.13)$$

where id is the identity function on $\{1, \dots, n\}$.

$$\theta_\delta(\theta_{\delta'}(u)) \simeq_R \theta_\gamma(u) \quad (2.14)$$

where γ is the equivalence generated by $\delta \cup \delta'$.

$$\theta_\Delta(u) \simeq_R u \quad (2.15)$$

where $\Delta = \{(i, i) \mid i \in \{1, \dots, n\}\}$ is the trivial equivalence.

$$\sigma_\alpha(u) \oplus \sigma_{\alpha'}(v) \simeq_R \sigma_\beta(v \oplus u) \quad (2.16)$$

where

$$\begin{aligned} \alpha : \{1, \dots, p\} &\rightarrow \{1, \dots, n\} \\ \alpha' : \{1, \dots, p'\} &\rightarrow \{1, \dots, n'\} \\ \beta : \{1, \dots, p + p'\} &\rightarrow \{1, \dots, n + n'\} \\ \text{with } \beta(i) &= \begin{cases} n' + \alpha(i) & \text{if } i \in \{1, \dots, p\} \\ \alpha'(i - p) & \text{if } i \in \{p + 1, \dots, p + p'\} \end{cases} \end{aligned}$$

$$\theta_\delta(u) \oplus \theta_{\delta'}(v) \simeq_R \theta_\gamma(u \oplus v) \quad (2.17)$$

where δ is an equivalence on $\{1, \dots, n\}$, δ' is an equivalence on $\{1, \dots, n'\}$ and γ is the equivalence on $\{1, \dots, n+n'\}$ generated by $\delta \cup \{(n+i, n+j) \mid (i, j) \in \delta'\}$.

$$\theta_\delta(u \oplus 1) \simeq_R \sigma_\alpha(\theta_\gamma(u)) \quad (2.18)$$

where δ is an equivalence on $\{1, \dots, n+1\}$ such that $(i, n+1) \in \delta$ for some $i \in \{1, \dots, n\}$. Furthermore

$$\begin{aligned} \gamma &= \delta \cap (\{1, \dots, n\} \times \{1, \dots, n\}) \\ \alpha &: \{1, \dots, n+1\} \rightarrow \{1, \dots, n\} \\ \text{where } \alpha(j) &= \begin{cases} j & \text{if } j \leq n \\ i & \text{if } j = n+1 \end{cases} \end{aligned}$$

$$\theta_\delta(\sigma_\alpha(u)) \simeq_R \sigma_\alpha(\theta_{\alpha(\delta)}(u)) \quad (2.19)$$

where δ is an equivalence on $\{1, \dots, n\}$, $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ and $\alpha(\delta)$ denotes the equivalence generated by $\{(\alpha(i), \alpha(j)) \mid (i, j) \in \delta\}$

$$\sigma_\alpha(\theta_\delta(u)) \simeq_R \sigma_\beta(\theta_\delta(u)) \quad (2.20)$$

where $\alpha, \beta : \{1, \dots, p\} \rightarrow \{1, \dots, n\}$ and $(\alpha(i), \beta(i)) \in \delta$ for all $i \in \{1, \dots, p\}$.

$$u \oplus 0 \simeq_R u \quad (2.21)$$

Proof: See [BC87], theorem (3.10). \square

As in proposition 2.2.21 in the previous section we will now show how to convert a co-limit construction into a graph expression and vice versa.

Proposition 2.3.5 (Graph Expression \leftrightarrow Context) *Let $g(x_1, \dots, x_n)$ be a graph expression without hyperedges of the form $(z, l)_n$ and let*

$$C\langle x_1, \dots, x_n \rangle := \text{val}_e(g(x_1, \dots, x_n))$$

For all graph expressions g_1, \dots, g_n with $\text{card}(\text{val}_e(g_i)) = \text{sort}(x_i)$ it follows that

$$C\langle \text{val}_e(g_1), \dots, \text{val}_e(g_n) \rangle \cong \text{val}_e(g(g_1, \dots, g_n)) \quad (2.22)$$

Let $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ be contexts. If both satisfy (2.22) for all g_1, \dots, g_n with $\text{card}(\text{val}_e(g_i)) = \text{sort}(x_i)$ it follows that

$$C\langle x_1, \dots, x_n \rangle \cong C'\langle x_1, \dots, x_n \rangle$$

And for all contexts $C\langle x_1, \dots, x_n \rangle$ there is a graph expression $g(x_1, \dots, x_n)$ such that

$$C\langle x_1, \dots, x_n \rangle \cong \text{val}_e(g(x_1, \dots, x_n))$$

Proof:

- $val_e(g(x_1, \dots, x_n)) \langle val_e(g_1), \dots, val_e(g_n) \rangle \cong val_e(g(g_1, \dots, g_n))$ can be shown by a straightforward structural induction on $g(x_1, \dots, x_n)$.
- We assume that $C \langle x_1, \dots, x_n \rangle, C' \langle x_1, \dots, x_n \rangle$ are contexts and both satisfy (2.22) for all g_1, \dots, g_n with $card(val_e(g_i)) = sort(x_i) =: m_i$. It follows that

$$\begin{aligned} C \langle x_1, \dots, x_n \rangle &\cong C \langle val_e(x_1), \dots, val_e(x_n) \rangle \cong val_e(g(x_1, \dots, x_n)) \\ &\cong C' \langle val_e(x_1), \dots, val_e(x_n) \rangle \cong C' \langle x_1, \dots, x_n \rangle \end{aligned}$$

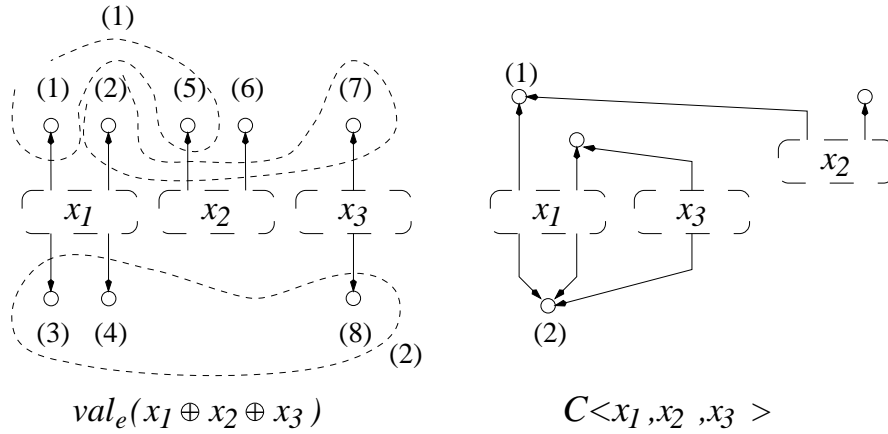
- Let $C \langle x_1, \dots, x_n \rangle$ be a context. We will now construct a graph expression $g(x_1, \dots, x_n)$ such that $C \langle x_1, \dots, x_n \rangle \cong val_e(g(x_1, \dots, x_n))$. This is described in [BC87], proposition (3.6).

□

Example: In the previous section we have shown how to describe the hypergraph H from figure 2.1 in the form $C \langle H_1, H_2, H_3 \rangle$. The graph expression corresponding to the context $C \langle x_1, x_2, x_3 \rangle$ is $\sigma_\alpha(\theta_\delta(x_1 \oplus x_2 \oplus x_3))$ where

$$sort(x_1) = 4, \quad sort(x_2) = 2, \quad sort(x_3) = 2$$

θ is generated by $\{(3, 4), (3, 8), (1, 5), (2, 7)\}$ and $\alpha : \{1, 2\} \rightarrow \{1, \dots, 7\}$ with $\alpha(1) = 1$ (or $\alpha(1) = 5$) and $\alpha(2) = 3$ (or $\alpha(2) = 4$ or $\alpha(2) = 8$).



The dashed lines indicate the equivalence classes of δ .

Notation: We define the following abbreviations:

- Let i_1, \dots, i_n be natural numbers and let H be a hypergraph with $card(H) = m \geq \max\{i_1, \dots, i_n\}$.

$$\sigma_{i_1 \dots i_n}(H) := \sigma_\alpha(H)$$

where $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ with $\alpha(j) := i_j$.

That is the former i_j -th external node becomes the j -th external node and all other external nodes are hidden.

- Let i, j be natural numbers and let H be a hypergraph with $\text{card}(H) = m \geq \max\{i, j\}$.

$$\theta_{i,j}(H) := \theta_\delta(H)$$

where δ is the smallest equivalence on $\{1, \dots, n\}$ satisfying $i \delta j$.

That is the i -th and the j -th external node are merged and the rest of the hypergraph stays unchanged.

2.4 Name-based Notation

From our point of view an expression in a process calculus is just a graph of interacting processes and messages. Many papers (see e.g. [MPW89a, MPW89b, Mil80]) contain graphical representations of processes suggesting such a view. In their syntax, however, the connections between processes, i.e. the channels, are not represented by a graph but by common names. We will show the correspondence between structures created by names and graphs.

Describing hypergraphs with the help of names seems to be very intuitive, especially if there is a meaning associated with the names.

Since we use terms and algebraic equations most of the disadvantages of graph expression mentioned in the previous section (especially the loss of locality) apply here as well. Furthermore we are burdened with the additional task of keeping track of free names.

Our name-based graph terms describe only hypergraphs whose external strings do not contain duplicates, since this is the approach used in all papers about process calculi. If we try to describe arbitrary hypergraphs, problems in graph rewriting pop up (see example at the end of this section).

Definition 2.4.1 (Name-based Graph Terms) Let N be a set of names. A *name-based graph term* h has one of the following forms:

Empty Graph: 0

Node: $[a], a \in N$

Edge of sort z , labelled l : $(z, l)[a_1 \dots a_n], z \in Z, l \in L, a_1, \dots, a_n \in N$

Parallel Composition: $h_1 | h_2$

Node Hiding: $(\nu a)h_1$

where h_1, h_2 are again name-based graph-terms.

We define the set of free names of a term h inductively as follows:

$$\begin{aligned} \text{fn}(0) &:= \emptyset \\ \text{fn}([a]) &:= \{a\} \\ \text{fn}((S)[a_1 \dots a_n]) &:= \{a_1, \dots, a_n\} \\ \text{fn}(h_1 | h_2) &:= \text{fn}(h_1) \cup \text{fn}(h_2) \\ \text{fn}((\nu a)h_1) &:= \text{fn}(h_1) \setminus \{a\} \end{aligned}$$

A *closed* name-based graph term is of the form $h[t]$ where h is a name-based graph term and $t \in fn(h)_{df}^*$ is a duplicate-free string which contains exactly the names in $fn(h)$. The cardinality of $h[t]$ is defined by $card(h[t]) := |t|$. \square

Our notation may trigger the suggestion that graph terms of the form h correspond to simple hypergraphs. In simple hypergraphs every node can still be made external, while in graph terms the set of external (i.e. free names) is already fixed and the string t only indicates their order.

Notation: Let s, t be duplicate-free strings of names where $Set(s) \subseteq Set(t)$, $n := |s|$, $m := |t|$. $\zeta_{s \rightarrow t} : \mathbf{n} \rightarrow \mathbf{m}$ is an embedding with

$$\zeta_{s \rightarrow t}(\lfloor \chi_{\mathbf{n}} \rfloor_i) = \lfloor \chi_{\mathbf{m}} \rfloor_j \iff \lfloor s \rfloor_i = \lfloor t \rfloor_j$$

It is quite obvious that $\zeta_{t \rightarrow u} \circ \zeta_{s \rightarrow t} = \zeta_{s \rightarrow u}$.

Definition 2.4.2 (Interpretation of Name-based Graph Terms)

Let $h[s]$ be a closed name-based graph term where $n := |s|$. We define:

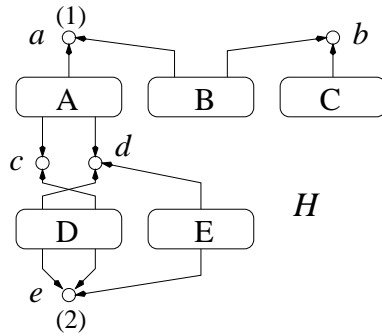
- $val_n(0[\varepsilon]) := \mathbf{0}$
- $val_n(\lceil a \rceil[a]) := \mathbf{1}$
- $val_n((h_1 | \dots | h_n)[t]) := \bigotimes_{i=1}^n (val_n(h_i[t_i]), \zeta_{t_i \rightarrow t})$
where $t_i := t \setminus (fn(h_1 | \dots | h_n) \setminus fn(h_i))$
- $val_n(((\nu a)h)[t]) := \begin{cases} \otimes(val_n(h[ta]), \zeta) & \text{if } a \in fn(h) \\ val_n(h[t]) & \text{otherwise} \end{cases}$

where ζ is the projection of $\mathbf{m} + \mathbf{1}$ into $\sigma_{1\dots m}(\mathbf{m} + \mathbf{1})$ with $m := |s|$.

- $val_n(((z, l)[t'])[t]) := \otimes(z_{|t'|}(l), \zeta_{t' \rightarrow t})$

\square

Example: We give a name-based graph term which corresponds to the hypergraph H in figure 2.1. The first step is to assign names to the nodes of H :



$$h = (\nu b)(\nu c)(\nu d)(\\ (A)[acd] \mid (B)[ab] \mid \\ (C)[b] \mid (D)[dcee] \mid \\ (E)[de])$$

$$H \cong val_n(h[ae])$$

As we have indicated above, name-based graph terms can only describe hypergraphs where the string of external nodes contains no duplicates. Everyone of these hypergraphs, however, has a representation in terms of name-based graph terms.

Proposition 2.4.3 (Construction of Name-based Graph Terms) *Let H be a hypergraph, where χ_H contains no duplicates, i.e. $\chi_H \in (V_H)_{df}^*$. Then there exists a closed name-based graph term $h[t]$ such that $H \cong \text{val}_n(h[t])$.*

Proof: See appendix A.1. □

There are two kinds of components we would like to substitute in name-based graph-terms: free names and entire terms. Terms are always substituted for an edge labelled with a variable. We will now define these two kinds of substitution

Definition 2.4.4 (Substitution in Name-based Graph Terms)

Substitution of free names: Let h be a name-based graph term and let $s, t \in N^*$ be duplicate-free strings of names with $|s| = |t|$. We define $h[s/t]$ inductively as follows: Let $\text{subst}_{[s/t]} : N \rightarrow N$ such that $\text{subst}_{[s/t]}([t]_i) = [s]_i$ and $\text{subst}_{[s/t]}(a) = a$ if $a \notin \text{Set}(s)$.

$$\begin{aligned}
0[s/t] &:= 0 \\
[a][s/t] &:= [\text{subst}_{[s/t]}(a)] \\
((S)[a_1 \dots a_m])[s/t] &:= (S)[\text{subst}_{[s/t]}(a_1 \dots a_m)] \\
(h_1|h_2)[s/t] &:= h_1[s/t]|h_2[s/t] \\
((\nu a)h)[s/t] &:= \begin{cases} (\nu a)(h[s/t]) & \text{if } a \notin \text{Set}(s) \cup \text{Set}(t) \\ (\nu a)(h[s \setminus i, t \setminus i]) & \text{if } [t]_i = a \\ (\nu b)((h[b/a])[s/t]) & \text{if } \exists i : [s]_i = a, [t]_i \neq a \text{ and } \\ & b \notin \text{Set}(s) \cup \text{Set}(t) \cup \text{fn}(h) \end{cases}
\end{aligned}$$

Substitution of terms: Let $h[t], h'[t']$ be two closed name-based terms. We define $h[t]\langle h'[t']/x \rangle$ with:

$$\begin{aligned}
h[t]\langle h'[t']/x \rangle &:= (h\langle h'[t']/x \rangle)[t] \\
0\langle h'[t']/x \rangle &:= 0 \\
[a]\langle h'[t']/x \rangle &:= a \\
((z, l)[a_1 \dots a_m])\langle h'[t']/x \rangle &:= \begin{cases} h'[a_1 \dots a_m/t'] & \text{if } l = x, z = \text{var} \\ (z, l)[a_1 \dots a_m] & \text{otherwise} \end{cases} \\
(h_1|h_2)\langle h'[t']/x \rangle &:= h_1\langle h'[t']/x \rangle|h_2\langle h'[t']/x \rangle \\
((\nu a)h)\langle h'[t']/x \rangle &:= (\nu a)(h\langle h'[t']/x \rangle)
\end{aligned}$$

□

We now present equations relating equivalent terms. Later we will show that these equations are correct and complete for the algebra of hypergraphs.

Proposition 2.4.5 (Equations for Name-based Graph Terms) *The following equation schemes generate the equivalence on name-based graph terms. Two closed terms $h[t], h'[t']$ are equivalent wrt. the following equations if and only if $val_n(h[t]) \cong_R val_n(h'[t'])$.*

Let h, h_1, h_2, h_3 be name-based graph terms.

$$(z, l)[t] \simeq_R (z, l')[t] \quad \text{if } l R l' \quad (2.23)$$

$$h_1|h_2 \simeq_R h_2|h_1 \quad (2.24)$$

$$h_1|(h_2|h_3) \simeq_R (h_1|h_2)|h_3 \quad (2.25)$$

$$h|0 \simeq_R h \quad (2.26)$$

$$(\nu a)0 \simeq_R 0 \quad (2.27)$$

$$(\nu a)(\nu b)h \simeq_R (\nu b)(\nu a)h \quad (2.28)$$

$$((\nu a)h_1)|h_2 \simeq_R (\nu a)(h_1|h_2) \quad \text{if } a \notin fn(h_2) \quad (2.29)$$

$$(\nu a)h \simeq_R (\nu b)(h[b/x]) \quad \text{if } b \notin fn(h) \quad (2.30)$$

$$[a]|[a] \simeq_R [a] \quad (2.31)$$

$$(z, l)[a_1 \dots a_n]|[a_i] \simeq_R (z, l)[a_1 \dots a_n] \quad (2.32)$$

and for closed terms:

$$h[s] \simeq_R (h[s'/s])[s'] \quad \text{if } |s| = |s'|, s' \text{ is duplicate-free} \quad (2.33)$$

Proof: See appendix A.1. The proof makes use of proposition 2.4.6. \square

In order to show the proposition above we introduce a normal form for name-based terms. Since every term has a normal form it is sufficient to prove the completeness result above for terms in normal form, which will considerably simplify the proof.

Proposition 2.4.6 (Normal Form of Name-based Notation)

A name-based graph term is in normal form if it has the form

$$((\nu b_1) \dots (\nu b_n)((z_1, l_1)[a_{11} \dots a_{1n_1}] \mid \dots \mid (z_m, l_m)[a_{m1} \dots a_{mn_m}] \mid [c_1] \mid \dots \mid [c_k]))[d_1 \dots d_l] \quad (2.34)$$

and

$$A \cap C = \emptyset, B \cap D = \emptyset, A \cup C = B \cup D, |B| = n, |C| = k$$

where

$$\begin{aligned} A &:= \{a_{ij} \mid i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}\} \\ B &:= \{b_i \mid i \in \{1, \dots, n\}\} \\ C &:= \{c_i \mid i \in \{1, \dots, k\}\} \\ D &:= \{d_i \mid i \in \{1, \dots, l\}\} \end{aligned}$$

For every graph term $h[t]$ there is a graph term $h'[t']$ in normal form such that $h[t] \simeq h'[t']$.

Proof: See appendix A.1. \square

Let $A = \{a_1, \dots, a_n\}$ be a finite subset of N . In the following we will use $(\nu A)h$ as an abbreviation for $(\nu a_1) \dots (\nu a_n)h$.

Analogous to hypergraphs we can now define n -ary name-based contexts $h[t]\langle x_1, \dots, x_n \rangle$ which contain exactly one edge $(var, x_i)[t_i]$ for every x_i , $i \in \{1, \dots, n\}$ and no other edges.

That is, an n -ary name-based contexts has the form

$$h[t] \simeq (\nu A)((var, x_1)[t_1] \mid \dots \mid (var, x_n)[t_n])$$

Substitution in contexts is defined as follows:

$$h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle := h[t]\langle h_1[t_1]/x_1, \dots, h_n[t_n]/x_n \rangle$$

(see the definition of substitution in definition 2.4.4).

And now we will give, as in the previous sections, the transformation of name-based terms with variables into contexts. That is, at least for graphs whose strings of external nodes do not contain duplicates, graph construction by co-limits and graph-construction with name-based contents is equivalent.

Proposition 2.4.7 (Context \leftrightarrow Name-based Notation)

Let $h[t]\langle x_1, \dots, x_n \rangle$ be a name-based context and let

$$C\langle x_1, \dots, x_n \rangle := val_n(h[t]\langle x_1, \dots, x_n \rangle)$$

For all closed name-based terms $h_1[t_1], \dots, h_n[t_n]$ with $|t_i| = sort(x_i)$ it follows that

$$C\langle val_n(h_1[t_1]), \dots, val_n(h_n[t_n]) \rangle \cong val_n(h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle) \quad (2.35)$$

Let $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ be contexts. If both satisfy (2.35) for all $h_1[t_1], \dots, h_n[t_n]$ with $|t_i| = sort(x_i)$ it follows that

$$C\langle x_1, \dots, x_n \rangle \cong C'\langle x_1, \dots, x_n \rangle$$

And furthermore, for all contexts $C\langle x_1, \dots, x_n \rangle$ there is a name-based context $h[t]\langle x_1, \dots, x_n \rangle$ such that

$$C\langle x_1, \dots, x_n \rangle \cong val_n(h[t]\langle x_1, \dots, x_n \rangle)$$

Proof:

- The following equation

$$\begin{aligned} & val_n(h[t]\langle x_1, \dots, x_n \rangle) \langle val_n(h_1[t_1]), \dots, val_n(h_n[t_n]) \rangle \\ & \cong val_n(h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle) \end{aligned}$$

can be shown by structural induction on $h[t]$ in a straightforward way.

- We assume that $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ are contexts and both satisfy (2.35) for all $h_1[t_1], \dots, h_n[t_n]$ with $|t_i| = \text{sort}(x_i) =: m_i$.

Let $s_1, \dots, s_n \in N_{df}^*$ with $|s_i| = m_i$. It follows that

$$\begin{aligned} C\langle x_1, \dots, x_n \rangle &\cong C\langle \text{val}_n((\text{var}, x_1)[s_1]), \dots, \text{val}_n((\text{var}, x_n)[s_n]) \rangle \\ &\cong \text{val}_n(h[t]\langle (\text{var}, x_1)[s_1], \dots, (\text{var}, x_n)[s_n] \rangle) \\ &\cong C'\langle \text{val}_n((\text{var}, x_1)[s_1]), \dots, \text{val}_n((\text{var}, x_n)[s_n]) \rangle \cong C'\langle x_1, \dots, x_n \rangle \end{aligned}$$

- Let $C\langle x_1, \dots, x_n \rangle$ be a context. According to proposition 2.4.3 there exists a closed name-based graph term $h[t]$ such that $C\langle x_1, \dots, x_n \rangle \cong \text{val}_n(h[t])$ and $h[t]$ contains exactly one occurrence of the edge labelled x_i , i.t. $h[t]$ is a context.

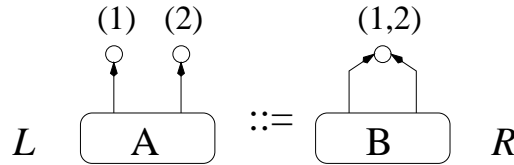
□

Corollary 2.4.8 *Let $h[t], h_1[t_1], \dots, h_n[t_n]$ be name-based graph terms where $h[t]$ is a context with holes of cardinality $|t_1|, \dots, |t_n|$. It follows that*

$$\text{val}_n(h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle) \cong \text{val}_n(h[t])\langle h_1[t_1], \dots, h_n[t_n] \rangle$$

We now discuss our choice of representing only graphs with duplicate-free strings of external nodes: it is clear that *representation* of graphs which have duplicates in their external nodes is no problem, we just have to extend the syntax of a name-based term $h[t]$ and to allow duplicates in t . It will be necessary to change definition 2.4.2, but that is not the real problem.

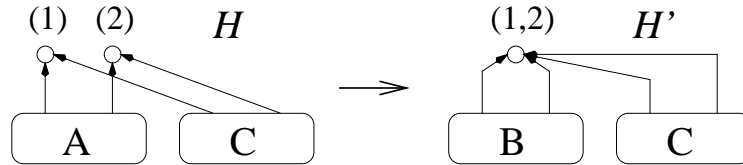
Problems, however, will arise in *graph rewriting*. Let us examine the following rewriting rule, consisting of a left hand side L and a right hand side R .



We assume that both edges have sort \diamond . Converting the rule into name-based notation yields

$$((\diamond, A)[ab])[ab] ::= ((\diamond, B)[aa])[aa]$$

Applying the rule above to the graph H yields the following reduction, fusing the two external nodes:



In name based-notation

$$\begin{aligned} H &\cong \text{val}_n(((\diamond, A)[ab] \mid (\diamond, C)[ab])[ab]) \\ H' &\cong \text{val}_n(((\diamond, B)[aa] \mid (\diamond, C)[aa])[aa]) \end{aligned}$$

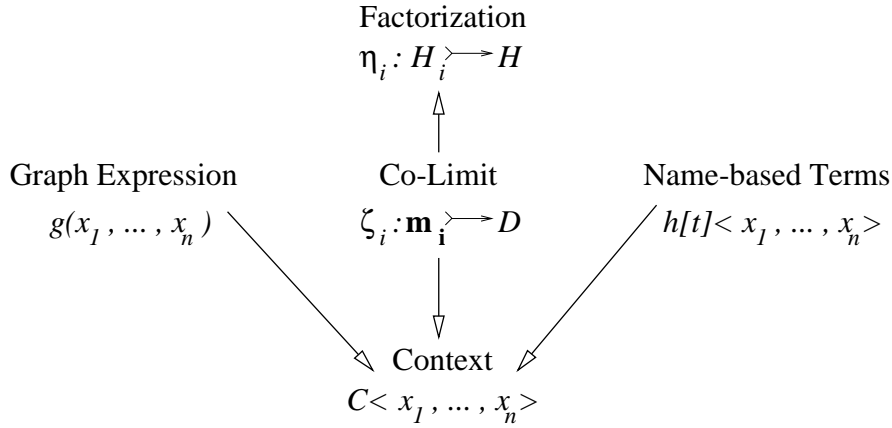
and we can observe that this reduction is *not local*, i.e. it also affects the edge labelled C . Therefore construction of graphs or graph rewriting cannot simply be modelled with contexts and the equivalent of proposition 2.4.7 would not work.

Thus we will refrain from duplicates in the external nodes. Since this notation is mainly useful for the comparison of our calculus with other process calculi (e.g. the π -calculus) and there are no duplicates in these calculi, no problems will arise.

2.5 Comparison of Hypergraph Construction Methods

In the previous sections we have introduced four methods of hypergraph construction: contexts (definition 2.1.10), co-limits (definition 2.2.9), graph expressions (definition 2.3.1) and name-based graph terms (definition 2.4.1). And then there is the concept of factorization of hypergraphs (definition 2.1.6).

In propositions 2.2.18, 2.2.21, 2.3.5 and 2.4.7 we have shown that these construction methods are basically the same. We have given the following transformations:



For arbitrary graphs there is a one-to-one correspondence between contexts, co-limits and graph expression, i.e. the transformations are bijective. The same holds for contexts, co-limits, graph expression and name-based terms in the case of graphs with duplicate-free sequences of external nodes. In all four cases the construction methods yield—together with graphs H_1, \dots, H_n of the correct cardinality—a canonical factorization $\eta_i : H_i \twoheadrightarrow H$, $i \in \{1, \dots, n\}$.

The main contribution in this chapter is the development of a new categorical graph construction mechanism (definition 2.2.9) and the transformation of

the different methods of graph construction into one another. To our knowledge, there is also no consistent treatment of a name-based representation of hypergraphs, although the issue is implicitly contained in many papers on process calculi (e.g. [Mil90]).

Chapter 3

Process Calculi: An Overview

Research in programming language has seen the development of several different programming paradigms: functional, imperative, logic, object-oriented and distributed or concurrent programming. (Note that the concepts of object-oriented and distributed or concurrent programming are orthogonal to the other paradigms, i.e. there is e.g. object-oriented, imperative or functional, concurrent programming).

These paradigms possess underlying calculi or machine models, clarifying their key ideas, forming the basis of their semantics and serving as a starting point for research.

Process calculi, describing processes communicating by message passing, claim to be the adequate model for concurrent and distributed programming. But let us first review the underlying models of other programming paradigms:

Functional Programming: the λ -calculus [Bar84, Bar90, HS86] is considered as the standard calculus for functional programming, representing the essence of the functional programming paradigm, by term rewriting and substitution.

It defines, in a nutshell, the meaning of function abstraction and application, as well as of free and bound variables. There are several approaches of giving a denotational semantics to the λ -calculus (see [Sto77] as well as the references cited above).

Another important area of research is the development of type systems for the λ -calculus, guaranteeing the normalization property, i.e. termination of reductions for every well-typed expression [Bar90].

Imperative Programming: the Turing machine, representing a von-Neumann-computer with linear memory, executing a program dependent on the content of the tape.

The theory of Turing machines has introduced important concepts, such as effective computability and complexity theory [Rog67, BDG88, Pap94].

Logic Programming: logic programming is, like functional programming, a declarative style of programming. It is based on first-order logic, namely

on Horn clauses, which give it a firm basis in a very well-known area of logic [Apt90].

In concurrent and distributed programming there is a diversity of calculi trying to model the essence of this programming paradigm. Common concepts and ideas, however, are visible. The rest of this chapter will deal with process calculi, introduce key ideas and describe what benefits for programming can be gained by research in this area. Concerning this matter, we will concentrate on bisimulation equivalence and type systems. Both concepts will appear in this work in connection with the SPIDER calculus.

It is left to mention, that some process calculi claim to be adequate models for object-oriented programming [MPW89a]. There is a large amount of work dealing with calculi for object-oriented programming [AC96]. In many cases these calculi are very similar to process calculi with an added construct for records (see also [AC96]).

3.1 Introduction to CCS

Among the first and one of the most successful process calculi to be proposed was CCS (Calculus of Communicating Systems) by Robin Milner [Mil80, Mil90]. It describes processes signalling on common channels. In contrast to other process calculi which will be described later in this chapter, communication in CCS does not involve the sending of values, processes or addresses, but only the exchange of signals. (There is, however, an extension, called value-passing CCS [Mil80].) Despite these limitations, CCS has full computational power.

We will now introduce the syntax and semantics of CCS: we assume that we have a set of labels $L = \{a, b, c, \dots\}$, representing actions. Elements of L are used as input prefixes, while elements of $\bar{L} = \{\bar{a}, \bar{b}, \bar{c}, \dots\}$ are output prefixes. We define $\bar{\bar{a}} = a$. τ represents the silent (internal) action.

Furthermore let X be a set of variables. A CCS process p is either a

Variable: $x \in X$

Prefix: $\alpha.p$ where $\alpha \in L \cup \bar{L} \cup \{\tau\}$

Summation: $p_1 + p_2$

Parallel Composition: $p_1 \mid p_2$

Restriction: $(\nu a)p$ where $a \in L$

Relabelling: $p[f]$ where $f : L \rightarrow L$ is a bijection (with $f(\bar{a}) := \overline{f(a)}$)

Recursion: $\text{rec } x.p, x \in X$

Dead Process: 0

where p, p_1, p_2 are again CCS processes. $\alpha.p$ is a process that can perform the action α and behave like p afterwards. $p_1 + p_2$ can either behave as p_1 or p_2 . $p_1 \mid p_2$ represents p_1 and p_2 acting in parallel, while $(\nu a)p$ acts like p where

actions a and \bar{a} are prohibited. In $p[f]$ the actions of p are relabelled by f . And $\text{rec } x.p$ is the fixed-point of the equation $x = p$. The dead process 0 denotes inaction.

In order to describe operational semantics we give a *labelled transition semantics* (see [Plo81]).

$\text{(CCS-ACT)} \quad \alpha.p \xrightarrow{\alpha} p$	
$\text{(CCS-SUM1)} \quad \frac{p_1 \xrightarrow{\alpha} p'_1}{p_1 + p_2 \xrightarrow{\alpha} p'_1}$	$\text{(CCS-SUM2)} \quad \frac{p_2 \xrightarrow{\alpha} p'_2}{p_1 + p_2 \xrightarrow{\alpha} p'_2}$
$\text{(CCS-COM0)} \quad \frac{p_1 \xrightarrow{a} p'_1, p_2 \xrightarrow{\bar{a}} p'_2}{p_1 \mid p_2 \xrightarrow{\tau} p'_1 \mid p'_2} \quad a \in L \cup \bar{L}$	
$\text{(CCS-COM1)} \quad \frac{p_1 \xrightarrow{\alpha} p'_1}{p_1 \mid p_2 \xrightarrow{\alpha} p'_1 \mid p_2}$	$\text{(CCS-COM2)} \quad \frac{p_1 \xrightarrow{\alpha} p'_2}{p_1 \mid p_2 \xrightarrow{\alpha} p_1 \mid p'_2}$
$\text{(CCS-RES)} \quad \frac{p \xrightarrow{\alpha} p'}{(\nu a)p \xrightarrow{\alpha} (\nu a)p'} \quad \text{if } \alpha \notin \{a, \bar{a}\}$	
$\text{(CCS-REL)} \quad \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$	$\text{(CCS-REC)} \quad \frac{p[\text{rec } x.p/x] \xrightarrow{\alpha} p'}{\text{rec } x.p \xrightarrow{\alpha} p'}$

$p[p'/x]$ denotes the substitution of x by p' in p . It may involve renaming of bound names in order to avoid capture of the free names in p' .

3.2 Reduction Semantics and the Chemical Abstract Machine

As an alternative to the labelled transition semantics presented above, Gérard Berry and Gérard Boudol proposed a different style of semantics with their chemical abstract machine (CHAM) [BB92]. This has initiated a new trend in defining the operational semantics of process calculi.

The key idea is to define structural congruence of processes a priori. E.g. $p_1 \mid p_2$ is expected to “behave” just like $p_2 \mid p_1$. In labelled transition semantics this fact can be expressed only after the introduction of bisimulation equivalence or similar equivalence relations. Berry and Boudol introduced the metaphor of a suspension of molecules which can be transformed by heating and cooling rules. In this case p_1 and p_2 are allowed to float around and change places.

In a more conventional way, we can say that there is a rule of equivalence $p_1 \mid p_2 \equiv p_2 \mid p_1$. Since this allows communicating processes to reach each other and communicate locally, labelled transitions are not necessary anymore.

We will introduce a new, unlabelled reduction relation \rightarrow . This new form of semantics is often called *reduction semantics* and is equivalent to labelled

transition semantics, i.e. in the case of CCS $p_1 \xrightarrow{\tau} p_2$ if and only if $p_1 \rightarrow p_2$. We will, however, leave out the relabelling operator, since it does not really fit into reduction semantics. Furthermore we will drop the summation operator, in order to facilitate the semantics. (It is, however, possible to integrate summation into reduction semantics, see e.g. [San92]).

CCS now looks as follows:

<i>Rules of Structural Congruence:</i>	
$p_1 p_2 \equiv p_2 p_1 \quad p_1 (p_2 p_3) \equiv (p_1 p_2) p_3 \quad p 0 \equiv p \quad (\nu a) 0 \equiv 0$	
$(\nu a)(\nu b)p \equiv (\nu b)(\nu a)p \quad ((\nu a)p_1) p_2 \equiv (\nu a)(p_1 p_2) \text{ if } a \notin fn(p_2)$	
$(\nu a)p \equiv (\nu b)(p[b/a]) \text{ if } b \notin fn(p) \quad rec\ x.p \equiv p[rec\ x.p/x]$	
$\frac{p \equiv p'}{\alpha.p \equiv \alpha.p'} \quad \frac{p_1 \equiv p'_1, p_2 \equiv p'_2}{p_1 p_2 \equiv p'_1 p'_2}$	
$\frac{p \equiv p'}{(\nu a)p \equiv (\nu a)p'} \quad \frac{p \equiv p'}{rec\ x.p \equiv rec\ x.p'}$	
<i>Reduction Rules:</i>	
$a.p \mid \bar{a}.p' \rightarrow p \mid p' \quad \frac{p_1 \rightarrow p'_1}{p_1 \mid p_2 \rightarrow p'_1 \mid p_2}$	
$\frac{p \rightarrow p'}{(\nu a)p \rightarrow (\nu a)p'} \quad \frac{q \equiv p, p \rightarrow p', p' \equiv q'}{q \rightarrow q'}$	

where $fn(p) \subseteq L$ is the set of all free labels in p , i.e. all labels that are not hidden by the operator ν . Furthermore $p[b/a]$ denotes syntactic substitution of names (with renaming of bound names in order to avoid capture of b).

By many reduction semantics is considered to be more natural and has the advantage that reduction can take place locally. In more recent works on process calculi [Mil92, San92] this form of semantics is actually dominant.

3.3 Mobility in the π -Calculus

We will now introduce the polyadic π -calculus [MPW89a, Mil92, Mil91] by extending the reduction semantics of CCS. The main additional feature of the π -calculus is mobility of port addresses, i.e. communicating processes will not only exchange signals but tuples of labels or names. The receiver process is then able to use these names for future communications. This means dynamic changes in the process and communication structure.

The syntax of CCS is changed as follows: input prefixes will now have the form $a(x_1, \dots, x_n)$ where $a, x_1, \dots, x_n \in L$ and $x_i \neq x_j$ if $i \neq j$. Output prefixes have the form $\bar{a}a_1 \dots a_n$, where $a, a_1, \dots, a_n \in L$. In the π -calculus no difference is made between names and variables representing names. ($n = 1$ for every prefix yields the monadic π -calculus as described in [MPW89a]).

The only change in the reduction semantics is that

$$a.p \mid \bar{a}.p' \rightarrow p \mid p'$$

is replaced by

$$a(x_1, \dots, x_n).p \mid \bar{a}a_1 \dots a_n.p' \rightarrow p[a_1/x_1, \dots, a_n/x_n] \mid p'$$

Let us further mention that in the π -calculus recursion is usually replaced by replication, i.e. there is a syntactic construct $!p$ where p is a π -calculus process. And the former rule of recursion respectively fixed-point expansion is replaced by

$$!p \rightarrow !p \mid p$$

Replication is as powerful as recursion and in this case has the added benefit that variables representing processes are unnecessary.

Furthermore the sum operator seems to be less essential than other operators and is left out in some versions of the π -calculus (see e.g. [Mil92]). This paper deals with the encoding of the λ -calculus into the π -calculus and thus ensures that the π -calculus without sum still has full computational power.

There are two phenomena that can occur in the π -calculus but not in CCS: the sending of a name out of its scope or into another scope where scope denotes the part of a term where the name is bound. The handling of scope extrusion and intrusion will also be of interest in higher-order calculi.

Scope Extrusion: a name is sent out of its scope, e.g.

$$a(x).p \mid (\nu b)(\bar{a}b.p') \equiv (\nu b)(a(x).p \mid \bar{a}b.p') \rightarrow (\nu b)(p[b/x] \mid p')$$

$$\text{if } b \notin fn(p)$$

Scope Intrusion: a name is sent into the scope of the same name. This is dealt with as follows:

$$\bar{a}b.p \mid (\nu b)(a(x).p') \equiv (\nu c)(\bar{a}b.p \mid a(x).(p'[c/b])) \rightarrow p \mid (\nu c)(p'[c/b][b/x])$$

$$\text{if } c \notin fn(p)$$

In both cases the rules of structural equivalence take care of the correct handling of the expression. In the case of labelled transition semantics, these two phenomena demand a careful choice of rules.

3.4 Higher-Order Communication—Static and Dynamic Binding

In the previous section we have described the mobility of port addresses. Another form of mobility is higher-order communication, i.e. the ability of processes to send other processes.

One way to realize this would be to change the communication rule to

$$\bar{a}p.p' \mid a(x).p'' \rightarrow p' \mid p''[p/x]$$

Let us now observe what happens if a process is sent out of the scope of one of its free names: let $b \in fn(p)$, $b \notin fn(p'')$ then:

$$(\nu b)(\bar{a}p.p') \mid a(x).p'' \equiv (\nu b)(\bar{a}p.p' \mid a(x).p'') \rightarrow (\nu b)(p' \mid p''[p/x])$$

That is, just as in the case of scope extrusion above, the scope of b is extended and the names in p stay bound as before. Therefore this concept is also called *static binding* and it is the binding mechanism of the higher-order π -calculus ($HO\pi$) by Davide Sangiorgi [San92].

This is, however, not the only way to define a higher-order calculus. The labelled transition semantics of CHOCS by Bent Thomsen [Tho89b, Tho95] is defined in such a way that the expression above would reduce in the following way:

$$(\nu b)(\bar{a}p.p') \mid a(x).p'' \xrightarrow{\tau} (\nu b)p' \mid p''[p/x]$$

which means that the names of p are taken out of their scope and rebound in their new context (*dynamic binding*).

Let us mention that our presentation of the higher-order π -calculus was somewhat superficial. In addition to the feature described above, it is also possible to parametrise and instantiate processes and to communicate port addresses as well as processes.

3.5 Bisimulation and Proof Techniques

We will now present *bisimulation* and *barbed congruence*, two equivalence relations which relate processes showing the same behaviour toward the environment. Both equivalences have strong and weak version, where the strong version demands that processes coincide after every step, while the weak version allows that one process may perform several steps while the equivalent process only performs one step.

It is more convenient to define bisimulation equivalence in labelled transition semantics, so we will start with this semantics and move on to reduction semantics later.

Bisimulation: Let R be a symmetric relation on CCS expression. We call R a *strong bisimulation* if

$$p R q, p \xrightarrow{\alpha} p' \Rightarrow \exists q' : q \xrightarrow{\alpha} q', p' R q'$$

where $\alpha \in L \cup \bar{L} \cup \{\tau\}$. That is if p can perform an α -step, q can match this and both resulting processes are equivalent wrt. R .

Two processes p, q are called *strongly bisimilar* ($p \sim q$) if there exists a strong bisimulation R such that $p R q$.

This definition seems to be too strong rather often since it attempts to match even internal actions of the processes. Thus we obtain *weak bisimulation* if we replace the condition above by:

$$\begin{aligned} p R q, p \xrightarrow{\alpha} p' &\Rightarrow \exists q' : q \xrightarrow{\tau^*} \xrightarrow{\alpha} \xrightarrow{\tau^*} q', p' R q' && \text{if } \alpha \in L \cup \bar{L} \\ p R q, p \xrightarrow{\tau} p' &\Rightarrow \exists q' : q \xrightarrow{\tau^*} q', p' R q' \end{aligned}$$

And, as above, two processes p, q are called *weakly bisimilar* ($p \approx q$) if there exists a weak bisimulation R such that $p R q$.

In CCS, bisimulation, although undecidable, still has some interesting properties. Slight modifications of the definition above (see [Mil90]) turn *bisimilarity* into a congruence, which is very important in practice: when we want to replace a process with another, maybe more efficient, process, it is important to check if the resulting systems behaves as before, and normally it is not sufficient to check if isolated processes behave the same. If bisimilarity was a congruence, however, this proof could be straightforward.

Furthermore we can give a set of algebraic equations describing strong bisimilarity, such that the equations are complete for important subsets of CCS. There are, e.g. equations corresponding to the laws of structural congruence or the expansion law linking parallel combination and summation. An instance of the expansion law would be: let $p := \alpha_1.p_1 + \alpha_2.p_2$, $q := \beta_1.q_1 + \beta_2.q_2$. Then

$$p \mid q \sim \alpha_1.(p_1 \mid q) + \alpha_2.(p_2 \mid q) + \beta_1.(p \mid q_1) + \beta_2.(p \mid q_2) + r_{11} + r_{12} + r_{21} + r_{22}$$

where $r_{ij} := \begin{cases} p_i \mid q_j & \text{if } \bar{\alpha}_i = \beta_j \\ 0 & \text{otherwise} \end{cases}$. As you can see the expansion law shows some similarities to the multiplication of sums.

Because of this algebraic representation, some process calculi are also called process algebras.

Let us now turn to another way of defining bisimulation, i.e. *barbed bisimulation* [San92, MS92b]. It tackles two problems at once: first it gives an acceptable solution for calculi with reduction semantics, second it defines bisimilarity such that it is a congruence in any case.

We will first define the notion of barbs. A process p has a barb a ($p \downarrow a$) if there exists a process p' such that either $p \xrightarrow{a} p'$ or $p \xrightarrow{\bar{a}} p'$. It is, of course, possible to define barbs without labelled transition semantics. Furthermore $p \Downarrow a$ if there exists a process p' such that $p \rightarrow p' \downarrow a$.

Barbed Congruence: A relation R on processes is called a *strong barbed bisimulation* if

$$p R q, p \downarrow a \Rightarrow q \downarrow a \tag{3.1}$$

$$p R q, p \rightarrow p' \Rightarrow \exists q' : q \rightarrow q', p' R q' \tag{3.2}$$

And p, q are called *strongly barbed-bisimilar* ($p \sim q$) if there exists a weak barbed bisimulation R such that $p R q$.

p, q ($p \sim q$) are called *strongly barbed-congruent* if for every context C : $C[p] \sim C[q]$.

\approx (the weak version of barbed congruence) is defined analogously, we only have to replace (3.1) and (3.2) by

$$p R q, p \Downarrow a \Rightarrow q \Downarrow a \quad (3.1')$$

$$p R q, p \rightarrow p' \Rightarrow \exists q' : q \rightarrow^* q', p' R q' \quad (3.2')$$

In practice, it is rather hard or even impossible to prove bisimilarity of processes on the basis of this definition. The hard thing is the quantification over all contexts. It is therefore necessary to prove a context lemma in order to reduce the number of relevant contexts [San96].

Another technique simplifying bisimulation proofs is the concept of *bisimulations up-to*. A relation R on processes is called a weak barbed bisimulation up-to strong barbed congruence if it satisfies (3.1') and

$$p R q, p \rightarrow p' \Rightarrow q \rightarrow^* q', p' \sim R \sim q'$$

Every R defined in such a way is a weak barbed bisimulation.

The main idea with bisimulations-up-to is to factor out annoying and less important aspects of the proof. Weak barbed bisimulation up-to weak barbed congruence, however, is not necessarily a weak barbed congruence, a counterexample can be found in [MS92a].

Apart from to strong bisimulation, there are other “up-to relations” which can be used, such as bisimulation up to context, up to expansion or up to almost-weak bisimulation [MS92a].

3.6 Comparison of Calculi and Full Abstraction

When working with calculi, it is often an interesting question, if they can be translated into one another, i.e. if there is an encoding $\Theta : A \rightarrow B$ of calculus A into calculus B .

It is necessary to impose some condition on Θ , e.g. we can demand that all reductions of an expression in A are matched by its encoding (see chapter 7).

Another desirable property of encodings is full abstraction, i.e. we demand that

$$p \approx_A q \iff \Theta(p) \approx_B \Theta(p')$$

where \approx_A and \approx_B are bisimilarities or other suitable behavioural equivalences.

Important results in this direction are the encoding of the higher-order π -calculus into the π -calculus [San92] and the encoding of the λ -calculus in the π -calculus [Mil92, SW98], in CHOCS [Tho89b, Tho95] or in the blue-calculus [Bou97] (see below).

3.7 Synchronicity and Asynchronicity

Up until now communication in the presented calculi has always been synchronous, i.e. both communication partners have to rendez-vous. On the syntactic level, asynchronicity can be obtained very easily: we demand that the only possible continuation of an output prefix is the dead process, i.e. if we have a process of the form $\bar{a}.p$ or $\bar{a}a_1 \dots a_n.p$ it follows that $p = 0$. These processes can also be regarded as messages.

That is synchronous sending $\bar{a}a_1 \dots a_n.p$ can now only be represented by its asynchronous version $\bar{a}a_1 \dots a_n.0 \mid p$.

Saying that an asynchronous calculus is simply a subcalculus of a synchronous calculus is, however, not correct. It is expected, that in an asynchronous calculus input action cannot be observed from the outside. E.g. in the case of barbed bisimulations there are only barbs for output actions, which leads to a different notion of bisimulation [ACS96].

3.8 Graph Calculi

Graph-based description of process calculi are widespread, they are mainly used for visualization purposes [Mil94, Mil90].

Describing the semantics of processes by graph rewriting is common in related areas: there are, for example, the description of actors by graphs [JR90], interaction nets and interaction combinators related to linear logic [Laf90, Laf97], Δ -grammars specifying concurrent languages and systems [KLG93, Loy92], a graph notation for concurrent combinators [Yos94] or the specification of the behaviour of petri nets by hypergraph grammars [Rei80].

The idea of describing the semantics of a concurrent or distributed programming language with a graph grammar semantics, similar to our approach, can also be found in [BS93].

In [Tae96] the graph transformation model is expanded in order to describe parallel and distributed graph transformation.

3.9 Type Systems

While in CCS and in the monadic π -calculus, no process can ever run into runtime-errors, this is not true for the polyadic π -calculus and other calculi. Therefore simple type systems—often called sort systems—ensuring the freedom of runtime errors were introduced [Mil91, Gay93].

These type systems were extended in order to handle polymorphism and recursive types [Vas94, Tur95] or to verify more complex properties of processes such as input/output capabilities [PS93], security and privacy issues [Aba97], absence of deadlocks [Kob97] or confluence [NS97].

For more details on type systems see chapter 8.

3.10 Other Process Calculi

This overview can necessarily not deal with all existing process calculi. We give a structured enumeration over important calculi and mention important work:

First-Order Calculi:

CCS (R. Milner) Calculus of Communicating Systems [Mil80, Mil90, Mil89]

CSP (C.A.R. Hoare) Communicating Sequential Processes [Hoa85]

Petri Nets [Rei85] Petri nets are modelling the execution of actions wrt. pre- and post-conditions (a representation of Petri nets by hypergraph grammars can be found in [Rei80]) [Rei85]

Mobile Calculi:

π -calculus (R. Milner) [MPW89a, MPW89b, Mil91]

Higher-order π -calculus (D. Sangiorgi) Mobility of port addresses and processes with static binding [San92, MS92b, San96]

CHOCs (Bent Thomsen) Calculus of higher-order communicating systems: Mobility of processes with dynamic binding [Tho89a, Tho89b, Tho95]

Join Calculus (C. Fournet and G. Gonthier) process calculus, based directly on the chemical abstract machine [FG96, FGL⁺96]

Fusion Calculus (J. Parrow and B. Victor) extension of the π -calculus with a different concepts of scopes [PV98]

Blue Calculus (G. Boudol) [Bou97] higher-order calculus, extension of both λ - and π -calculus, an early version was the γ -calculus [Bou89]

Foundations of Process Calculi:

Action Structures (R. Milner) framework for different calculi based on interactions with the environment [Mil96, Mil93]

Interaction Categories (S. Abramsky) Models processes in terms of category theory where processes are represented as morphisms and interfaces are represented as objects [AGN95]

Calculi with additional Features:

Distributed Calculi (J. Riely and M. Hennessy) Calculi modelling locations, movement of processes or threads to another location and failure of locations [RH97, RH98].

Spi Calculus (M. Abadi) a calculus which describes encryption of messages, useful for modelling security protocols and privacy [Aba97, AG97]

Object Calculus (M. Abadi, L. Cardelli) [AC96]

Chapter 4

The SPIDER Calculus: Syntax and Semantics

4.1 Syntax

We will now describe the syntax and semantics of the graph-based calculus SPIDER. It is an asynchronous calculus featuring higher-order communication and mobility of port addresses.

A SPIDER expression is simply a hypergraph, consisting of two different types of edges: processes and messages. The nodes of a hypergraph are meant to represent ports.

Messages are always connected to a certain port—the port they are sent to—and are labelled with their content, which is again a SPIDER expression.

Processes are either waiting for messages, which means they are labelled with a *process abstraction*, detailing the behaviour of a process upon receipt of a message. Or they are labelled with a *replication operator*, meaning that this process can create copies of its content at will. They can also be labelled with a *variable*, which indicates that they will be replaced by an entire process graph in the course of message reception.

Definition 4.1.1 (SPIDER expression) Let X be a set of variables and let $Z := \{proc, mess, var\}$ be a set of edge sorts.

The class \mathcal{S} of all SPIDER expressions is the smallest class satisfying:

- Every element of \mathcal{S} is a hypergraph.
- If $H_1, \dots, H_n \in \mathcal{S}$ and $C\langle x_1, \dots, x_n \rangle$ is a discrete context with $x_1, \dots, x_n \in X$ and $sort(x_i) = card(H_i)$ it follows that

$$(\textbf{Context}) \quad C\langle H_1, \dots, H_n \rangle \in \mathcal{S}$$

- If $H \in \mathcal{S}$ with $n := card(H)$ it follows that

$$(\textbf{Message}) \quad mess_m(H) \in \mathcal{S}, \quad \text{where } m \geq 1$$

$$(\textbf{Process with Replication}) \quad proc_n(!H) \in \mathcal{S}$$

(Process with Process Abstraction)

$$proc_m(\lambda_k x.H) \in \mathcal{S}, \quad \text{where } m \leq card(H), \quad k \in \{1, \dots, m\}$$

(Process with a Variable)

$$proc_n(x) \in \mathcal{S}, \quad \text{where } x \in X, \quad sort(x) = n$$

Labels of the form $!H$, $\lambda_k x.H$ or x are also called *process descriptions*.

$free(H) \subseteq X$ is the set of all variables occurring in a SPIDER expression H , which are not bound by an abstraction $\lambda_n x$. \square

The set of all processes is denoted by P_H (i.e. $P_H := E_H^{proc}$) and the set of all messages is denoted by M_H (i.e. $M_H := E_H^{mess}$).


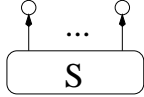
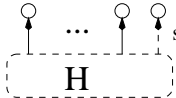
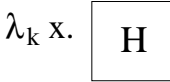
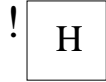
The last node of a message (if it exists) is also called its send-node, i.e. the node it is sent to. Let $e \in E_H$. We define $send_H(e) := (s_H(e))_{card(e)}$.

If there is no free variable x in H , then $\lambda_k x.H$ can be abbreviated to $\lambda_k.H$.

In the following we will use the name “node” and “port” interchangeably, since our use of nodes corresponds to the concept of a port in process calculi.

4.2 Graphical Representation

SPIDER has an intuitive graphical representation which we will use extensively in the rest of this work. The components are represented as follows:

Port	
Process	
Message	
Process Abstraction	
Replication Operator	

The name SPIDER for the calculus was chosen because of the resemblance of an expression with a spider net with many spiders (i.e. processes) in it.

4.3 Semantics

We will now define substitution of variables $H[J/x]$ in the entire expression.

Definition 4.3.1 (Substitution of Variables) Let H, J be SPIDER expressions and let x be a variable with $\text{sort}(x) = \text{card}(J)$. We define $H[J/x]$ inductively as follows:

$$\begin{aligned}
(C\langle H_1, \dots, H_n \rangle)[J/x] &:= C\langle H_1[J/x], \dots, H_n[J/x] \rangle \\
\text{mess}_n(H)[J/x] &:= \text{mess}_n(H[J/x]) \\
\text{proc}_n(y)[J/x] &:= \begin{cases} J & \text{if } x = y \\ \text{proc}_n(y) & \text{otherwise} \end{cases} \\
\text{proc}_n(!H)[J/x] &:= \text{proc}_n(!(H[J/x])) \\
\text{proc}_n(\lambda_k y. H)[J/x] &:= \begin{cases} \text{proc}_n(\lambda_k y. H) & \text{if } x = y \\ \text{proc}_n(\lambda_k y. (H[J/x])) & \text{if } x \neq y, y \notin \text{free}(J) \\ \text{proc}_n(\lambda_k z. ((H[\text{proc}_n(z)/y])[J/x])) & \text{if } x \neq y, y \in \text{free}(J), z \notin \text{free}(H) \cup \text{free}(J), \\ & n = \text{sort}(y) = \text{sort}(z) \end{cases}
\end{aligned}$$

□

In the spirit of the Chemical Abstract Machine (CHAM) [BB92] we now define rules of structural congruence for SPIDER, i.e. expressions are considered to be equivalent if they have the same structure and if they can be converted into one another by simple renaming of variables (α -conversion).

Definition 4.3.2 (Structural Congruence) \equiv is the least equivalence satisfying:

Rules of Structural Congruence:

$$(C\text{-MESS}) \quad \frac{H_1 \equiv H_2}{\text{mess}_n(H_1) \equiv \text{mess}_n(H_2)}$$

$$(C\text{-PA}) \quad \frac{H_1 \equiv H_2}{\text{proc}_n(\lambda_k x. H_1) \equiv \text{proc}_n(\lambda_k x. H_2)}$$

$$(C\text{-REPL}) \quad \frac{H_1 \equiv H_2}{\text{proc}_n(!H_1) \equiv \text{proc}_n(!H_2)}$$

$$(C\text{-}\alpha) \quad \text{proc}_n(\lambda_k x. H) \equiv \text{proc}_n(\lambda_k y. (H[\text{proc}_m(y)/x])) \\ \text{if } y \notin \text{free}(H), \text{sort}(x) = \text{sort}(y) = m$$

$$(C\text{-CON}) \quad \frac{C \cong C', H_i \equiv J_i, i \in \{1, \dots, n\}}{C\langle H_1, \dots, H_n \rangle \equiv C'\langle J_1, \dots, J_n \rangle}$$

where $H, H_1, J_1 \in \mathcal{S}, n \in \mathbb{N}, x \in X, S_1, S_2$ are process descriptions and C is a context with n holes where the i -th hole has sort $\text{card}(H_i) = \text{card}(J_i)$. \square

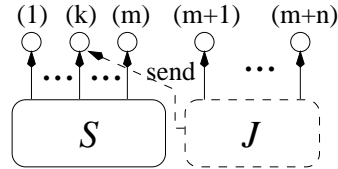
The rules of structural congruence are equivalent to the following rules, where \equiv is also defined on process descriptions:

(C-PA')	$\frac{H_1 \equiv H_2}{\lambda_n x. H_1 \equiv \lambda_n x. H_2}$	(C-REPL')	$\frac{H_1 \equiv H_2}{!H_1 \equiv !H_2}$
(C- α)	$\lambda_k x. H \equiv \lambda_k y. (H[\text{proc}_m(y)/x])$ if $y \notin \text{free}(H), \text{sort}(x) = \text{sort}(y) = m$		
(C-CON')	$\frac{H \cong H'}{H \equiv H'}$		

Table 4.1: Alternative rules of structural congruence

We will make use of the alternative definition in chapter 7 where we compare different calculi.

We now define a redex, i.e. a process with a message attached to one of its nodes. Let S be a process description and let J be a SPIDER expression. A redex $\text{Red}^{k,m,n}(S, J)$ in graphical notation is of the following form:



$$\text{Red}^{k,m,n}(S, J) := (\text{proc}_m(S), \zeta_P) \otimes (\text{mess}_{n+1}(J), \zeta_M)$$

where

$$\zeta_P : \mathbf{m} \rightarrow \mathbf{m} + \mathbf{n}, \quad \zeta_M : \mathbf{n} + \mathbf{1} \rightarrow \mathbf{m} + \mathbf{n}$$

with $\zeta_P(\chi_{\mathbf{m}}) := \lfloor \chi_{\mathbf{m}+\mathbf{n}} \rfloor_{1\dots m}$ and $\zeta_M(\chi_{\mathbf{n}+\mathbf{1}}) := \lfloor \chi_{\mathbf{m}+\mathbf{n}} \rfloor_{m+1\dots m+\mathbf{n}k}$.

We are now ready to give the reduction semantics of the calculus:

Definition 4.3.3 (Reduction Semantics) Let \rightarrow be the least relation satisfying

Reduction Rules:

$$(R-MR) \quad Red^{k,m,n}(\lambda_k x.H, J) \rightarrow H[J/x] \quad \text{if } m+n = card(H), \\ card(J) = sort(x)$$

$$(R-REPL) \quad proc_n(!H) \rightarrow proc_n(!H) \square H \quad \text{if } n = card(H)$$

$$(R-CON) \quad \frac{H_i \rightarrow H'_i}{C\langle H_1, \dots, H_i, \dots, H_n \rangle \rightarrow C\langle H_1, \dots, H'_i, \dots, H_n \rangle}$$

$$(R-EQU) \quad \frac{J \equiv H, H \rightarrow H', H' \equiv J'}{J \rightarrow J'}$$

where $H, J, H', J' \in \mathcal{S}$ and $k, m, n \in \mathbb{N}$ and $x \in X$ and C is a context with a hole of sort $card(H)$. \square

The two basic rules of reduction are (R-MR) and (R-REPL) (see figure 4.1). (R-MR) describes the reception of a message $mess_{n+1}(J)$ by a process $proc_m(\lambda_k x.H)$. The message can be received if its send-port is identical with the k -th port of the process. The content J of the message is substituted for x in H and the entire graph H is embedded into the existing graph. Note that the n ports which are attached to the message are taken over by H and are merged with the n last ports in χ_H . This corresponds exactly to the mobility of port addresses which can also be found in the π -calculus.

(R-REPL) describes the replication of a process, i.e. it can create arbitrarily many copies of its content, which are still attached to its string of external ports.

Definition 4.3.4 (Bad Redex) We say that a SPIDER expression H contains a *bad redex* if it has either a factor $Red_{k,m,n}(\lambda_k x.J_1, J_2)$ where $m+n \neq card(J_1)$ or $sort(x) \neq J_2$ or a factor $proc_n(!J)$ where $n \neq card(J)$. \square

We will now give an alternative description of reduction which will facilitate some proofs. Our aim is to show that the reduction semantics remains unchanged if we use only binary contexts in (R-CON) and allow the application of (R-EQU) at most once.

Proposition 4.3.5 *If $H \rightarrow H'$ it follows that there exist a binary context C and process graphs H_1, H_2 such that*

$$H \equiv C\langle H_1, H_2 \rangle \quad H' \equiv C\langle H'_1, H_2 \rangle$$

$$\text{and } H_1 \xrightarrow{(R-MR)} H'_1 \text{ or } H_1 \xrightarrow{(R-REPL)} H'_1$$

Proof: We proceed by induction on the reduction rules:

- In the case of (R-MR) and (R-REPL) we have $H \rightarrow H'$ and can choose the context $var_n(x_1) \oplus var_0(x_2)$ where $n := card(H)$ and $H_1 := H, H_2 := \mathbf{0}$.

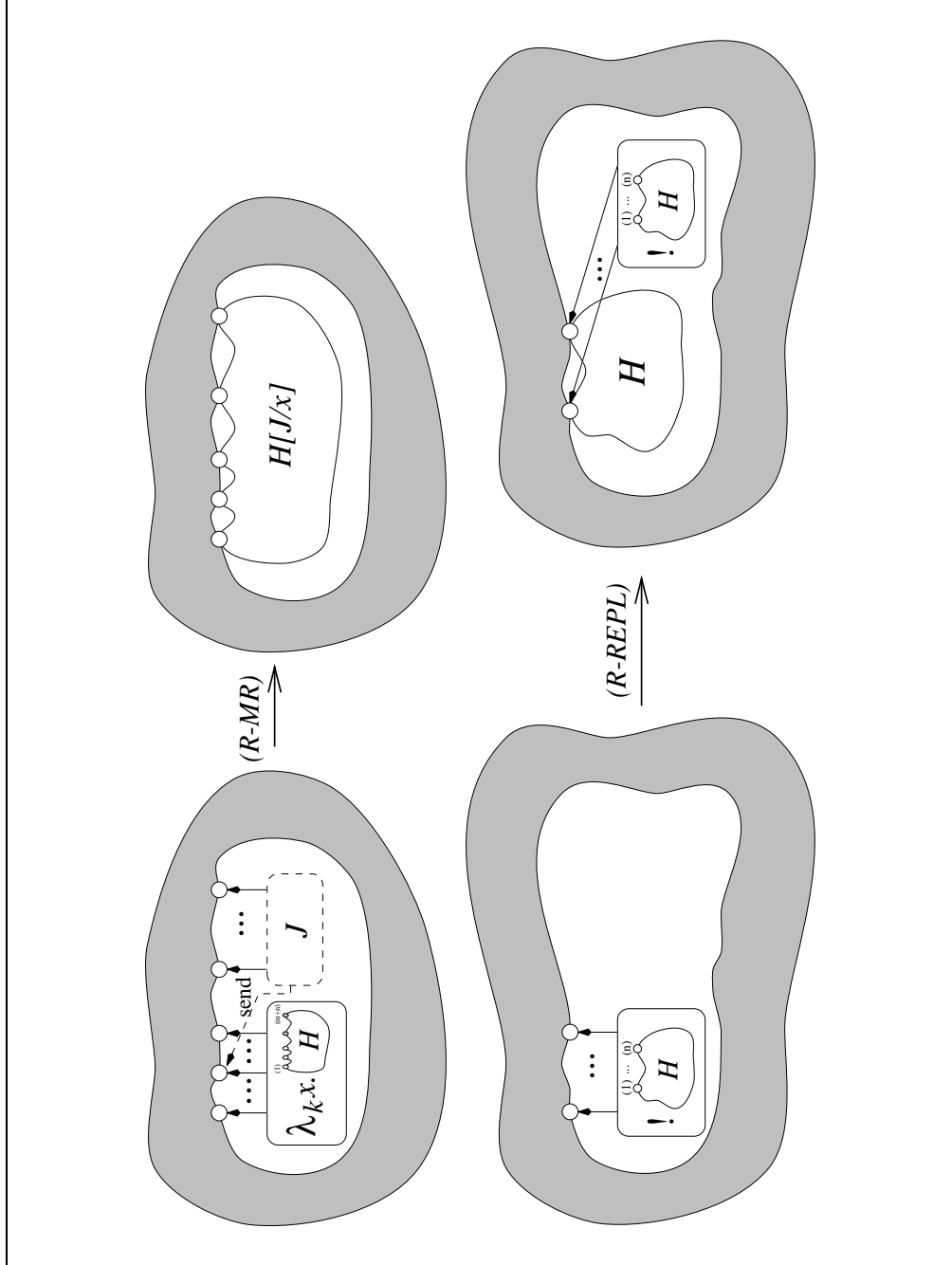


Figure 4.1: Message Reception and Replication

- In the case of (R-EQU) the proposition follows immediately with the induction hypothesis and the transitivity of \equiv .
- In the case of (R-CON) we have $H \rightarrow H'$ where

$$H = C\langle H_1, \dots, H_n \rangle \quad H' = C\langle H_1, \dots, H'_i, \dots, H_n \rangle$$

and $H_i \rightarrow H'_i$. The induction hypothesis implies that

$$H_i = D\langle J_1, J_2 \rangle \quad H'_i \equiv D\langle J'_1, J_2 \rangle$$

and $J_1 \xrightarrow{(R-MR)} J'_1$ or $J_1 \xrightarrow{(R-REPL)} J'_1$.

It follows with propositions 2.2.13 and 2.2.15 that there exist contexts \tilde{C}, \tilde{D} such that

$$\begin{aligned} & C\langle x_1, \dots, x_{i-1}, D\langle y_1, y_2 \rangle, x_{i+1}, \dots, x_n \rangle \\ & \cong \tilde{C}\langle y_1, D\langle x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n, y_2 \rangle \rangle \end{aligned}$$

Therefore

$$\begin{aligned} H & \equiv \tilde{C}\langle J_j, \tilde{D}\langle H_1, \dots, H_{i-1}, H_{i+1}, \dots, H_n, J_2 \rangle \rangle \\ H' & \equiv \tilde{C}\langle J'_j, \tilde{D}\langle H_1, \dots, H_{i-1}, H_{i+1}, \dots, H_n, J_2 \rangle \rangle \end{aligned}$$

and $J_j \rightarrow J'_j$.

□

4.4 Example: Client-Server-Interaction

Server :=

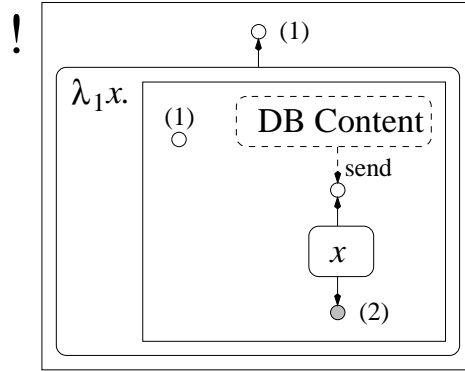


Figure 4.2: A simple database server

We model the following client-server interaction: A client sends to a server a database query with a return address. The query is itself a process, able to

process the database content and to filter the desired information. The server (see figure 4.2) will replicate itself, in order to be able to process other queries, and will then receive the query process and insert it in such a way that the query process is able to receive the database content (see figure 4.3).

In the first step the server creates a copy of itself according to (R-REPL). Both processes (the server and its copy) are still attached to the same port. Then the copy receives a message attached to the very same port whose content is the query process. (R-MR) states that the redex is replaced by the hypergraph inside the process, which means that, in this case, the port attached to the message is taken over by the second port of the hypergraph. At the same time the content of the message is substituted for the variable x .

Since this example is there to give a first glimpse onto SPIDER we do not further specify the database content, the query process or the client.

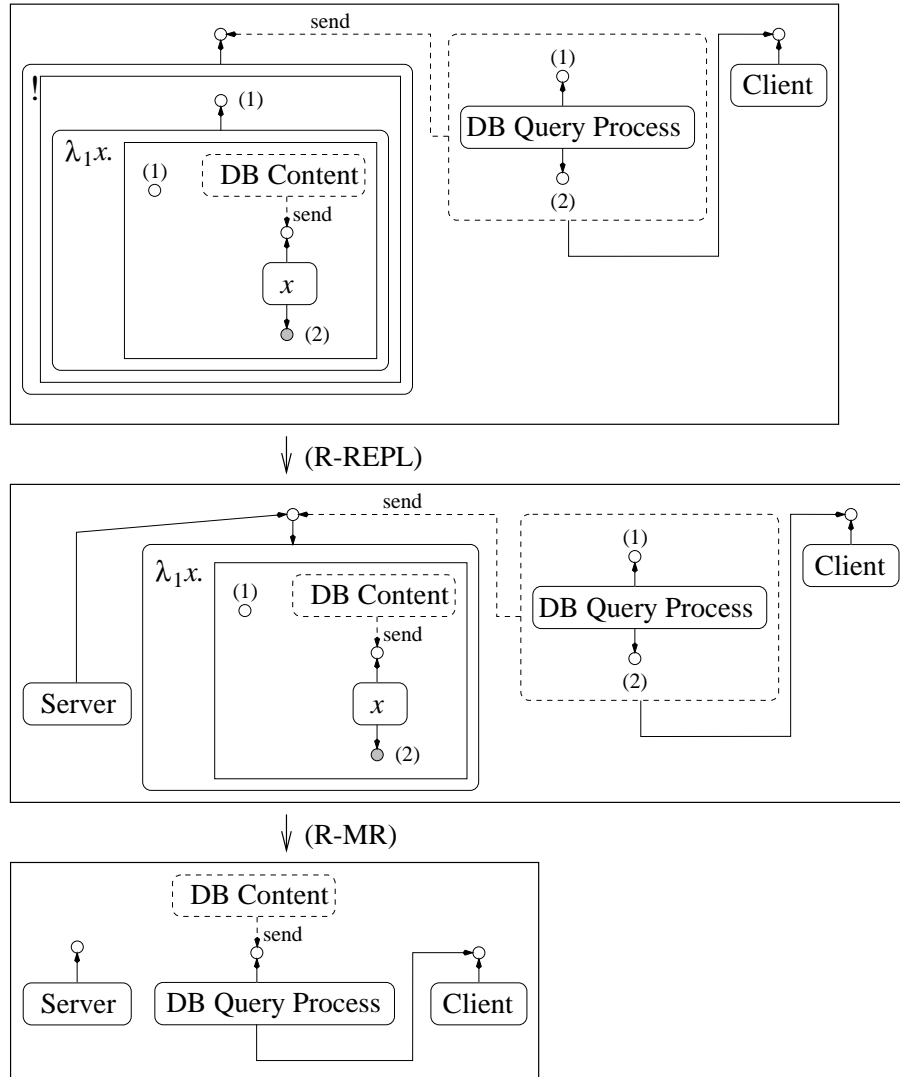


Figure 4.3: Client-Server interaction

In order to emphasize the ports of a hypergraph H in a process abstraction $\lambda_k x.H$ which are merged with the ports attached to the message, they are filled with gray.

An extended example (mail delivery system) can be found in chapter 5.

Chapter 5

Extended Example: Mail Delivery System

5.1 Informal Description

We will now show how SPIDER's features can be exploited in order to specify a Mail Delivery System.

Imagine several processes who want to communicate with one another. A communication structure connecting these processes shall be developed. It is not the aim of this chapter to describe virtual connections between processes, i.e. a scenario where there is a link between each pair of processes. The aim is to model a communication structure closer to reality by describing a physical net of components and by specifying the routing information we need to reach a certain process in the net.

Every process has a *binary mail address*. The mail addresses are organized hierarchically. For every binary string being a prefix to more than one mail address there is a process acting as a router for this prefix. A mail sent to mail address "100" for example is first sent to the topmost router who passes it on to the router representing "1". From there it is sent to the router "10" who finally passes it to the receiver process. Such a hierarchical structure is depicted as a hypergraph in figure 5.1. The processes labelled P are the actual receiver and sender processes having access only to the topmost router. The process marked grey has the mail address "100".

The set of all mail addresses is partitioned into *domains*. There is a local administrator for each domain, handling requests such as the entering and leaving of processes participating in the mail system. A process leaving the system gives up its mail address and sends a message containing its mail address to the local administrator. A process entering the system sends a request to the administrator and receives a reply with a free mail address if available.

We can see that we have to use SPIDER's ability to send ports. Later we will see that higher-order communication is also very helpful.

In the following sections we will implement a router and an administrator and will show how a message can be enriched with routing information. We will also specify a protocol for processes wishing to enter or leave the system.

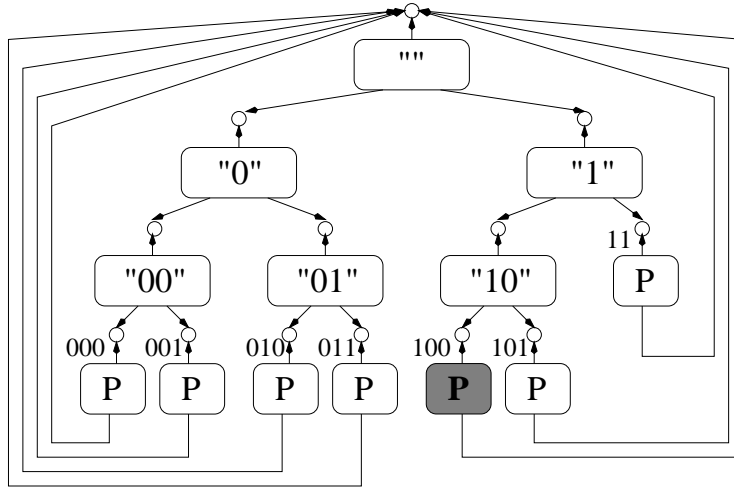


Figure 5.1: A hierarchical communication structure

5.2 Routers and Routing Information

Routing information and routers depend on one another. If we use another kind of routing information we need another kind of router which can interpret this information correctly. Messages with routing information are called *mails*.

A router is a process of cardinality 3 (see figure 5.1). It receives mails on its first source port and sends them on its second or third port depending on the routing information. It is depicted in figure 5.2.

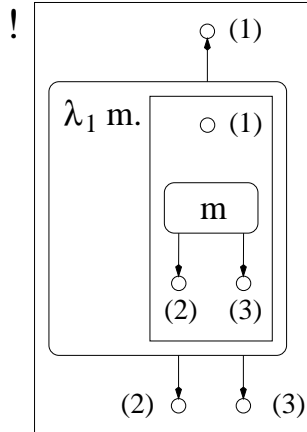


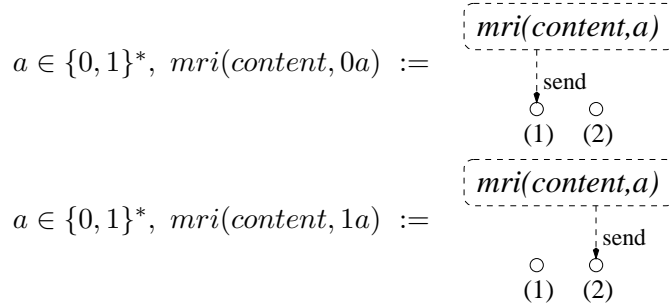
Figure 5.2: A router as a SPIDER expression

Because a router is not only used once but several times we use a replication operator $!$. Prior to receiving a mail a router has to duplicate itself in order to be available afterwards. (The ability of a router to replicate itself not only ensures its constant availability, but also solves the bottleneck problem, since there can be several routers acting concurrently.)

After replication a message is received and its content is substituted for m . This content has to be a process graph which ensures that the rest of the message is passed to the correct destination.

Let me state this more clearly: we will code both the content of a mail and its routing information into a SPIDER expression. Let $content \in \mathcal{S}$ be the actual content of the mail and let $a \in \{0, 1\}^*$ be the mail address of the recipient. $mri(content, a)$ is the corresponding **m**essage with **r**outing **i**nformation. It is defined as follows:

$mri(content, \varepsilon) := content$, where ε is the empty string



A mail with content $content$ and routing information to the mail address “100” is shown in figure 5.3.

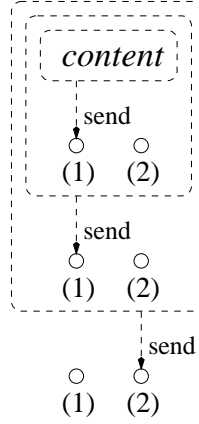


Figure 5.3: Mail with routing information to mail address “100”

Let us see what happens when we send this mail with routing information to the topmost router in figure 5.1. The reductions that result in the arrival of the content of the message are depicted in figure 5.4. Only the relevant parts of the communication structure are shown. Note that the routing information diminishes during reduction and in the end only the mail content is left.

A process has to be able to send its mail address to another process. When sending the routing information as an expression it is convenient to treat the content as a formal parameter. A process therefore sends a process abstraction as a mail address which the receiver can use to insert a content and send it

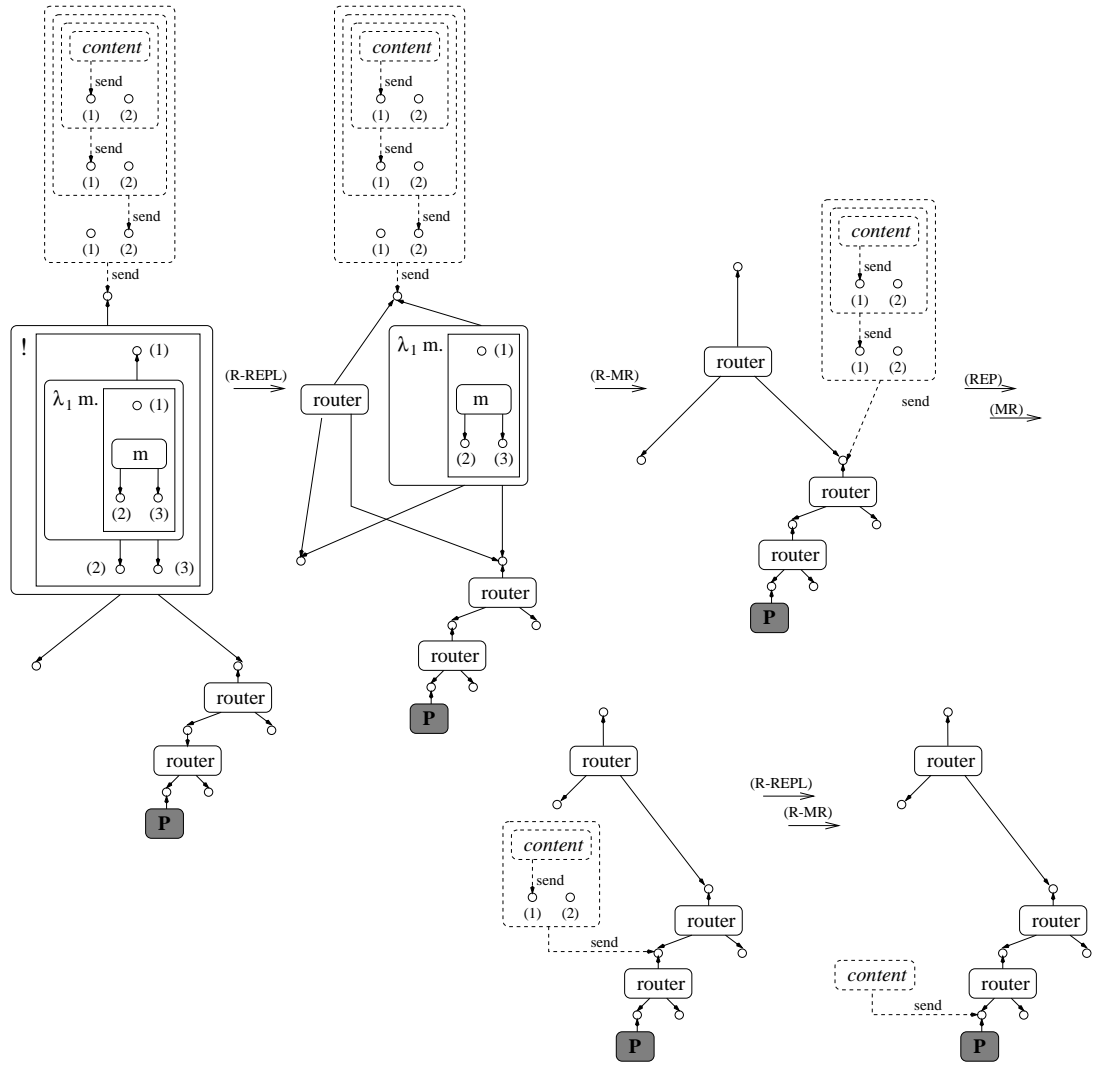


Figure 5.4: Routing of a message

back to the address. Such a process abstraction of cardinality 2 is depicted in figure 5.5 for a process with mail address a . It is also called an *address abstraction*.

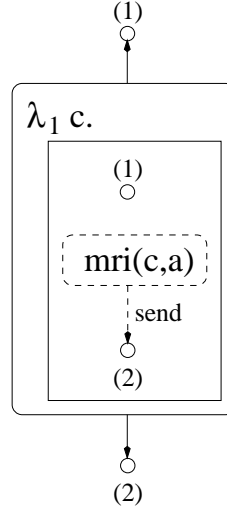


Figure 5.5: Address abstraction representing the mail address a

Figure 5.6 shows a process receiving a mail address via its first input port and afterwards sending *content* to this very mail address.

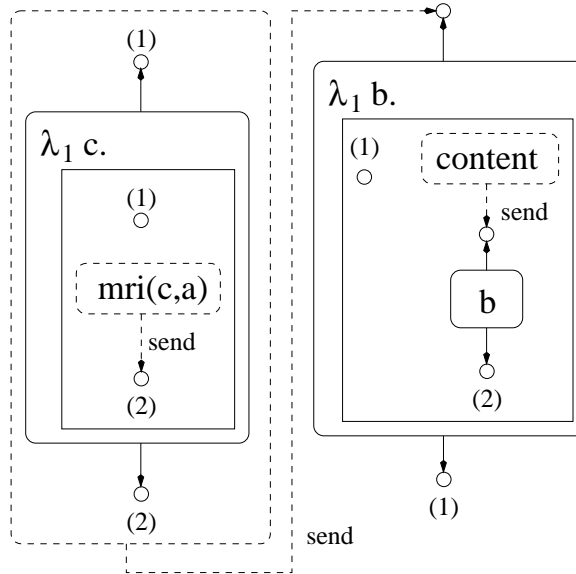


Figure 5.6: Process replying to a mail address

5.3 Local Administrators

We can establish domains of processes by grouping processes with the same prefix of their mail addresses. Every domain has a local administrator handling requests from processes who want to leave or enter the system. An administrator has cardinality 3. It receives leaving requests on its first port and entering requests on its second port. The third port is connected to the topmost router.

If a process wants to give up its mail address and leave the system it sends a message with its address abstraction to the administrator. Attached to the message is a source port which was the connection of the process to the mail system.

There might be several such messages waiting at the first source port of the administrator. The administrator replicates itself, receives one of these messages and waits for an entering request with a reply port attached to it. To this reply port it sends the free mail address with its corresponding source port. Attached to this message is also a port which establishes a connection to the topmost router.

A local administrator is depicted in figure 5.7.

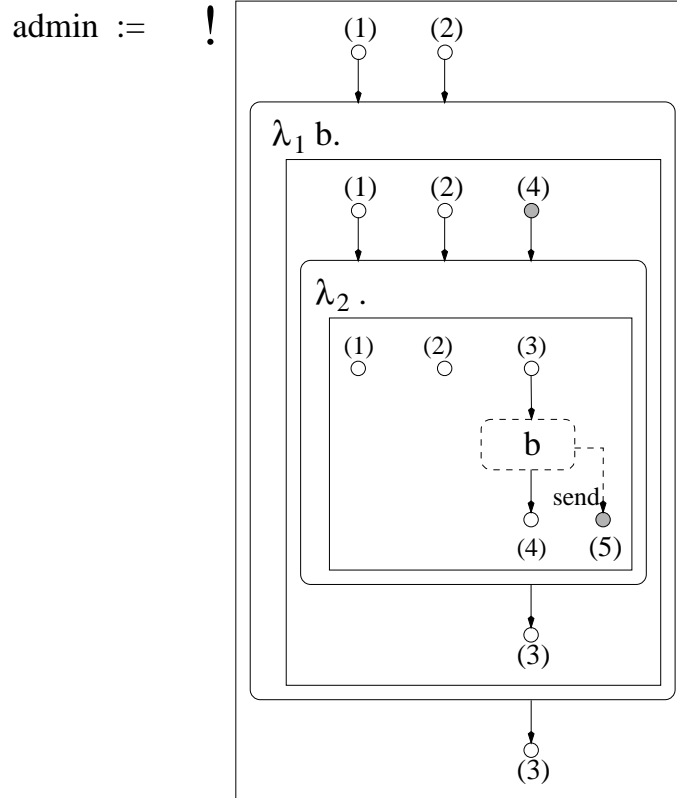


Figure 5.7: Local administrator for a domain of processes

Administrators are added to the communication structure of figure 5.1. The resulting mail system is shown in figure 5.8. The processes are grouped into

domains according to the first digit of their mail addresses. All the participants of the mail system know the first source port of the administrator in order to be able to send a leaving request. The second source port could be connected to potential participants not yet connected to the system.

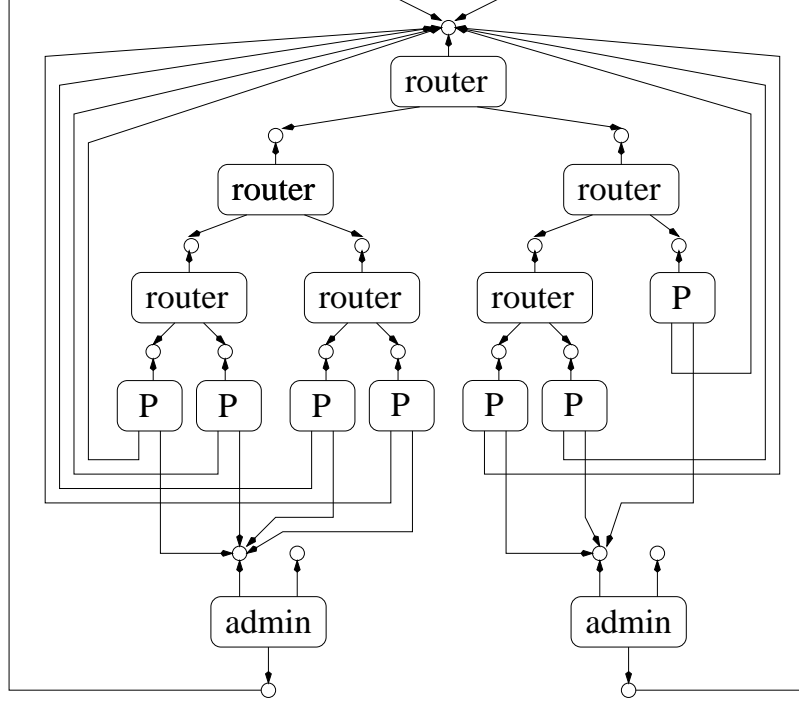


Figure 5.8: Communication structure with administrators

A process who wishes to leave the system has to obey the following protocol:

- It sends a message with its address abstraction to the first port of the local administrator.
- Attached to the message is the port via which the process has received the mails. The process must not retain any connection to this port.

A process wishing to enter the system must proceed as follows:

- It sends a message with dummy content to the second port of the administrator, signalling its request.
- Attached to the message is a reply port on which it listens out for the reply which contains the address abstraction for its new mail address.
- The reply also brings a port to which the process attaches itself in order to receive mails and a target port which connects the process with the topmost router and enables it to send mails to other participants of the system.

5.4 Possible Extensions

- **What happens if there are more potential participants than mail addresses?**

We can always enlarge the communication structure by inserting a router at the place of a participating process. The router then has to announce the availability of its two mail addresses to the local administrator.

- **What happens if a former participant keeps its mail address instead of giving it back?**

In our current scenario this situation can happen. But we might specify a router which creates a new port every time a new process is inserted. This effectively keeps the former participant from receiving mails which are not meant for it.

Chapter 6

Bisimulation and Proof Techniques

6.1 Barbed Congruence

We now show how to incorporate a very important concept of process calculi into our formalism: the notion of bisimulation (for more information on bisimulation in process calculi see section 3.5).

Structural equivalence, as defined in chapter 4 is much too fine-grained for most purposes. We do not want to distinguish processes showing the same behaviour with respect to their environment. The following notion of barbed congruence is close to a definition of Davide Sangiorgi in [San92, MS92b]. We will modify it slightly in order to avoid complications in the rest of this chapter.

It is important that this relation is a congruence. This gives us the possibility to replace a process with an equivalent process in a larger context, without changing the behaviour of the entire system.

We assume that we can only distinguish processes by checking their communication abilities (“black-box view”). Therefore we will define the notion of active ports.

Definition 6.1.1 (Active Ports) Let H be a SPIDER expression. We say that H outputs on the k -th port or respectively that $\lfloor \chi_H \rfloor_k$ is an *active output port* iff there exists a message $q \in M_H$ such that $\text{send}_H(q) = \lfloor \chi_H \rfloor_k$. We say $\text{card}(q)$ is the cardinality of this active output port.

And we say that H expects input on the k -th port or respectively that $\lfloor \chi_H \rfloor_k$ is an *active input port* iff there exists a process $p \in P_H$ such that $l_H(p) = \lambda_i x. J$ and $\lfloor s_H(p) \rfloor_i = \lfloor \chi_H \rfloor_k$. We say that $\text{card}(J) - \text{card}(p)$ is the cardinality of this active input port (= cardinality of the expected message). \square

(Note that in the definition above an input/output port may have several cardinalities. This, however, can lead to runtime errors which can be detected by a type system (see chapter 8). In a well-typed SPIDER expression, a port can have only one cardinality.)

In asynchronous calculi we are not expected to be able to observe if a process is expecting input (for more details on bisimulation in the asynchronous π -

calculus see [ACS96]). Thus we will only distinguish processes by their output capabilities.

We are now ready to define weak barbed congruence. While the following definition of weak barbed congruence is very intuitive, it turns out that it can be very hard to prove anything with respect to this definition. Therefore the rest of the chapter will be devoted to proof techniques which can greatly simplify our proofs.

Definition 6.1.2 (Weak Barbed Congruence)

A relation R is a *weak barbed congruence* on SPIDER expressions iff

- R is an equivalence
- $H R J$ implies $\text{card}(H) = \text{card}(J)$
- $H_i R J_i$ for $i \in \{1, \dots, n\}$ implies

$$C\langle H_1, \dots, H_n \rangle R C\langle J_1, \dots, J_n \rangle \quad (6.1)$$

for every discrete n -ary context $C\langle x_1, \dots, x_n \rangle$ with holes of cardinality $\text{card}(H_1), \dots, \text{card}(H_n)$.

- $H R J$ and $H \rightarrow H'$ implies the existence of an expression J' with $J \rightarrow^* J'$ and $H' R J'$
- $H R J$ and H outputs on the external port $[\chi]_k$ implies that there exists an expression J' with $J \rightarrow^* J'$, J' outputs on $[\chi]_k$ and $H R J'$.

Two SPIDER expressions H, J are called *weakly barbed congruent* ($H \approx J$) iff there is a weak barbed congruence R such that $H R J$. \square

It is not difficult to show that \approx is also a weak barbed congruence. Note also that since R is reflexive, (6.1) holds also for non-discrete contexts $C\langle x_1, \dots, x_n \rangle$.

Note: The definition above does not quite match the definition of barbed bisimulation in Sangiorgi's work [San92, MS92b]. The last condition should actually read:

- $H R J$ and H outputs on the external port $[\chi]_k$ implies that there exists an expression J' with $J \rightarrow^* J'$ and J' outputs on $[\chi]_k$.

This means that our definition is stronger and it is not quite clear if both definitions coincide. We chose the definition above since it is easier to handle and since it enables us to prove the \Rightarrow -part of proposition 6.3.4. And since bisimulation is not the main objective of this paper, we consider it appropriate to keep matters clear and manageable.

We will now conduct our first step to simplify proofs for the weak barbed equivalence. The main point in this case is to avoid the transitivity of R .

Proposition 6.1.3 *Let $R \subseteq \mathcal{S} \times \mathcal{S}$ be a reflexive and symmetric relation on SPIDER expressions and let R^* be its transitive closure. Furthermore:*

- $H R J$ implies $\text{card}(H) = \text{card}(J)$
- $H_i R J_i$ for $i \in \{1, \dots, n\}$ implies

$$C\langle H_1, \dots, H_n \rangle R^* C\langle J_1, \dots, J_n \rangle$$

for every discrete n -ary context $C\langle x_1, \dots, x_n \rangle$ with holes of cardinality $\text{card}(H_1), \dots, \text{card}(H_n)$.

- $H R J$ and $H \rightarrow H'$ implies the existence of an expression J' with $J \rightarrow^* J'$ and $H' R J'$.
- $H R J$ and H outputs on the external port $[\chi]_k$ implies that there exists an expression J' with $J \rightarrow^* J'$, J' outputs on $[\chi]_k$ and $H R^* J'$.

Then R^* is a weak barbed congruence.

Proof: R^* can be characterised in the following way:

$$H R^* J \iff \exists n \in \mathbb{N}, K_1, \dots, K_n \in \mathcal{S} : H R K_1 R \dots R K_n R J$$

- If $H R^* J$ then $\text{card}(H) = \text{card}(K_1) = \dots = \text{card}(K_n) = \text{card}(J)$.
- If $H_i R^* J_i$, $i \in \{1, \dots, n\}$ it follows that there are process graphs K_i^j , $i \in \{1, \dots, n\}$, $j \in \{1, \dots, m_i\}$ such that

$$H_i R K_i^1 R \dots R K_i^{m_i} R J_i$$

Since R is reflexive we can choose the K_i^j such that $m_1 = \dots = m_n =: m$.

It follows that

$$\begin{aligned} C\langle H_1, \dots, H_n \rangle R^* C\langle K_1^1, \dots, K_n^1 \rangle R^* \dots R^* C\langle K_1^m, \dots, K_n^m \rangle \\ R^* C\langle J_1, \dots, J_n \rangle \end{aligned}$$

- We will show that $H R^* J$, $H \rightarrow H'$ implies $J \rightarrow^* J'$, $H' R^* J'$ by induction on n where n is the number of R -steps between H and J : the case $H R J$ is obvious. If $H R^n K R J$ and $H \rightarrow H'$, the induction hypothesis implies that $K \rightarrow^* K'$ and $H' R^* K'$. By induction on the length of $K \rightarrow^* K'$ we can show the existence of J' with $J \rightarrow^* J'$ and $K' R J'$.

- We will show again by induction on n that if $H R^* J$ and H outputs on the k -th port then there exists an expression J' with $J \rightarrow^* J'$, J' also outputs on the k -th port and $H' R^* J'$: The case $H R J$ is obvious.

If $H R^n K R J$ and H outputs on the k -th output port, the induction hypothesis implies that $K \rightarrow^* K'$, K' outputs on the k -th port and $H R^* K'$. We know (see above) that there exists a J' with $J \rightarrow^* J'$ and $K' R J'$. Since K' outputs on the k -th port it follows that $J' \rightarrow^* J''$, J'' outputs on the k -th output port and $K' R^* J''$ which implies that $H R^* J''$.

□

6.2 Garbage Collection

In this section we present a first application of barbed congruence: we show that garbage, i.e. edges and nodes that have no connection to external ports, can be removed from a process graph without changing its behaviour. This fact will be also needed in the proof of proposition 6.3.4.

Definition 6.2.1 (Garbage) Let H be a SPIDER expression such that $H \equiv H' \oplus G_H$ where $\text{card}(G_H) = 0$. Then G_H is called *garbage* of H . \square

We now show that removing this garbage yields another process which is weakly barbed congruent to the original process.

Proposition 6.2.2 (Garbage Collection) *Let H be a SPIDER expression with $H \equiv H' \oplus G_H$ such that $\text{card}(G_H) = 0$. Then $H \approx H'$.*

Proof: We define

$$R := \{(K \oplus G_H, K \oplus G_J) \mid K \in \mathcal{S}, \text{card}(G_H) = \text{card}(G_J) = 0\}$$

It is obvious that R is reflexive and symmetric and that $H R H'$. We will now show that all conditions of proposition 6.1.3 are satisfied:

- If $K \oplus G_H R K \oplus G_J$ with $\text{card}(G_H) = \text{card}(G_J) = 0$ then

$$\text{card}(K \oplus G_H) = \text{card}(K) = \text{card}(K \oplus G_J)$$

- Let $K \oplus G_H \rightarrow H'$. Then either $H' \equiv K' \oplus G_H$ and $K \rightarrow K'$. In this case $K \oplus G_J \rightarrow K' \oplus G_J R H'$.

If on the other hand $H' \equiv K \oplus G'_H$ and $G_H \rightarrow G'_H$ (and thus $\text{card}(G'_H) = 0$) then $K \oplus G_J \rightarrow^0 K \oplus G_J R K \oplus G'_H$. Since R is symmetric it follows that $K \oplus G'_H R K \oplus G_J$.

- Let $K_i \oplus G_H^i R K_i \oplus G_J^i$ and let $C\langle x_1, \dots, x_n \rangle$ be a given context. It follows with propositions 2.2.13 and 2.2.15 that there are contexts $C'\langle y_1, \dots, y_{n+1} \rangle$ and $C''\langle z_1, \dots, z_n \rangle$ with

$$\begin{aligned} C\langle K_1 \oplus G_H^1, \dots, K_n \oplus G_H^n \rangle &\equiv C'\langle K_1, \dots, K_n, C''\langle G_H^1, \dots, G_H^n \rangle \rangle \\ C\langle K_1 \oplus G_J^1, \dots, K_n \oplus G_J^n \rangle &\equiv C'\langle K_1, \dots, K_n, C''\langle G_J^1, \dots, G_J^n \rangle \rangle \end{aligned}$$

With proposition 2.2.13 we can show that

$$C\langle K_1, \dots, K_n, G \rangle \equiv C\langle K_1, \dots, K_n, \mathbf{0} \rangle \oplus G$$

if $\text{card}(G) = 0$. It follows immediately that

$$C\langle K_1 \oplus G_H^1, \dots, K_n \oplus G_H^n \rangle R C\langle K_1 \oplus G_J^1, \dots, K_n \oplus G_J^n \rangle$$

- Let $K \oplus G_H R K \oplus G_J$ and $K \oplus G_H$ outputs on the k -th port, then, of course, K outputs on the k -th port and the same is true for $K \oplus G_J$.

\square

6.3 Normal Bisimulation

Bisimulation proofs can be much more complicated than the proof of proposition 6.2.2. The main problem is the great number of contexts we have to deal with. But, as it turns out, we do not need to consider arbitrary contexts. Only messages received from and sent to the exterior are really important. This is similar to the bisimulation equivalence based on labelled transition semantics.

Furthermore we consider bisimulation up-to-context. That is, as in proposition 6.1.3 there is a stronger relation R on the left side and a weaker relation \hat{R} on the right side of an implication.

Definition 6.3.1 Let $R \subseteq \mathcal{S} \times \mathcal{S}$ be a symmetric relation on SPIDER expression. We define

$$\hat{R} := \{(C\langle H_1, \dots, H_n \rangle, C\langle J_1, \dots, J_n \rangle) \mid C \text{ is a } n\text{-ary discrete context, } H_i R J_i\}$$

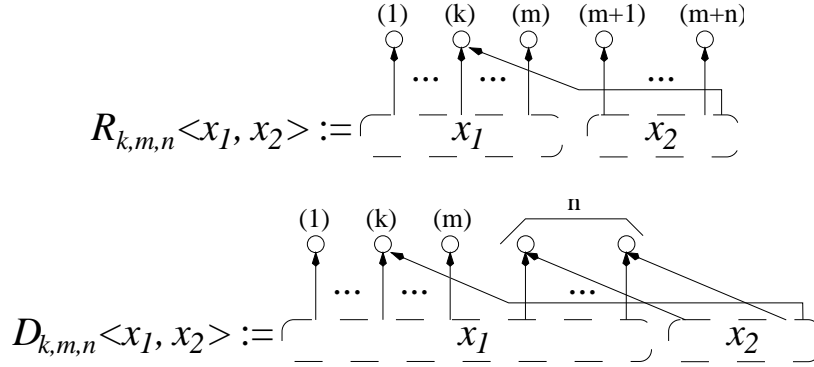
□

Proof: Immediate with proposition 2.2.13. □

Lemma 6.3.2 (Properties of \hat{R}) Let R be a symmetric relation. Let H_i, J_i be SPIDER expressions with $H_i \hat{R} J_i$ and let C be a discrete n -ary context with holes of cardinality $\text{card}(H_1), \dots, \text{card}(H_n)$. Then

$$C\langle H_1, \dots, H_n \rangle \hat{R} C\langle J_1, \dots, J_n \rangle$$

For the following definitions we need two special contexts: $R_{k,m,n}$ (representing a process x_1 receiving a message x_2) and $D_{k,m,n}$ (representing a process sending a message x_2 , where x_1 is the rest of the process).



We are now ready to define the notion of normal bisimulation.

Definition 6.3.3 (Normal Bisimulation) A relation R on SPIDER expressions is a *normal bisimulation* iff

- R is reflexive and symmetric
- $H R J$ implies $\text{card}(H) = \text{card}(J)$

- $H R J$ and $H \rightarrow H'$ implies the existence of an expression J' with $J \rightarrow^* J'$ and $H' \hat{R} J'$
- If $H R J$ and $H \equiv D_{k,m,n}\langle H', \text{mess}_{n+1}(M_H) \rangle$ where $H', M_H \in \mathcal{S}$ then there exist $K', M_K \in \mathcal{S}$ such that

$$\begin{array}{ccc} H' & \hat{R} & K' \\ \text{mess}_{n+1}(M_H) & R & \text{mess}_{n+1}(M_K) \\ J & \rightarrow^* & D_{k,m,n}\langle K', \text{mess}_{n+1}(M_K) \rangle \end{array}$$

(If H outputs a message on the k -th port, J can output an equivalent message, and the rest of the expressions is also equivalent.)

- If $H R J$ and $[\chi]_k$ is an active input port (of cardinality n) in H then $J \rightarrow^* K$ such that for every pair $\text{mess}_{n+1}(M) R \text{mess}_{n+1}(N)$:

$$R_{k,m,n}\langle H, \text{mess}_{n+1}(M) \rangle \hat{R} R_{k,m,n}\langle K, \text{mess}_{n+1}(N) \rangle$$

where $m := \text{card}(H)$, $n \in \mathbb{N}$.

(If H expects a message on the k -th port, H and J behave the same if they receive equivalent messages from the environment.)

Two SPIDER expressions H, J are called *normally bisimilar* ($H \approx_N J$) iff there is a normal bisimulation R such that $H \hat{R}^* J$ \square

Proposition 6.3.4 *Barbed congruence and normal bisimulation are identical, i.e. for every $H, J \in \mathcal{S}$:*

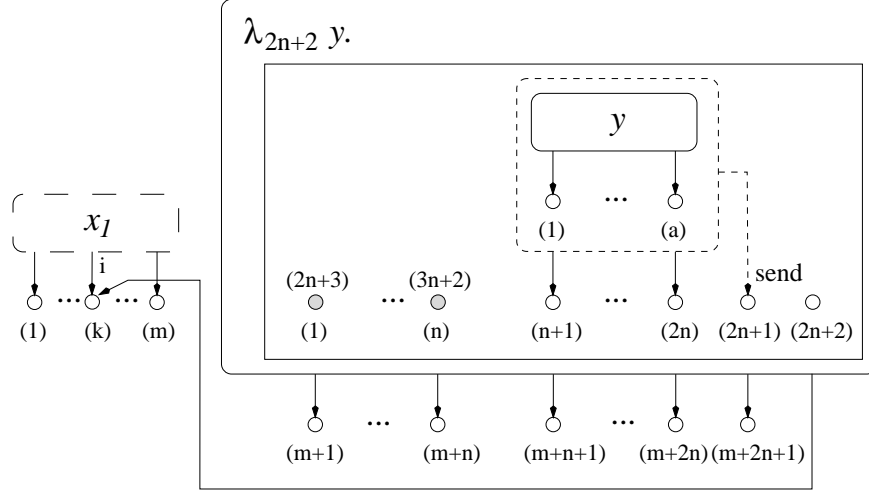
$$H \approx J \iff H \approx_N J$$

Proof:

\Rightarrow : Let $H \approx J$. We will show that the conditions of definition 6.3.3 are satisfied for $R = \approx$. Since \approx is a bisimulation it is easy to see that it satisfies the first three conditions of definition 6.3.3. The fifth condition is satisfied since \approx is a congruence by definition. We will now check the fourth condition:

Let $H \equiv D_{k,m,n}\langle H', \text{mess}_{n+1}(M_H) \rangle$ where $\text{card}(M_H) =: a$.

We define the following context $C\langle x_1 \rangle$:



Since $H \approx J$, we conclude that also $C\langle H \rangle \approx C\langle J \rangle$. Since in H a message of content M_H is sent to $\lfloor \chi \rfloor_k$, $C\langle H \rangle$ can perform a reduction where the process on the right hand side receives this very message. It is easy to see that the resulting process has the form $H' \oplus \text{mess}_{n+1}(M_H) =: \tilde{H}$.

Since $H' \oplus \text{mess}_{n+1}(M_H)$ outputs on the $(m+2n+1)$ -th port we conclude that $C\langle J \rangle \rightarrow^* \tilde{J} \approx H$ and \tilde{J} also outputs on the $(n+2n+1)$ -th port. This works only if at some time in the reduction process a message is received by the right hand process in C . This implies that

$$\begin{aligned} C\langle J \rangle &\rightarrow^* C\langle D_{k,m,n}\langle K, \text{mess}_{n+1}(M_K) \rangle \rangle \rightarrow K \oplus \text{mess}_{n+1}(M_K) \\ &\rightarrow^* K' \oplus \text{mess}_{n+1}(M_K) \equiv \tilde{J} \end{aligned}$$

It follows that

$$\begin{aligned} C\langle J \rangle &\rightarrow^* C\langle D_{k,m,n}\langle K, \text{mess}_{n+1}(M_K) \rangle \rangle \rightarrow^* D_{k,m,n}\langle K', \text{mess}_{n+1}(M_K) \rangle \\ &\rightarrow K' \oplus \text{mess}_{n+1}(M_K) \end{aligned}$$

Since $H' \oplus \text{mess}_{n+1}(M_H) \approx K' \oplus \text{mess}_{n+1}(M_K)$ and \approx is a congruence it follows that

$$\begin{aligned} H' \oplus \sigma_\varepsilon(\text{mess}_{n+1}(M_H)) &\equiv \sigma_{1\dots m+n}(H' \oplus \text{mess}_{n+1}(M_H)) \\ &\approx \sigma_{1\dots m+n}(K' \oplus \text{mess}_{n+1}(M_K)) \equiv H' \oplus \sigma_\varepsilon(\text{mess}_{n+1}(M_K)) \\ \sigma_\varepsilon(H') \oplus \text{mess}_{n+1}(M_H) &\equiv \sigma_{m+n+1\dots m+2n+1}(H' \oplus \text{mess}_{n+1}(M_H)) \\ &\approx \sigma_{1\dots m+n}(K' \oplus \text{mess}_{n+1}(M_K)) \equiv \sigma_\varepsilon(H') \oplus \text{mess}_{n+1}(M_K) \end{aligned}$$

With proposition 6.2.2 it follows that $H' \approx K'$ and $\text{mess}_{n+1}(M_H) \approx \text{mess}_{n+1}(M_K)$.

\Leftarrow : Let R be a normal bisimulation. We will show that \hat{R} is a weak barbed congruence with the help of proposition 6.1.3.

- \hat{R} is symmetric since R is symmetric.

- $H \hat{R} J$ implies $H \equiv C\langle H_1, \dots, H_n \rangle$ and $J \equiv C\langle J_1, \dots, J_n \rangle$ for some context $C\langle x_1, \dots, x_n \rangle$. It follows that $\text{card}(H) = \text{card}(C\langle x_1, \dots, x_n \rangle) = \text{card}(J)$.
- It follows immediately with lemma 6.3.2 that $H_i \hat{R} J_i$ implies

$$C\langle H_1, \dots, H_n \rangle \hat{R} C\langle J_1, \dots, J_n \rangle$$

- Let $H \hat{R} J$, ($H \equiv C\langle H_1, \dots, H_n \rangle$, $J \equiv C\langle J_1, \dots, J_n \rangle$, $H_i R J_i$), H outputs on χ_k .

Then there is a process p of C , labelled x_i with $\lfloor s_C(p) \rfloor_j = \lfloor \chi \rfloor_k$ for some j and H_i outputs on $\lfloor \chi \rfloor_j$, i.e. $H_i \equiv D_{j, m_i, n_i} \langle H'_i, \text{mess}_{n_i+1}(M_{H_i}) \rangle$. Since $H_i R J_i$ we conclude that

$$J_i \rightarrow^* K_i \equiv D_{j, m_i, n_i} \langle K'_i, \text{mess}_{n_i+1}(M_{K_i}) \rangle$$

with $H'_i \hat{R} K'_i$ and $\text{mess}_{n+1}(M_{H_i}) R \text{mess}_{n+1}(M_{K_i})$.

It follows with lemma 6.3.2 that $H_i \hat{R} K_i$ and that

$$H \equiv C\langle H_1, \dots, H_n \rangle \hat{R} C\langle J_1, \dots, J_{i-1}, K_i, J_{i+1}, \dots, J_n \rangle =: K$$

Furthermore $J \rightarrow^* K$ and since K_i is active in the j -th output port and $\lfloor s_C(p) \rfloor_j = \lfloor \chi \rfloor_k$ it follows that K is active in the k -th output port.

- Let $H \hat{R} J$, ($H \equiv C\langle H_1, \dots, H_n \rangle$, $J \equiv C\langle J_1, \dots, J_n \rangle$, $H_i R J_i$), and let $H \rightarrow H'$. We have to show that $J \rightarrow^* K$ and $H' \hat{R} K$.

There are two cases:

- The reduction takes place in one of the H_i , i.e. $H_i \rightarrow H'_i$. In this case the properties of R assure that $J_i \rightarrow^* K_i$ and $H'_i \hat{R} K_i$. It follows with lemma 6.3.2 that

$$\begin{aligned} C\langle J_1, \dots, J_n \rangle &\rightarrow^* \\ C\langle J_1, \dots, J_{i-1}, K_i, J_{i+1}, \dots, J_n \rangle &\hat{R} C\langle H_1, \dots, H_n \rangle \end{aligned}$$

- The reduction takes place in $H \equiv C\langle H_1, \dots, H_n \rangle$ and H_i receives a message of H_j . H_i expects an input on $\lfloor \chi_{H_i} \rfloor_{k_i}$ and H_j outputs on $\lfloor \chi_{H_j} \rfloor_{k_j}$. That is

$$H_j \equiv D_{k_j, m_j, n} \langle H'_j, \text{mess}_{n+1}(M_{H_j}) \rangle$$

We assume that C has edges e_i, e_j with $l_C(e_i) = x_i$, $l_C(e_j) = x_j$ and $\lfloor s_C(e_i) \rfloor_{k_i} = \lfloor s_C(e_j) \rfloor_{k_j}$.

We can show that there exists a context $C'\langle x_1, \dots, x_n \rangle$ (with $m_j + n_j = \text{sort}(x_j)$, $m_j = \text{sort}(x_i)$ and $\text{sort}(x_l) = \text{sort}(y_l)$ if $l \neq i, j$) such that

$$\begin{aligned} &C\langle x_1, \dots, x_{j-1}, D_{k_j, m_j, n} \langle x_j, x_{n+1} \rangle, x_{j+1}, \dots, x_n \rangle \\ &\equiv C'\langle x_1, \dots, x_{i-1}, R_{k_i, m_i, n} \langle x_i, x_{n+1} \rangle, x_{i+1}, \dots, x_n \rangle \end{aligned}$$

This implies that

$$H \equiv C' \langle H_1, \dots, R_{k_i, m_i, n} \langle H_i, \text{mess}_{n+1}(M_{H_j}) \rangle, \dots, H'_j, \dots, H_n \rangle$$

Since $H_j R J_j$ and H_j outputs on $\lfloor \chi \rfloor_{k_j}$ it follows that $J_j \rightarrow^* K_j$ where $K_j \equiv D_{k_j, m_j, n} \langle K'_j, \text{mess}_{n+1}(M_{K_j}) \rangle$ and

$$\begin{array}{ccccc} H'_j & \hat{R} & K'_j & & \\ \text{mess}_{n+1}(M_{H_j}) & R & \text{mess}_{n+1}(M_{K_j}) & & \end{array}$$

Since $H_i R J_i$ and H_i expects an input on $\lfloor \chi \rfloor_{k_i}$ it follows that $J_i \rightarrow^* K_i$ such that

$$R_{k_i, m_i, n} \langle H_i, \text{mess}_{n+1}(M_{H_j}) \rangle \hat{R} R_{k_i, m_i, n} \langle K_i, \text{mess}_{n+1}(M_{K_j}) \rangle$$

Therefore

$$\begin{aligned} J &\rightarrow^* C \langle J_1, \dots, K_i, \dots, K_j, \dots, J_n \rangle \\ &\equiv C \langle J_1, \dots, K_i, \dots, D_{k_j, m_j, n} \langle K'_j, \text{mess}_{n+1}(M_{K_j}) \rangle, \dots, J_n \rangle \\ &\equiv C' \langle J_1, \dots, R_{k_j, m_j, n} \langle K_i, \text{mess}_{n+1}(M_{K_j}) \rangle, \dots, K'_j, \dots, J_n \rangle \\ &\hat{R} \quad H \end{aligned}$$

□

6.4 Example: Simulating Replication

We now show the usefulness of normal bisimulation by simulating the replication operator. This actually means that the replication operator is not necessary in order to obtain full computational power in a higher-order calculus. This result is not unexpected, and a similar result was shown for the fixed-point operator by Bent Thomsen in [Tho95] for the calculus of higher-order communicating systems CHOCS.

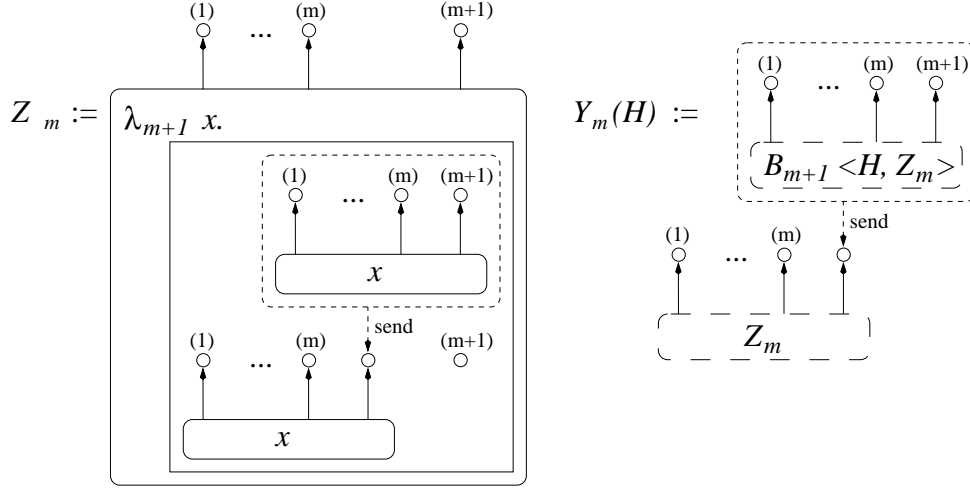
The corresponding expression in the λ -calculus is the paradoxical combinator Y .

It is, however, sensible to keep the replication operator in order to facilitate programming and in order to retain full computational power when we restrict SPIDER to its non-higher-order fragment.

We will need the following context:

$$B_{m+1} \langle x_1, x_2 \rangle := \begin{array}{c} \begin{array}{ccc} (1) & \dots & (m) & (m+1) \\ \circ & & \circ & \circ \end{array} \\ \begin{array}{c} \diagup \quad \quad \quad \diagdown \\ \text{---} \quad \quad \quad \text{---} \\ \text{---} \quad \quad \quad \text{---} \end{array} \\ \begin{array}{cc} \text{---} & \text{---} \\ x_1 & x_2 \end{array} \end{array}$$

Furthermore:



where $m := \text{card}(H)$.

And the only reduction $Y_m(H)$ can perform is

$$Y_m(H) \rightarrow \begin{array}{c} \begin{array}{c} (1) \quad \dots \quad (m) \quad (m+1) \\ \uparrow \quad \quad \quad \uparrow \quad \uparrow \\ \text{---} \quad \text{---} \quad \text{---} \quad \text{---} \\ \boxed{B_{m+1} \langle H, Z_m \rangle} \end{array} \\ \text{send} \\ \begin{array}{c} (1) \quad \dots \quad (m) \quad (m+1) \\ \uparrow \quad \quad \quad \uparrow \quad \uparrow \\ \text{---} \quad \text{---} \quad \text{---} \quad \text{---} \\ \boxed{B_{m+1} \langle H, Z_m \rangle} \end{array} \end{array} \equiv (H \square Y_m(H)) \oplus \bar{\mathbf{1}}$$

Proposition 6.4.1 (Simulating Replication) *Let H be a SPIDER expression with $m = \text{card}(H)$.*

$$Y_m(H) \cong \text{proc}_m(!H)$$

Proof: In order to show that $Y_m(H) \approx \text{proc}_m(!H)$ we have to show that the reflexive and symmetric closure of

$$R := \{(Y_m(H) \oplus \bar{\mathbf{n}}, \text{proc}_m(!H)) \mid n \in \mathbb{N}\}$$

is a normal bisimulation.

The first two conditions are obvious and the last two need not be checked since none of the processes contains any active ports. So only the third condition remains: for the reflexive part of R this is obvious.

If $Y_m(H) \oplus \bar{\mathbf{n}} \rightarrow (H \square Y_m(H)) \oplus \overline{\mathbf{n} + \mathbf{1}} \equiv H \square (Y_m(H) \oplus \overline{\mathbf{n} + \mathbf{1}})$ it follows that $\text{proc}_m(!H) \rightarrow H \square \text{proc}_m(!H)$ and

$$H \square (Y_m(H) \oplus \overline{\mathbf{n} + \mathbf{1}}) \hat{R} H \square \text{proc}_m(!H)$$

Finally, the case $proc_m(!H) \rightarrow H \sqcap proc_m(!H)$ is nearly analogous:

$$Y_m(H) \oplus \bar{n} \rightarrow H \sqcap (Y_m(H) \oplus \overline{\mathbf{n} + \mathbf{1}}) \hat{R} H \sqcap proc_m(!H)$$

□

Chapter 7

Comparison with other Formalisms

7.1 How to Compare Calculi

In this chapter we will relate SPIDER to important calculi, like the λ -calculus and the π -calculus. We will encode expressions from these calculi into SPIDER. To be able to show that such a translation is in some way “correct” we need a notion of what it means that a translation is preserving the semantics of a calculus. The following definitions are inspired by G. Boudol [Bou89] and appear, in a similar form, also in [PS93].

Definition 7.1.1 (Strong (Weak) Simulation)

Let $A = (\mathcal{A}, \rightarrow_A)$, $B = (\mathcal{B}, \rightarrow_B)$ be two calculi (or transition systems) where \mathcal{A}, \mathcal{B} are sets of expressions and $\rightarrow_A, \rightarrow_B$ are reduction relations. Furthermore there are equivalence relations \equiv_A, \equiv_B and we demand that in both calculi there is the rule

$$\frac{x \equiv x', y \equiv y', x \rightarrow y}{x' \rightarrow y'}$$

Let $a, a' \in \mathcal{A}$, $b, b' \in \mathcal{B}$. We say that B *strongly simulates* A if there exists a mapping $\Delta : \mathcal{B}' \rightarrow \mathcal{A}$ with $\mathcal{B}' \subseteq \mathcal{B}$ such that:

$$\forall a \in \mathcal{A} \exists b \in \mathcal{B}' : \quad \Delta(b) \equiv_A a \quad (7.1)$$

$$b \equiv_B b' \Rightarrow \Delta(b) \equiv_A \Delta(b') \quad (7.2)$$

$$b \rightarrow_B b' \Rightarrow \Delta(b) \rightarrow_A \Delta(b') \quad (7.3)$$

$$\Delta(b) \rightarrow_A a' \Rightarrow \exists b' \in \mathcal{B}' : b \rightarrow_B b', \Delta(b') \equiv_A a' \quad (7.4)$$

A is *weakly simulated* by B if we replace $\rightarrow_A, \rightarrow_B$ by $\rightarrow_A^*, \rightarrow_B^*$ in the definition above. \square

It is possible to gain simulations out of existing simulations:

Proposition 7.1.2 (Transitivity of Simulation)

If A can be strongly (weakly) simulated by B and B can be strongly (weakly) simulated by C then A can be strongly (weakly) simulated by C .

Proof: Straightforward by composition of the Δ -mappings. \square

Let us now consider a special case where we can show simulation with a mapping $\Theta : \mathcal{A} \rightarrow \mathcal{B}$ instead of a mapping $\Delta : \mathcal{B}' \rightarrow \mathcal{A}$. Note that the conditions given below are less general than the conditions in definition 7.1.1.

Proposition 7.1.3 *Let A, B be calculi as defined in definition 7.1.1. And let $\Theta : \mathcal{A} \rightarrow \mathcal{B}$ be a mapping satisfying:*

$$\Theta(a) \equiv_B \Theta(a') \iff a \equiv_A a' \quad (7.5)$$

$$a \rightarrow_A a' \implies \Theta(a) \rightarrow_B \Theta(a') \quad (7.6)$$

$$\Theta(a) \rightarrow_B b' \implies \exists a' \in \mathcal{A} : a \rightarrow_A a', \Theta(a') \equiv_B b' \quad (7.7)$$

Then A can be strongly simulated by B .

Proof: Let $\mathcal{B}' := \Theta(\mathcal{A})$. Since $\Theta|_{\mathcal{B}'}$ is a bijection from the equivalence classes of \mathcal{A} into the equivalence classes of \mathcal{B}' (see condition 7.5) there exists a mapping $\Delta : \mathcal{B}' \rightarrow \mathcal{A}$ with

$$b \equiv_B b' \implies \Delta(b) \equiv_A \Delta(b')$$

$$\Theta(\Delta(b)) \equiv_B b \text{ for every } b \in \mathcal{B}'$$

$$\Delta(\Theta(a)) \equiv_A a \text{ for every } a \in \mathcal{A}$$

This immediately implies (7.1) and (7.2).

We will now show that the other two conditions of definition 7.1.1 are satisfied:

(7.3) Let $b \rightarrow_B b'$. We define $a := \Delta(b)$ with $\Theta(a) \equiv b$. It follows that $\Theta(a) \rightarrow_B b'$. (7.7) implies that there exists a a' such that $a \rightarrow_A a'$ and $\Theta(a') \equiv b'$. Furthermore $\Delta(b') \equiv_A a'$.

It follows that

$$\Delta(b) \equiv_A a \rightarrow_A a' \equiv_A \Delta(b')$$

(7.4) Let $\Delta(b) \rightarrow_A a'$. With 7.6 it follows that

$$b \equiv_B \Theta(\Delta(b)) \rightarrow \Theta(a')$$

If we define $b' := \Theta(a')$ it follows that $b \rightarrow b'$ and $\Delta(\Theta(a')) \equiv_A a'$.

\square

When dealing with process calculi it is also desirable to compare the input/output capabilities of two related expressions and make sure that they are the same.

Normally it is expected that the translation is *fully abstract*, i.e. that it preserves the semantical equivalences of the different calculi. That is if \approx_A, \approx_B are the semantical equivalences (e.g. some sort of bisimulation or barbed congruence) we expect that

$$b \approx_B b' \iff \Delta(b) \approx_A \Delta(b')$$

We will now assume that in both calculi the semantical equivalence is weak barbed congruence. In order to be able to define barbed congruence in a calculus A we will assume that there are n -ary contexts c_A such that $c_A\langle a_1, \dots, a_n \rangle \in \mathcal{A}$ if it is defined. We assume that if $c_A\langle a_1, \dots, a_n \rangle \in \mathcal{A}$ is defined and $a_i \equiv a'_i$, $i \in \{1, \dots, n\}$, then $c_A\langle a'_1, \dots, a'_n \rangle$ is also defined and

$$c_A\langle a_1, \dots, a_n \rangle \equiv c_A\langle a'_1, \dots, a'_n \rangle$$

Furthermore we assume that each expression a is associated with a set of active ports $active_A(a)$. Where $a \equiv a'$ implies $active_A(a) = active_A(a')$.

Now it is possible to define weak barbed congruence as in definition 6.1.2:

Definition 7.1.4 (Weak Barbed Congruence for Arbitrary Calculi) Let $A = (\mathcal{A}, \rightarrow)$ be a calculus with contexts and active ports as defined above.

A relation R is a *weak barbed congruence* on \mathcal{A} iff

- R is an equivalence
- $a_i R a'_i$ for $i \in \{1, \dots, n\}$ implies

$$c_A\langle a_1, \dots, a_n \rangle R c_A\langle a'_1, \dots, a'_n \rangle \quad (7.8)$$

for every discrete n -ary context c_A . Furthermore if $c_A\langle a_1, \dots, a_n \rangle$ is defined then $c_A\langle a'_1, \dots, a'_n \rangle$ is also defined.

- $a R a'$ and $a \rightarrow \hat{a}$ implies the existence of an expression \hat{a}' with $\hat{a} \rightarrow^* \hat{a}'$ and $\hat{a} R \hat{a}'$
- $a R a'$ and if $act \in active_A(a)$ then there exists an expression \hat{a}' with $a' \rightarrow^* \hat{a}'$, $act \in active_A(\hat{a}')$ and $a R \hat{a}'$.

Two expressions a, a' are called *weakly barbed congruent* ($a \approx a'$) iff there is a weak barbed congruence R such that $a R a'$ \square

We will now investigate what conclusions can be drawn from a simulation Δ concerning full abstraction.

Proposition 7.1.5 Let $\Delta : \mathcal{B}' \rightarrow \mathcal{A}$ be a mapping describing the simulation of calculus A by calculus B . Furthermore let \approx_A, \approx_B be weak barbed congruences on A respectively B .

We extend Δ to contexts such that for every context c_A of A there is a context c_B of B such that $\Delta(c_B) \equiv c_A$. And

$$\Delta(c_B\langle b_1, \dots, b_n \rangle) \equiv_B \Delta(c_B)\langle \Delta(b_1), \dots, \Delta(b_n) \rangle$$

(if one of both sides is defined) and there is a bijection f with $active_B(b) = f(active_A(\Delta(b)))$.

It follows that

$$b \approx_B b' \Rightarrow \Delta(b) \approx_A \Delta(b')$$

Proof: Let $b \approx_B b'$. This implies that there exists a weak barbed congruence R_B with $b R_B b'$. We define

$$R_A := \{(a, a') \mid \exists b, b' : b R_B b', \Delta(b) \equiv a, \Delta(b') \equiv a'\}$$

- R_A is reflexive because of (7.1). And since R_B is symmetric and transitive it follows that also R_A is symmetric and transitive.
- Let $\Delta(b_i) \equiv a_i R_A a'_i \equiv \Delta(b'_i)$ with $b_i R_B b'_i$. Let c_A be a context in A where $c_A\langle a_1, \dots, a_n \rangle \equiv c_A\langle \Delta(b_1), \dots, \Delta(b_n) \rangle$ is defined. With the condition above it follows that there is a context c_B in B satisfying $\Delta(c_B) \equiv_A c_A$. It follows that $\Delta(c_B\langle b_1, \dots, b_n \rangle) \equiv_A \Delta(c_B)\langle \Delta(b_1), \dots, \Delta(b_n) \rangle$ and $c_B\langle b_1, \dots, b_n \rangle$ is defined.

This implies that $c_B\langle b'_1, \dots, b'_n \rangle$ is defined and since R_B is a weak barbed congruence it follows that

$$c_B\langle b_1, \dots, b_n \rangle R_B c_B\langle b'_1, \dots, b'_n \rangle$$

Furthermore

$$\begin{aligned} c_A\langle a_1, \dots, a_n \rangle &\equiv c_A\langle \Delta(b_1), \dots, \Delta(b_n) \rangle \equiv_A \Delta(c_B\langle b_1, \dots, b_n \rangle) \\ R_A \quad \Delta(c_B\langle b'_1, \dots, b'_n \rangle) &\equiv_A c_A\langle \Delta(b'_1), \dots, \Delta(b'_n) \rangle \equiv c_A\langle a'_1, \dots, a'_n \rangle \end{aligned}$$

- Let $a \equiv \Delta(b) R_A \Delta(b') \equiv a'$ with $b R_B b'$ and let $a \rightarrow \hat{a}$ and thus also $\Delta(b) \rightarrow \hat{a}$. It follows with 7.4 that $b \rightarrow^* \hat{b}$ and $\Delta(\hat{b}) \equiv_A \hat{a}$.

Since R_B is a weak barbed congruence it follows that $b' \rightarrow^* \hat{b}'$ and $\hat{b} R_B \hat{b}'$.

(7.3) then implies that $a' \equiv \Delta(b') \rightarrow^* \Delta(\hat{b}')$ and $\hat{a} R_A \Delta(\hat{b}) R_A \Delta(\hat{b}')$.

- Let $a \equiv \Delta(b) R_A \Delta(b') \equiv a'$ with $b R_B b'$ and let $act \in active_A(a) = active_A(\Delta(b))$. This implies that $f(act) \in active_B(b)$.

It follows that $b' \rightarrow^* \hat{b}'$, $f(act) \in active_B(\hat{b}')$ and $b R_B \hat{b}'$.

It follows with (7.3) that $a' \equiv \Delta(b') \rightarrow^* \Delta(\hat{b}')$ with $a \equiv \Delta(b) R_A \Delta(\hat{b})$. And since $active_A(\Delta(\hat{b}')) = f^{-1}(active_B(\hat{b}'))$ it follows that

$$act \in active_A(\Delta(\hat{b}'))$$

□

7.2 The λ -Calculus

7.2.1 Informal Comparison

The original idea behind SPIDER was to combine the λ -calculus with concepts from graph rewriting in order to describe concurrent programs only by graph syntax, functional abstraction and application.

Since the semantics of SPIDER is based on lazy evaluation it is sensible to compare it with the lazy-evaluation semantics of the λ -calculus. Its syntax is as follows:

$$E ::= x \mid \lambda x.E \mid (E_1 E_2)$$

Renaming of bound names, i.e. α -conversion, leads to equivalent expressions. That is $\lambda x.E \equiv_\alpha \lambda y.E[y/x]$ if y does not occur free in E . The reduction rules are the following:

<i>Rules of Structural Rules:</i>	
(C- λ - α)	$\frac{E \equiv_\alpha E'}{E \equiv E'}$
(C- λ -APPL)	$\frac{E_1 \equiv E'_1, E_2 \equiv E'_2}{(E_1 E_2) \equiv (E'_1 E'_2)}$
(C- λ -ABSTR)	$\frac{E \equiv E'}{\lambda x.E \equiv \lambda x.E'}$
<hr/>	
<i>Reduction Rules:</i>	
(R- λ - β)	$((\lambda x.E_1)E_2) \rightarrow E_1[E_2/x]$
(R- λ -APPL)	$\frac{E_1 \rightarrow E'_1}{(E_1 E_2) \rightarrow (E'_1 E_2)}$
(R- λ -EQU)	$\frac{E_1 \equiv E'_1, E_2 \equiv E'_2, E_1 \rightarrow E_2}{E'_1 \rightarrow E'_2}$

We assume that $E_1[E_2/x]$ denotes the syntactical substitution of x by E_2 in E_1 (with change of bound variables in order to avoid capture of free variables in E_2).

The set of all λ -expressions will be denoted by Λ .

Some elements of the λ -calculus reappear in SPIDER with the gravest changes being made in the application of a function to a parameter. Instead of only two expressions $(E_1 E_2)$ there can be any number of connected processes and messages in a hypergraph. We can say that processes play the role of functions while messages play the role of parameters. The processes and messages are linked via ports (the nodes of the hypergraph).

Whereas an abstraction $\lambda x.E$ in the λ -calculus has access to only one parameter, a process can receive a message from any port at any time. The number k in the process abstraction $\lambda_k x.H$ indicates at which port the process is listening. Another difference to the λ -calculus is the fact that ports can be attached to a message, which are taken over by the receiver allowing it to gain access to even more messages. The port via which a message is received does not get lost but is available afterwards as well.

In the λ -calculus it is immediately obvious how to replace a substring by another string. In a graph-based calculus it is necessary to introduce a mechanism for the construction of graphs, as we have done in chapter 2.

Unlike in the λ -calculus, which is confluent, there are two forms of non-determinism which can occur in SPIDER:

- *Several messages* might be waiting at one port. A process listening at this port nondeterministically receives any of them.
- *Several processes* might be listening at the same port. If a message arrives at this port any one of the processes will receive it.

Consequently SPIDER is not confluent any more. In section 8.3, however, we will present a type system which will guarantee confluence for typed expressions.

We will now introduce two ways of encoding the λ -calculus into SPIDER. Although both simulate lazy evaluation, they are different in their approach.

7.2.2 Encoding the λ -Calculus into SPIDER, Version I

In the λ -calculus a function substitutes a parameter and so *becomes* the result. That is there is no real output. In this spirit we will give an encoding of the λ -calculus into SPIDER.

In section 7.2.3 we will follow a different approach, that respects the paradigm of process calculi. That is a function is simulated by a process which receives its parameters and outputs the result as a message.

The mapping Θ_λ^I described in the following encodes every λ -expression into a SPIDER expression of cardinality 1 (see figure 7.1). On this one external port a message, i.e. the parameter, is expected. This message is expected to bring along a port attached to it, from which the next parameter is to be received. This port is also called the *continuation*.

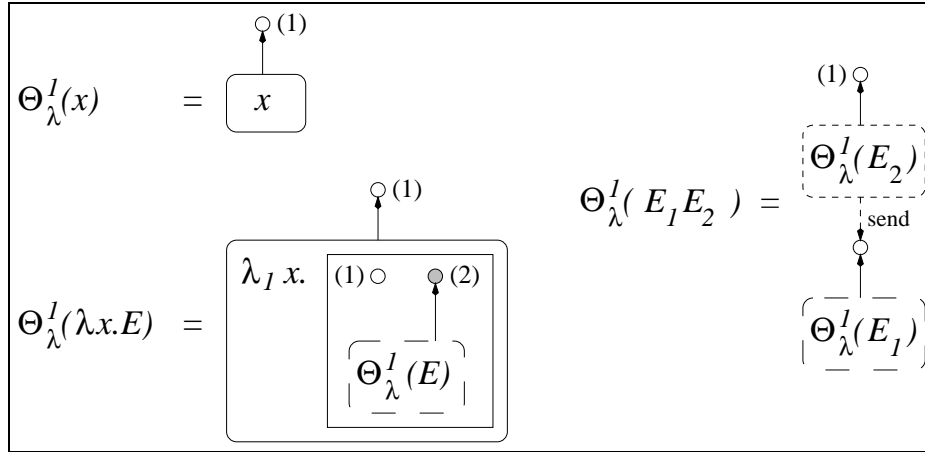


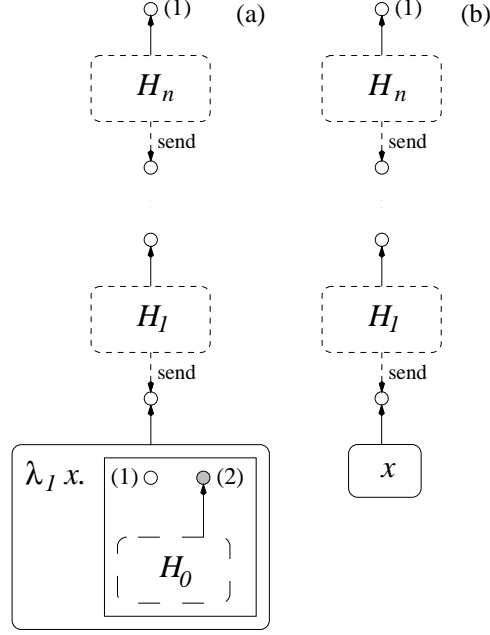
Figure 7.1: Encoding the λ -Calculus into SPIDER, Version I

(For the meaning of the long dashed lines see the explanation at the end of section 2.1.)

Proposition 7.2.1 *The λ -calculus can be strongly simulated by SPIDER.*

Proof: Our aim is to define an inverse translation $\Delta_\lambda^1 : \Lambda \rightarrow \mathcal{S}$ and to show that it satisfies the conditions of definition 7.1.1.

By induction on the length of the reduction we can show that if $\Theta_\lambda^1(E) \rightarrow^* H$ then $H \equiv H' \oplus \overline{\mathbf{m}}$ where $m \in \mathbb{N}$ and H' is equivalent either to (a) or (b) in the following figure, where H_0, H_1, \dots, H_n also have form (a) or (b) (for some $n \in \mathbb{N}$).



We define

$$\Delta_\lambda^1(H) := \begin{cases} \lambda x. \Delta_\lambda^1(H_0) \Delta_\lambda^1(H_1) \dots \Delta_\lambda^1(H_n) & \text{if } H' \text{ has form (a)} \\ x \Delta_\lambda^1(H_1) \dots \Delta_\lambda^1(H_n) & \text{if } H' \text{ has form (b)} \end{cases}$$

We can show by induction that

$$\begin{aligned} \Theta_\lambda^1(E_1[E_2/x]) &\equiv \Theta_\lambda^1(E_1)[\Theta_\lambda^1(E_2)/x] \\ \Delta_\lambda^1(H[J/x]) &\equiv \Delta_\lambda^1(H)[\Delta_\lambda^1(J)/x] \\ \Delta_\lambda^1(\Theta_\lambda^1(E) \oplus \overline{\mathbf{n}}) &\equiv E \\ \forall H \exists m \in \mathbb{N} : \Theta_\lambda^1(\Delta_\lambda^1(H)) \oplus \overline{\mathbf{n}} &\equiv H \end{aligned}$$

We will now show that the four conditions of definition 7.1.1 are satisfied:

(7.1) because of $\Delta_\lambda^1(\Theta_\lambda^1(E)) \equiv E$

(7.2) By induction on the (alternative) rules of structural congruence of SPI-DER (see table 4.1).

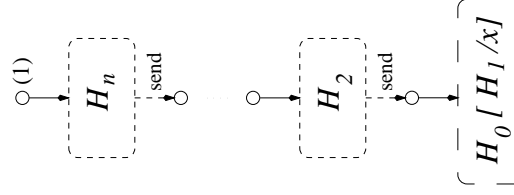
(7.3) Let $H \rightarrow H'$.

According to proposition 4.3.5 it follows that

$$H \equiv C\langle J_1, J_2 \rangle \quad H' \equiv C\langle J'_1, J_2 \rangle$$

and $J_1 \xrightarrow{(M-MR)} J'_1$.

This is only possible if H has form (a) and H' has the following form:



It implies that

$$\begin{aligned}
 \Delta_\lambda^1(H) &= \lambda x. \Delta_\lambda^1(H_0) \Delta_\lambda^1(H_1) \dots \Delta_\lambda^1(H_n) \\
 &\rightarrow \Delta_\lambda^1(H_0) [\Delta_\lambda^1(H_1)/x] \Delta_\lambda^1(H_2) \dots \Delta_\lambda^1(H_n) \\
 &\equiv \Delta_\lambda^1(H_0[H_1/x]) \Delta_\lambda^1(H_2) \dots \Delta_\lambda^1(H_n) \\
 &= \Delta_\lambda^1(H')
 \end{aligned}$$

(7.4) Let $\Delta_\lambda^1(H) \rightarrow E'$. We have to show that $H \rightarrow H'$ such that $\Delta_\lambda^1(H) \equiv E'$.

- We will first show that $E_1 \equiv E_2 \Rightarrow \Theta_\lambda^1(E_1) \equiv \Theta_\lambda^1(E_2)$ (by induction on the rules of structural congruence)
- Then it can be show that $E \rightarrow E'$ implies $\Theta_\lambda^1(E) \rightarrow \Theta_\lambda^1(E') \oplus \bar{\mathbf{1}}$. This can be done by straightforward induction on the reduction rules.

Therefore $\Theta_\lambda^1(\Delta_\lambda^1(H)) \rightarrow \Theta_\lambda^1(E') \oplus \bar{\mathbf{1}}$ and

$$H \equiv \Theta_\lambda^1(\Delta_\lambda^1(H)) \oplus \bar{\mathbf{n}} \rightarrow \Theta_\lambda^1(E') \oplus \overline{\mathbf{n} + \mathbf{1}} := H'$$

where $\Delta_\lambda^1(H') \equiv E'$.

□

7.2.3 Encoding the λ -Calculus into SPIDER, Version II

The encoding above is correct in the sense defined in section 7.1, but it has one weakness: it is hard to combine encoded λ -expressions with other SPIDER processes. If we regard an encoded expression as a black box, we see a process which only receives but never sends messages. Its interaction with the environment is thus rather limited, especially in our asynchronous case where the reception of a message cannot be observed. Since there are no active ports to observe, we have problems with full abstraction.

What we want to do now is create functions receiving their parameters and sending the result as a message. We will define the following two operators.

- $\boxed{\lambda x} \text{expr}$ ($x \in Id$, $\text{expr} \in \mathcal{E}$), a SPIDER expression of cardinality 2, which can receive an expression, substitute this expression for x in expr and send the result of the substitution.
- $@$, a process description which receives a function on its first port, a parameter on its second port and which will insert the function such that it receives the parameter and outputs the result.

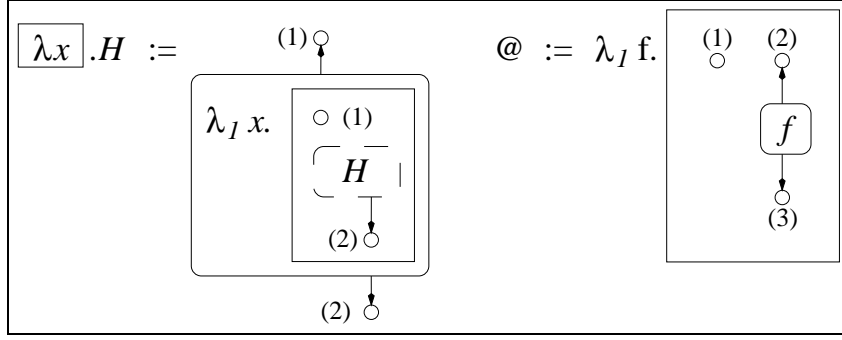
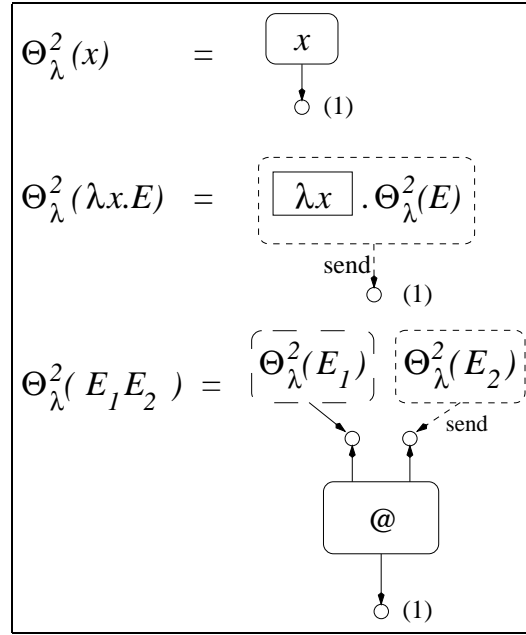


Figure 7.2: Operators for abstraction and application

The operators $\boxed{\lambda x}.expr$, $@$ are defined in figure 7.2.

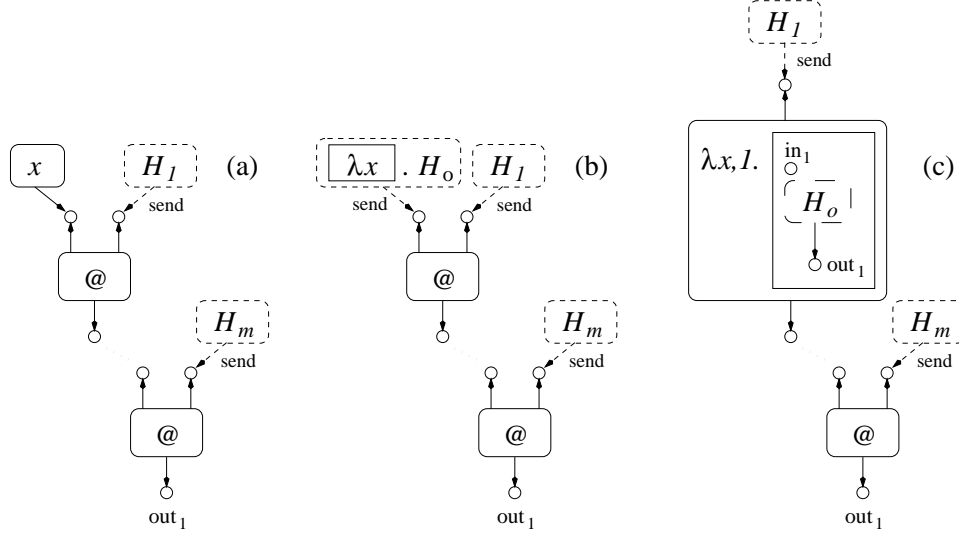
We use the operators to define the encoding depicted in figure 7.3. It takes a λ -expression and maps it onto a SPIDER expression of cardinality 1. Note that $\Theta_\lambda^2(E)$ is active in its output port if and only if E is of the form $\lambda x.E'$.

Figure 7.3: Encoding the λ -Calculus into SPIDER, Version II

Proposition 7.2.2 *The λ -calculus can be weakly simulated by SPIDER.*

Proof: By induction on the length of the reduction we can show that if $\Theta_\lambda^2(E) \rightarrow^* H$ then $H \equiv H' \oplus \bar{\mathbf{m}}$ where H' either has form (a), (b) or (c) in figure 7.4 (where the expressions H_i have form (a) or (b)).

We can define an inverse mapping Δ_λ^2 for every SPIDER expression H with

Figure 7.4: Encoded λ -expressions

$\Theta_\lambda^2(E) \rightarrow^* H$:

$$\Delta_\lambda^2(H) := \begin{cases} x\Delta_\lambda^2(H_1) \dots \Delta_\lambda^2(H_m) & \text{if } H \text{ has form (a)} \\ (\lambda x. \Delta_\lambda^2(H_0))\Delta_\lambda^2(H_1) \dots \Delta_\lambda^2(H_m) & \text{if } H \text{ has form (b)} \\ \Delta_\lambda^2(H_0)[\Delta_\lambda^2(H_1)/x]\Delta_\lambda^2(H_2) \dots \Delta_\lambda^2(H_m) & \text{if } H \text{ has form (c)} \end{cases}$$

We can show that

$$\begin{aligned} \Theta_\lambda^2(E_1[E_2/x]) &\equiv \Theta_\lambda^1(E_1)[\Theta_\lambda^2(E_2)/x] \\ \Delta_\lambda^2(H[J/x]) &\equiv \Delta_\lambda^2(H)[\Delta_\lambda^2(J)/x] \\ \Delta_\lambda^2(\Theta_\lambda^2(E) \oplus \bar{\mathbf{n}}) &\equiv E \\ \forall H \exists m \in \mathbb{N} : H &\rightarrow^* \Theta_\lambda^2(\Delta_\lambda^2(H)) \oplus \bar{\mathbf{m}} \end{aligned}$$

Now we will show that all four conditions of definition 7.1.1 are satisfied.

(7.1) Obvious since $\Delta_\lambda^2(\Theta_\lambda^2(E)) \equiv E$

(7.2) By induction on the (alternative) rules of structural congruence of SPI-DER.

(7.3) Let $H \rightarrow H'$.

According to proposition 4.3.5 it follows that

$$H \equiv C\langle J_1, J_2 \rangle \quad H' \equiv C\langle J'_1, J_2 \rangle$$

and $J_1 \xrightarrow{(M-MR)} J'_1$.

There are two cases:

- H has form (b) and the reduction is the reception of the message $\boxed{\lambda x}.H_0$. Then H' has form (c) and

$$\begin{aligned}\Delta_\lambda^2(H) &\rightarrow \Delta_\lambda^2(H_0)[\Delta_\lambda^2(H_1)/x]\Delta_\lambda^2(H_2)\dots\Delta_\lambda^2(H_m) \\ &= \Delta_\lambda^2(H_0[H_1/x])\Delta_\lambda^2(H_2)\dots\Delta_\lambda^2(H_m) = \Delta_\lambda^2(H')\end{aligned}$$

- H has form (c) and the reduction is the reception of the message H_1 . Then

$$\begin{aligned}\Delta_\lambda^2(H') &= \Delta_\lambda^2(H_0[H_1/x])\Delta_\lambda^2(H_2)\dots\Delta_\lambda^2(H_m) \\ &= \Delta_\lambda^2(H_0)[\Delta_\lambda^2(H_1)/x]\Delta_\lambda^2(H_2)\dots\Delta_\lambda^2(H_m) \\ &= \Delta_\lambda^2(H)\end{aligned}$$

(7.4) Let $\Delta_\lambda^2(H) \rightarrow E'$. We have to show that $H \rightarrow H'$ such that $\Delta_\lambda^2(H') \equiv E'$.

- We will first show that $E_1 \equiv E_2 \Rightarrow \Theta_\lambda^1(E_1) \equiv \Theta_\lambda^1(E_2)$ (by induction on the rules of structural congruence).
- Now we can show that $E \rightarrow E'$ implies $\Theta_\lambda^1(E) \rightarrow^2 \Theta_\lambda^1(E') \oplus \bar{\mathbf{2}}$. This can be done by then by straightforward induction on the reduction rules.

Therefore $\Theta_\lambda^1(\Delta_\lambda^2(H)) \rightarrow^* \Theta_\lambda^1(E') \oplus \bar{\mathbf{2}}$ and

$$H \equiv \Theta_\lambda^1(\Delta_\lambda^2(H)) \oplus \bar{\mathbf{n}} \rightarrow \Theta_\lambda^1(E') \oplus \overline{\mathbf{n} + \mathbf{2}} := H'$$

Furthermore $\Delta_\lambda^2(H') \equiv E'$.

□

7.3 SPIDER with Graph Expressions

In chapter 2 we have shown that there are alternative representations of hypergraphs, e.g. graph expressions. When we represent SPIDER expressions in this notation, it is useful to have a representation of reduction rules in this approach.

We will now introduce an equivalent version of the SPIDER calculus, based on the graph expressions.

Syntax: A SPIDER expression can be represented as a graph expression g where g has the form

Process with a Variable: $proc_m(x)$ where $x \in X$, $m = sort(x)$

Process with Replication: $proc_m(!g)$

Process with Process Abstraction: $proc_m(\lambda_k x.g)$ where $m \leq card(g)$, $k \in \{1, \dots, n\}$

Message: $mess_n(g)$ where $n \geq 1$

Sum: $g_1 \oplus g_2$

Redefinition of External Nodes: $\sigma_\alpha(g)$

Node Fusion: $\theta_\delta(g)$

where g, g_1, g_2 are again SPIDER expressions represented as graph expressions.

As in definition 4.3.1 we can define substitution of variables by terms as follows:

Definition 7.3.1 (Substitution of Variables) Let g, g' be SPIDER expressions represented as graph expressions and let $card(g') = sort(x)$.

We define $g[g'/x]$ inductively as follows:

$$\begin{aligned}
(g_1 \oplus g_2)[g'/x] &:= g_1[g'/x] \oplus g_2[g'/x] \\
\sigma_\alpha(g)[g'/x] &:= \sigma_\alpha(g[g'/x]) \\
\theta_\delta(g)[g'/x] &:= \theta_\delta(g[g'/x]) \\
proc_n(y)[g'/x] &:= \begin{cases} g' & \text{if } x = y \\ proc_n(y) & \text{otherwise} \end{cases} \\
mess_n(g)[g'/x] &:= mess_n(g[g'/x]) \\
proc_n(!g)[g'/x] &:= proc_n(!(g[g'/x])) \\
proc_n(\lambda_k y. g)[g'/x] &:= \begin{cases} proc_n(\lambda_k y. g) & \text{if } x = y \\ proc_n(\lambda_k y. (g[g'/x])) & \text{if } x \neq y, y \notin free(g') \\ proc_n(\lambda_k z. ((g[z/y])[g'/x])) & \text{if } x \neq y, y \in free(J), z \notin free(g) \cup free(g') \end{cases}
\end{aligned}$$

□

The operational semantics of the SPIDER calculus based on graph expressions is given in figure 7.5.

Proposition 7.3.2 *The algebraic calculus above can be strongly simulated by SPIDER.*

Proof: We will define a mapping Θ_A which takes an algebraic SPIDER expression and converts it into an ordinary SPIDER expression:

$$\begin{aligned}
\Theta_A(g) &:= val_a(\Theta'_A(g)) \\
\Theta'_A(g_1 \oplus g_2) &:= \Theta'_A(g_1) \oplus \Theta'_A(g_2) \\
\Theta'_A(\theta_\delta(g)) &:= \theta_\delta(\Theta'_A(g)) \\
\Theta'_A(\sigma_\alpha(g)) &:= \sigma_\alpha(\Theta'_A(g)) \\
\Theta'_A(proc_m(!g)) &:= proc_m(!\Theta_A(g)) \\
\Theta'_A(proc_m(\lambda_k x. g)) &:= proc_m(\lambda_k x. \Theta_A(g)) \\
\Theta'_A(mess_{n+1}(g)) &:= mess_{n+1}(\Theta_A(g))
\end{aligned}$$

<i>Rules of Structural Congruence:</i>	
(C-A-PA)	$\frac{g_1 \equiv g_2}{\lambda_k x. g_1 \equiv \lambda_k x. g_2}$
(C-A-REPL)	$\frac{g_1 \equiv g_2}{!g_1 \equiv !g_2}$
(C-A-CON)	$\frac{g_1 \simeq \equiv g_2}{g_1 \equiv g_2}$
(C-A- α)	$\lambda_n x. g \equiv \lambda_n y. (g[proc_n(y)/x])$ if $y \notin free(g)$, $sort(x) = sort(y) = n$
<i>Reduction Rules:</i>	
(R-A-REPL)	$proc_m(!g) \rightarrow \sigma_\alpha(\theta_\delta(proc_m(!g) \oplus g))$
(R-A-MR)	$\sigma_{1\dots m+n}(\theta_{k,m+n+1}(proc_m(\lambda_i x. g_1) \oplus mess_{n+1}(g_2))) \rightarrow g_1[g_2/x]$
(R-A-EQU)	$\frac{h \equiv g, g \rightarrow g', g' \equiv h'}{h \rightarrow h'}$
(R-A- σ)	$\frac{g \rightarrow g'}{\sigma_\alpha(g) \rightarrow \sigma_\alpha(g')}$
(R-A- θ)	$\frac{g \rightarrow g'}{\theta_\delta(g) \rightarrow \theta_\delta(g')}$
(R-A- \oplus)	$\frac{g_1 \rightarrow g'_1}{g_1 \oplus g_2 \rightarrow g'_1 \oplus g_2}$
where	
δ is the transitive closure of $\{(i, m+i) \mid 1 \leq i \leq m\}$	
$\alpha : \{1, \dots, m\} \rightarrow \{1, \dots, 2m\}$ with $\alpha(i) := i \bmod m$	

Figure 7.5: The SPIDER calculus based on graph expressions

It is left to show that Θ_A satisfies the conditions of proposition 7.1.3. This proof is more or less analogous to the proof of proposition 7.4.3 in the following section. \square

7.4 SPIDER in the Name-based Notation

In the name-based notation it is only possible to describe hypergraphs without duplicates in the sequence of external nodes.

Let $\mathcal{S}_n \subseteq \mathcal{S}$ be the class of all SPIDER expressions, which contains only hypergraphs H where χ_H does not contain any duplicates, i.e. all graphs can be represented in the form $H \cong val_n(h[t])$ where $h[t]$ is name-based graph term.

We will first make sure that \mathcal{S}_n is closed under reduction. It is also important

that any hypergraph in \mathcal{S}_n containing a redex can always be described as the composition of the redex and of a graph in \mathcal{S}_n .

Proposition 7.4.1 (Subcalculus \mathcal{S}_n) *Let $H \in \mathcal{S}_n$. If $H \rightarrow H'$ then*

- *There exists a binary context $C\langle x_1, x_2 \rangle$ where χ_C is duplicate-free and SPIDER expressions $H_1, H_2 \in \mathcal{S}_n$ such that*

$$H_1 \xrightarrow{(R-MR)} H'_1 \quad \text{or} \quad H_1 \xrightarrow{(R-REPL)} H'_1 \\ H' \cong C\langle H'_1, H_2 \rangle$$

- *$H' \in \mathcal{S}_n$, i.e. \mathcal{S}_n is closed under reduction.*

Proof: According to proposition 4.3.5 there is a binary context $H \cong C\langle H_1, H_2 \rangle$ such that $H_1 \xrightarrow{(R-MR)} H'_1$ or $H_1 \xrightarrow{(R-REPL)} H'_1$ and $H' \cong C\langle H'_1, H_2 \rangle$. However H_2 is not necessarily an element of \mathcal{S}_n .

We will regard the non-discrete context $J\langle x_1 \rangle := C\langle \text{var}_{m_1}(x_1), H_2 \rangle$ where $m_1 := \text{sort}(x_1)$. With proposition 2.1.7 and its proof it follows that there exists a factorization $\eta_1 : \text{var}_{m_1}(x_1) \rightarrow J$, $\eta_2 : J_2 \rightarrow J$ such that $J_2 \in \mathcal{S}_n$.

Propositions 2.2.18 and 2.2.21 imply that there exists a context $C'\langle y_1, y_2 \rangle$ such that $J\langle x_1 \rangle \cong C'\langle \text{var}_{m_1}(x_1), J_2 \rangle$. Thus

$$H \cong J\langle H_1 \rangle \cong C'\langle H_1, J_2 \rangle \quad H' \cong J\langle H'_1 \rangle \cong C'\langle H'_1, J_2 \rangle$$

and $H_1 \rightarrow H'_1$.

$H' \in \mathcal{S}_n$ follows immediately since \mathcal{S}_n is closed under graph construction with contexts C where χ_C is duplicate-free. \square

Now that we know that a name-based subcalculus is closed under reduction we can define its syntax as follows:

Syntax: A name-based SPIDER expression can be represented as a name-based graph term $h[t]$ where h has the form

Process with a Variable: $(x)_P[t]$ where $x \in X$, $|t| = \text{sort}(x)$

Process with Replication: $(!h[t'])_P[t]$

Process with Process Abstraction:

$$(\lambda_k x. h[t'])_P[t] \text{ where } |t| \leq |t'|, k \in \{1, \dots, |t|\}$$

Message: $(h[t'])_M[t]$ where $|t| \geq 1$

Parallel Composition: $h_1 | h_2$

Node Hiding: $(\nu a)h$

where h, h_1, h_2 are again name-based SPIDER expressions and t, t' are duplicate-free sequences of names.

As in definition 4.3.1 we can define substitution of variables by terms as follows:

Definition 7.4.2 (Substitution) We define $h[t][h'[t']/x]$ inductively as follows:

$$\begin{aligned}
h[t][h'[t']/x] &:= (h[h'[t']/x])[t] \\
(h_1 \mid h_2)[h'[t']/x] &:= h_1[h'[t']/x] \mid h_2[h'[t']/x] \\
(\nu a)h[h'[t']/x] &:= (\nu a)(h[h'[t']/x]) \\
((y)_P[s])[h'[t']/x] &:= \begin{cases} h[s/t'] & \text{if } x = y \\ (y)_P[s] & \text{otherwise} \end{cases} \\
((h[t])_M[s])[h'[t']/x] &:= (h[t][h'[t']/x])_M[s] \\
((!h[t])_P[s])[h'[t']/x] &:= (!h[t][h'[t']/x])_P[s] \\
((\lambda_k y.h[t])_P[s])[h'[t']/x] &:= \begin{cases} (\lambda_k y.h[t])_P[s] & \text{if } x = y \\ (\lambda_k y.(h[t][h'[t']/x]))_P[s] & \text{if } x \neq y, y \notin \text{free}(g') \\ (\lambda_k y.((h[t][((z)_P[u])[u]/y)][h'[t']/x]))_P[s] & \text{if } x \neq y, y \in \text{free}(J), z \notin \text{free}(g) \cup \text{free}(g') \end{cases}
\end{aligned}$$

□

The operational semantics of the SPIDER calculus based on the name-based notation is given in figure 7.5.

Proposition 7.4.3 *The name-based calculus above can be strongly simulated by SPIDER.*

Proof: We will give another set of reduction rules (reduction relation \rightsquigarrow) and show that this calculus can be strongly simulated by SPIDER.

But first we have to prove that the new reduction rules are equivalent to the old rules. After showing

$$\begin{aligned}
h \rightsquigarrow h' &\Rightarrow h[t] \rightarrow h'[t] \\
h[t] \rightarrow h'[t'] &\Rightarrow h \rightsquigarrow h'[t/t']
\end{aligned}$$

by induction on the reduction rules ((R-N-CON') additionally needs induction on h) it follows immediately that

$$h[t] \rightarrow h'[t'] \iff h[t] \rightsquigarrow h'[t']$$

<i>Rules of Structural Congruence:</i>	
(C-N-CON)	$\frac{h_1[t_1] \simeq \equiv h_2[t_2]}{h_1[t_1] \equiv h_2[t_2]}$
(C-N-PA)	$\frac{h_1[t_1] \equiv h_2[t_2]}{\lambda_n x. h_1[t_1] \equiv \lambda_n x. h_2[t_2]}$
(C-N-REPL)	$\frac{h_1[t_1] \equiv h_2[t_2]}{!h_1[t_1] \equiv !h_2[t_2]}$
(C-N- α)	$\lambda_k x. h[t] \equiv \lambda_k y. (h[t][((y)_P[s])[s]/x])$ if $y \notin \text{free}(g)$, $\text{sort}(x) = \text{sort}(y) = s $
<hr/>	
<i>Reduction Rules:</i>	
(R-N-REPL)	$(!h[s])_P[s'] \rightarrow (!h[s])_P[s'] \mid h[s'/s]$
(R-N-MR)	$(\lambda_k x. h_1[s_1])_P[a_1 \dots a_m] \mid (h_2[s_2])_M[a_{m+1} \dots a_{m+n} a_k] \rightarrow (h_1[h_2[s_2]/x])[a_1 \dots a_{m+n}/s]$
(R-N-PAR)	$\frac{h_1 \rightarrow h'_1}{h_1 \mid h_2 \rightarrow h'_1 \mid h_2}$
(R-N-RESTR)	$\frac{h \rightarrow h'}{(\nu a)h \rightarrow (\nu a)h'}$
(R-N-CL)	$\frac{h \rightarrow h'}{h[t] \rightarrow h'[t]}$
(R-N-EQU)	$\frac{h_2[t_2] \equiv h_1[t_1], h_1[t_1] \rightarrow h'_1[t'_1], h'_1[t'_1] \equiv h'_2[t'_2]}{h_2[t_2] \rightarrow h'_2[t'_2]}$

Figure 7.6: The SPIDER calculus in the name-based notation

<i>Reduction Rules:</i>	
(R-N-REPL')	$((!h[s])_P[t])[t] \rightsquigarrow ((!h[s])_P[t] \mid h[s'/s])[t]$
(R-N-MR')	$((\lambda_k x. h_1[s_1])_P[a_1 \dots a_m] \mid (h_2[s_2])_M[a_{m+1} \dots a_{m+n} a_k])[t] \rightsquigarrow h((h_1[h_2[s_2]/x])[a_1 \dots a_{m+n}/s])[t]$ where $t = a_1 \dots a_{m+n}$
(R-N-CON')	$\frac{h_i[t_i] \rightsquigarrow h'_i[t'_i]}{h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle \rightsquigarrow h[t]\langle h_1[t_1], \dots, h'_i[t'_i], \dots, h_n[t_n] \rangle}$
(R-N-EQU')	$\frac{h_2[t_2] \equiv h_1[t_1], h_1[t_1] \rightsquigarrow h'_1[t'_1], h'_1[t'_1] \equiv h'_2[t'_2]}{h_2[t_2] \rightsquigarrow h'_2[t'_2]}$

We therefore define the mapping Θ_N which takes a name-based SPIDER expression and converts it into an ordinary SPIDER expression:

$$\begin{aligned}\Theta_N(h[t]) &:= \text{val}_n(\Theta'_N(h)[t]) \\ \Theta'_N(h_1|h_2) &:= \Theta'_N(h_1)|\Theta'_N(h_2) \\ \Theta'_N((\nu a)h) &:= (\nu a)\Theta'_N(h) \\ \Theta'_N((!h[t])_P[t']) &:= (!\Theta_N(h[t]))_P[t'] \\ \Theta'_N((\lambda_k x.h[t])_P[t']) &:= (\lambda_k x.\Theta_N(h[t]))_P[t'] \\ \Theta'_N((h[t])_M[t']) &:= (\Theta_N(h[t]))_M[t']\end{aligned}$$

Proposition 2.4.5 and induction on the hierarchy levels imply that $h[t] \equiv h'[t']$ if and only if $\Theta_N(h[t]) \equiv \Theta_N(h'[t'])$.

With proposition 2.4.7 it follows that

$$\Theta_N(h[t]\langle h_1[t_1], \dots, h_n[t_n] \rangle) \cong \Theta_N(h[t])\langle \Theta_N(h_1[t_1]), \dots, \Theta_N(h_n[t_n]) \rangle \quad (7.9)$$

We can show that Θ_N satisfies the three conditions of proposition 7.1.3:

(7.5) By induction on the rules of structural equivalence and with

We have to show that $\Theta_N(h[t]) \equiv \Theta_N(h'[t'])$ implies $h[t] \equiv h'[t']$. This can be done by induction on $h[t]$ with the help of proposition 2.4.5. (For SPIDER we will use the alternative set of rules of structural congruence).

(7.6) By induction on the reduction rules and with equation (7.9).

(7.7) By induction on the reduction rules and with propositions 7.4.1, 2.4.7.

It is now not difficult to show that the alternative reduction rules are equivalent to the original reduction rules. \square

7.5 The π -Calculus

Milner's π -calculus [MPW89a, MPW89b, Mil92, Mil91] is a calculus describing communicating systems which models mobile processes being able to send port addresses. The π -calculus is a widespread and thoroughly analyzed paradigm describing concurrency and communication.

In the previous sections we have shown that higher-order functions of the λ -calculus can be modelled in SPIDER. We will now encode the asynchronous polyadic π -calculus without sum into SPIDER. We will use a slight variant of the π -calculus in order to simplify our proof.

$$p ::= \mathbf{0} \mid (\nu a)p \mid \bar{a}a_1, \dots, a_n \mid a(x_1, \dots, x_n).p \mid p_1|p_2 \mid !p$$

where $a, a_1, \dots, a_n, x_1, \dots, x_n$ are taken from a fixed set of names.

The set of all free names in a process p will be denoted by $fn(p)$. We will translate SPIDER expressions with the help of name-based graph terms introduced in section 2.4. We have shown in section 7.4 that name-based SPIDER

<i>Rules of Structural Congruence:</i>	
(C- π -COM) $p_1 p_2 \equiv p_2 p_1$	(C- π -ASSO) $p_1 (p_2 p_3) \equiv (p_1 p_2) p_3$
(C- π -RESTR1) $(\nu a)\mathbf{0} \equiv \mathbf{0}$	(C- π -RESTR2) $(\nu a)(\nu b)p \equiv (\nu b)(\nu a)p$
(C- π -RESTR3) $((\nu a)p_1) p_2 \equiv (\nu a)(p_1 p_2)$ if $a \notin fn(p_2)$	
(C- π -0) $p \mathbf{0} \equiv p$	(C- π -REN1) $(\nu a)p \equiv (\nu b)(p[b/a])$ if $b \notin fn(p)$
(C- π -REN2) $a.(x_1, \dots, x_n).p \equiv a.(y_1, \dots, y_n).p[y_1 \dots y_n/x_1 \dots x_n]$ if $y_1, \dots, y_n \notin fn(p)$	
<i>Reduction Rules:</i>	
(R- π -COMM) $a(x_1, \dots, x_n).p \mid \bar{a}a_1 \dots a_n \rightarrow p[a_1 \dots a_n/x_1 \dots x_n]$	
(R- π -REPL) $!p \rightarrow !p p$	
(R- π -PAR) $\frac{p \rightarrow p'}{p q \rightarrow p' q}$	(R- π -RESTR) $\frac{p \rightarrow p'}{(\nu a)p \rightarrow (\nu a)p'}$
(R- π -EQU) $\frac{q \equiv p, p \rightarrow p', p' \equiv q'}{q \rightarrow q'}$	

Figure 7.7: π -Calculus, Operational Semantics

can be strongly simulated by the original SPIDER calculus. We encode the π -calculus into name-based SPIDER, the rest follows with the transitivity of strong simulation.

Let \leq be a total order on the set of all names. Let p be a process in the π -calculus. Then fn_p is a duplicate-free string, ordered according to \leq , of all names in $fn(p)$.

$$\Theta_\pi^s(p) := (\Theta'_\pi(p)|[\![s]_1]\!|\dots|[\![s]_m]\!|)[s]$$

where s is a duplicate-free string with $m := |s|$, $fn(p) \subseteq Set(s)$ and where Θ'_π is defined inductively as follows:

$$\begin{aligned} \Theta'_\pi(p_1|p_2) &:= \Theta'_\pi(p_1)|\Theta'_\pi(p_2) \\ \Theta'_\pi((\nu a)p) &:= (\nu a)\Theta'_\pi(p) \\ \Theta'_\pi(\mathbf{0}) &:= \mathbf{0} \\ \Theta'_\pi(\bar{a}a_1, \dots, a_m) &:= (\mathbf{0})_M[a_1 \dots a_m a] \end{aligned}$$

$$\begin{aligned}\Theta(!p) &:= (!\Theta_\pi^{fn_p}(p))_P[fn_p] \\ \Theta(a(x_1, \dots, x_n).p) &:= (\lambda_k. \Theta_\pi^{fn_{px_1 \dots x_n}}(p))_P[fn_p] \text{ where } \lfloor fn_p \rfloor_k = a\end{aligned}$$

The following partial mapping Δ_π^s “inverts” Θ_π^s :

$$\Delta_\pi^s(h[t]) := (\Delta_\pi(h))[s/t]$$

where

$$\begin{aligned}\Delta_\pi(h_1|h_2) &:= \Delta_\pi(h_1)|\Delta_\pi(h_2) \\ \Delta_\pi((\nu a)h) &:= (\nu a)\Delta_\pi(h) \\ \Delta_\pi(0) &:= \mathbf{0} \\ \Delta_\pi(\lceil a \rceil) &:= \mathbf{0} \\ \Delta_\pi((0)_M[a_1 \dots a_{n+1}]) &:= \overline{a_{n+1}}a_1 \dots a_n \\ \Delta_\pi((!h[s])_P[a_1 \dots a_m]) &:= !\Delta_\pi^{a_1 \dots a_m}(h[s]) \\ \Delta_\pi((\lambda_k.h(s))_P[a_1 \dots a_m]) &:= a_k(x_1 \dots x_n).\Delta_\pi^{a_1 \dots a_m x_1 \dots x_n}(h[s])\end{aligned}$$

where x_1, \dots, x_n are fresh names
and $n := |s| - m$

Proposition 7.5.1 (Simulation of the π -calculus) *The π -calculus can be strongly simulated by the name-based SPIDER calculus wrt. the encoding Δ_π^t .*

Proof: Let $\mathcal{N}' := \{h[t] \mid \Delta_\pi(h[t]) \text{ is defined}\}$ be the part of the name-based SPIDER calculus which simulates the π -calculus.

Furthermore we define $\bar{n} := \underbrace{(\nu a)\lceil a \rceil \dots (\nu a)\lceil a \rceil}_{n \text{ times}}$

We can show by induction on p that

$$\begin{aligned}p \equiv p' &\Rightarrow \Theta_\pi^s(p) \equiv \Theta_\pi^s(p') \\ \Delta_\pi^s(\Theta_\pi^t(p)) &\equiv p[s/t]\end{aligned}$$

Furthermore if $h'[t'] := \Theta_\pi^s(\Delta_\pi^s(h[t]))$ then there is a natural number n such that $(h'|\bar{n})[t'] \equiv h[t]$ (proof by induction on $h[t]$). That is $\Theta_\pi^s(\Delta_\pi^s(h[t]))$ and $h[t]$ are equivalent up to isolated ports, if $\Delta_\pi^s(h[t])$ is defined.

We will check that the four conditions of definition 7.1.1 are satisfied.

(7.1) Obvious since $\Delta_\pi^t(\Theta_\pi^t(p)) \equiv p$

(7.2) Can be shown by induction on the rules of structural congruence of the name-based SPIDER calculus.

(7.3) We have to show that $h[t] \rightarrow h'[t']$ implies $\Delta_\pi^s(h[t]) \rightarrow \Delta_\pi^s(h'[t'])$. We will additionally show that $h \rightarrow h'$ implies $\Delta_\pi(h) \rightarrow \Delta_\pi(h')$.

Both can be shown by induction on the reduction rules.

(7.4) We have to prove that if $\Delta_\pi^s(h[t]) \rightarrow p'$ there exists a name based expression $h'[t']$ such that $h[t] \rightarrow h'[t']$ and $\Delta_\pi^s(h'[t']) \equiv p'$.

We can show by straightforward induction on the reduction rules of the π -calculus that $p \rightarrow p'$ implies $\Theta'_\pi(p) \rightarrow \Theta'_\pi(p') \mid [a_1] \mid \dots \mid [a_l] \mid \bar{n}$ for some natural number n and where $a_1, \dots, a_l \in fn(p)$.

Now let $p \rightarrow p'$ and $h[s] := \Theta_\pi^s(p)$, $h'[s] := \Theta_\pi^s(p')$.

$$\begin{aligned} h &\equiv \Theta'_\pi(p) \mid [s]_1 \mid \dots \mid [s]_m \\ &\rightarrow \Theta'_\pi(p') \mid [s]_1 \mid \dots \mid [s]_m \mid [a_1] \mid \dots \mid [a_l] \mid \bar{n} \\ &\equiv \Theta'_\pi(p') \mid [s]_1 \mid \dots \mid [s]_m \mid \bar{n} \end{aligned}$$

where $m := |s|$ and $\{1, \dots, a_l\} \subseteq fn(p) = Set(s)$.

This implies that $h[t] \rightarrow (h'|\bar{n})[t]$.

If $\Delta_\pi^s(h[t]) \rightarrow p'$ we define

$$\begin{aligned} h_1[t_1] &:= \Theta_\pi^s(\Delta_\pi^s(h[t])) \\ h_2[t_2] &:= \Theta_\pi^s(p') \end{aligned}$$

It follows that $h[t] \equiv (h_1|\bar{n})[t]$, $h_1[t_1] \rightarrow (h_2|\bar{m})[t_2]$ for natural numbers n, m .

Therefore

$$h[t] \rightarrow (h_2|\overline{n+m})[t_2] =: h'[t']$$

and $\Delta_\pi^s((h_2|\overline{n+m})[t_2]) \equiv p'$.

□

Chapter 8

Generating Type Systems

Type systems are an important tool for programming since they allow us to check for runtime-errors and verify programs, revealing, for instance, security problems.

In many cases a type is meant to represent a set of objects, e.g. the integers or the booleans. But this is *not* the view we will take here. Actually a type is, in our case, a partial behaviour description of a process that allows us to infer certain properties of a process, one of them being absence of runtime errors. We now summarize the requirements for types in process calculi:

Verification: a type makes assertions on the behaviour of a process. It is however not necessary that every process showing this very behaviour actually has a type. Therefore it is not uncommon to design type systems for properties that are undecidable. The point is to capture as many meaningful processes as possible.

Subject Reduction Property: types stay invariant under reduction in the calculus, i.e. if a process has a type T then all its successors have this type as well.

Compositionality: the type of an expression can be derived from the types of its subterms. This property normally manifests itself in the fact that there is a typing rule for every syntactic construction.

There are two more properties often associated with type systems but which are not necessarily satisfied by all of them. Our type systems will, however, satisfy them.

Principal Types: if several types can be assigned to one expression, there is one most general type, called the *principal type*. That is every other type of the expression can be derived from the principal type by some simple operation (e.g. substitution, in our case: hypergraph morphisms).

Type Inference: there is an algorithm which takes an expression as input and assigns a type to it, if one exists. That is, typing is effectively computable. Normally we expect the algorithm to output the principal type of an expression.

There exists a large theory of type systems for the λ -calculus (see e.g. [DM82, Bar90]). One aim of these type systems is to guarantee strong normalization for λ -terms.

In [Mil91] Milner introduced a type system, which was actually called a sort system, for the polyadic π -calculus. A corresponding sort inference algorithm was proposed by Simon Gay [Gay93]. Unlike in the λ -calculus or in the monadic π -calculus, reductions in the polyadic π -calculus can actually lead to runtime errors, which are detected by the sort system. Milner's sort system was extended by recursive types and polymorphism [Tur95, VH93, Vas94] in order to be able to type meaningful processes. While arbitrary recursive types in the λ -calculus are meaningless since they allow the typing of *all* expressions (see [Urz95]), they are essential for the typing of processes. For example, a process emitting its own address—a very common case—can only be typed by recursive types.

Examples of type systems checking other properties than the absence of runtime errors can be found in [PS93], where the input/output capabilities of a port are controlled, and in [Aba97], where security in communication protocols is enforced. In both cases, ports are associated with lattice elements, in the first case the lattice contains the elements $+$ (input capabilities), $-$ (output capabilities), \pm (input and output capabilities) where $+$ $<$ \pm and $-$ $<$ \pm , whereas in the second case it contains the elements *Secret* (secret channel), *Public* (public channel) and *Any* (any channel) where *Secret* $<$ *Any* and *Public* $<$ *Any*.

There are other specialized type systems checking confluence properties [NS97] and the absence of deadlocks [Kob97].

As far as we know there is no type system that takes a more general approach and provides the framework for a type system based on arbitrary lattices and parametrized typing rules for the basic components of a process calculus, i.e. processes and messages. Our contribution is to propose a generation method for type systems, i.e. in our case designing a type system for a certain property only involves filling the parameters of the typing system.

In the following section we will introduce a generic type system for SPIDER, show that it satisfies all the properties of type systems mentioned above, but will leave open some parameters, namely the actual lattice and methods of assigning type information to hyperedges. In a second step we will extend the type system and replace lattices with lattice-ordered monoids.

Since our process calculus is based on graphs, the type of an expression will also be a graph. As it turns out, graphs are very useful for the following reasons:

Recursive Types: in a graph notation recursive types can be represented by cycles in a graph. This relieves us of inserting extra rules for recursive types into our system. This approach is, for example, also taken in [RV97].

Labelling: since in a graph it is possible to explicitly represent ports, processes and messages, it is easy to superimpose additional labelling. We will, among others, design a type system where it is necessary to assign lattice elements to arbitrary pairs of ports. Doing such a thing for types represented as terms seems to be rather difficult to us.

Morphisms: it will turn out that there exists a process graph morphism from an expression into its type. So, if a property is preserved by inverse graph morphism, and this property is true for the type graph, it is also true for the process graph and (because of the subject reduction property) for all its successors.

Furthermore the existence of such a morphism provides an intuitive method to show what has gone wrong if the type inference algorithm does not succeed.

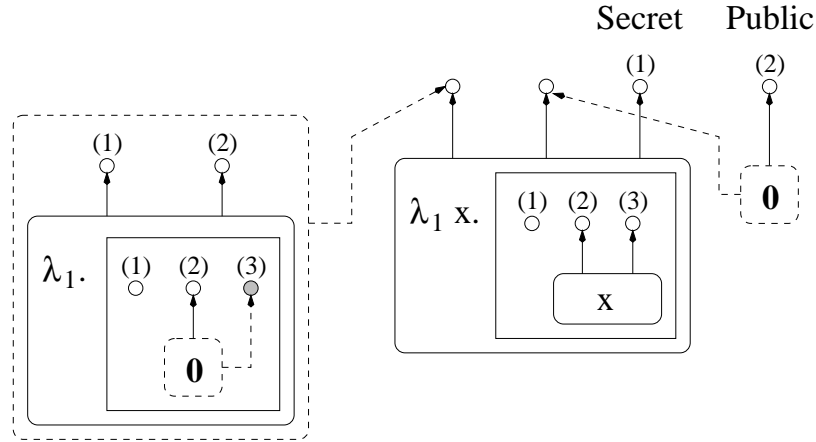
8.1 Motivation

We introduce the features of the type system by giving an example concerning security issues in process communication.

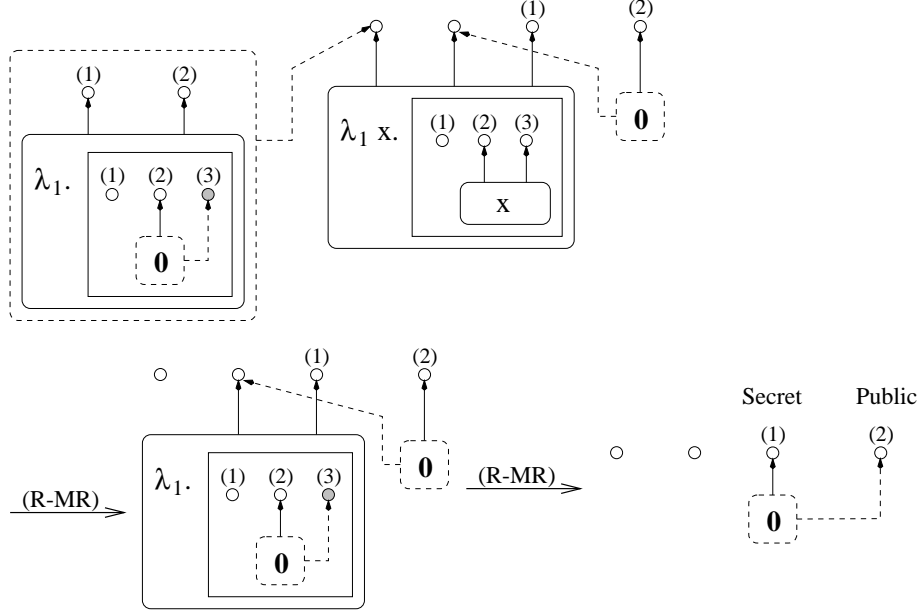
Suppose we have a process which communicates with two types of partners via different ports: first, ports for secret communication, and second, ports for general communication. Not only does a process have to check that no secret information is passed to general communication partner, but it also has to make sure that no port via which secret communication is conducted, is passed to a general communication partner.

We now assume the following situation: the set of external ports of an expression is partitioned into two subsets: the set of public and the set of secret ports. Both can be used to exchange messages with the outside but the process is not allowed to send a message with a secretport attached to it to a public port.

Consider the following SPIDER expression. We will assume that the first external port is secret and that the second external port is public.



Reducing this expression we obtain the following steps:



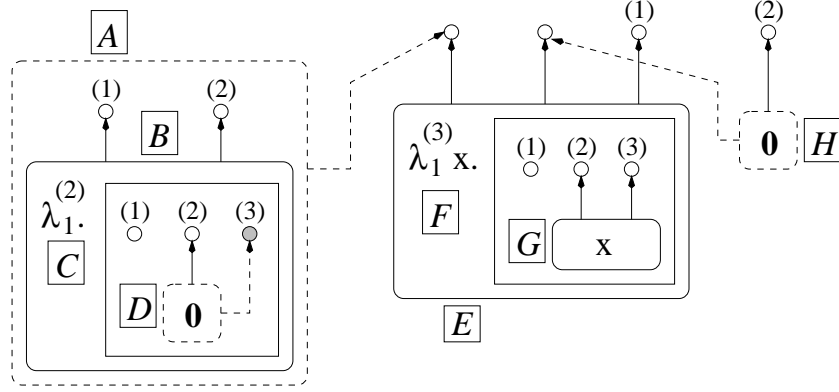
As we can see, this expression violates the conditions imposed on it, i.e. a secret port is sent to a public port, a fact not visible at first glance. The question is: is it possible to decide statically (before runtime) if an expression might run into problems? Our solution is the following: we will compute a type for each expression which does not change while the expression is reduced, and which, at the same time, captures the relevant structural properties of an expression.

Our basic requirements for types are the following: types are hypergraphs and an expression can be mapped onto its type by a graph morphism. Furthermore a type can contain extra annotation, not to be found in the original expression.

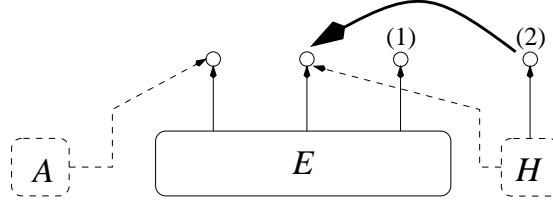
We now proceed and motivate types in a very intuitive, constructive way. Later we will formalize our approach and give non-constructive methods of typing. These non-constructive methods will be better suited for proving the central theorem of a type system: the subject reduction property, i.e. a theorem that says that types do not change while an expression is reduced. But for now we choose a rather informal approach.

We first assign labels to the edges and sub-expressions of our expression. This is just for convenience, so that we can find the edges again in the type graph.

We will now transform the hierarchical process graph into a flat graph, introduce additional content-edges, representing the contents of messages, and fold the resulting type graph in such a way that it respects the subject reduction property:



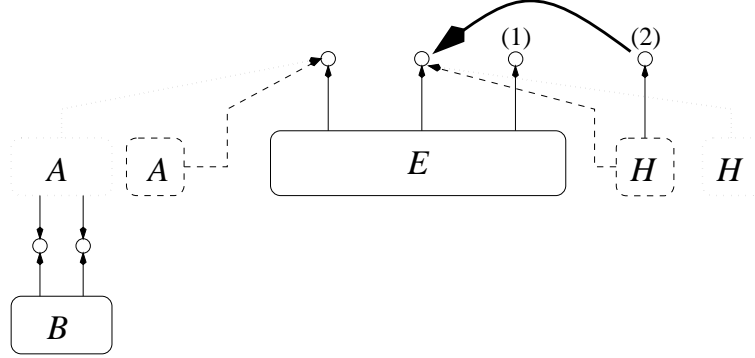
- Our first step in constructing a type graph is to simply take the edges of the “outer level” and to annotate them. Annotation, in this case, simply means that we draw arrows from ports attached to a message to the send-port of the same message. Thus, if an arrow from a secret to a public port appears, this indicates a violation of our conditions. We obtain the following graph:



Looking at the reduction of the expression, it is, of course, obvious that this graph does not correspond to our requirements, namely the subject reduction property. It is intuitively clear that the other levels of the expression have to be integrated into the type graph as well. We are, however, not interested in constructing just another hierarchical graph, but a flat graph, respecting the fact that the levels can merge during reduction.

- Our next step is to introduce edges that represent the content of a message. These kinds of edges do not exist in ordinary SPIDER expressions. We draw them with dotted lines and assume that the last node (in the string of nodes associated with the edge) is the port, the message is sent to. The cardinality of such a content-edge will be the cardinality of the content incremented by one. The edge(s) representing the context can thus be attached to the ports of the content-edge.

In this case only the message with the label A has an actual content: the edge B . This edge is attached accordingly which yields the following graph:

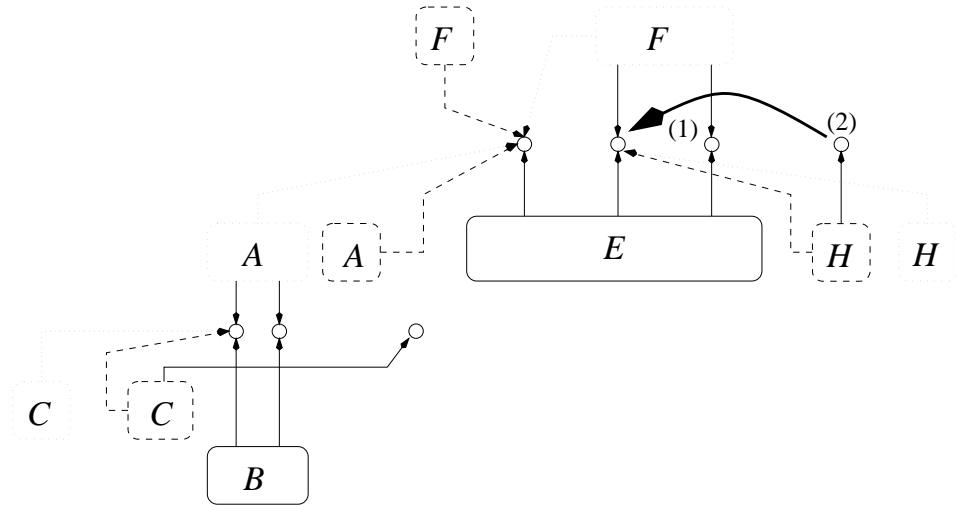


- Our next step involves finding a way to encode the process abstractions λ_1 and $\lambda_1 x$ into the type graph. Each process abstraction will be represented by a message-edge and by a content-edge, whose send-ports are the receiver port. The message-edge connects the receiver port with the ports, which are going to be attached to the ports brought by an arriving message. And the content-edge connects the receiver port with the ports connected to the variable, the content of the arriving message will be substituted for.

In the case of the first process abstraction λ_1 (which is denoted by the label C) the content edge is quite trivial and will be connected to nothing save the receiver port. The message-edge however will be connected to the receiver port and to another port, symbolizing the port that the message brings with it.

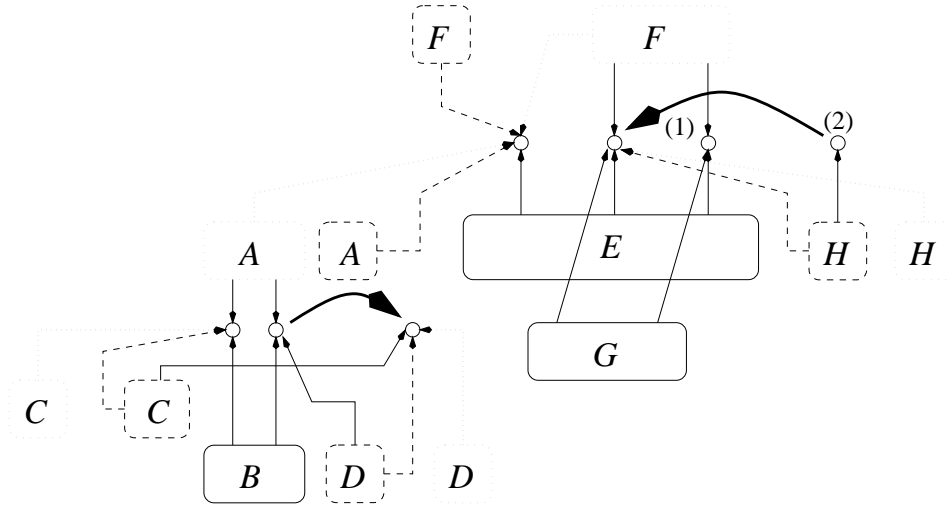
In the case of the other process abstraction $\lambda_1 x$ (denoted by the label F) it is the other way round. The message-edge is quite trivial, but the content-edge is connected to the ports which correspond to the ports of the edge labelled x .

The additional edges are labelled with the letter of the corresponding process abstraction.

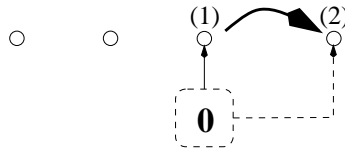


Note that we have drawn messages, but have not annotated them (by a broad arrow). This is because these messages can be considered as virtual, simply representing messages which are yet to come.

- We have now almost finished adding edges. In the final step we will now add an edge labelled G , representing the process with the variable x , and a message- and a content-edge labelled D , representing the corresponding message. Since this is now again a real message, we add the annotation, i.e. a broad arrow between ports.



- Is this the type graph we are looking for? The answer is no and can be motivated rather easily. The following type graph is derived from the final graph in the reduction sequence, by simply annotating it:



It is our intention that there exists a strong morphism, preserving the annotations, from this graph into the type graph. (What it formally means to preserve annotations we have not yet defined.) But there is no way to define such a morphism. This fact is no surprise since we have neglected one important feature of the calculus: during message reception previously distinct ports may merge. That is, it is now our task to find sets of ports which might be merged during the reduction process and to actually fuse them in the type graph.

Some consideration reveals two simple rules for merging ports (and, at the same time, hyperedges):

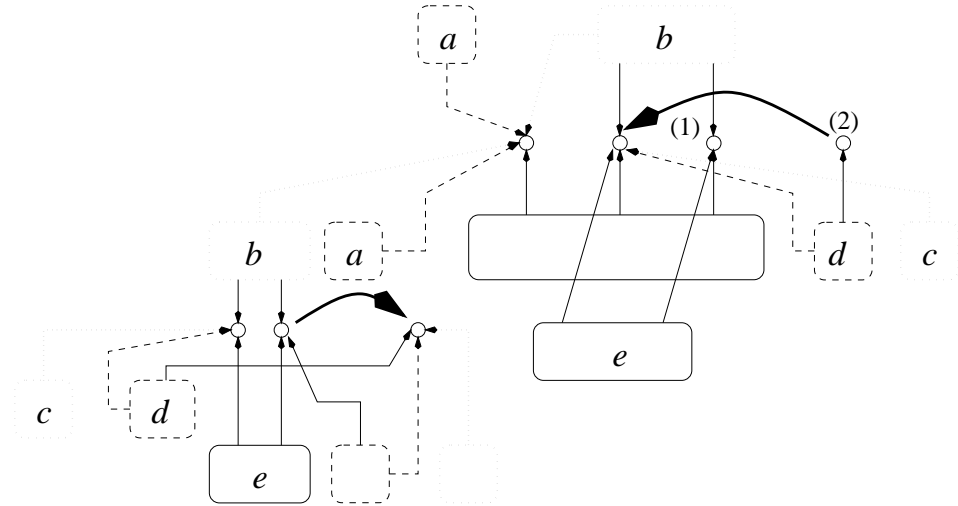
- If two message-edges are sent to the same port these messages are to be merged (and all of the ports attached to them are to be merged in the correct order)
- If two content-edges are sent to the same port these messages are to be merged (and all of the ports attached to them are to be merged in the correct order)

We will add a third rule whose intent will not be clear now, but will be revealed in the following sections:

- If two process-edges are associated with the very same string of ports, they are also to be merged.

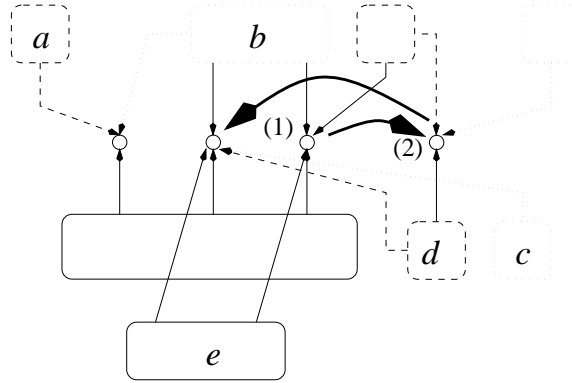
It seems clear that this process has to be executed several times, before all relevant edges are merged.

We will now compute the sets of edges in the type graph, which are to be merged. Edges which we will now fuse are denoted by the same letter.



It now becomes clear why it was necessary to insert dummy edges representing the process abstractions. These dummy edges are now essential to ensure that the correct ports are merged.

The folding process (which can actually be described by a surjective morphism) yields the following result:



This is now the final type graph and it has all the desirable properties. It also contains, as annotation, an arrow leading from the first (secret) external port to the second (public) external port, which means that a secret port is sent to a public port, thereby violating our conditions on secrecy.

Note that in all other cases (second external port is secret, or both external ports are public) the process graph would have been well-typed.

8.2 Lattices, Monoids and Type Graphs

We have stated above that we want to associate components of a process graph with lattice elements, describing e.g. their input/output capabilities or their security status. The first idea is to associate type graphs with an additional labelling of the ports, that maps each port onto a lattice element. As it turns out, this approach is not sufficiently general. Sometimes we might want to associate a lattice element with a pair of ports or even with a hyperedge. E.g. in the example in section 8.1 the labelling function for a graph G was $l : V_G^2 \rightarrow \{true, false\}$ where $l(v_1, v_2) = true$ indicates that there exists an arrow from v_1 to v_2 .

We solve this problem by associating only *one* lattice element to every hypergraph. This works because the set of all mappings assigning labels to ports can also be regarded as a lattice if we consider pointwise order.

Later we will extend our type system to work with monoids. One example where this is needed is a type system which guarantees upper bounds for the number of messages attached to each port. When composing a type it is not appropriate to take the supremum of two numbers, we actually have to add them. It turns out that we need lattice-ordered commutative monoids. Since lattices, at least lattices with bottom elements, are only a special case of these monoids we will concentrate on monoids and only mention simplifications for the case of lattices.

The following definitions are from [Bir67, Cro93]

8.2.1 Lattice-Ordered Commutative Monoids

Definition 8.2.1 (Lattice) A *lattice* is a partially ordered set (or poset) (I, \leq) such that every finite nonempty subset F of I has a greatest lower bound or infimum $(\bigwedge F)$ and a least upper bound or supremum $(\bigvee F)$.

(We define $a \wedge b := \bigwedge \{a, b\}$, $a \vee b := \bigvee \{a, b\}$.) \square

If (I, \leq) has a smallest element it is called *bottom* (\perp), the greatest element, if it exists, is called *top* (\top).

$$\perp := \bigvee \emptyset \quad \top := \bigwedge \emptyset$$

Example: The subtype relation in most programming languages is a lattice if all types have a common supertype.

Definition 8.2.2 (Lattice-ordered Commutative Monoid)

A *lattice-ordered commutative monoid* (*l-monoid*) is a tuple $(I, +, \leq)$ (I for short) where I is a set, $+: I \times I \rightarrow I$ is a binary operation and \leq is a partial order which satisfy:

- $(I, +)$ is a commutative monoid, i.e. $+$ is associative and commutative and there is a neutral element 0 with $\forall a \in I : 0 + a = a$
- (I, \leq) is a lattice.
- For $a, b, c \in I$: $a + (b \vee c) = (a + b) \vee (a + c)$

\square

Every lattice with a bottom element \perp (I, \vee, \leq) is a lattice-ordered commutative monoid where \perp is the neutral element. And an l-monoid $(I, +, \leq)$ is called a lattice if $+$ and \vee coincide.

Definition 8.2.3 (Residuated l-monoid)

Let I be an l-monoid. Let $a, b \in I$. The *residual* $a - b$ is the smallest x (if it exists) such that $a \leq x + b$. I is called *residuated* iff all residuals $a - b$ exist in I for $a, b \in I$. \square

If I is a group where $-a$ denotes the inverse of a , a residual $a - b$ is the same as $a + (-b)$.

Definition 8.2.4 (Idempotency)

Let I be an l-monoid. An element $a \in I$ is called *idempotent* iff $a + a = a$. \square

In a lattice all elements are idempotent.

Examples: We will give some examples for l-monoids:

- The integers with the usual \leq -order and sum form a residuated l-monoid.

- The positive integers form a residuated l-monoid where $+$ is integer multiplication and

$$a \leq b \iff (a \text{ divides } b)$$

In this case the supremum is the smallest common multiple, the infimum is the greatest common divisor and the residual of a and b is $\lceil a \div b \rceil$ (divide a by b and take the smallest integer which is greater or equal than the result).

- The set Int_∞ —the integers with infinity (∞) and minus infinity ($-\infty$)—forms a residuated l-monoid where \leq is the usual order on the integers and $+$ is integer summation with

$$\forall x \in Int_\infty : \infty + x := \infty, \quad -\infty + x := \begin{cases} -\infty & \text{if } x \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

Supremum and infimum are the usual maximum and minimum. For integers a, b the residual is $a - b = a + (-b)$. Otherwise we obtain

$$\begin{aligned} (-\infty) - x &= -\infty, & x - \infty &= -\infty \\ \infty - x &= \begin{cases} \infty & \text{if } x \neq (-\infty) \\ -\infty & \text{otherwise} \end{cases} \\ x - (-\infty) &= \begin{cases} \infty & \text{if } x \neq (-\infty) \\ -\infty & \text{otherwise} \end{cases} \end{aligned}$$

- Let I be a residuated l-monoid with residuals $a - b$. Then $\{a \mid a \in I, a \geq 0\}$ (the positive cone of I) is also a residuated l-monoid where the residual of a, b is $(a - b) \vee 0$.
- Let $(I, +, \leq)$ be a residuated l-monoid and let S be an arbitrary set. Then $\{S \rightarrow I\}$, i.e. the set of all functions from S into I is also a residuated l-monoid (sum and residuals are taken pointwise).
- Let $(I, +, \leq)$ be a residuated l-monoid and let $n \in \mathbb{N}$. Then I^n with pointwise order is also a residuated l-monoid (sum and residuals are taken pointwise).
- Let $(I, +, \leq)$ be an l-monoid with a smallest element \perp . We define $a \oplus b := (a + b) \vee a \vee b$. Then (I, \oplus, \leq) is an l-monoid with neutral element \perp . For monoid elements $a_j \in I, j \in J$ it follows that

$$\sum_{j \in J} a_j \leq \bigoplus_{j \in J} a_j$$

if the index set J contains more than one element. If $J = \emptyset$ it follows that

$$\sum_{j \in J} a_j = 0 \geq \perp = \bigoplus_{j \in J} a_j$$

If $+$ equals \vee , i.e. if I is a lattice, the operations \oplus and \vee coincide.

Proposition 8.2.5 *The following laws hold in a residuated l-monoid I . Let $a, b, c \in I$.*

$$\text{Monotonicity:} \quad a \leq b \Rightarrow a + c \leq b + c \quad (8.1)$$

$$a \leq b \Rightarrow a - c \leq b - c \quad (8.2)$$

$$b \geq c \Rightarrow a - b \leq a - c \quad (8.3)$$

$$\text{Residuals:} \quad a \leq (a - b) + b \quad (8.4)$$

$$(a + b) - b \leq a \quad (8.5)$$

Proof:

(8.1) Since $a \leq b$ it follows that $a \vee b = b$. Thus $b + c = (a \vee b) + c = (a + c) \vee (b + c)$ which implies that $a + c \leq b + c$.

(8.4) Straightforward with the definition of $a - b$.

(8.2) $(b - c) + c \stackrel{(8.4)}{\geq} b \geq a$. $a - c$ is the smallest x with $x + c \geq a$. Therefore $a - c \leq b - c$.

(8.3) $(a - c) + b \geq (a - c) + c \stackrel{(8.4)}{\geq} a$. $a - b$ is the smallest x with $x + b \geq a$. Therefore $a - c \leq b - c$.

(8.5) $(a + b) - b$ is the smallest x such that $a + b \leq x + b$. Since a satisfies this inequality as well we obtain $(a + b) - b \leq a$.

□

Definition 8.2.6 (l-monoid Morphism) Let I, I' be l-monoids. A mapping $t : I \rightarrow I'$ is called *l-monoid morphism* (or monoid morphism) if for all $a, b \in I$:

$$a \leq b \Rightarrow t(a) \leq t(b) \quad (8.6)$$

$$t(0) = 0 \quad (8.7)$$

$$t(a + b) = t(a) + t(b) \quad (8.8)$$

□

The composition of two monoid morphisms is again a monoid morphism. Since composition is associative and identity morphisms exist, the l-monoid morphisms form a category \mathcal{M} .

In the case where $+$ equals \vee , law (8.8) implies (8.6), i.e. in this case a monoid morphism coincides with a join-morphism preserving \perp .

8.2.2 Type Graphs and Type Functors

As described above, type graphs will be arbitrary, non-hierarchical hypergraphs, where each graph $G[\chi]$ is associated with an l-monoid element a . Every type graph has the form $G[\chi, a]$. We do not work with one single l-monoid but there will be one l-monoid for every type graph.

Type functors map type graphs onto their corresponding l-monoids and describe how l-monoid elements are transformed by morphisms.

We will now use a few concepts of category theory, for a short introduction see section 2.2.1

Definition 8.2.7 (Type Functor and Type Graphs) Let

$$F : \mathcal{G}(Z, L) \rightarrow \mathcal{M}$$

be a functor from the category of simple hypergraph morphisms $\mathcal{G}(Z, L)$ into the category of l-monoid morphisms \mathcal{M} .

Then F is called a *type functor*.

A *type graph* $T = G[\chi, a]$ consists of a hypergraph $G[\chi]$ and a monoid element $a \in F(G)$. \square

Notation: We also write F_ϕ instead of $F(\phi)$ (F applied to a morphism ϕ). Let $H = G[\chi]$ be a hypergraph and let $a \in F(G)$. Then $H[a] := G[\chi, a]$.

Example: Let F be a type functor that maps every graph G to a lattice consisting of all labellings of the form $a : V^G \rightarrow \text{Int}_\infty$ where $(\text{Int}_\infty, \vee, \leq)$ is a lattice. Let $\phi : G \rightarrow G'$ be a graph morphism. We assume that $a' := F_\phi(a)$ where

$$a'(v') := \bigvee_{\phi(v)=v'} a(v) \text{ if } v \in V_G$$

If we regard the l-monoid $(\text{Int}_\infty, +, \leq)$, one possible type functor is, e.g., J with $a' := J_\phi(a)$ and

$$a'(v') := \sum_{\phi(v)=v'} a(v) \text{ if } v \in V_G$$

Type graph morphisms are expected not only to preserve the graph structure, but also the order in the underlying monoid.

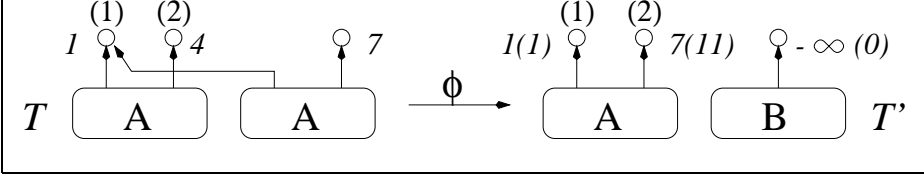
Definition 8.2.8 (Type Graph Morphisms) Let F be a type functor and let $\phi : G[\chi] \rightarrow G'[\chi']$ be a graph morphism. Let $a \in F(G)$, $a' \in F(G')$ and let $\theta \in \{=, \leq, \geq\}$. We say that ϕ is a (F, θ) -morphism from $G[\chi, a]$ into $G'[\chi', a']$

$$\phi : G[\chi, a] \xrightarrow{F, \theta} G'[\chi', a']$$

iff $F_\phi(a) \theta a'$. \square

The category of type graph morphisms with respect to a type functor F is called \mathcal{T}_F .

Example: The following morphism ϕ (the two hyperedges labelled A are mapped onto the corresponding hyperedge on the right hand side) is a $(F, =)$ -morphism (and a $(J, =)$ -morphism for the numbers in brackets). F, J are the two functors from the previous example, where F takes the supremum of monoid elements and J adds them up. Note that 0 is the empty sum and $-\infty$ (the bottom element) is the supremum of the empty set.



Our method of graph construction can also be applied to type graphs.

Definition 8.2.9 (Constructing Type Graphs) Let F be a type functor.

Let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ be a discrete factorization. Furthermore let $T_i := G_i[\chi_i, a_i]$ be type graphs with $|\chi_i| = m_i$.

We define

$$G[\chi] := \bigotimes_{i=1}^n (G_i[\chi_i], \zeta_i)$$

Let $\eta_i : G_i[\chi_i] \rightarrow G[\chi]$ be the corresponding embeddings. We define $a := \sum_{i=1}^n F_{\eta_i}(a_i)$. Then

$$\bigotimes_{i=1}^n (T_i, \zeta_i)_F := G[\chi, a]$$

□

Note: We can now easily define the construction of type graphs for other graph representations: let C be an n -ary context and let T_1, \dots, T_n be type graphs. Then

$$C\langle T_1, \dots, T_n \rangle_F := \bigotimes_{i=1}^n (T_i, \zeta_i)_F$$

where $(\zeta_i)_{i \in \{1, \dots, n\}}$ is the unique factorization with

$$C\langle x_1, \dots, x_n \rangle \cong \bigotimes_{i=1}^n (var_{m_i}(x_i), \zeta_i)$$

(see proposition 2.2.18).

Note: Let F be a type functor such that $F(G)$ is a lattice for every graph G . Then $G[\chi, a]$ is the co-limit of

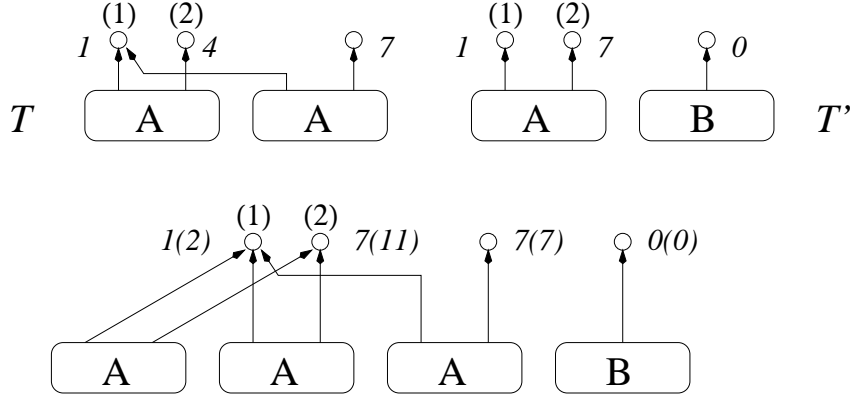
$$\zeta_i : \mathbf{m}_i[\perp] \xrightarrow{F, \leq} D[\perp] \text{ and } \phi_i : \mathbf{m}_i[\perp] \xrightarrow{F, \leq} G_i[\chi_i, a_i]$$

in the category of type graph morphisms.

This is not true if $F(G)$ is not always a lattice. Consequently type systems based on monoids are more difficult to handle than type systems based on lattices. Therefore we will start with type systems based on lattices in section 8.3 and deal with the other variant in section 8.4.

Example: We define $T_1 \sqcup_F T_2 := C\langle T_1, T_2 \rangle_F$ where $C\langle x_1, x_2 \rangle := \text{proc}_n(x_1) \sqcup \text{proc}_n(x_2)$ with $n = \text{card}(T_1) = \text{card}(T_2)$.

If we construct $T \sqcup_F T'$ with T, T' from the example above, the result is (the numbers in brackets denote the labelling of $T \sqcup_F T'$):



Note that the co-limit in the category of (J, \leq) -morphisms would be (at least in this case) identical to $T \sqcup_F T'$.

Definition 8.2.10 (Isomorphism of Type Graphs) $G[\chi, a]$ and $G'[\chi', a']$ are called isomorphic

$$G[\chi, a] \cong_F G'[\chi', a']$$

if there exists an isomorphism $\phi : G[\chi] \rightarrow G'[\chi']$ with $F_\phi(a) = a'$. We write

$$G[\chi, a] \tilde{<}_F G'[\chi', a']$$

if $F_\phi(a) \leq a'$.

$\tilde{<}_F$ is reflexive and transitive and

$$G[\chi, a] \tilde{<}_F G'[\chi', a'], G[\chi, a] \tilde{>}_F G'[\chi', a'] \Rightarrow G[\chi, a] \cong_F G'[\chi', a']$$

□

Just as we can construct new graphs with a context, we can also pack together morphisms ψ_1, \dots, ψ_n in order to form a morphism $C\langle \psi_1, \dots, \psi_n \rangle$ (see definition 2.2.22). We now demonstrate that this kind of construction is also feasible for (F, θ) -morphisms.

Lemma 8.2.11 (Composition of Type Graph Morphisms)

Let ψ_1, \dots, ψ_n be (F, θ) -morphisms and let C be a discrete context with holes of cardinality $\text{card}(\psi_1), \dots, \text{card}(\psi_n)$. Then

$$C\langle \psi_1, \dots, \psi_n \rangle \text{ is also a } (F, \theta)\text{-morphism} \quad (8.9)$$

Proof: Let $\psi_i : G_i[\chi_i, a_i] \rightarrow G'_i[\chi'_i, a'_i]$ and let $\psi := C\langle\psi_1, \dots, \psi_n\rangle_F$ with

$$\psi : C\langle G_1[\chi_1], \dots, G_n[\chi_n]\rangle \rightarrow C\langle G'_1[\chi'_1], \dots, G'_n[\chi'_n]\rangle$$

such that $\eta'_i \circ \psi_i = \psi \circ \eta_i$ where

$$\begin{aligned} \eta_i : G_i[\chi_i] &\rightarrow C\langle G_1[\chi_1], \dots, G_n[\chi_n]\rangle \\ \eta'_i : G'_i[\chi'_i] &\rightarrow C\langle G'_1[\chi'_1], \dots, G'_n[\chi'_n]\rangle \end{aligned}$$

are canonical factorizations for $i \in \{1, \dots, n\}$

It is left to show that ψ is a (F, θ) -morphism:

$$F_\psi\left(\sum_{1 \leq i \leq n} F_{\eta_i}(a_i)\right) = \sum_{1 \leq i \leq n} F_\psi(F_{\eta_i}(a_i)) = \sum_{1 \leq i \leq n} F_{\eta'_i}(F_{\psi_i}(a_i)) \theta \sum_{1 \leq i \leq n} F_{\eta'_i}(a'_i)$$

□

With the proposition above we can show another important property of $\tilde{<}$:

Lemma 8.2.12 (Properties of $\tilde{<}$) $\tilde{<}_F$ is preserved by substitution, i.e. if

$$T_i \tilde{<}_F T'_i \quad \text{for } i \in \{1, \dots, n\} \text{ then}$$

$$C\langle T_1, \dots, T_n \rangle \tilde{<}_F C\langle T'_1, \dots, T'_n \rangle$$

for any discrete context C with holes of cardinality $\text{card}(T_1), \dots, \text{card}(T_n)$.

Proof: There are (F, \leq) -isomorphisms $\psi_i : T_i \rightarrow T'_i$. With lemma 8.2.11 it follows that there is a strong (F, \leq) -morphism

$$\psi = C\langle\psi_1, \dots, \psi_n\rangle : C\langle T_1, \dots, T_n \rangle \rightarrow C\langle T'_1, \dots, T'_n \rangle$$

Since the ψ_i are isomorphisms and ψ is constructed as described in proposition 2.2.12, (B) by a co-limit and is therefore unique it follows that ψ is an isomorphism and therefore

$$C\langle T_1, \dots, T_n \rangle \tilde{<} C\langle T'_1, \dots, T'_n \rangle$$

□

We need a method of converting each process graph into a type graph. This is done by a linear mapping, which is linear in the sense defined below. This mapping is one of the parameters of the type system, i.e. we do not fix it a priori. In the following section we define conditions having to be satisfied by such a linear mapping in order to be acceptable for the type system.

Proposition 8.2.13 (Linear Mapping) *Let F be a type functor. We define a function A which maps basic graphs of the form $z_n(l)$ onto type graphs, i.e. $A(z_n(l)) = T$ with $\text{card}(T) = n$. Furthermore $H_1 \cong H_2$ for two basic graphs H_1, H_2 implies $A(H_1) \cong_F A(H_2)$.*

We expand A to arbitrary hypergraphs in the following way:

$$A(C\langle H_1, \dots, H_n \rangle) := C\langle A(H_1), \dots, A(H_n) \rangle_F$$

where H_1, \dots, H_n are basic graphs. A is well-defined, i.e. $H \cong H'$ implies $A(H) \cong_F A(H')$, and is called a linear mapping wrt. F . Furthermore

$$A(C\langle H_1, \dots, H_n \rangle) := C\langle A(H_1), \dots, A(H_n) \rangle_F$$

for arbitrary hypergraphs H_1, \dots, H_n .

Proof:

- We will first show that $A(H)$ is well-defined:
 - We will first consider the case where $C\langle H_1 \rangle \cong H$ is a basic graph. We have to show that $A(H) \cong C\langle A(H_1) \rangle$. Since H_1 is also a basic graph and since factorization into basic graphs is unique (proposition 2.2.20) it follows that $H_1 \cong H$ and $C\langle x_1 \rangle \cong \text{var}_{\text{card}(H)}(x_1)$. Thus

$$A(H) \cong_F A(H_1) \cong_F C\langle A(H_1) \rangle$$

- Let $C\langle H_1, \dots, H_n \rangle \cong C'\langle H'_1, \dots, H'_k \rangle$ where C, C' are discrete contexts and the $H_1, \dots, H_n, H'_1, \dots, H'_k$ are basic graphs. Proposition 2.2.20 implies that $n = k$ and there exists a permutation $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ such that $H_{\alpha(i)} \cong H'_i$ for every i and $C\langle x_1, \dots, x_n \rangle \cong C'\langle x_{\alpha(1)}, \dots, x_{\alpha(n)} \rangle$. It follows that

$$\begin{aligned} C\langle A(H_1), \dots, A(H_n) \rangle &\cong_F C'\langle A(H_{\alpha(1)}), \dots, A(H_{\alpha(n)}) \rangle \\ &\cong_F C'\langle A(H'_1), \dots, A(H'_n) \rangle \end{aligned}$$

- We will now prove that A satisfies

$$A(C\langle H_1, \dots, H_n \rangle) \cong_F C\langle A(H_1), \dots, A(H_n) \rangle_F$$

even if H_1, \dots, H_n are not necessarily basic graphs. Every H_i has the form $C_i\langle H_{i1}, \dots, H_{im_i} \rangle$ where C_i is a discrete context and H_{i1}, \dots, H_{im_i} are basic graphs.

We define $N_n := \sum_{i=1}^n m_i$ and

$$\tilde{C}\langle x_1, \dots, x_{x_n} \rangle := C\langle C_1\langle x_1, \dots, x_{N_1} \rangle, \dots, C_n\langle x_{N_{n-1}+1}, \dots, x_{N_n} \rangle \rangle$$

and \tilde{C} is also a discrete context

$$\begin{aligned} A(C\langle H_1, \dots, H_n \rangle) &\cong_F A(\tilde{C}\langle H_{11}, \dots, H_{1m_1}, \dots, H_{n1}, \dots, H_{nm_n} \rangle) \\ &\cong_F \tilde{C}\langle A(H_{11}), \dots, A(H_{1m_1}), \dots, A(H_{n1}), \dots, A(H_{nm_n}) \rangle_F \\ &\cong_F C\langle C_1\langle A(H_{11}), \dots, A(H_{1m_1}) \rangle_F, \dots, C_n\langle A(H_{n1}), \dots, A(H_{nm_n}) \rangle_F \rangle_F \\ &\cong_F C\langle A(H_1), \dots, A(H_n) \rangle_F \end{aligned}$$

□

If we draw a parallel between the factorization of graphs and the representation of a vector as a linear composition of base vectors, the proposition above has its equivalent in the fact that a linear mapping between vector spaces is determined uniquely by the images of the base vectors.

8.3 A Type System Based on Lattices

We now present the first version of the type system, which enables us to type higher-order communication, detects runtime-errors and produces type graphs labelled with lattices. With this type system, we are able to check a few interesting properties of SPIDER expressions, namely input/output capabilities and secrecy in communication.

8.3.1 The Type System

In this section we assume that there is a fixed type functor F such that for every simple hypergraph G $F(G)$ is a lattice with a smallest element \perp .

Furthermore we assume that the set of Z of edge sorts contains at least the elements *proc*, *mess*, *cont*, representing “process”, “message” and “message content” and that the set L of edge labels contains the dummy element \diamond .

Definitions: We will define $z_n := z_n(\diamond)$. The subsets of the hyperedge set E_T of a type graph containing processes, messages or message contents are denoted by $P_T := E_T^{proc}$, $M_T := E_T^{mess}$, $Co_T := E_T^{cont}$. For $c \in Co_T$ we define $send_T(c) := \lfloor s_T(c) \rfloor_{card(c)}$.

Furthermore there is a basic mapping A satisfying the restraints specified in table 8.1.

Definition 8.3.1 (True Type Graphs) G is a *true graph* if it satisfies:

$$\forall p_1, p_2 \in P_G : \quad s_G(p_1) = s_G(p_2) \Rightarrow p_1 = p_2 \quad (8.10)$$

$$\forall q_1, q_2 \in M_G : \quad send_G(q_1) = send_G(q_2) \Rightarrow q_1 = q_2 \quad (8.11)$$

$$\forall c_1, c_2 \in Co_G : \quad send_G(c_1) = send_G(c_2) \Rightarrow c_1 = c_2 \quad (8.12)$$

$T = G[\chi, a]$ is called a *true type graph* iff G is a true graph. □

In a true type graph T there is, for every port v , at most one message m and at most one content c such that $send_T(m) = v$ and $cont_T(c) = v$. Thus we can define unique partial functions $send_T$ and $cont_T$ on V_T with $send_T(v) := m$ and $cont_T(v) := c$.

Condition (1) of the linear mapping makes sure that equivalent processes have the same type. Conditions (2) and (3) are utilized in lemma 8.3.8 and in proposition 8.3.9 where invariance of type under substitution is shown.

$$(1) H_1 \equiv H_2 \Rightarrow A(H_1) \cong_F A(H_2)$$

$$(2) A(proc_n(x)) \cong_F A(proc_n(y)) \text{ if } sort(x) = n = sort(y)$$

$$(3) A(proc_n(S)) \cong_F A(proc_n(S[H/x])) \text{ for every process graph } H \text{ with } card(H) = sort(x) \text{ and for every process description } S \notin X.$$

$$A(mess_n(S)) \cong_F A(mess_n(H[H'/x])) \text{ for process graphs } H, H' \text{ with } card(H') = sort(x).$$

(4) There is a strong morphism

$$\phi : proc_n[\perp] \xrightarrow{F, \leq} A(proc_n(S))$$

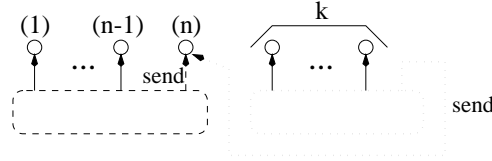
for every $n \in \mathbb{N}$ and every process description S .

(5) There is a strong morphism

$$\phi : \sigma_{1, \dots, n}(\theta_{n, n+k+1}(mess_n \oplus cont_{k+1}))[\perp] \xrightarrow{F, \leq} A(mess_n(H))$$

where $n \in \mathbb{N}$, H is a process graph and $k := card(H)$.

$\sigma_{1, \dots, n}(\theta_{n, n+k+1}(mess_n \oplus cont_{k+1}))$ has the following form



This means that a message is represented not only by a message but also by another hyperedge (drawn with a dotted line) attached to the send-port representing the content of the message.

Table 8.1: Conditions for the linear mapping

Conditions (5), (8.11) and (8.12) are necessary because of the mobility of ports and processes in SPIDER. They will have the effect that ports which might be merged during reduction, are already merged in the type graph. This ensures the subject reduction property (see proposition 8.3.11).

Conditions (4) and (8.10) are less central and could be discarded without harm. We will, however, need them in the type system based on monoids. Thus we will introduce them now in order to facilitate comparison of the two type systems.

Now we present the typing rules of this type system. Note that two parameters of the type system are still not fixed: the functor F (see definition 8.2.7)

and the linear mapping A (see proposition 8.2.13).

Definition 8.3.2 (Typing Rules) A *type environment* of G is a set E containing assignments of the form $x : \eta_x$, where each x occurs at most once. $\eta_x : \mathbf{m}_x \rightarrow G[\varepsilon]$ is an embedding where $m_x := \text{sort}(x)$ and G is a true graph.

$E \setminus x$ denotes the set E without assignments of the form $x : \eta_x$.

A process graph or process description has a *type* of the form E, T where E is a type environment for G and $T = G[\chi, a]$. We write $E, T \vdash_{A,F}^L S$ (or $E, T \vdash S$ for short).

The following rules describe how a type can be assigned to an expression.

$$\begin{array}{l}
 \text{(TL-VAR)} \quad E, G[\eta_x(\chi_{\mathbf{m}_x}), a] \vdash x \text{ if } x : \eta_x \in E \\
 \\
 \text{(TL-REPL)} \quad \frac{E, G[\chi, a] \vdash H}{E, G[\chi, a] \vdash !H} \\
 \\
 \text{(TL-CON)} \quad \frac{E, G[\phi(\zeta_i(\chi_{\mathbf{m}_i})), a_i] \vdash H_i, \quad \zeta_i : \mathbf{m}_i \rightarrow D, \quad \phi : D \rightarrow G[\chi]}{E, G[\chi, \bigvee a_i] \vdash \bigotimes_{i=1}^n (H_i, \zeta_i)} \\
 \\
 \text{(TL-PROC)} \quad \frac{E, G[\chi, a] \vdash S, \quad \phi : A(\text{proc}_n(S)) \xrightarrow{F, \leq} G[\chi, a]}{E, G[\chi, a] \vdash \text{proc}_n(S)} \\
 \\
 \text{(TL-MESS)} \quad \frac{E, G[\chi', a] \vdash H, \quad \phi : A(\text{mess}_n(H)) \xrightarrow{F, \leq} G[\chi, a]}{E, G[\chi, a] \vdash \text{mess}_n(H)} \\
 \text{if } \exists c \in Co_G : s_G(c) = \chi' \circ [\chi]_n \\
 \\
 \text{(TL-PA)} \quad \frac{E \setminus x \cup \{x : \eta_x\}, G[\chi, a] \vdash H}{E, G[[\chi]_{1\dots m}, a] \vdash \lambda_k x. H} \\
 \text{if } \exists q \in M_G, \quad c \in Co_G, \quad p \in P_G : \\
 s_G(q) = [\chi]_{m+1\dots m+n k}, \quad s_G(c) = \eta_x(\chi_{\mathbf{m}_x}) \circ [\chi]_k, \quad s_G(p) = [\chi]_{1\dots m}
 \end{array}$$

□

The rules were constructed with two purposes in mind: first if $E, T \vdash H$, then there exists a morphism $\phi : A(H) \xrightarrow{F, \leq} T$. And second, the subject reduction property is satisfied, i.e. the type stays invariant under reduction.

The morphisms in the preconditions of (TL-PROC) and (TL-MESS) (and of course also rule (TL-CON)) are there to ensure the existence of ϕ . The conditions in (TL-MESS) and (TL-PA), demanding the existence of certain hyperedges, are central for subject reduction property. They ensure that ports that might be merged during reduction, are already merged in the type graph.

8.3.2 Subject Reduction Property

We now concern ourselves with one of the central properties of type systems: the subject reduction property, which basically says that the type of an expression

does not change during reduction. That is if $E, T \vdash H$ and $H \rightarrow^* H'$ it follows that $E, T \vdash H'$. Note, however, that H' might have “more” types than H , i.e. the inverse implication is normally not true.

We will first prove a few simple, but useful, lemmas:

Lemma 8.3.3 *If $E, G[\chi, a] \vdash H$ then $\text{card}(H) = |\chi|$.*

Proof: By induction on the typing rules. \square

Lemma 8.3.4 *Let S be a process description.*

If $E, G[\chi, a] \vdash S$ and $a \leq b$ then $E, G[\chi, b] \vdash S$.

Proof: Straightforward by induction on the typing rules. \square

Lemma 8.3.5 (Weakening Law) *Let $E, G[\chi, a] \vdash S$ and let $E \subseteq E'$ where E' is a type environment. It follows that $E', G[\chi, a] \vdash S$.*

Proof: Straightforward by induction on the typing rules. \square

Lemma 8.3.6 *Let $E, G[\chi, a] \vdash S$ and let $x \notin \text{free}(S)$.*

It follows that $E \setminus x, G[\chi, a] \vdash S$.

Proof: Straightforward by induction on the typing rules. \square

In the following proofs it will sometimes be necessary to “reverse” the typing process, i.e. to find out which rules were used in order to type an expression. While this reversal is unambiguous in the case of the rules (TL-VAR), (TL-REPL), (TL-PROC), (TL-MESS) and (TL-PA), it is not so obvious in the case of (TL-CON). We will therefore need the following lemma:

Lemma 8.3.7 *Let $E, G[\chi, a] \vdash \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow D$.*

It follows that there is a strong morphism $\psi : D \rightarrow G[\chi]$ such that

$$E, G[\psi(\eta_i(\chi_{\mathbf{m}_i})), a] \vdash H_i$$

Proof: See appendix A.2.1. \square

We will now investigate how types behave under substitution and alpha-conversion:

Lemma 8.3.8 *Let $E, T \vdash S$ such that $y \notin \text{free}(S)$ and $\{x : \eta_x\} \in E$ where $\text{sort}(x) = \text{sort}(y) = n$. It follows that*

$$E \setminus x \setminus y \cup \{y : \eta_x\}, T \vdash S[\text{proc}_n(y)/x]$$

Proof: By induction on the typing of H and with condition (2) of the linear mapping. \square

Proposition 8.3.9 (Substitution)

If $E \setminus x \cup \{x : \eta_x\}, G[\chi, a] \vdash S$ and $E, G[\eta_x(\chi_{\mathbf{m}_x}), a] \vdash J$ then

$$E, G[\chi, a] \vdash S[J/x]$$

Proof: We proceed by induction on the typing of H :

(TL-PROC), (TL-MESS)

- $S = \text{proc}_n(x)$: in this case it follows with (TL-VAR) and (TL-PROC) $\eta_x(\chi_{\mathbf{m}_x}) = \chi$. And since $S[J/x] := J$ it follows that

$$E, G[\chi, a] \vdash S[J/x]$$

- $S = \text{proc}_n(y)$ where $x \neq y$. Since $x \notin \text{free}(\text{proc}_n(y))$ it follows with lemma 8.3.6 that $(E \setminus x \cup \{x : \eta_x\}) \setminus x, G[\chi, a] \vdash S$. And since $S[J/x] := S$ and $(E \setminus x \cup \{x : \eta_x\}) \setminus x = E \setminus x$ it follows that

$$E \setminus x, G[\chi, a] \vdash S[J/x]$$

The rest follows with lemma 8.3.5.

- $S = \text{proc}_n(S')$ or $S = \text{mess}_n(H)$: the proposition follows with the induction hypothesis and condition (3) of the linear mapping.

(TL-REPL) immediate with the induction hypothesis.

(TL-CON) with lemma 8.3.7 and the induction hypothesis.

(TL-PA) $S = \lambda_k y. H$. There are the following cases:

- $y = x$: in this case $S[J/x] := S$ and as in the case $S = \text{proc}_n(y)$ above the proposition follows with lemmas 8.3.6 and 8.3.5.
- $y \notin \text{free}(J)$: immediate with the induction hypothesis.
- $y \in \text{free}(J), z \notin \text{free}(H) \cup \text{free}(J)$, i.e.

$$(\lambda_k y. H)[J/x] := \lambda_k z. (H[\text{proc}_n(z)/y])[J/x]$$

Since $E \setminus x \cup \{x : \eta_x\}, G[\chi, a] \vdash \lambda_k. H$ it follows with (TL-PA) that

$$E \setminus x \setminus y \cup \{x : \eta_x\} \cup \{y : \eta_y\}, G[\chi \circ \chi', a] \vdash H$$

Lemma 8.3.8 implies that

$$E \setminus x \setminus y \setminus z \cup \{x : \eta_x\} \cup \{z : \eta_y\}, G[\chi \circ \chi', a] \vdash H[\text{proc}_n(z)/y]$$

With the induction hypothesis and lemma 8.3.5 it follows that

$$E \setminus z \cup \{z : \eta_y\}, G[\chi \circ \chi', a] \vdash (H[\text{proc}_n(z)/y])[J/x]$$

And (TL-PROC) implies that

$$E, G[\chi, a] \vdash \lambda_k z. (H[\text{proc}_n(z)/y])[J/x]$$

□

We will now undertake the first step in order to prove the subject reduction property and show that equivalent expressions have the same type:

Proposition 8.3.10 (Equivalence)

Let H be a process graph with $E, G[\chi, a] \vdash H$. Furthermore let $H \equiv H'$. Then $E, G[\chi, a] \vdash H'$.

Proof: By induction on the rules of structural congruence. For (C- α) use lemma 8.3.8 □

The subject reduction property is the main result in this section. It states that during reduction the type of an expression stays invariant.

Proposition 8.3.11 (Subject Reduction Property)

Let $E, T \vdash H$ and $H \rightarrow^ H'$. Then $E, T \vdash H'$. Furthermore H contains no bad redexes.*

Proof: See appendix A.2.1. □

8.3.3 Type Inference Algorithm

The following algorithm will compute the principal type of a process. We will specify first what “principal” means in this context.

Definition 8.3.12 (Principal Type) E, T is called a *principal type* of S if

- $E, T \vdash S$
- If $\hat{\phi} : T \xrightarrow{F, \leq} T'$ it follows that $\hat{\phi}(E), T' \vdash S$
- If $E', T' \vdash S$ then there exists a morphism $\hat{\phi} : T \xrightarrow{F, \leq} T'$ such that $\hat{\phi}(E) = E'$.

□

That is, the principal type is the smallest type of an expression. We will show later that if an expression has a type, then it also has a principal type.

We now show how to construct type graphs. One important concept, which is to be exploited here, is the modularity of the type system, i.e. a type of an expression can be computed out of the types of its subexpressions. The algorithm basically proceeds as follows: we compute the types of the subexpressions, attach them accordingly by forming a quotient graph (see definition 2.1.9) and then fold the resulting type graph as described in proposition 8.3.13. The folding is necessary since the building of a new type graph out of true type graphs does not always yield a true type graph.

Proposition 8.3.13 (Folding Type Graphs)

Let $T = G[\chi, a]$ be a type graph. Folding a type graph works as follows: let \approx be the smallest equivalence such that:

$$\begin{aligned} p_1, p_2 \in P_G, \quad s_G(p_1) = s_G(p_2) &\Rightarrow p_1 \approx p_2 \\ m_1, m_2 \in M_G, \quad \text{send}_G(m_1) = \text{send}_G(m_2) &\Rightarrow m_1 \approx m_2 \\ c_1, c_2 \in Co_G, \quad \text{send}_G(c_1) = \text{send}_G(c_2) &\Rightarrow c_1 \approx c_2 \end{aligned}$$

If \approx is consistent we define

$$FOLD_F(T) := (G/\approx)[FOLD^T(\chi), F_{FOLD^T}(a)]$$

where $FOLD^T = FOLD^G : G \rightarrow G/\approx$ is the projection of G into G/\approx .

(1) $FOLD_F(T)$ is the “smallest” true type graph into which exists a morphism from T . That is, $FOLD_F(T)$ is a true type graph and for every true type graph T' with a strong morphism $\psi : T \xrightarrow{F, \leq} T'$ it follows that $FOLD_F(T)$ is defined and there exists a unique strong morphism $\phi : FOLD_F(T) \xrightarrow{F, \leq} T'$ such that

$$\phi \circ FOLD^T = \psi$$

(2) Let $T = G[\chi, a]$ be a type graph and let \approx be the equivalence defined as above. Furthermore let $\approx' \subseteq \approx$. Let $p : G \rightarrow G/\approx'$ be the projection of G into G/\approx' . It follows that

$$FOLD^G \cong FOLD^{G/\approx'} \circ p$$

(3) Let $T \cong_F C\langle T_1, \dots, T_n \rangle_F$ be a type graph and let $i \in \{1, \dots, n\}$. Furthermore let $T' \cong_F C\langle T_1, \dots, T_{i-1}, FOLD_F(T_i), T_{i+1}, \dots, T_n \rangle_F$. It follows that

$$\begin{aligned} FOLD_F(T) &\cong_F FOLD_F(T') \\ FOLD^T &\cong FOLD^{T'} \circ C\langle id_{T_1}, \dots, id_{T_{i-1}}, FOLD^{T_i}, id_{T_{i+1}}, \dots, id_{T_n} \rangle_F \end{aligned}$$

Proof: See appendix A.2.1. □

We first specify the algorithm and prove its correctness afterwards.

Algorithm 8.3.14 (Type Inference Algorithm)

Input: a process description S , a natural number m indicating the cardinality of S , an environment E with embeddings $\eta_x : \mathbf{m}_x \rightarrow G$ for all $x \in \text{free}(S)$ and a lattice element $a \in F(G)$.

Output: a morphism $\phi : G \rightarrow G'$, a string $\chi' \in V_{G'}^*$ and $a' \in F(G')$ such that

$$\phi(E), G'[\chi', a'] \vdash S$$

The mapping $(\phi, \chi', a') = W(S, m, E, a)$ is defined inductively as follows:

Variable: $S = x$

Return $(id_G, \eta_x(\chi_{\mathbf{m}_x}), a)$ if $x : \eta_x \in E$, $\eta_x : \mathbf{m}_x \rightarrow G$ and $id_G : G \rightarrow G$ is the identity.

Replication: $S = !H$

Return $W(!H, card(H), E, a)$

Process: $S = proc_n(S')$

Let $(\phi, \chi', a') := W(S', n, E, a)$ where $\phi : G \rightarrow G'$ and let \approx be the smallest equivalence on $P[\chi_P, a_P] := A(proc_n(S))$ and $G'[\chi', a']$ with

$$(\lfloor \chi_P \rfloor_i, 0) \approx (\lfloor \chi' \rfloor_i, 1) \quad i \in \{1, \dots, n\}$$

Let $\tilde{G} := (PG')/\approx$, let $p_P : P \rightarrow \tilde{G}$ be the projection of P into \tilde{G} and let $p_{G'} : G' \rightarrow \tilde{G}$ be the projection of G' into \tilde{G} .

Return $(FOLD^{\tilde{G}}_{op_{G'} \circ \phi}, FOLD^{\tilde{G}}(p_P(\chi_P)), F_{FOLD^{\tilde{G}}}(F_{p_P}(a_P) \vee F_{p_{G'}}(a')))$.

Message: $S = mess_{n+1}(H)$

Let $(\phi, \chi', a') := W(H, m, E, a)$ where $m := card(H)$, $\phi : G \rightarrow G'$ and let \approx be the smallest equivalence on $M[\chi_M, a_M] := A(mess_{n+1}(H))$ and $G'[\chi', a']$ with

$$(\lfloor s_M(cont_M(\lfloor \chi_M \rfloor_{n+1})) \rfloor_i, 0) \approx (\lfloor \chi' \rfloor_i, 1) \quad i \in \{1, \dots, n+1\}$$

Let $\tilde{G} := (MG')/\approx$, let $p_M : M \rightarrow \tilde{G}$ be the projection of M into \tilde{G} and let $p_{G'} : G' \rightarrow \tilde{G}$ be the projection of G' into \tilde{G} .

Return $(FOLD^{\tilde{G}}_{op_{G'} \circ \phi}, FOLD^{\tilde{G}}(p_M(\chi_M)), F_{FOLD^{\tilde{G}}}(F_{p_M}(a_M) \vee F_{p_{G'}}(a')))$.

Process Graph: $S = H = \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow D$ and $\phi : D \rightarrow H$.

We define $a_0 := a$ and $E_0 := E$. Furthermore

$$(\phi_i, \chi_i, a_i) := W(H_i, card(H_i), E_{i-1}, a_{i-1})$$

where $\phi_i : G_{i-1} \rightarrow G_i$ and

$$E_i := \{x : \phi_i \circ \eta_x^{i-1} \mid x : \eta_x^{i-1} \in E_{i-1}\}$$

We define $\chi'_i := (\phi_n \circ \dots \circ \phi_{i+1})(\chi_i) \in V_{G_n}^*$. Let \approx be the smallest equivalence on D, G_n satisfying

$$(\lfloor \zeta_i(\chi_{\mathbf{m}_i}) \rfloor_j, 0) \approx (\lfloor \chi'_i \rfloor_j, 1) \quad j \in \{1, \dots, m_i\}$$

Define¹ $\tilde{G} := (D^0 G_n)/\approx$ and let $p_D : D^0 \rightarrow \tilde{G}$ and $p_{G_n} : G_n \rightarrow \tilde{G}$ be the projections into \tilde{G} .

Return $(FOLD^{\tilde{G}}_{op_{G_n} \circ \phi_n \circ \dots \circ \phi_1}, FOLD^{\tilde{G}}(p_D(\chi_D)), F_{FOLD^{\tilde{G}}}(F_{p_{G_n}}(a_n)))$.

¹If $H = G[\chi]$ we define $H^0 := G$.

Process Abstraction: $S = \lambda_k x. H$

Let $\bar{G} := G \oplus \mathbf{m}_x^0$ and let $p_G : G \rightarrow \bar{G}$, $p_{\mathbf{m}_x} : \mathbf{m}_x^0 \rightarrow \bar{G}$ be the corresponding projections.

Furthermore let $(\phi, \chi', a') := W(H, \text{card}(H), p_G(E) \setminus x \cup \{x : p_{\mathbf{m}_x}\}, F_{p_G}(a))$ where $\phi : \bar{G} \rightarrow G'$.

We define $P[\chi_P] := \text{proc}_M$, $M[\chi_M] := \text{mess}_{n+1}$, $Co[\chi_{Co}] := \text{cont}_{\text{sort}(x)+1}$.

Let \approx be the smallest equivalence on G', P, M, Co satisfying:

$$\begin{aligned} (\lfloor \chi' \rfloor_i, 0) &\approx (\lfloor \chi_P \rfloor_i, 1) \quad i \in \{1, \dots, m\} \\ (\lfloor \chi' \rfloor_{m+i}, 0) &\approx (\lfloor \chi_M \rfloor_i, 2) \quad i \in \{1, \dots, n\} \\ (\lfloor \chi' \rfloor_k, 0) &\approx (\lfloor \chi_M \rfloor_{n+1}, 2) \\ (\lfloor (\phi \circ p_{\mathbf{m}_x})(\chi_{\mathbf{m}_x}) \rfloor_i, 0) &\approx (\lfloor \chi_{Co} \rfloor_i, 3) \quad i \in \{1, \dots, \text{sort}(x)\} \\ (\lfloor \chi' \rfloor_k, 0) &\approx (\lfloor \chi_{Co} \rfloor_{\text{sort}(x)+1}, 3) \end{aligned}$$

Define $\tilde{G} := (G' P M Co) / \approx$ and let $p_{G'} : G' \rightarrow \tilde{G}$ be the projection of $G'[\chi']$ into \tilde{G} .

Return $(FOLD^{\tilde{G}} \circ p_{G'} \circ \phi \circ p_G, [FOLD^{\tilde{G}}(p_{G'}(\chi'))]_{1..m}, F_{FOLD^{\tilde{G}}}(F_{p_{G'}}(a')))$.

The type inference algorithm fails if a quotient graph or a fold operation is undefined. \square

We will now show that the algorithm above has the following property: if an expression has any type at all, the algorithm computes its principal type.

First we show that the result of the algorithm is a type of the expression (in the case where $\hat{\phi}' = id_{\tilde{G}}$), and that every type which can be derived from the result by a morphism, is also a type of the expression.

Lemma 8.3.15 *Let $(\phi, \chi', a') := W(S, m, E, a)$. It follows that $F_{\phi}(a) \leq a'$.*

Proof: Straightforward by induction on S . \square

Proposition 8.3.16 (Correctness of the Type Inference Algorithm) *Let $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(S, m, E, a)$ where $\hat{\phi} : G \rightarrow \hat{G}$ and let*

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

Then

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash S$$

Proof: See appendix A.2.1. \square

We will now show that if an expression has a type E, T , then the algorithm yields a result. And furthermore, there exists a morphism from the result into E, T .

Proposition 8.3.17 (Soundness of the Type Inference Algorithm)

Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash S$ and let $\hat{\phi} : G \xrightarrow{F, \leq} \hat{G}$ with $\hat{E} = \hat{\phi}(E)$ and $F_{\hat{\phi}}(a) \leq \hat{a}$ then

$$(\phi, \chi', a') := W(S, |\hat{\chi}|, E, a)$$

(where $\phi : G \rightarrow G'$) is defined and there exists a morphism

$$\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

such that $\hat{\phi} = \psi \circ \phi$.

Proof: See appendix A.2.1. □

The type inference algorithm computes the principal type of an expression, if it exists:

Proposition 8.3.18 (Principal Types) Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash S$.

Let $X := \text{free}(S)$, $m_x := \text{sort}(x)$ and let $G := \bigoplus_{x \in X} \mathbf{m}_x$ (i.e. G is a discrete hypergraph with $\sum_{x \in X} m_x$ nodes). Furthermore let $\eta_x : \mathbf{m}_x \rightarrow G$ be the canonical projections into G .

We define $E := \{x : \eta_x \mid x \in X\}$. It follows that

$$(\phi, \chi', a') := W(S, |\hat{\chi}|, E, \perp)$$

where $\phi : G \rightarrow G'$ is defined and that $\phi(E), G'[\chi', a']$ is the principal type of S .

Proof: According to lemma 8.3.6 we can assume that \hat{E} is of the form $\{x : \hat{\eta}_x \mid x \in \text{free}(S)\}$.

We have to show that there exists a morphism $\hat{\phi} : G \rightarrow \hat{G}$ such that $\hat{\phi}(E) = \hat{E}$. We define $\hat{\phi}(\eta_x(\chi_{\mathbf{m}_x})) := \hat{\eta}_x(\chi_{\mathbf{m}_x})$ where $x : \hat{\eta}_x \in \hat{E}$. It is straightforward to check that $\hat{\phi}$ is well-defined.

The rest is straightforward with propositions 8.3.16 and 8.3.17. □

8.3.4 Verification with the Type System

We will now show how to exploit the type system for verification purposes. That is, for every typed expression H , there is a morphism from $A(H)$ into the type of H . So if the type satisfies a certain property which is closed under inverse morphisms, $A(H)$ satisfies this property as well.

Proposition 8.3.19 Let $E, T \vdash H$. This implies that there is a morphism

$$\psi : A(H) \xrightarrow{F, \leq} T$$

Proof: See appendix A.2.1 □

Proposition 8.3.20 *Let F be a type functor and let A be a linear mapping.*

Let P be a predicate on SPIDER expressions and let Q be a predicate on type graphs. P, Q satisfy

$$Q(A(H)) \Rightarrow P(H) \quad (8.13)$$

$$\phi : T \xrightarrow{F, \leq} T', Q(T') \Rightarrow Q(T) \quad (8.14)$$

i.e. Q is closed under inverse (F, \leq) -morphisms.

Then $E, T \vdash H$ and $Q(T)$ imply $P(H')$ for all $H \rightarrow^ H'$.*

Proof: The subject reduction property (proposition 8.3.11) implies $E, T \vdash H'$ for every $H \rightarrow^* H'$.

If $E, T \vdash H'$ it follows with proposition 8.3.19 that there exists a (F, \leq) -morphism $\phi' : A(H') \rightarrow T$. Since $Q(T)$ and Q is closed under inverse morphisms it follows that $Q(A(H'))$ which implies $P(H')$. \square

A type system for checking a predicate P thus consists of a type functor F , a linear mapping A and a predicates Q on type graphs as defined above. That is $TS = (F, A, P, Q)$.

8.3.5 Composing Type Systems

Let $TS_i := (F_i, A_i, Q_i, P_i)$, $i = 1, 2$ be two type systems. We define

$$F(T) := F_1(T) \times F_2(T)$$

$$A(H) := G[\chi, (a_1, a_2)] \text{ if } A_1(H) \cong_F G[\chi, a_1] \text{ and } A_2(H) \cong_F G[\chi, a_2]$$

That is we demand that the linear mappings map equivalent process graphs onto type graphs of the same structure.

We define

$$Q_\wedge(G[\chi, (a_1, a_2)]) := Q_1(G[\chi, a_1]) \wedge Q_2(G[\chi, a_2])$$

$$Q_\vee(G[\chi, (a_1, a_2)]) := Q_1(G[\chi, a_1]) \vee Q_2(G[\chi, a_2])$$

It is easy to check that

$$TS_\wedge := (F, A, Q_\wedge, P_1 \wedge P_2) \text{ and}$$

$$TS_\vee := (F, A, Q_\vee, P_1 \wedge P_2)$$

are also type systems, checking the conjunction respectively disjunction of P_1 and P_2 .

8.3.6 Examples

The lattice will contain mappings of the form $a : V_T^k \rightarrow I$ or $a : E_T \rightarrow I$ where I is a lattice. The type functor F has to satisfy

$$F_\phi(a)(s') = \bigvee_{\phi(s)=s'} a(s) \quad \text{if } s' \in V_T^k$$

in the first case and

$$F_\phi(a)(e') = \bigvee_{\phi(e)=e'} a(e) \quad \text{if } e' \in E_T$$

in the second case. It is straightforward to verify that both are indeed type functors.

In the following examples we will assume a linear mapping A with

$$\begin{aligned} A(\text{proc}_n(S)) &:= P[\chi_P, a_P] \text{ where } P[\chi_P] := \text{proc}_n \\ A(\text{mess}_n(H)) &:= M[\chi_M, a_M] \text{ where} \\ &\quad M[\chi_M] := \sigma_{1,\dots,n}(\theta_{n,n+k+1}(\text{mess}_n \oplus \text{cont}_{k+1})) \end{aligned}$$

That is the graph structure of $A(H)$ is fixed. Only the lattice elements will vary.

Avoiding Run-Time Errors

We set $k := 0$ and take the trivial lattice $I = \{0\}$. In this case the mapping A is constant, i.e. $a_P = 0$ and $a_M = 0$ where $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(H)) := M[\chi_M, a_M]$ (see remark above). We get a standard type system, that only types process graphs without runtime errors.

Applying this type system to the example in section 8.1 yields the type graph on page 113 without any extra labels or arrows.

Input/Output-Capabilities

We want to ensure that some external ports are only used as input ports and that some are only used as output ports.

We choose $k = 1$, $I = \{\perp, in, out, both\}$ where $\perp < in < both$ and $\perp < out < both$, i.e. $in \vee out = both$.

The mapping a is defined in the following way:

$$\begin{aligned} a_P(\lfloor \chi_P \rfloor_i) &:= \begin{cases} in & \text{if } S = \lambda_i.H \\ \perp & \text{otherwise} \end{cases} \\ a_M(\lfloor \chi_M \rfloor_i) &:= \begin{cases} out & \text{if } i = n \\ \perp & \text{otherwise} \end{cases} \\ a_M(\lfloor cont(\lfloor \chi_M \rfloor_n) \rfloor_i) &:= \perp \end{aligned}$$

where $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(H)) := M[\chi_M, a_M]$.

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

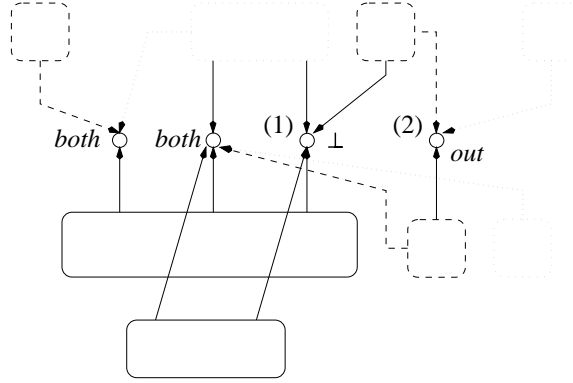
$$Q(G[\chi, a]) := (a(\lfloor \chi \rfloor_i) \leq in)$$

If, on the other hand the type of H satisfies

$$Q(G[\chi, a]) := (a(\lfloor \chi \rfloor_i) \leq out)$$

we can conclude that H will never reduce to a process graph H' where a process listens at $\lfloor \chi_H \rfloor_i$.

Applying this type system to the example in section 8.1 yields the following principal type graph:



That is the second external port is only used as an output port and the first external port is not used for any input/output-operations at all.

A more sophisticated version of this type system is presented in [PS93]. There is, of course, a trade-off between generality and the percentage of processes which can be typed: e.g. in this case our type system can type *fewer* processes than the type system introduced in [PS93], which can partly be explained by the very general nature of our type system and partly by the fact that the type system in [PS93] does not have principal types.

Secrecy of Message Contents

We assume that there exists a subset $SECRET \subseteq \mathcal{S}$ of SPIDER expressions which are to be considered secret and which are not to be communicated to the outside.

Because of restraints of the type system we have to assume that

$$H \equiv H', H \in SECRET \Rightarrow H' \in SECRET$$

$$H[H'/x] \in SECRET \iff H \in SECRET$$

$$\forall x, y \in X \text{ with } sort(x) = sort(y) = n :$$

$$proc_n(x) \in SECRET \iff proc_n(y) \in SECRET$$

Our lattice is the set of all mappings $a : Co_T \rightarrow \{sec, pub\}$ where $pub < sec$. The mapping a_M has the following form:

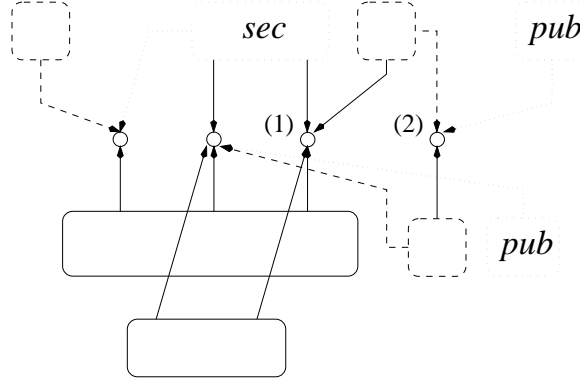
$$a_M(cont(\lfloor \chi_M \rfloor_n)) := \begin{cases} sec & \text{if } H \in SECRET \\ pub & \text{otherwise} \end{cases}$$

where $A(mess_n(H)) := M[\chi_M, a_M]$ (see remark above).

We want to make sure that H never reduces to a process graph H' which contains a message labelled with a secret content and sent to an external port. The corresponding predicate Q is:

$$Q(G[\chi, a]) := (\forall i \in \{1, \dots, \text{card}(H)\} : a(\text{cont}_G(\lfloor \chi \rfloor_i)) = \text{pub})$$

We investigate again the example from section 8.1 and assume that the content of the message on the left is considered secret and that all messages labelled $\mathbf{0}$ are considered public. This yields the following principal type graph:



That is, no secret message is sent to an external port.

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 messages, but it is not allowed to forward a secret port to a receiver listening at a public port.

We choose $k = 2$, $I = \{\text{true}, \text{false}\}$ where $\{\text{true}, \text{false}\}$ is the boolean lattice with $\text{false} < \text{true}$.

The mapping a has the following form:

$$\begin{aligned} a_P(\lfloor \chi_P \rfloor_i, \lfloor \chi \rfloor_j) &:= \text{false} \\ a_M(\lfloor \chi_M \rfloor_i, \lfloor \chi \rfloor_j) &:= \begin{cases} \text{true} & \text{if } i = n, j \neq n \\ \text{false} & \text{otherwise} \end{cases} \\ a_M(\lfloor \text{cont}(\lfloor \chi_M \rfloor_n) \rfloor_i) &:= \text{false} \end{aligned}$$

where $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(H)) := M[\chi_M, a_M]$ (see remark above).

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

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

it follows that no message, with secret ports attached to it, is ever sent to a public port.

In section 8.1 we demonstrated the construction of a type graph for this type system.

8.4 A Type System Based on Monoids

We will now present another type system that is, in some ways, more advanced, since it allows us to handle monoids instead of lattices making it possible to count.

Restrictions: We will, however, make an important restriction and drop higher-order communication. That is during this chapter we demand that messages are always labelled $\mathbf{0}$. As a consequence we drop all variables $x \in X$ and will abbreviate $\lambda_n x$ with λ_n .

Furthermore we demand that all expressions are in the subcalculus \mathcal{S}_n (see section 7.4). That is, we restrict ourselves to the part of SPIDER into which the π -calculus can be coded (see section 7.5).

Discussion of Problems: Taking l-monoids instead of lattices leads to some drawbacks. We have already mentioned the first: construction of type graphs can not, any more, be represented as a co-limit of (F, \leq) -morphisms.

A second problem arises since we might have *negative numbers*. In the example in the introduction chapter, we constructed the type graph in two steps: first attach all parts and fold them afterwards (these two steps were integrated into one in the type system based on lattices). If we look at this issue closely we find out that we need two kinds of monoid operations: in the first step it is ordinary summation, but in the folding step we have to make sure that the monoid label of every components does not *decrease*, otherwise the type system would be useless for verification purposes. This, however, can happen if we use the ordinary summation for folding and if we work with negative numbers. As it turns out, we need the operator \oplus with $a \oplus b := (a + b) \vee a \vee b$ (defined in the introduction) for the folding process.

So we actually need *two* type functors: one, which we will call F and which is used for type construction, and the other one, which we will call J and which is used for folding type graphs. We cannot demand that F, J satisfy the equivalent of condition (3) in proposition 8.3.13, i.e. normally

$$FOLD_J(C\langle T_1, \dots, T_n \rangle_F) \not\approx_F FOLD_J(C\langle T_1, \dots, FOLD_J(T_i), \dots, T_n \rangle_F)$$

since for many monoids and type functors this property is not satisfied.

So, instead of mixing the two phases of type graph construction and type graph folding, as we have done in the previous chapter, we now first construct the type graph and fold it afterwards. This has some disadvantages, especially for the compositionality of types, but we do not see any other way to do this.

8.4.1 The Type System

We will now assume that there are two type functors F and J .

A type graph $G[\chi, a]$ consists (as before) of a hypergraph $G[\chi]$ and of $a \in F(G)$.

We will use the same definitions of true type graphs as in the type system based on lattices. But for the linear mapping A we change conditions (4) and (5) to

- (4) If $P[\chi_P, a_P] := A(proc_n(S))$ it follows that $P[\chi_P] \cong proc_n$
- (5) If $M[\chi_M, a_M] := A(mess_n(S))$ it follows that either $M[\chi_M] \cong mess_n$ or $M[\chi_M] \cong \sigma_{1,\dots,n}(\theta_{n,n+k+1}(mess_n \oplus cont_{k+1}))$. (This has to be handled consistently within a type system.)

Conditions (4) and (5) will be needed in the proof of proposition 8.4.8 (subject reduction property for (R-MR)). Actually they are no real restriction since relations between ports can always be modelled by extra monoid elements.

Definition 8.4.1 (Compatible Type Functors) Let F, J be two type functors.

F, J are called *compatible*, if for every graph G , $F(G) = (I, +, \leq)$ and $J(G) = (I, \oplus, \leq)$ where $(I, +, \leq), (I, \oplus, \leq)$ are l-monoids with $a \oplus b := (a + b) \vee a \vee b$ (see section 8.2). Furthermore in $(I, +, \leq)$ the residual $0 - a$ is defined for every $a \in I$ and I has a smallest element \perp .

Furthermore

$$J_\phi \leq F_\phi \quad \text{if } \phi \text{ is an injective morphism} \quad (8.15)$$

$$J_\phi \geq F_\phi \quad \text{if } \phi \text{ is a surjective morphism} \quad (8.16)$$

Let ψ_1, \dots, ψ_n be nice (J, \leq) -morphisms (nice hypergraph morphisms are defined in definition 2.1.4) and let C be a discrete context with holes of cardinality $card(\psi_1), \dots, card(\psi_n)$. Then

$$C\langle\psi_1, \dots, \psi_n\rangle_F \text{ is also a } (J, \leq)\text{-morphism} \quad (8.17)$$

□

In section 8.4.6 we will present a pair of compatible type functors.

Since $G_\phi = F_\phi$ if ϕ is an isomorphism the relations $\tilde{<}_F$ and $\tilde{<}_J$ coincide.

Notes: We will need a few abbreviations which we will define now: Since we do not work with higher-order message passing any more, a redex can be described by

$$Red_{k,m,n}(S) := Red_{k,m,n}(S, \mathbf{0})$$

Furthermore we define

$$(G[\chi, a])^- := G[\chi, 0 - a]$$

We will now introduce the typing rules of the type system based on monoids. It differs from the type system based on lattices in that it is more constructive and that the type environment E is missing. The type environment is unnecessary since we have excluded expressions with variables.

Again, parameters of the type system are the functors F, G and the linear mapping A .

Definition 8.4.2 (Typing Rules) Let F, G be compatible type functors and let A be a linear mapping as defined above.

A *type* of a process graph or process description S consists of a (not necessarily true!) type graph T . We write $T \Vdash_{A,F,G}^M S$ or $T \Vdash S$ for short.

The following rules describe how a type can be assigned to an expression.

$$\begin{array}{c}
\text{(TM-}\leq\text{)} \quad \frac{T \Vdash S, T \lesssim T'}{T' \Vdash S} \\
\\
\text{(TM-REPL)} \quad \frac{T \Vdash H, T \sqcap_F T \xrightarrow{J, \leq} T}{T \Vdash !H} \\
\\
\text{(TM-PA)} \quad \frac{T \Vdash H}{\sigma_{1\dots m}(T \sqcap_F A(\text{Red}_{k,m,n}(\lambda_k.H))^-) \Vdash \lambda_k.H} \\
\\
\text{(TM-MESS)} \quad A(\text{mess}_n(\mathbf{0})) \Vdash \text{mess}_n(\mathbf{0}) \\
\\
\text{(TM-PROC)} \quad \frac{T \Vdash S, \mathbf{n}[0] \xrightarrow{J, \leq} T}{A(\text{proc}_n(S)) \sqcap_F T \Vdash \text{proc}_n(S)} \\
\\
\text{(TM-CON)} \quad \frac{T_i \Vdash H_i, i \in \{1, \dots, n\}}{C\langle T_1, \dots, T_n \rangle_F \Vdash C\langle H_1, \dots, H_n \rangle}
\end{array}$$

□

The intuitive meaning of some of the rules can be informally explained as follows:

(TM-REPL) In order to type $!H$ with T it is necessary that the type graph can be folded into itself, in order to ensure the subject reduction property. Because of the idempotency of lattice elements this condition is not necessary for (TL-REPL).

The existence of a hypergraph morphism from $T \sqcap T$ into T is clear, it is left to check that it is a (J, \leq) -morphism.

(TM-PROC) Here, the problem with the negative numbers surfaces again. Since the external nodes of T might be labelled with negative elements, it might be the case that the labels of $A(\text{proc}_n(S))$ are decreased. For the purpose of verification it is, however, essential that there is a morphism $A(\text{proc}_n(S)) \xrightarrow{J, \leq} A(\text{proc}_n(S)) \sqcap_F T$. The side condition is there to ensure the existence of this morphism.

(TM-PA) In this rule we subtract the labels of the redex (or at least the labels of the external nodes) from the labels of T . By doing so we take into account the fact that parts of a graph are removed during reduction.

Note: Let $T \Vdash H$. Since we assume that H is in the subcalculus \mathcal{S}_n , i.e. every hypergraph inside of H has duplicate-free sequences of external nodes we can conclude, by induction on the typing rules, that χ_T is also duplicate-free. We need this fact for proposition 8.4.10.

Definition 8.4.3 (True Type) Let S be a process graph or a process description. We say that S can be typed by the *true type* TT

$$TT \vdash S$$

if TT is a true type graph, $T \Vdash S$ and there is a (J, \leq) -morphism $\phi : T \rightarrow TT$.
□

8.4.2 Subject Reduction Property

In this section we assume that there are fixed compatible type functors F, J and a basic mapping A .

Lemma 8.4.4 *If $T \Vdash H$ then $\text{card}(H) = \text{card}(T)$.*

Proof: By simple induction on the typing rules. □

As in the previous chapter we need some way to unravel the typing of an expression and to retrace our steps. In the case of (TM- \leq), (TM-REPL), (TM-PA), (TM-MESS) and (TM-PROC) this is easy. But in the case of (TM-CON) ambiguity appears. The following lemma is essential in unravelling the type of a process graph. It corresponds to lemma 8.3.7.

Lemma 8.4.5 *Let $T \Vdash C\langle H_1, \dots, H_n \rangle$. Then there exist type graphs T_1, \dots, T_n such that*

$$\begin{aligned} T_i &\Vdash H_i \\ T &\widetilde{>}_F C\langle T_1, \dots, T_n \rangle_F \end{aligned}$$

Proof: Let $H \cong C'\langle H'_1, \dots, H'_m \rangle$ such that C' is a discrete context and H'_1, \dots, H'_m are basic graphs. During the typing of H a type graph T'_i is assigned to each basic graph H'_i . We define a linear mapping with

$$B(J) := \begin{cases} T'_i & \text{if } J \cong H'_i \\ \mathbf{k}[0] \text{ with } k := \text{card}(J) & \text{if } J \text{ is any other basic graph} \end{cases}$$

By induction on the typing of H we can show that $B(H) \widetilde{<}_F T$. And furthermore it follows with (TM-CON) that $B(H) \Vdash H$.

If we define $T_i := B(H_i)$ it follows that $T_i \Vdash H_i$ and

$$T \widetilde{>}_F B(H) \cong_F C\langle T_1, \dots, T_n \rangle_F$$

□

As in the previous chapter (see proposition 8.4.6) we have to check that equivalent process graphs have the same type.

Proposition 8.4.6 (Equivalence) *Let $H_1, H_2 \in \mathcal{S}$ with $H_1 \equiv H_2$.*

$T \Vdash H_1 \iff T \Vdash H_2$ for any type graph T .

This implies immediately $TT \vdash H_1 \iff TT \vdash H_2$ for any true type graph TT .

Proof: See appendix A.2.2. □

It is now our aim to show that no type contains bad redexes and that the type is invariant under reduction. In the case of replication this is not very difficult, because of the precondition $T \sqcap_F T \xrightarrow{J, \leq} T$ in rule (TL-REPL).

Proposition 8.4.7 (Replication) *Let $TT \vdash H$. If H contains a subgraph of the form $proc_n(!K)$ then replication is defined, i.e. $n = card(K)$.*

Let $H \xrightarrow{(R-REPL)} H'$. It follows that H' has the same type as H , i.e. $TT \vdash H'$.

Proof: Let $T \Vdash H$ such that there is a (J, \leq) -morphism $\psi : T \rightarrow TT$.

It follows with proposition 4.3.5 that $H \equiv C\langle H_1, H_2 \rangle$, $H' \equiv C\langle H'_1, H_2 \rangle$ and $H_1 \xrightarrow{(R-REPL)} H'_1$. In this case $H_1 \equiv proc_n(!K)$ and $H'_1 = proc_n(!K) \sqcap K$.

Lemma 8.4.5 implies that there are type graphs T_1, T_2 with $T_i \Vdash H_i$. It is left to show that $T_1 \Vdash H'_1$.

It follows with (TM-PROC) and (TM-REPL) that

$$T_1 \widetilde{>}_F A(proc_n(!K)) \sqcap T_K \Vdash proc_n(!K)$$

and $T_K \Vdash K$. Furthermore there is a morphism $\phi : T_K \sqcap T_K \xrightarrow{J, \leq} T_K$.

Since $A(proc_n(!K)) \sqcap T_K$ is defined we conclude $n = card(T_K) = card(K)$.

(TM-CON) implies that $T' := A(proc_n(!K)) \sqcap T_K \sqcap T_K \Vdash proc_n(!K) \sqcap K$.

Since ϕ is a nice morphism it follows with (8.17) that there exists a (J, \leq) -morphisms $\phi' : T' \rightarrow T$.

It follows that $\psi \circ \phi' : T' \xrightarrow{J, \leq} TT$. □

In the case of message reception it is more difficult to prove the subject reduction property. The main reason for these difficulties is the fact that the folding of a type graph into a true type graph is done at the very end of the typing process. Typing a process after message reception, however, causes adjustments deep inside the typing process. These adjustments include the merging of ports which are fused during the folding.

In the case of message reception $T \Vdash Red_{k,m,n}(\lambda_k.H)$ does not imply $T \Vdash H$. And in the case of \vdash local arguments (showing that $T \vdash Red_{k,m,n}(\lambda_k.H)$ implies $T \vdash H$) are not sufficient and we have to take a global view on the whole typing process.

Proposition 8.4.8 (Message Reception) *Let $TT \vdash H$. If H contains a redex*

$$Red := Red_{k,m,n}(\lambda_k x.K)$$

then message-reception is defined, i.e. $m + n = card(K)$.

Let $H \xrightarrow{(R-MR)} H'$. It follows that H' has the same type as H , i.e. $TT \vdash H'$.

Proof: See appendix A.2.2. \square

Proposition 8.4.9 (Subject Reduction) *Let $TT \vdash H$ and $H \rightarrow^* H'$. Then $TT \vdash H'$. Furthermore H contains no bad redexes.*

Proof: Straightforward with propositions 8.4.6, 8.4.7 and 8.4.8. \square

8.4.3 Type Inference

The type system presented in this section is of a more constructive nature than the type system presented in section 8.3. Only the rules (TM-REPL) and (TM-PROC) are non-constructive and may require the additional use of (TM- \leq). For the existence of principal types we therefore demand that the sets

$$\begin{aligned} &\{T' \in \mathcal{T}_J \mid T' \succ_F T, T' \sqcap_F T' \xrightarrow{J, \leq} T'\} \\ &\{T' \in \mathcal{T}_J \mid T' \succ_F T, \mathbf{n}[0] \xrightarrow{J, \leq} T'\} \end{aligned}$$

contain a smallest element wrt. \prec_F for every type graph T . These smallest elements are denoted by $R(T)$ respectively $P(T)$.

If this smallest element is computable, the type inference algorithm corresponds exactly to the typing rules.

As mentioned above, compositionality of types is only true for \Vdash and not for \vdash . This is unfortunate, but we did not find any elegant solution for a more compositional type system.

8.4.4 Verification with the Type System

We now show how to exploit the type system for verification purposes. Note that in the proof of the following proposition we utilize the fact that the sequence of external nodes of a type graph does not contain duplicates.

Proposition 8.4.10 *Let $T \Vdash H$. This implies that there is a morphism*

$$A(H) \xrightarrow{J, \leq} T$$

This implies immediately that if $TT \vdash H$, then there also exists a morphism $A(H) \xrightarrow{J, \leq} TT$.

Proof: We consider the following cases:

(TM-MESS) If $T \Vdash \text{mess}_n(\mathbf{0})$, then clearly $T \cong_F A(\text{mess}_n(\mathbf{0}))$ and there is a nice (J, \leq) -morphism (the identity morphism) $\text{id} : A(\text{mess}_n(\mathbf{0})) \rightarrow T$.

(TM-PROC) If $T \Vdash \text{proc}_n(S)$, then $T \cong_F A(\text{proc}_n(S)) \sqcap_F T'$ and there exists a morphism $\psi : \mathbf{n}[0] \xrightarrow{J, \leq} T'$. Since $\chi_{T'}$ is duplicate-free, it follows that ψ is a nice morphism.

It follows with lemma 8.2.11 that

$$C\langle id, \psi \rangle : A(proc_n(S)) \square_F \mathbf{n}[0] \xrightarrow{J, \leq} A(proc_n(S)) \square_F T'$$

where $A(proc_n(S)) \cong_F A(proc_n(S)) \square_F \mathbf{n}[0]$.

$C\langle id, \psi \rangle$ is again a nice morphism.

(TM-CON) If $T \Vdash C\langle H_1, \dots, H_n \rangle$ where H_1, \dots, H_n are basic graphs, it follows with lemma 8.4.5 that there exist type graphs T_1, \dots, T_n with $T_i \Vdash H_i$ and $C\langle T_1, \dots, T_n \rangle \widetilde{\prec}_F T$.

We have shown that for basic graphs H_i there exists a nice morphism $\psi_i : A(H_i) \xrightarrow{J, \leq} T_i$.

Lemma 8.2.11 implies that

$$C\langle \psi_1, \dots, \psi_n \rangle : C\langle H_1, \dots, H_n \rangle \xrightarrow{J, \leq} C\langle T_1, \dots, T_n \rangle \widetilde{\prec}_F T$$

□

Proposition 8.4.11 *Let F, J be a pair of compatible type functors and let A be a basic mapping.*

Let P be a predicate on SPIDER expressions and let Q be a predicate on type graphs. P, Q satisfy

$$Q(A(H)) \Rightarrow P(H) \tag{8.18}$$

$$\phi : T \xrightarrow{J, \leq} T', Q(T') \Rightarrow Q(T) \tag{8.19}$$

i.e. Q is closed under inverse (J, \leq) -morphisms.

Let H be a representative. Then $TT \vdash H$ and $Q(TT)$ imply $P(H')$ for all $H \rightarrow^ H'$.*

Proof: The subject reduction property (proposition 8.4.9) implies $TT \vdash H'$ for every $H \rightarrow^* H'$.

If $TT \vdash H'$ then there exists a (J, \leq) -morphism $\phi' : A(H') \rightarrow TT$. Since $Q(TT)$ and Q is closed under inverse morphisms it follows that $Q(A(H'))$ which implies $P(H')$. □

A type system for checking a predicate P thus consists of a pair of compatible type functors (F, J) , a linear mapping A and a predicates Q on type graphs as defined above. That is $TS = ((F, J), A, P, Q)$.

8.4.5 Composing Type Systems

We will now show that the conjunction and disjunction of predicates checked by type systems can also be checked by a type system.

Let $TS_i := ((F_i, J_i), A_i, Q_i, P_i)$, $i = 1, 2$ be two type systems. We define

$$F(T) := F_1(T) \times F_2(T)$$

$$J(T) := J_1(T) \times J_2(T)$$

$$A(H) := G[\chi, (a_1, a_2)] \text{ if } A_1(H) \cong_{F_1} G[\chi, a_1] \text{ and } A_2(H) \cong_{F_2} G[\chi, a_2]$$

That is we demand that the linear mappings map equivalent process graphs onto type graphs of the same structure.

We define

$$\begin{aligned} Q_{\wedge}(G[\chi, (a_1, a_2)]) &:= Q_1(G[\chi, a_1]) \wedge Q_2(G[\chi, a_2]) \\ Q_{\vee}(G[\chi, (a_1, a_2)]) &:= Q_1(G[\chi, a_1]) \vee Q_2(G[\chi, a_2]) \end{aligned}$$

It is easy to check that

$$\begin{aligned} TS_{\wedge} &:= ((F, J), A, Q_{\wedge}, P_1 \wedge P_2) \text{ and} \\ TS_{\vee} &:= ((F, J), A, Q_{\vee}, P_1 \wedge P_2) \end{aligned}$$

are also type systems, checking the conjunction respectively disjunction of P_1 and P_2 .

8.4.6 Compatible Type Functors

We will now give an example for l-monoids and compatible type functors which will be needed for our examples and which will be used in the rest of this section.

Let $(I, +, \leq)$ be an arbitrary residuated l-monoid with a smallest element \perp and let $k \in \mathbb{N}$. For any type graph T we define $F(T)$ as the set of all mappings from V_T^k into I .

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

$$F_{\phi}(a)(s') := \sum_{\phi(s)=s'} a(s) \quad (8.20)$$

$$J_{\phi}(a)(s') := \bigoplus_{\phi(s)=s'} a(s) \quad (8.21)$$

where $a \oplus b := (a + b) \vee a \vee b$.

We will now check that F, J satisfy the conditions of definition 8.4.1:

(8.15) For any set I with $|I| \leq 1$ it follows that $\sum_{i \in I} i \geq \bigoplus_{i \in I} i$.

Since ϕ is injective there is at most one $s \in V_T^k$ for every $s' \in V_{T'}^k$ such that $\phi(s) = s'$. Therefore

$$F_{\phi}(a)(s') = \sum_{\phi(s)=s'} a(s) \geq \bigoplus_{\phi(s)=s'} a(s) = J_{\phi}(a)(s')$$

(8.16) For any non-empty set I $\sum_{i \in I} i \leq \bigoplus_{i \in I} i$.

Since ϕ is surjective there is at least one $s \in V_T^k$ for every $s' \in V_{T'}^k$ such that $\phi(s) = s'$. Therefore

$$F_{\phi}(a)(s') = \sum_{\phi(s)=s'} a(s) \leq \bigoplus_{\phi(s)=s'} a(s) = J_{\phi}(a)(s')$$

(8.17) Let $\psi_i : T_i \xrightarrow{J_i^{\leq}} T'_i$ with $T_i = G_i[\chi_i, a_i]$, $T'_i = G'_i[\chi'_i, a'_i]$. Furthermore let C be a context and let η_i be the embedding of T_i into $C\langle T_1, \dots, T_n \rangle$.

Let $\psi := C\langle \psi_1, \dots, \psi_n \rangle$ and let η'_i be the embedding of T'_i into $C\langle T'_1, \dots, T'_n \rangle$.

Let $s' \in V_{T'}^k$, $S := \psi^{-1}(s')$, $S'_i := (\eta'_i)^{-1}(s')$, $S_i := \eta_i^{-1}(S) = \psi_i^{-1}(S'_i)$.

- Let $s' \in \left(\bigcup_{1 \leq i \leq n} \eta'_i(EXT_{T'_i}) \right)^k$. Since the ψ_i are nice morphisms it follows that there is exactly one $s_0 \in V_T$ such that $\psi(s_0) = s'$. Therefore $S = \{s_0\}$. And since the ψ_i are nice morphisms the mapping $\psi_i|_{S_i} : S_i \rightarrow S'_i$ is injective.

Therefore

$$\begin{aligned} J_\psi \left(\sum_{1 \leq i \leq n} F_{\eta_i}(a_i) \right)(s') &= \sum_{1 \leq i \leq n} \sum_{s_i \in S_i} a_i(s_i) \\ \sum_{1 \leq i \leq n} F_{\eta'_i}(J_{\psi_i}(a_i))(s') &= \sum_{1 \leq i \leq n} \sum_{\eta'_i(s'_i)=s'} \bigoplus_{\psi_i(s_i)=s'_i} a_i(s_i) \\ &= \sum_{1 \leq i \leq n} \sum_{(\eta'_i \circ \psi)(s_i)=s'} a_i(s_i) = \sum_{1 \leq i \leq n} \sum_{s_i \in S_i} a_i(s_i) \end{aligned}$$

- Let $s' \in (\eta'_j(V_{T'_j} \setminus EXT_{T'_j}))^k$. Because of the properties of embeddings and nice morphisms it follows that $S_i = \emptyset$, $S'_i = \emptyset$ for $j \neq i$. It also follows that η'_j has only one element and that $(\eta_j)|_{S_j}, (\eta'_j)|_{S'_j}$ are injective.

Therefore

$$\begin{aligned} J_\psi \left(\sum_{1 \leq i \leq n} F_{\eta_i}(a_i) \right)(s') &= J_\psi(F_{\eta_j}(a_j))(s') = \bigoplus_{s_i \in S_i} a_i(s_i) \\ \sum_{1 \leq i \leq n} F_{\eta'_i}(J_{\psi_i}(a_i))(s') &= F_{\eta'_j}(J_{\psi_j}(a_j))(s') = \bigoplus_{s_i \in S_i} a_i(s_i) \end{aligned}$$

- For any other $s' \in V_{T'}$ we have $S_i = S'_i = \emptyset$ for every i . If $S = \emptyset$ we obtain

$$J_\psi \left(\sum_{1 \leq i \leq n} F_{\eta_i}(a_i)(s') \right) = \perp \leq 0 = \sum_{1 \leq i \leq n} F_{\eta'_i}(J_{\psi_i}(a_i))(s')$$

If $S \neq \emptyset$ we get

$$J_\psi \left(\sum_{1 \leq i \leq n} F_{\eta_i}(a_i)(s') \right) = \bigoplus_{s \in S} 0 = 0 = \sum_{1 \leq i \leq n} F_{\eta'_i}(J_{\psi_i}(a_i))(s')$$

We will now show under which conditions the requirements for the existence of principal types and for type construction given in section 8.4.3 are satisfied: We assume that in both monoids $(I, +, \leq)$ and (I, \oplus, \leq) the sets $\{a' \mid a' \geq a, a' + a' = a'\}$ respectively $\{a' \mid a' \geq a, a' \oplus a' = a'\}$ have a smallest element. These smallest element are denoted by \bar{a}^+ respectively \bar{a}^\oplus .

That is we demand that every element has an upper bound which is idempotent. In the l-monoid of integers with ∞ and $-\infty$ these upper bounds are as follows:

$$\bar{a}^+ = \begin{cases} -\infty & \text{if } a = -\infty \\ 0 & \text{if } -\infty < a \leq 0 \\ \infty & \text{if } a > 0 \end{cases} \quad \bar{a}^\oplus = \begin{cases} a & \text{if } a \leq 0 \\ \infty & \text{if } a > 0 \end{cases}$$

That is $R(T)$ and $P(T)$ of a type graph T can be defined as follows:

$$\begin{aligned} R(G[\chi, a]) &:= G[\chi, a'] \text{ where } a'(s) := \begin{cases} \overline{a(s)}^+ & \text{if } \text{Set}(s) \subseteq \text{Set}(\chi) \\ \overline{a(s)}^\oplus & \text{otherwise} \end{cases} \\ P(G[\chi, a]) &:= G[\chi, a'] \text{ where } a'(s) := \begin{cases} a(s) \vee 0 & \text{if } \text{Set}(s) \subseteq \text{Set}(\chi) \\ a(s) & \text{otherwise} \end{cases} \end{aligned}$$

8.4.7 Examples

We now present various type systems useful for the verification of mobile processes. These type systems are of different quality, i.e. the percentage of correct process which can be typed varies. While in the type system checking confluence there is hardly any loss of information, mapping a process graph into a type in the type system checking the absence of vicious circles, may result in the creation of circles not present in the original process graph.

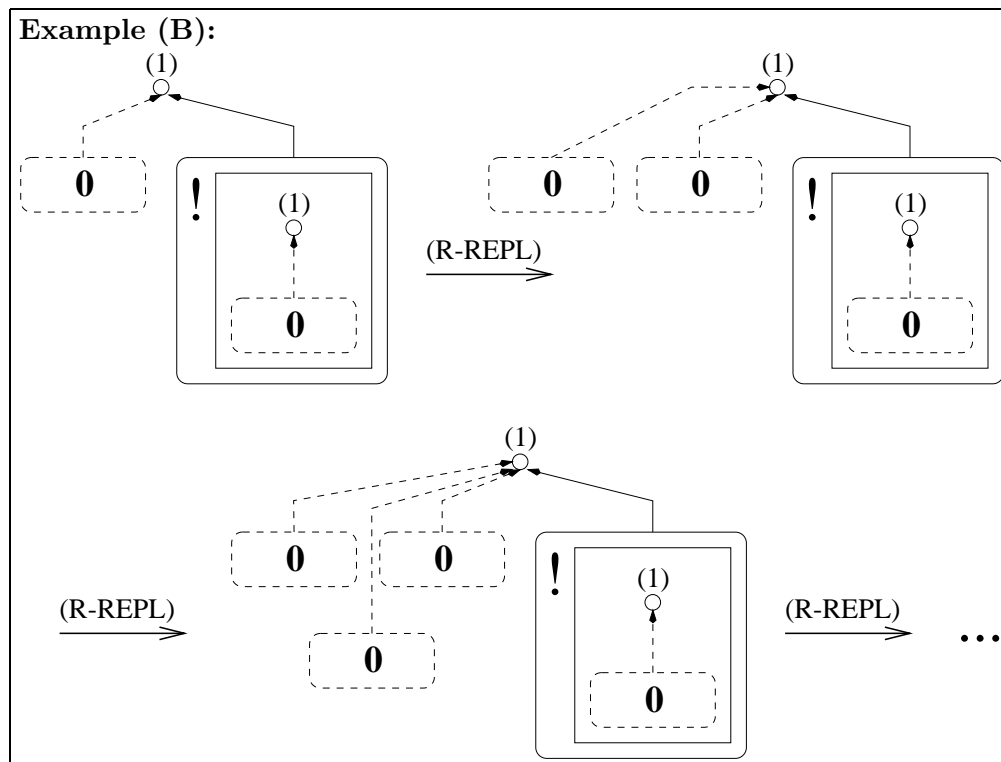
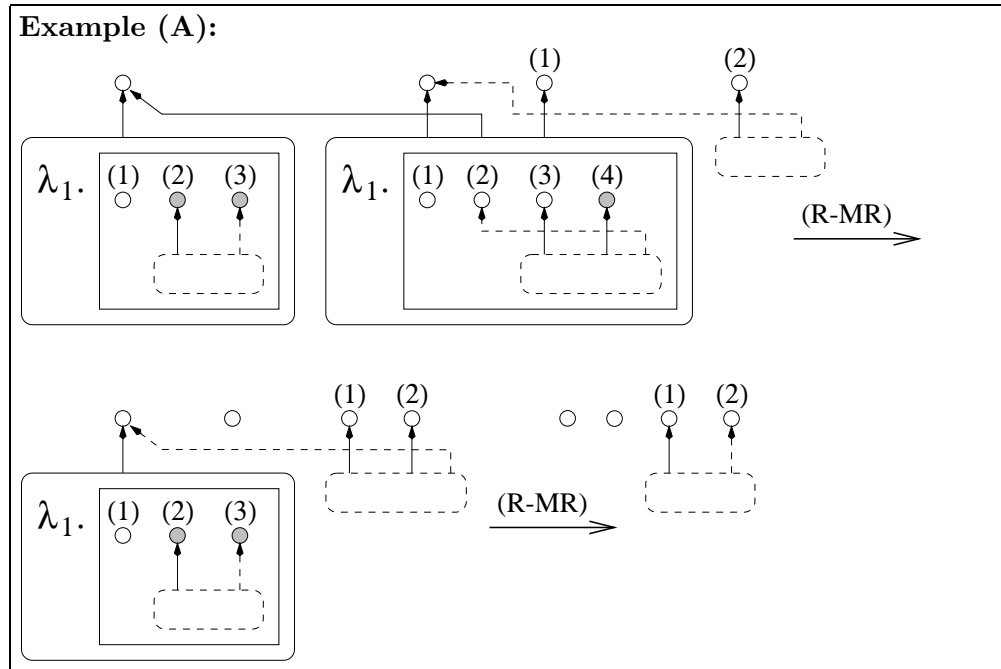
In the following examples we will assume a linear mapping A with

$$\begin{aligned} A(\text{proc}_n(S)) &:= P[\chi_P, a_P] \text{ where } P[\chi_P] := \text{proc}_n \\ A(\text{mess}_n(\mathbf{0})) &:= M[\chi_M, a_M] \text{ where } M[\chi_M] := \text{mess}_n \end{aligned}$$

and $a_P : V_P^k \rightarrow I$ and $a_M : V_M^k \rightarrow I$.

That is the graph structure of $A(H)$ is fixed. Only the monoid elements and $k \in \mathbb{N}$ will vary. Note that in contrast to the type system based on lattices we do not demand content-edges since we dropped higher-order communication.

We will apply the type systems in this section to the following two examples, **(A)** demonstrating mainly message reception and mobility of port addresses, and **(B)** demonstrating unlimited replication of messages. We will type the first process in either of the two reduction sequences.



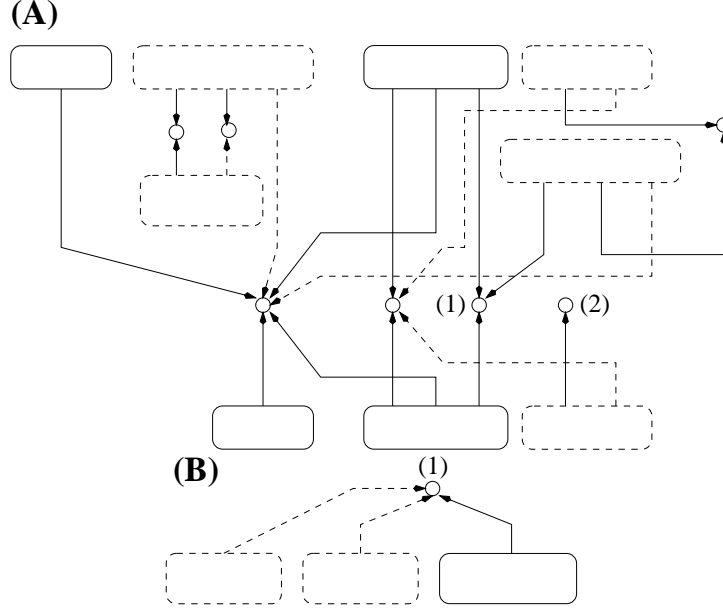
Avoiding Run-Time Errors

We set $k := 0$ and take the trivial monoid $I = \{0\}$. In this case the mapping A is constant, i.e. $a_M = 0$ and $a_P = 0$ where $A(proc_n(S)) := P[\chi_P, a_P]$, $A(mess_n(\mathbf{0})) := M[\chi_M, a_M]$. We get a standard type system, that only types

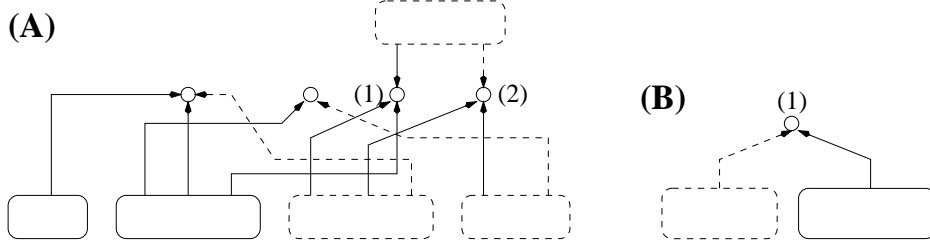
process graphs without runtime errors, i.e. no typed process contains a bad redex (see definition 4.3.4).

For a detailed comparison between this type system and standard type systems for the π -calculus see section 8.6.

If we type the examples **(A)** and **(B)** above, we obtain the following type graphs (before folding)



and after folding



Bounded Number of Processes/Messages

We want to check that there are at most K processes (messages) in a process graph. We choose I as Int_∞ (the integers with ∞ and $-\infty$, see the examples of l-monoids above) and set $k := 0$, i.e. every type graph is associated with only one integer. We define

$$\begin{aligned} a_P &:= 1 \\ a_M &:= 0 \end{aligned}$$

if we want to count processes and

$$\begin{aligned} a_P &:= 0 \\ a_M &:= 1 \end{aligned}$$

if we want to count messages. $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(\mathbf{0})) := M[\chi_M, a_M]$.

Furthermore we define

$$Q(G[\chi, a]) := (a \leq K)$$

Since the number associated to a type graph can only become greater when a (J, \leq) -morphism is applied, it follows that Q is closed under (J, \leq) -morphisms.

Applying this type system to the example process **(A)** yields the results 2 for the number of processes and 1 for the number of messages. The latter result is quite interesting since it can only be obtained by the subtraction mechanism in typing rule (TM-PA).

In the case of example **(B)** the results are 1 for the number of processes and ∞ for the number of messages.

Bounded Number of Messages at one Port

We want to check that there are at most K messages waiting at a certain external port $\lfloor \chi_H \rfloor_n$, $n \in \mathbb{N}$ i.e.

$$P(H) := (|\{q \in M_H \mid \text{send}_H(q) = \lfloor \chi_H \rfloor_n\}| \leq K)$$

We choose I as Int_∞ and set $k := 1$.

$$\begin{aligned} a_P(\lfloor \text{ext}_P \rfloor_i) &:= 0 \text{ for any } i \\ a_M(\lfloor \text{ext}_M \rfloor_i) &:= \begin{cases} 1 & \text{if } i = n \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(\mathbf{0})) := M[\chi_M, a_M]$.

Furthermore we define

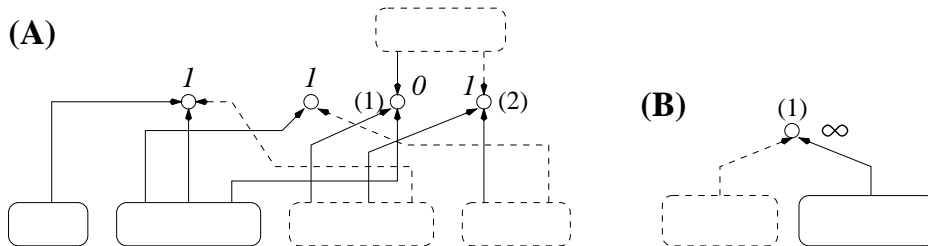
$$Q(G[\chi, a]) := (a(\lfloor \chi \rfloor_n) \leq K)$$

It is left to show that Q is closed under inverse (J, \leq) -morphisms: Let

$$\phi : G[\chi, a] \rightarrow G'[\chi', a']$$

such that $J_\phi(a) \leq a'$ and $Q(G'[\chi', a'])$. It follows that $K \geq a'(\lfloor \chi' \rfloor_n) = \bigoplus_{\phi(v)=\lfloor \chi' \rfloor_n} a(v) \geq a(\lfloor \chi \rfloor_n)$.

Applying this type system to the examples yields the following results (we will not present the typing process in detail):



That is, in the case of example **(A)**, the type graph correctly states that there is at most one message attached to any port at any time. Furthermore no message is ever sent to the first external port. In example **(B)** the precondition of rule (TM-REPL) leads to the label ∞ at the external port, which is quite true, since the process is able to generate infinitely many messages.

Minimum Number of Messages at one Port

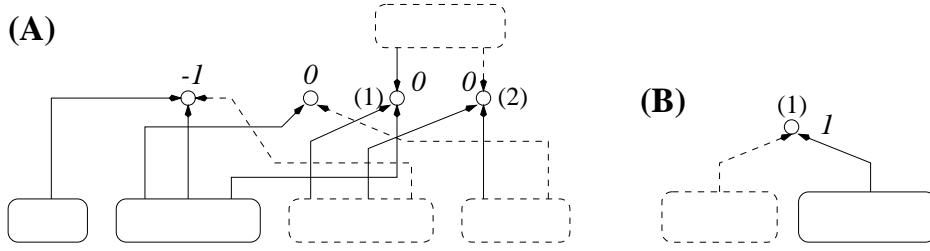
This time we want to check

$$P(H) := (|\{q \mid \text{send}_H(q) = (\chi_H)_n\}| \geq K)$$

We only have to change the underlying monoid $(I, +, \leq)$ from the previous example to $(I, +, \geq)$. The basic mapping A stays the same and Q is modified to

$$Q(G[\chi, a]) := (a(\lfloor \chi \rfloor_n) \geq K)$$

Applying this type system to the examples yields the following results:



In the case of **(A)** there is no guaranteed number of message attached to any port. The leftmost port is even labelled -1 which means that it can take away messages without replacing them by new ones. In the other example **(B)** the type system correctly states that there is always at least one message attached to the port.

Starving Processes

In a process graph H a starving process listens out for a message on an internal port. But all other hyperedges attached to that port are also processes waiting for a message via the same port. It is clear that such a message will never arrive.

We will present two versions of type systems preventing such a situation. The first one is as follows: We will use as monoid a pair of integers from Int_∞ and set $k := 1$. Furthermore we define our partial order to be $(a_1, a_2) \sqsubseteq (b_1, b_2) \iff (a_1 \leq b_1) \wedge (a_2 \geq b_2)$.

$$\begin{aligned} a_P(\lfloor \text{ext}_P \rfloor_i) &:= \begin{cases} (1, 0) & \text{if } S = \lambda_i.H \\ (0, 1) & \text{otherwise} \end{cases} \\ a_M(\lfloor \text{ext}_M \rfloor_i) &:= (0, 1) \text{ for any } i \end{aligned}$$

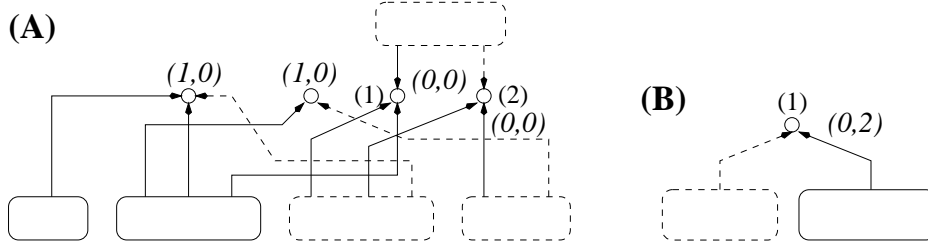
where $A(proc_n(S)) := P[\chi_P, a_P]$, $A(mess_n(\mathbf{0})) := M[\chi_M, a_M]$.

Furthermore

$$Q(G[\chi, a]) := (\forall v \in V_G \setminus Set(\chi) : \text{if } (a_1, a_2) = a(v) \text{ then } (a_1 > 0 \Rightarrow a_2 > 0))$$

That is we demand that if there is a process waiting at a certain port, there is at least one other hyperedge attached to it, which seems to be a reasonable approach. Because of the folding of type graphs information might get lost.

Applying this type system to the examples yields the following results:



In case **(A)** the type system fails for the two leftmost ports. No starving processes can appear on any of these ports, but the type system cannot confirm this fact. The predicate Q is satisfied for **(B)** which is not very surprising, since there are no listening processes at all.

The second version of the type system also prohibits starving processes, but this time our approach is less direct: we demand that the number of processes waiting at one port is smaller or equal than the number of the rest of the hyperedges.

We will use the Int_∞ as monoid and set $k := 1$.

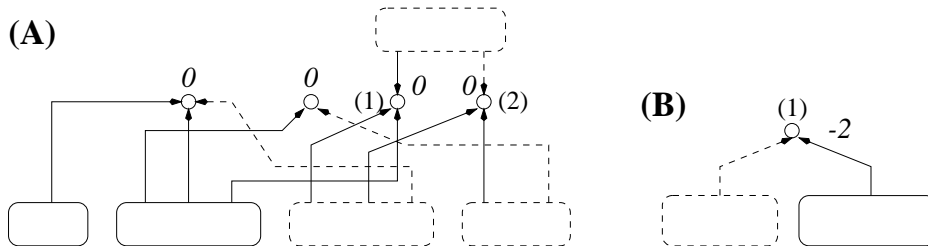
$$\begin{aligned} a_P(\lfloor ext_P \rfloor_i) &:= \begin{cases} 1 & \text{if } S = \lambda_i.H \\ -1 & \text{otherwise} \end{cases} \\ a_M(\lfloor ext_M \rfloor_i) &:= -1 \text{ for any } i \end{aligned}$$

where $A(proc_n(S)) := P[\chi_P, a_P]$, $A(mess_n(\mathbf{0})) := M[\chi_M, a_M]$.

Furthermore

$$Q(G[\chi, a]) := (\forall v \in V_G \setminus Set(\chi) : a(v) \leq 0)$$

Applying this type system to the examples yields the following results:



The first version of a type system checking for starving processes failed on **(A)**, but fortunately the second version works and Q is satisfied. The same is true for example **(B)**.

Since for both type systems there exist process which can be typed by it but not by the other system, the best results can be obtained by the disjunction of the two systems.

An even more convincing type system can be obtained by performing the disjunction on the level of node labels and not on the level of entire predicates.

Blocked Messages

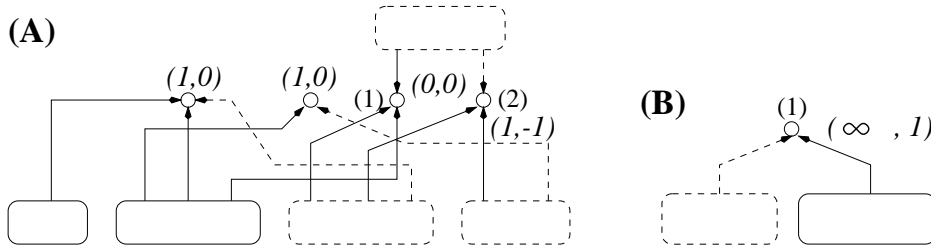
A blocked message is a message sent to an internal port where the rest of the hyperedges attached to that port are only messages whose aim is the very same port. Blocked message are never received by a process.

By using slightly different versions of the linear mapping A we can use the same type systems as for starving processes, in order to avoid blocked messages. The respective predicates Q stay the same.

In the first case we set:

$$\begin{aligned} a_P(\lfloor ext \rfloor_i) &:= (0, 1) \\ a_M(\lfloor ext \rfloor_i) &:= \begin{cases} (1, 0) & \text{if } i = n \\ (0, 1) & \text{otherwise} \end{cases} \end{aligned}$$

Applying this type system to the examples yields the following results:



Again in the case of **(A)** the result of the type system is not really satisfying: the two leftmost ports never have blocked messages attached to them, their labels, however, do not satisfy Q . The result is correct for the rightmost port to which indeed a blocked message is attached at the end of reduction. (In this case this is not grave since it is an external port.)

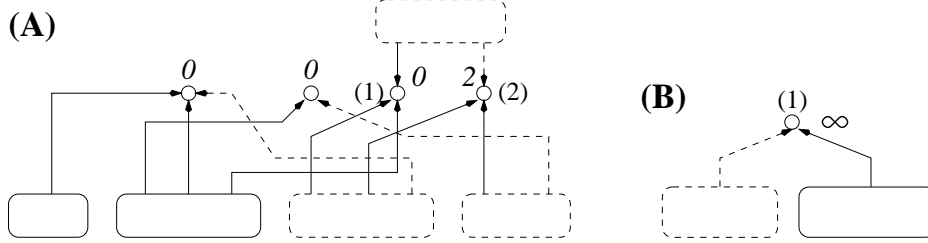
In example **(B)** there are no blocked messages by definition since there is still a process attached to the same port. The type system confirms this fact.

And in the second case:

$$a_P(\lfloor ext \rfloor_i) := -1$$

$$a_M(\lfloor ext \rfloor_i) := \begin{cases} 1 & \text{if } i = n \\ -1 & \text{otherwise} \end{cases}$$

Applying this type system to the examples yields the following results:



In the case of **(A)** the type system yields exactly the desired results: only the rightmost port may have blocked messages attached to it. For example **(B)**, however, the type system gets the wrong result: it does not guarantee absence of blocked message. Here the first variant of the type system succeeded and the second did not.

8.4.8 Confluence

Our aim is to identify confluent processes with the help of a type system. As already mentioned in section 7.2, SPIDER is *not* confluent. The reason for this are overlapping redexes, i.e. several messages are sent to one port or several processes are listening at one port.

We will now show how to avoid overlaps destroying the confluence of the calculus:

Proposition 8.4.12 *Let H be a SPIDER expression satisfying:*

- *If $q_1, q_2 \in M_H$ and $send_H(q_1) = send_H(q_2)$ then*
 - *either $q_1 = q_2$*
 - *or $card(q_1) = card(q_2) = 1$ and $l_H(q_1) \equiv l_H(q_2)$*
- *If $p_1, p_2 \in P_H$, $l_H(p_1) = \lambda_{k_1}.J_1$, $l_H(p_2) = \lambda_{k_2}.J_2$ and $\lfloor s_H(p_1) \rfloor_{k_1} = \lfloor s_H(p_2) \rfloor_{k_2}$ then*
 - *either $p_1 = p_2$*
 - *or $card(p_1) = card(p_2) = 1$, $k_1 = k_2 = 1$ and $J_1 \equiv J_2$*

If furthermore $H \rightarrow H_1$ and $H \rightarrow H_2$, then either $H_1 \equiv H_2$ or there exists an expression H' with $H_1 \rightarrow H'$ and $H_2 \rightarrow H'$.

Proof: There are the following cases:

- the two redexes in H do not overlap, i.e. $H \equiv C\langle K_1, K_2, K_3 \rangle$, $K_1 \rightarrow K'_1$, $K_2 \rightarrow K'_2$ and

$$H_1 \equiv C\langle K'_1, K_2, K_3 \rangle$$

$$H_2 \equiv C\langle K_1, K'_2, K_3 \rangle$$

Then we define $H' := C\langle K'_1, K'_2, K_3 \rangle$.

- the redexes overlap. Since the messages and processes satisfy the constraints above it follows that $H_1 \equiv H_2$.

□

Now we will define a type system that ensures the property defined above. As a monoid I we take the set of all mappings from $(\mathcal{S}/\equiv) \times \mathbb{N}$ into Int_∞ (with pointwise summation and pointwise partial order).

Let H be a process graph and let $f := a(v) \in \{(\mathcal{S}/\equiv) \times \mathbb{N} \rightarrow Int_\infty\}$. $f([mess_n(\mathbf{0})]_\equiv, i) = m$ means intuitively that there are m messages q_1, \dots, q_m with $[s_H(q_i)]_i = v$. $f([proc_n(S)]_\equiv, i) = m$ means that there are m processes p_1, \dots, p_m with $l_H(p_i) \equiv S$ and $[s_H(p_i)]_i = v$.

That is, we keep track of all hyperedges that can be attached to a port. The linear mapping is more complex than the linear mappings defined above, but since for every node v only a finite number of tuples are mapped by $a(v)$ to a value different from 0 it is still effectively computable.

The linear mapping A is defined as follows:

$$\begin{aligned} a_P([\chi_P]_i)([H]_\equiv, j) &:= \begin{cases} 1 & \text{if } H \equiv proc_n(S), i = j \\ 0 & \text{otherwise} \end{cases} \\ a_M([\chi_M]_i)([H]_\equiv, j) &:= \begin{cases} 1 & \text{if } H \equiv mess_n(\mathbf{0}), i = j \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where $A(proc_n(S)) := P[\chi_P, a_P]$, $A(mess_n(\mathbf{0})) := M[\chi_M, a_M]$. Furthermore we define

$$\begin{aligned} Q(G[\chi, a]) &:= (\forall v \in V_G : \\ &\quad (a(v)([proc_{n_1}(\lambda_{k_1}.H_1)]_\equiv, k_1) =: l_1 \geq 1 \wedge \\ &\quad a(v)([proc_{n_2}(\lambda_{k_2}.H_2)]_\equiv, k_2) =: l_2 \geq 1 \\ &\quad \Rightarrow n_1 = n_2 \wedge k_1 = k_2 \wedge H_1 \equiv H_2 \wedge \\ &\quad (l_1 = l_2 = 1 \vee n_1 = n_2 = 1)) \\ &\quad a(v)([mess_{n_1}(\mathbf{0})]_\equiv, n_1) =: l_1 \geq 1 \wedge \\ &\quad a(v)([mess_{n_2}(\mathbf{0})]_\equiv, n_2) =: l_2 \geq 1 \\ &\quad \Rightarrow n_1 = n_2 \wedge (l_1 = l_2 = 1 \vee n_1 = n_2 = 1)) \end{aligned}$$

It is straightforward to check that $Q(A(H))$ implies that H satisfies the conditions of proposition 8.4.12. Furthermore Q is closed under inverse (J, \leq) -morphisms.

Proposition 8.4.13 (Confluence) *Let $T \vdash H$, $H \rightarrow^* H_1$ and $H \rightarrow^* H_2$. Then there exists an expression H' such that $H_1 \rightarrow^* H'$ and $H_2 \rightarrow^* H'$.*

Proof: By induction on the number of reduction steps. □

Another type system ensuring confluence for the π -calculus is described in [NS97].

Applying this type system to **(A)** yields the following labelling (see below for which nodes v_1, v_2, v_3, v_4 stand). We assume that A_1 is the process description

of the leftmost process and that A_2 is the process description of the rightmost process.

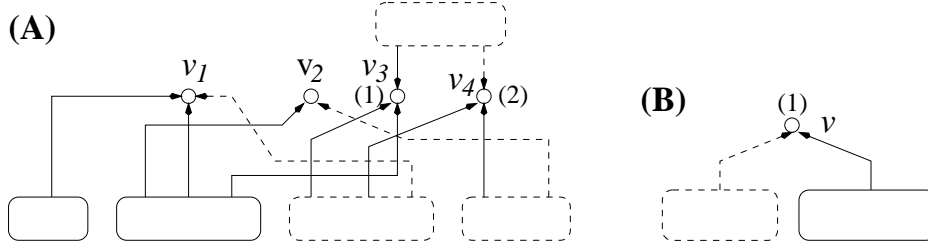
$$\begin{array}{ll}
a(v_1)([proc_1(A_1)]_{\equiv}, 1) &= 1 & a(v_3)([proc_1(A_2)]_{\equiv}, 3) &= 1 \\
a(v_1)([proc_3(A_2)]_{\equiv}, 2) &= 1 & a(v_3)([mess_3(\mathbf{0})]_{\equiv}, 1) &= 1 \\
a(v_1)([mess_3(\mathbf{0})]_{\equiv}, 3) &= 1 & a(v_3)([mess_2(\mathbf{0})]_{\equiv}, 1) &= 1 \\
\\
a(v_2)([proc_3(A_2)]_{\equiv}, 1) &= 1 & a(v_4)([mess_3(\mathbf{0})]_{\equiv}, 2) &= 1 \\
a(v_2)([mess_2(\mathbf{0})]_{\equiv}, 2) &= 1 & a(v_4)([mess_2(\mathbf{0})]_{\equiv}, 1) &= 1 \\
& & a(v_4)([mess_2(\mathbf{0})]_{\equiv}, 2) &= 1
\end{array}$$

Note that the equivalence class $[mess_2(\mathbf{0})]_{\equiv}$ contains two elements. Q is satisfied and thus the process in example **(A)** is confluent. This is not very surprising, since, in fact, there is only one possible sequence of reductions.

In the case of example **(B)** we get the labelling

$$\begin{array}{l}
a(v)([proc_1(!mess_1(\mathbf{0}))]_{\equiv}, 1) = 1 \\
a(v)([mess_1(\mathbf{0})]_{\equiv}, 1) = \infty
\end{array}$$

which also satisfies Q . The function value ∞ is compensated by the fact that the corresponding message has cardinality 1.



8.4.9 Avoiding Deadlocks

We will first investigate processes being incapable of performing any further reductions. One of the reasons is good-natured, i.e. the process is merely waiting for input or output operations. The other two cases can be considered as a deadlock and we will propose a type system for avoiding such deadlocks.

Proposition 8.4.14 (Deadlock) *Let H be a process graph with a non-empty edge set such that there is no process graph H' with $H \rightarrow H'$. 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.*
- (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.*

That is there is a port $v \in V_H$ with at most one hyperedge attached to it such that either

- if there is an edge $e \in E_H$ with $\lfloor s_H(e) \rfloor_i = v$ then $z_H(e) = \text{mess}$ and $i = \text{card}(e)$
 - if there is an edge $e \in E_H$ with $\lfloor s_H(e) \rfloor_i = v$ then $z_H(e) = \text{proc}$ and $l_H(e) = \lambda_i.J$.
- (3) There is a vicious circle, i.e. a sequence $v_0, \dots, v_n = v_0 \in V_H$ such that for every pair v_i, v_{i+1} there is
- either a message q with $\text{send}_H(q) = v_i$ and $\lfloor s_H(q) \rfloor_j = v_{i+1}$ for some $j \in \{1, \dots, \text{card}(q)\}$
 - or a process p with $l_H(p) = \lambda_k.Q$, $\lfloor s_H(p) \rfloor_k = v_i$ and $\lfloor s_H(p) \rfloor_j = v_{i+1}$ for some $j \in \{1, \dots, \text{card}(p)\}$

Proof: We will traverse the hypergraph H with the following algorithm and show that either condition (1), (2) or (3) will occur. We start with $D := \emptyset$ and let e be an arbitrary edge of H .

◇ $D := D \cup \{e\}$

There are the following cases:

$z_H(e) = \text{mess}$: Let $v := \text{send}_H(e)$

- v is an external port \Rightarrow (1)
- the only edges attached to v are messages sent to $v \Rightarrow$ (2)
- there is a message q with $\lfloor s_H(q) \rfloor_j = v$, $j < \text{card}(q)$. There are two cases:
 - $q \in D \Rightarrow$ (3)
 - $q \notin D$: take q as the new e and continue with ◇
- there is a process p with $\lfloor s_H(p) \rfloor_j = v$. There are the following cases
 - $l_H(p) = !J$ or $l_H(p) = \lambda_j.J$: then reductions are possible (either (R-REPL) or (R-MR)), which is a contradiction
 - $l_H(p) = \lambda_k.J$, $k \neq j$, $p \in D \Rightarrow$ (3)
 - $l_H(p) = \lambda_k.J$, $k \neq j$, $p \notin D$: take p as the new e and continue with ◇

$z_H(e) = \text{proc}$: If $l_H(e) = !J$ reductions would be possible which is a contradiction. We can therefore conclude that $l_H(e) = \lambda_k.J$. Let $v := \lfloor s_H(e) \rfloor_k$

- v is an external port \Rightarrow (1)
- the only edges attached to v are processes listening on $v \rightarrow$ (2)
- there is a process p with $\lfloor s_H(p) \rfloor_j = v$, $j \neq k$. There are two cases:
 - $p \in D \Rightarrow$ (3)
 - $p \notin D$: take p as the new e and continue with ◇
- there is a message q with $\lfloor s_H(q) \rfloor_j = v$. There are the following cases:

$j = \text{card}(q)$ that is $\text{send}_H(q) = v$. This is a contradiction since a reduction by rule (R-MR) is possible.

$j \neq \text{card}(q), q \in D \Rightarrow \mathbf{(3)}$

$j \neq \text{card}(q), q \notin D$: take q as the new e and continue with \diamond

Since E_H is finite and D increases in every step, the algorithm will always terminate. \square

Condition **(1)** cannot be considered as a deadlock, the process is just blocked since it is waiting for an external communication partner. Condition **(2)** corresponds to the existence of blocked messages or starving processes, while **(3)** can be checked as follows:

Absence of Vicious Circles

We want to avoid circles as described in **(3)** in a process graph (this is the first step to avoid deadlocks). We set $k := 2$ and I is again the integers with ∞ and $-\infty$ and the conventional \leq -relation.

We define:

$$a_P([ext_P]_i, [ext_P]_j) := \begin{cases} 1 & \text{if } S = \lambda_i.H, j \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$a_M([ext_M]_i, [ext_M]_j) := \begin{cases} 1 & \text{if } i = n, j \neq n \\ 0 & \text{otherwise} \end{cases}$$

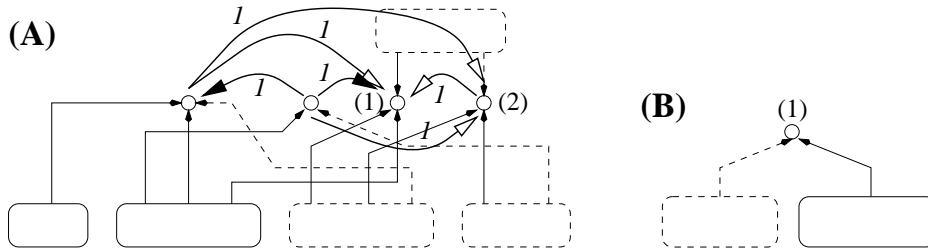
where $A(\text{proc}_n(S)) := P[\chi_P, a_P]$, $A(\text{mess}_n(\mathbf{0})) := M[\chi_M, a_M]$.

Furthermore:

$$Q(G[\chi, a]) := (\exists v_0, \dots, v_n = v_0 \in V_G : a(v_i, v_{i+1}) \geq 1, 0 \leq i < n)$$

Another type system ensuring deadlock freedom for the π -calculus is described in [Kob97].

Applying this type system to the examples yields the following type graphs:



where in the case of example **(A)** an arrow labelled x from node v_1 to node v_2 indicates that $a(v_1, v_2) = x$. On all other pairs, a has value 0. The arrows created by processes have a black tip, while the arrows created by messages have a white tip. Since there is no circle of arrows we can guarantee that the expression will never generate a vicious circle.

Things are much less complicated for example **(B)**: there are no arrows at all and therefore there is no danger of a vicious circle

8.5 Transformation of Type Systems

We will now show that both type systems presented in this work are equivalent when they overlap. That is we will show that if a process without higher-order messages can be typed in the first system it can be typed in the second system as well. On the other hand if the monoid in the second system is, in fact, a lattice, we can show that typings are equivalent for non-higher-order processes.

In this section we impose the following restrictions on the linear mapping A :

$$\begin{aligned} G[\chi, a] := A(proc_n(S)) &\Rightarrow G[\chi] \cong proc_n \\ G[\chi, a] := A(mess_n(H)) &\Rightarrow G[\chi] \cong \sigma_{1,\dots,n}(\theta_{n,n+k}(mess_n \oplus cont_k)) \end{aligned}$$

where $n \in \mathbb{N}$, H is a process graph and $k := card(H)$.

All example type systems based on lattices satisfy this condition. And all example type systems based on monoids can easily be converted in order to satisfy it.

8.5.1 Conversion Lattice \rightarrow Monoid

We now show that every type which can be assigned to an expression by the rules of the type system based on monoids (TL-*) can also be assigned to it by the rules of the type system based on monoids (TM-*). This is, of course, only true for expressions satisfying the restraints imposed in section 8.4, i.e. for process without higher-order communication belonging to the subcalculus \mathcal{S}_n .

In the following proposition we will exploit the fact that a lattice is always a lattice-ordered commutative monoid where summation of two elements is equal to their least upper bound.

Proposition 8.5.1 (Lattice \rightarrow Monoid) *Let S be a process description or process graph in \mathcal{S}_n without variables whose messages are all labelled $\mathbf{0}$.*

If $E, G[\chi, a] \vdash_{A,F}^L S$ it follows that $G[\chi, a] \vdash_{A,F,F}^M S$.

Proof: See appendix A.2.3. □

8.5.2 Conversion Monoid \rightarrow Lattice

We now show that the conversion works also in the opposite direction, at least under certain circumstances. We assume that the monoids $F(G)$ in the type system are, in reality lattices, i.e. the sum operation corresponds to the supremum or least upper bound.

We will show that in this case, every expression having a type in the system based on monoids has a type in the system based on lattices as well, even with an arbitrary type environments E .

Proposition 8.5.2 (Monoid \rightarrow Lattice) *Let S be a process description or process graph in \mathcal{S}_n without variables whose messages are all labelled $\mathbf{0}$.*

We consider a type system where the linear mapping A satisfies the constraints of the type system based on lattices and where the monoid operations in $F(G) = (I, +, \leq)$ coincide with the supremum, i.e. $+$ = \vee = \oplus .

If $\hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F,F}^M S$ it follows that for any environment E for \hat{G} :

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L S$$

Proof: See appendix A.2.3. □

8.6 Comparison of Type Systems: SPIDER \leftrightarrow π -Calculus

In this section we assume that there is a fixed type functor F .

We will now show that simple types with recursion for the asynchronous π -calculus (for the semantics of the π -calculus see section 7.5) can be translated into a SPIDER type and vice versa. The following type system corresponds more or less to the type system presented in [PS93] (without keeping track of input/output capabilities).

In the π -calculus types or sorts are normally represented by terms with a fixed-point operator μx . This representation is not suited for our translation and we will therefore assume that a type in the π -calculus is a potentially infinite tree with only finitely many subtrees (for conversion of a term into a tree and vice versa see e.g. [PS93, Urz95]). The leaves of such an infinite tree are labelled with variables from the set \mathcal{A} .

We can compose n type trees tr_1, \dots, tr_n to another type tree

$$t := [tr_1, \dots, tr_n]$$

by creating a new root and regarding the old root nodes as the sons of the new root. $\alpha \in \mathcal{A}$ is a variable representing a type tree. We define $\text{card}([tr_1, \dots, tr_n]) := n$.

In the following Γ is a set of type assignments of the form $a : t$ where t is a type tree and a is a name in the π -calculus. If $a : t \in \Gamma$ then $\Gamma(a) := t$.

The typing rules are

$$\begin{array}{c}
\text{(T}\pi\text{-NIL)} \quad \Gamma \vdash \mathbf{0} \qquad \text{(T}\pi\text{-PAR)} \quad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p|q} \\
\\
\text{(T}\pi\text{-REPL)} \quad \frac{\Gamma \vdash p}{\Gamma \vdash !p} \qquad \text{(T}\pi\text{-RESTR)} \quad \frac{\Gamma, a : tr \vdash p}{\Gamma \vdash (\nu a)p} \\
\\
\text{(T}\pi\text{-IN)} \quad \frac{\Gamma(a) = [tr_1, \dots, tr_m] \quad \Gamma, x_1 : tr_1, \dots, x_m : tr_m \vdash p}{\Gamma \vdash a(x_1, \dots, x_m).p} \\
\\
\text{(T}\pi\text{-OUT)} \quad \frac{\Gamma(a) = [\Gamma(a_1) \dots, \Gamma(a_m)] \quad \Gamma \vdash p}{\Gamma \vdash \bar{a}a_1 \dots a_m.p}
\end{array}$$

We will also write $\Gamma \vdash^\pi p$ if it is not clear which type system is under consideration.

The type system for the π -calculus satisfies the following properties:

- (1) $\Gamma \vdash p, p \equiv p' \Rightarrow \Gamma \vdash p'$
- (2) $\Gamma \vdash p, p \rightarrow p' \Rightarrow \Gamma \vdash p'$
(Subject Reduction Property)
- (3) $a_1 : tr_1, \dots, a_n : tr_n \vdash p \Rightarrow b_1 : tr_1, \dots, b_n : tr_n \vdash p[b_1/a_1, \dots, b_n/a_n]$
(Substitution Law)
- (4) $\Gamma \vdash p, z \notin \Gamma \Rightarrow \Gamma, a : tr \vdash p$
(Weakening Law)

We will now show how to translate a type graph into a type tree:

Definition 8.6.1 (Type Graph \rightarrow Type Tree) Let G be a true graph and let σ be a function mapping the nodes in $\{v \in V_G \mid \nexists q \in E_G : z_G(q) = \text{mess}, \text{send}_G(q) = v\}$ (i.e. all nodes that are not in the range of send_G) onto type trees.

Let $v \in V_G$. We define:

$$Tree_\sigma^G(v) := \begin{cases} [Tree_\sigma^G(v_1), \dots, Tree_\sigma^G(v_n)] & \text{if } \exists q \in E_G : z_G(q) = \text{mess}, \\ & s_G(q) = v_1 \dots v_n v \\ \sigma(v) & \text{otherwise} \end{cases}$$

□

8.6.1 SPIDER \rightarrow π -Calculus

We will now show that if the SPIDER version of a π -calculus expression is typable, then the original expression is typable in the type system for the π -calculus. For SPIDER we will use the type system based on lattices since it makes the proof easier to handle. Note, however, that neither the type environment E nor the lattice element a do have any influence on the construction of the type of the π -calculus expression.

We need the encodings introduced in sections 7.4 and 7.5: Θ_N transforms a SPIDER expression in name-based notation into an ordinary SPIDER expression and Δ_π^t transforms a SPIDER expression in name-based notation into a π -calculus expression.

Proposition 8.6.2 *Let $E, G[\chi, a] \vdash \Theta_N(h[t])$ and $n := |\chi|$. It follows that*

$$[u]_1 : Tree_\sigma^G([\chi]_1), \dots, [u]_n : Tree_\sigma^G([\chi]_n) \vdash \Delta_\pi^u(S)$$

where u is a duplicate-free string with $|u| = n$ and σ is an arbitrary substitution as defined in definition 8.6.1.

Proof: See appendix A.2.4.

□

8.6.2 π -Calculus \rightarrow SPIDER

We will now show that if there exists a type for a π -calculus expression, there exists a type environment and a type graph such that the encoding of the expression into SPIDER can be typed as well.

We will assume that the linear mapping A satisfies the properties specified at the beginning of section 8.5.

In the proof of proposition 8.6.4 we will construct the type graph of an encoding of a π -calculus expression by forming quotient graphs. The following lemma will tell us under what conditions these quotient graphs are defined and how they relate to the set of type assignments Γ .

Lemma 8.6.3 *Let G_1, \dots, G_n be true graphs (where all content-edges have cardinality 0), let \sim be an equivalence on the nodes of the type graphs and let the σ_i be mappings satisfying*

$$v_i \in V_{G_i}, v_j \in V_{G_j}, (v_i, i) \sim (v_j, j) \Rightarrow Tree_{\sigma_i}^{G_i}(v_i) = Tree_{\sigma_j}^{G_j}(v_j) \quad (8.22)$$

Then there is a smallest consistent equivalence \approx containing \sim such that $G_1 \dots G_n / \approx$ is a true graph.

Let $p_i : G_i \rightarrow G_1 \dots G_n / \approx$ be the i -th projection into the quotient graph. We define $\sigma(p_i(v_i)) := \sigma_i(v_i)$. Then σ is well-defined and:

$$Tree_{\sigma}^{G_1 \dots G_n / \approx}(p_i(v_i)) = Tree_{\sigma_i}^{G_i}(v_i)$$

for all $v_i \in V_{G_i}$.

Proof: See appendix A.2.4. □

If a π -calculus expression can be typed, we can construct a corresponding type graph and a type environment, which types the encoding of the π -calculus expression into SPIDER.

Proposition 8.6.4 *Let $\Gamma \vdash p$ and let t be a duplicate-free string such that $fn(p) = Set(t) \subseteq Set(\Gamma)$.*

Then there exists a true type graph $G[\chi, a]$, a type environment E and a mapping σ such that

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^t(p))$$

and $Tree_\sigma^G([\chi]_i) = \Gamma([t]_i)$.

Proof: See appendix A.2.4. □

Chapter 9

Conclusion

In this work we presented a specification method SPIDER for process calculi based on hypergraphs and hypergraph rewriting. There are several approaches to hypergraph construction: the set-based approach, the categorical or algebraic approach, graph expressions and a name-based notation. We have shown that these methods of graph construction are equivalent and can be converted into one another. This enables us to choose a suitable representation for a process graph for every problem.

We introduced barbed congruence for SPIDER and presented proof techniques for facilitating proofs of barbed congruence. Furthermore we investigated the connection between the λ - and the π -calculus and SPIDER, and we demonstrated how to encode λ -expressions and π -calculus expressions into SPIDER.

A central part of this work is the generation of type systems for our process calculus. Types are also graphs and we attempt to conserve information about the process in its type by labelling the type with lattice or monoid elements. The type system is parametrized in the sense that the assignment of lattice or monoid elements to type graphs is not fixed and can be instantiated according to the property which is to be checked.

We presented two versions of the type system: one is based on lattices and can type processes with higher-order communication, the other one is based on monoids, of which lattices are only a special case, but excludes higher-order communication. We have shown how to verify interesting properties such as absence of deadlocks, confluence or data security with the help of the type system.

When typing a process, information often gets lost, i.e. not all processes without deadlocks are actually typable by the corresponding type system. Future work could therefore involve the improvement of the type system, changing it in order to type as many processes as possible. This might be achieved by combining type systems and temporal logic.

Another area of research would be the closer combination of bisimulation and type systems. There are type systems, e.g. checking privacy in process calculi [Aba97], that yield results of bisimulation equivalence for processes. That

is, proving the bisimilarity of processes can be achieved by finding a type in a corresponding type system.

A related line of research is the definition of barbed congruence wrt. to a type. That is, we demand that congruent processes have the same type. It would be interesting to investigate how this combination of barbed congruence and our system could contribute to the verification of processes.

It would also be desirable to expand SPIDER into a full-fledged programming language and to test the type system in “real life”. SPIDER as a real programming language would require additional syntactic constructs for easier programming, a graphical user interface for “drawing” processes and an efficient implementation. A distributed implementation could be achieved with the distributed graph rewriting system by Boris Reichel [Rei98]. This work describes efficient execution of graph rewriting in a multi-processor environment.

The main contribution of this thesis is the supplying of graph rewriting and graph construction techniques for the specification of the semantics of a mobile process calculus, and the development of generic type systems for the verification of processes.

Appendix A

Proofs

A.1 Methods of Hypergraph Construction

Proposition 2.2.11 (Graph Construction and Factorizations)

Let $(\phi, \eta'_1, \dots, \eta'_n) := \lim_{i=1}^n (\phi_i, \eta_i)$ with $\eta_i : G_i[\chi_i] \rightarrowtail G[\chi]$, $\phi_i : G_i[\chi_i] \rightarrow G'_i[\chi'_i]$.

If the η_i are a factorization of $G[\chi]$ then the $\eta'_i : G'_i[\chi'_i] \rightarrow G'[\phi(\chi)]$, $i \in \{1, \dots, n\}$ are a factorization.

Proof: In the following we will assume that G' is constructed as a quotient graph with an equivalence \approx as described in proposition 2.2.10.

- Let \triangleright be the smallest relation on G, G'_1, \dots, G'_n satisfying:

$$\begin{aligned} \text{for all } v_i \in V_{G_i} : & (\phi_i(v_i), i) \triangleright (\eta_i(v_i), 0) \\ \text{for all } e_i \in E_{G_i} : & (\phi_i(e_i), i) \triangleright (\eta_i(e_i), 0) \end{aligned}$$

Since \approx is the reflexive, symmetric and transitive closure of \triangleright it follows that

$$\begin{aligned} (v'_i, i) \approx (v'_j, j) & \iff \exists k : (v'_i, i) (\triangleright \circ \triangleleft)^k (v'_j, j) \\ (v, 0) \approx (v', 0) & \iff \exists k : (v, 0) (\triangleleft \circ \triangleright)^k (v', 0) \\ (v'_i, i) \approx (v, 0) & \iff \exists k : (v'_i, i) (\triangleright \circ \triangleleft)^k \circ \triangleright (v, 0) \end{aligned}$$

- We will now show that $\eta'_i : G'_i[\chi'_i] \rightarrowtail G'[\phi(\chi)]$ satisfies the four conditions of definition 2.1.6:

(2.2) Let $\eta'_i(v'_i) = \eta'_j(v'_j)$. This implies that there exists a natural number k with $(v'_i, i) (\triangleright \circ \triangleleft)^k (v'_j, j)$. We proceed by induction on k :

$k = 0$: it follows immediately that $i = j$, $v'_i = v'_j$.

$k \rightarrow k + 1$: Let

$$\begin{aligned} (v'_i, i) (\triangleright \circ \triangleleft)^{k-1} (v'_l, l) &= (\phi_l(v_l), l) \\ \triangleright & (\eta_l(v_l), 0) = (\eta_j(v_j), j) \\ \triangleleft & (\phi_j(v_j), j) = (v'_j, j) \end{aligned}$$

With the induction hypothesis it follows that

$$i = l, v'_i = v'_l \text{ (Case A) or } v'_i \in \text{Set}(\chi'_i), v'_l \in \text{Set}(\chi'_l) \text{ (Case B)}$$

And since (η_i) is a factorization it follows that

$$l = j, v_l = v_j \text{ (Case C) or } v_l \in \text{Set}(\chi_l), v_j \in \text{Set}(\chi_j) \text{ (Case D)}$$

This leads to the following four cases:

$$\mathbf{A+C} \Rightarrow i = j, v'_i = v'_l = \phi_l(v_l) = \phi_j(v_j) = v'_j$$

$$\mathbf{A+D} \Rightarrow v'_i = \phi_i(v_i) = \phi_k l(v_l) = v'_l \in \text{Set}(\chi'_l) = \text{Set}(\chi'_i) \text{ and } v'_j = \phi_j(v_j) \in \text{Set}(\chi'_j)$$

$$\mathbf{B+C} \Rightarrow v'_i \in \text{Set}(\chi'_i), v'_j = \phi_j(v_j) = \phi_l(v_l) = v'_l \in \text{Set}(\chi'_l) = \text{Set}(\chi'_j)$$

$$\mathbf{B+D} \Rightarrow v'_i \in \text{Set}(\chi'_i), v'_l = \phi_l(v_l) \in \text{Set}(\chi'_l), v'_j = \phi(v_j) \in \text{Set}(\chi'_j)$$

(2.1) The proof for (2.1) is quite similar to the proof of (2.2) and there is only the case A+C to consider.

(2.3) Let $\eta'_i(v'_i) \in \text{Set}(\phi(\chi))$. It follows that there exists a $v \in \text{Set}(\chi)$ such that $\eta'_i(v'_i) = \phi(v)$. Since η'_i and ϕ are projections into the quotient graph, this implies that there exists a natural number k such that $(v'_i, i) (\triangleright \circ \triangleleft)^k \circ \triangleright (v, 0)$.

We will show that $v'_i \in \text{Set}(\chi'_i)$ by induction on k :

$k = 0$: That is $(v, 0) = (\eta_i(v_i), 0) \triangleleft (\phi_i(v_i), i) = (v'_i, i)$. Since η_i is an embedding and $v \in \text{Set}(\chi)$ it follows that $v_i \in \text{Set}(\chi_i)$ and thus $v'_i = \phi_i(v_i) \in \text{Set}(\chi'_i)$.

$k \rightarrow k + 1$: Let

$$\begin{aligned} (v, 0) \quad (\triangleright \circ \triangleleft)^{k-1} \circ \triangleright \quad (v'_i, i) &= (\phi_i(v_i), i) \\ &\triangleright \quad (\eta_i(v_i), 0) = (\eta_j(v_j), 0) \\ &\triangleleft \quad (\phi_j(v_j), j) = (v'_j, j) \end{aligned}$$

The induction hypothesis implies that $v'_i \in \text{Set}(\chi'_i)$.

We will now assume that $v'_j \notin \text{Set}(\chi'_j)$. It follows that $v_j \notin \text{Set}(\chi_j)$ and since $\eta_j(v_j) = \eta_i(v_i)$ and (η_i) is a factorization we conclude that $i = j, v_i = v_j$. Thus $v'_j = \phi_j(v_j) = \phi_i(v_i) = v'_i \in \text{Set}(\chi'_i) = \text{Set}(\chi'_j)$ which is a contradiction.

(2.4) $E_{G'} = \bigcup_{i=1}^n \eta'_i(E_{G'_i}) \cup \phi(E_G)$. We have to show that $\phi(E_G) \subseteq \bigcup_{i=1}^n \eta'_i(E_{G'_i})$.

Since the η_i form a factorization it follows that $E_G = \bigcup_{i=1}^n \eta_i(E_{G_i})$ and thus

$$\phi(E_G) = \bigcup_{i=1}^n \phi(\eta_i(E_{G_i})) = \bigcup_{i=1}^n \eta'_i(\phi_i(E_{G_i})) \subseteq \bigcup_{i=1}^n \eta'_i(E_{G'_i})$$

□

Proposition 2.2.12 (Combination of Co-Limits) Let $\eta_i : H_i \rightarrowtail H$, $\phi_i : H_i \rightarrow H'_i$, $\eta'_i : H'_i \rightarrowtail H'$ and $\phi : H \rightarrow H'$ with

$$(\phi, \eta'_1, \dots, \eta'_n) = \lim_{i=1}^n (\phi_i, \eta_i)$$

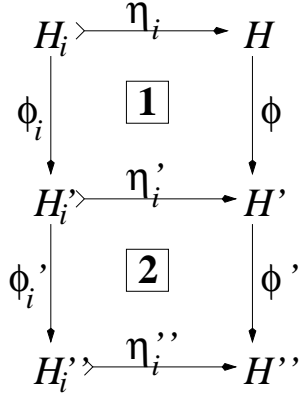
(A) $\boxed{1}, \boxed{2}$ co-limits $\Rightarrow \boxed{1} + \boxed{2}$ co-limit:

Let $\phi'_i : H'_i \rightarrow H''_i$ and let

$$(\phi', \eta''_1, \dots, \eta''_n) := \lim_{i=1}^n (\phi'_i, \eta'_i)$$

Then

$$(\phi' \circ \phi, \eta''_1, \dots, \eta''_n) \cong \lim_{i=1}^n (\phi'_i \circ \phi_i, \eta_i)$$



(B) $\boxed{1}, \boxed{1} + \boxed{2}$ co-limits $\Rightarrow \boxed{2}$ co-limit:

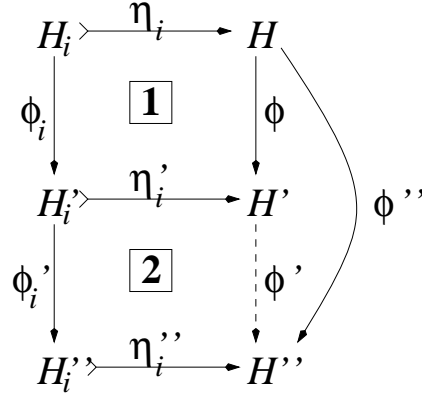
Let $\phi'_i : H'_i \rightarrow H''_i$ and let

$$(\phi'', \eta''_1, \dots, \eta''_n) := \lim_{i=1}^n (\phi'_i \circ \phi_i, \eta_i)$$

with $\phi'' : H \rightarrow H''$, $\eta''_i : H''_i \rightarrowtail H''$.

Then there exists a strong morphism $\phi' : H' \rightarrow H''$ such that $\phi'' = \phi' \circ \phi$ and

$$(\phi', \eta''_1, \dots, \eta''_n) \cong \lim_{i=1}^n (\phi'_i, \eta'_i)$$



Now let $\eta_{ij} : H_{ij} \rightarrowtail H_i$, $\phi_{ij} : H_{ij} \rightarrow H'_{ij}$, $\eta'_{ij} : H'_{ij} \rightarrowtail H'_i$ and $\phi_i : H_i \rightarrow H'_i$ with $(\phi_i, \eta'_{i1}, \dots, \eta'_{in_i}) = \lim_{j=1}^{n_i} (\phi_{ij}, \eta_{ij})$ ($i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}$). (Note that in this case $\boxed{1}$ consists of m co-limits.)

(C) $\boxed{1}, \boxed{2}$ co-limits $\Rightarrow \boxed{1} + \boxed{2}$ co-limit:

Let $\eta_i : H_i \rightarrowtail H$, $i \in \{1, \dots, m\}$ and let

$$(\phi, \eta'_1, \dots, \eta'_m) := \lim_{i=1}^m (\phi_i, \eta_i)$$

Then

$$(\phi, \eta'_i \circ \eta'_{i1}, \dots, \eta'_i \circ \eta'_{in_i}) \cong \lim_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (\phi_{ij}, \eta_{ij} \circ \eta_{ij})$$

$$\begin{array}{ccccc}
H_{ij} & \xrightarrow{\eta_{ij}} & H_i & \xrightarrow{\eta_i} & H \\
\downarrow \phi_{ij} & \boxed{1} & \downarrow \phi_i & \boxed{2} & \downarrow \phi \\
H'_{ij} & \xrightarrow{\eta'_{ij}} & H'_i & \xrightarrow{\eta'_i} & H'
\end{array}$$

(D) $\boxed{1}, \boxed{1} + \boxed{2}$ co-limits \Rightarrow $\boxed{2}$ co-limit:

Let $\eta_i : H_i \rightarrow H$ and let

$$(\phi, \eta_{11}, \dots, \eta_{mn_m}) := \lim_{i \in \{1, \dots, m\}, 1 \leq j \leq n_i} (\phi_{ij}, \eta_i \circ \eta_{ij})$$

with $\eta_{ij} : H'_{ij} \rightarrow H'$ and $\phi : H \rightarrow H'$. Then there exist embeddings $\eta'_i : H'_i \rightarrow H'$ such that $\eta_{ij} = \eta'_i \circ \eta'_{ij}$ and

$$(\phi, \eta'_1, \dots, \eta'_n) := \lim_{i=1}^m (\phi_i, \eta_i)$$

$$\begin{array}{ccccc}
H_{ij} & \xrightarrow{\eta_{ij}} & H_i & \xrightarrow{\eta_i} & H \\
\downarrow \phi_{ij} & \boxed{1} & \downarrow \phi_i & \boxed{2} & \downarrow \phi \\
H'_{ij} & \xrightarrow{\eta'_{ij}} & H'_i & \xrightarrow{\eta'_i} & H'
\end{array}$$

ζ_{ij}

Proof: Let $H = G[\chi]$, $H' = G'[\chi']$, $H_i = G_i[\chi_i]$ and so on. We will now give proofs for the four cases above.

(A) We assume that there are morphisms $\psi : G \rightarrow \hat{G}$, $\psi_i : G'_i \rightarrow \hat{G}$ such that $\psi \circ \eta_i = \psi_i \circ (\phi'_i \circ \phi_i) = (\psi_i \circ \phi'_i) \circ \phi_i$.

It follows with co-limit $\boxed{1}$ that there exists a unique morphism $\psi' : G' \rightarrow \hat{G}$ such that $\psi = \psi' \circ \phi$, $\psi' \circ \eta'_i = \psi_i \circ \phi'_i$.

And it follows with co-limit $\boxed{2}$ that there exists a unique morphism $\psi'' : G'' \rightarrow \hat{G}$ such that $\psi' = \psi'' \circ \phi'$, $\psi_i = \psi'' \circ \eta''_i$ which implies that $\psi'' \circ \phi' \circ \phi = \psi' \circ \phi = \psi$.

It is left to show that ψ'' is in fact unique: let $\bar{\psi} : G'' \rightarrow \hat{G}$ such that $\bar{\psi} \circ \phi' \circ \phi = \psi$, $\bar{\psi} \circ \eta''_i = \psi_i$. This implies that $\psi = (\bar{\psi} \circ \phi') \circ \phi$, $(\bar{\psi} \circ \phi') \circ \eta'_i = \bar{\psi} \circ \eta''_i \circ \phi'_i = \psi_i \circ \phi'_i$.

$$\begin{array}{ccccc}
G_i & \xrightarrow{\eta_i} & G & & \\
\downarrow \phi_i & \boxed{1} & \downarrow \phi & & \downarrow \psi \\
G'_i & \xrightarrow{\eta'_i} & G' & & \\
\downarrow \phi'_i & \boxed{2} & \downarrow \phi' & & \downarrow \psi' \\
G''_i & \xrightarrow{\eta''_i} & G'' & & \\
& & \downarrow \psi'' & & \downarrow \psi \\
& & \hat{G} & & \hat{G}
\end{array}$$

ψ_i

Since ψ' is unique (co-limit $\boxed{1}$), it follows that $\bar{\psi} \circ \phi' = \psi'$. And since furthermore $\bar{\psi} \circ \eta'_i = \psi_i$ and ψ'' is unique (co-limit $\boxed{2}$), it follows that $\bar{\psi} = \psi''$. That is ψ'' is unique.

(B) We define $(\bar{\phi}, \bar{\eta}_i) := \lim_{i=1}^n (\phi'_i, \eta'_i)$ where $\bar{\phi} : G' \rightarrow \bar{G}$, $\bar{\eta}_i : G'_i \rightarrow \bar{G}$.

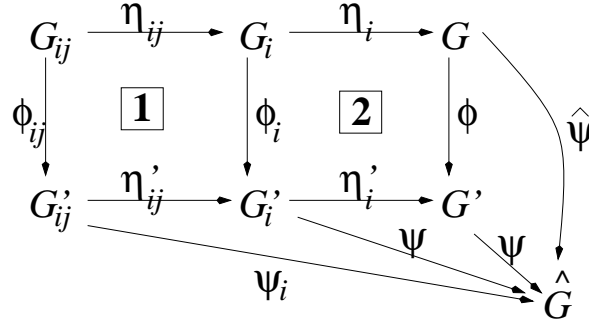
It follows with the proof above that

$$(\bar{\phi} \circ \phi, \bar{\eta}_i) \cong \lim_{i=1}^n (\phi'_i \circ \phi_i, \eta'_i) = (\phi'', \eta''_i)$$

This implies that there is an isomorphism $\psi : \bar{G} \rightarrow G''$ such that $\psi \circ \bar{\phi} \circ \phi = \phi''$, $\psi \circ \bar{\eta}_i = \eta''_i$.

We define $\phi' := \psi \circ \bar{\phi} : G' \rightarrow G''$ and $(\phi', \eta''_i) \cong \lim_{i=1}^n (\phi'_i, \eta'_i)$.

(C) We assume that there are morphisms $\hat{\psi} : G \rightarrow \hat{G}$, $\psi_{ij} : G'_{ij} \rightarrow \hat{G}$ such that $\hat{\psi} \circ (\eta_i \circ \eta_{ij}) = \psi_{ij} \circ \phi_{ij}$. This implies that $(\hat{\psi} \circ \eta_i) \circ \eta_{ij} = \psi_{ij} \circ \phi_{ij}$.



It follows with co-limit $\boxed{1}$ that there exist unique morphisms $\psi_i : G'_i \rightarrow \hat{G}$ such that $\psi_i \circ \phi_i = \hat{\psi} \circ \eta_i$, $\psi_i \circ \eta'_{ij} = \psi_{ij}$.

And it follows with co-limit $\boxed{2}$ that there exists a unique morphism $\psi : G' \rightarrow \hat{G}$ such that $\psi \circ \phi = \hat{\psi}$, $\psi \circ \eta'_i = \psi_i$ which implies that $\psi \circ \eta'_i \circ \eta'_{ij} = \psi_i \circ \eta'_{ij} = \psi_{ij}$.

It is left to show that ψ is unique: let $\bar{\psi} : G' \rightarrow \hat{G}$ be a morphism such that $\bar{\psi} \circ \eta'_i \circ \eta'_{ij} = \psi_{ij}$, $\bar{\psi} \circ \phi = \hat{\psi}$.

It follows that $(\bar{\psi} \circ \eta'_i) \circ \eta'_{ij} = \psi_{ij}$, $(\bar{\psi} \circ \eta'_i) \circ \phi_i = \bar{\psi} \circ (\phi \circ \eta_i) = \hat{\psi} \circ \eta_i$.

Since the ψ_i are unique (co-limit $\boxed{1}$) it follows that $\bar{\psi} \circ \eta'_i = \psi_i$. And since furthermore $\bar{\psi} \circ \phi = \hat{\psi}$ and ψ is unique (co-limit $\boxed{2}$) it follows that $\bar{\psi} = \psi$.

(D) We define $(\bar{\phi}, \bar{\eta}_i) := \lim_{i=1}^n (\phi_i, \eta_i)$ where $\bar{\phi} : G \rightarrow \bar{G}$, $\bar{\eta}_i : G \rightarrow \bar{G}$.

It follows with the proof above that

$$(\bar{\phi}, \bar{\eta}_i \circ \eta'_{ij}) \cong \lim_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (\phi_{ij}, \eta_i \circ \eta_{ij}) = (\phi, \eta_{ij})$$

This implies that there is an isomorphism $\psi : \bar{G} \rightarrow G'$ such that $\psi \circ \bar{\phi} = \phi$, $\psi \circ \bar{\eta}_{ij} \circ \eta'_{ij} = \eta_{ij}$.

We define $\eta'_i := \psi \circ \bar{\eta}_i$ and $(\phi, \eta'_i) \cong \lim_{i=1}^n (\phi_i, \eta_i)$.

□

Proposition 2.2.18 (Co-Limit \leftrightarrow Factorization)

Let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ be a discrete factorization and let H_1, \dots, H_n be hypergraphs with $m_i := \text{card}(H_i)$. We define:

$$(\phi, \eta_1, \dots, \eta_n) := \lim_{i=1}^n (H_i, \zeta_i)$$

Then $(\eta_i)_{i \in \{1, \dots, n\}}$ with $\eta_i : H_i \rightarrow H$ is a factorization of H .

If $\eta_i : H_i \rightarrow H$, $i \in \{1, \dots, n\}$ is a factorization of H it follows that there exists a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$ and a strong morphism $\phi : D \rightarrow H$ such that

$$(\phi, \eta_1, \dots, \eta_n) \cong \lim_{i=1}^n (H_i, \zeta_i)$$

If the χ_{H_i} are duplicate-free, the discrete factorization $(\zeta_i)_{i \in \{1, \dots, n\}}$ is unique up to isomorphism.

Proof:

- If $(\phi, \eta_1, \dots, \eta_n) := \lim_{i=1}^n (H_i, \zeta_i)$ then (η_i) is a factorization according to proposition 2.2.11.
- Let $\eta_i : H_i \rightarrow H$ be a factorization, we will now construct the corresponding discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$ where $m_i := H_i$:

Let D be a discrete hypergraph with

$$V_D := V_H \setminus \bigcup_{i=1}^n \eta_i(V_{H_i} \setminus \text{EXT}_{H_i}) \text{ and } \chi_D := \chi_H$$

We define

$$\zeta_i(\chi_{\mathbf{m}_i}) := \eta_i(\chi_{H_i})$$

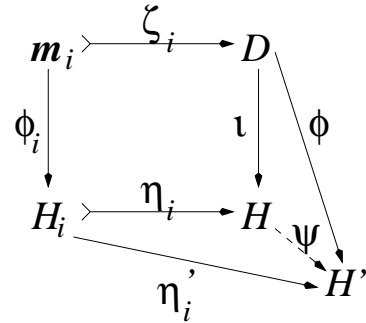
Since (η_i) is a factorization it follows that $\chi_D, \eta_i(\chi_{H_i}) \in V_D^*$.

•

Let $\iota : D \rightarrow H$ (where $\iota(v) := v$ for $v \in V_D$) be the canonical morphism of D into H .

We now show that $(\iota, \eta_1, \dots, \eta_n) \cong \lim_{i=1}^n (H_i, \zeta_i)$.

Let $\eta'_i : H_i \rightarrow H'$, $\phi : D \rightarrow H'$ be morphisms with $\eta'_i \circ \phi_i = \phi' \circ \zeta_i$.



We define a strong morphism $\psi : H \rightarrow H'$ with

$$v \in V_H, \psi(v) := \begin{cases} \phi(v) & \text{if } v \in V_D \\ \eta'_i(v_i) & \text{if } \eta_i(v_i) = v \end{cases}$$

$$e \in E_H, \psi(e) := \eta'_i(e_i) \text{ if } \eta_i(e_i) = e$$

ψ clearly satisfies $\psi \circ \iota = \phi$ and $\psi \circ \eta_i = \eta'_i$. We now show that

ψ is well-defined: there may be a conflict in the definition of $\psi(v)$ if $\eta_i(v_i) = v \in V_D$. In this case it follows with the definition of V_D that $v_i \in EXT_{H_i}$. Let $w_i \in V_{\mathbf{m}_i}$ such that $\phi_i(w_i) = v_i$. It follows that

$$\begin{aligned}\eta'_i(v_i) &= \eta'_i(\phi_i(w_i)) = \phi'(\zeta_i(w_i)) = \phi'(\iota(\zeta_i(w_i))) \\ &= \phi'(\eta_i(\zeta_i(w_i))) = \phi(v)\end{aligned}$$

ψ is unique: Let $\bar{\psi} : H \rightarrow H'$ be another morphism with $\bar{\psi} \circ \iota = \phi$ and $\bar{\psi} \circ \eta_i = \eta'_i$ and let $v \in V_H$.

If $v \in V_D$ then $\bar{\psi}(v) = \bar{\psi}(\iota(v)) = \phi(v) = \psi(\iota(v)) = \psi(v)$. And if $v = \eta_i(v_i)$ then $\bar{\psi}(v) = \bar{\psi}(\eta_i(v_i)) = \eta'_i(v_i) = \psi(\eta_i(v_i)) = \psi(v)$. An analogous argument can be applied to the edge set.

Therefore $\iota, \eta_1, \dots, \eta_n$ is the co-limit.

- Let $\eta_i : H_i \rightarrow H$ be a factorization of H where the χ_{H_i} are without duplicates.

Let $\zeta_i : \mathbf{m}_i \rightarrow D$ be a discrete factorization constructed as described above, and let $\zeta'_i : \mathbf{m}_i \rightarrow D$ be another discrete factorization such that there exists a strong morphisms $\phi' : D' \rightarrow H'$ satisfying

$$(\phi', \eta_1, \dots, \eta_n) \cong \lim_{i=1}^n (H_i, \zeta'_i)$$

We show that $(\zeta'_i) \cong (\zeta_i)$.

First we prove that $V_D = \phi'(V_{D'})$ and that $\phi'|^D : D' \rightarrow D$ (the restriction of the range of ϕ' to D) is an isomorphism:

- Let \approx be the equivalence for the construction of $\bigotimes_{i=1}^n (H_i, \zeta'_i)$ defined in proposition 2.2.10. Because of the special form of the morphism ϕ_i ($\phi_i : \mathbf{m}_i \rightarrow H_i$ and χ_{H_i} is without duplicates) it follows that

$$\begin{aligned}(v, 0) \approx (v_i, i) &\iff \exists w_i \in V_{\mathbf{m}_i} : \zeta'_i(w_i) = v_i, \phi_i(w_i) = v \\ (v, 0) \approx (v', 0) &\iff v = v'\end{aligned}$$

- Since $\phi'(v) := [(v, 0)]_{\approx}$ it follows immediately that ϕ'_V is injective.
- $\phi'(V_{D'}) \subseteq V'_D$: We assume that $\phi'(V_{D'}) \not\subseteq V_D$, i.e. that there exists a $v_i \in V_{H_i} \setminus EXT_{H_i}$ such that $\phi'(v) = \eta_i(v_i)$. This implies that $(v, 0) \approx (v_i, i)$. It follows that there exists a $w_i \in V_{\mathbf{m}_i}$ such that $v_i = \zeta'_i(w_i) \in EXT_{H_i}$ which is a contradiction.
- $\phi'(V_{D'}) \supseteq V_D$: Let $v \in V_D \subseteq V_H$, i.e. either $v' \in \phi'(V_{D'})$ or $v \in \eta_i(EXT_{H_i})$.

In the second case it follows that there is a $v_i \in EXT_{H_i}$ such that $v = \eta_i(v_i)$ because of the definition of D' and since (η_i) is a factorization. Thus there exists a $w_i \in V_{\mathbf{m}_i}$ such that $v_i = \phi_i(w_i)$ and therefore

$$v = \eta_i(v_i) = \eta_i(\phi_i(w_i)) = \phi'(\zeta'_i(w_i)) \in \phi'(V_{D'})$$

$$- \phi'(\chi_{D'}) = \chi_H = \chi_D$$

It follows that $(\phi' \circ \zeta'_i)(\chi_{\mathbf{m}_i}) = (\eta_i \circ \phi_i)(\chi_{\mathbf{m}_i}) = \eta_i(\chi_{H_i}) = \zeta_i(\chi_{\mathbf{m}_i})$ because of the definition of the ζ_i .

Since $\phi' \circ \zeta'_i = \zeta_i$ and $\phi'|^D : D' \rightarrow D$ is an isomorphism it follows that $(\zeta_i) \cong (\zeta'_i)$.

□

Proposition 2.2.21 (Co-Limit \leftrightarrow Context) *Let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ be a discrete factorization and let*

$$C\langle x_1, \dots, x_n \rangle := \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

For all hypergraphs H_1, \dots, H_n with $m_i = \text{card}(H_i)$ it follows that

$$C\langle H_1, \dots, H_n \rangle \cong \bigotimes_{i=1}^n (H_i, \zeta_i) \quad (2.9)$$

Let $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ be contexts with holes of cardinality m_1, \dots, m_n . If both satisfy (2.9) for all H_1, \dots, H_n with $m_i = \text{card}(H_i)$ it follows that

$$C\langle x_1, \dots, x_n \rangle \cong C'\langle x_1, \dots, x_n \rangle$$

And for all contexts $C\langle x_1, \dots, x_n \rangle$ with holes of cardinality m_1, \dots, m_n there is a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ such that

$$C\langle x_1, \dots, x_n \rangle \cong \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

Proof:

- We first define a transformation Θ that maps every n -ary context $C\langle x_1, \dots, x_n \rangle$ with holes of cardinality m_1, \dots, m_n onto a discrete factorization $\zeta'_i : \mathbf{m}_i \rightarrow D$.

Let D be a discrete hypergraph with $V_D := V_C$ and $\chi_D := \chi_C$. Let $e_i \in E_C$ be the edge labelled x_i . We define a factorization $(\zeta'_i) := \Theta(C\langle x_1, \dots, x_n \rangle)$ where $\eta_i(\chi_{\mathbf{m}_i}) := s_C(e_i)$.

According to definition 2.1.10 $C\langle H_1, \dots, H_n \rangle$ is defined as $DH_1 \dots H_n / \approx$ where \approx is the smallest equivalence such that

$$(\lfloor \chi_{H_i} \rfloor_l, i) \approx (\lfloor s_C(e_i) \rfloor_l, 0)$$

where $l_H(e_i) = x_i$. This is equivalent to

$$(\lfloor \phi_i(\chi_{\mathbf{m}_i}) \rfloor_l, i) \approx (\lfloor \zeta'_i(\chi_{\mathbf{m}_i}) \rfloor_l, 0)$$

which corresponds to the equivalence generated by

$$(\phi_i(v_i), i) \approx (\zeta'_i(v_i), 0)$$

for every $v_i \in V_{\mathbf{m}_i}$.

Since this is the equivalence used in the proof of proposition 2.2.10 it follows that

$$C\langle H_1, \dots, H_n \rangle \cong \bigotimes_{i=1}^n (H_i, \zeta'_i)$$

- Now let $\zeta_i : \mathbf{m}_i \rightarrow D$, $i \in \{1, \dots, n\}$ be a given discrete factorization and let

$$C\langle x_1, \dots, x_n \rangle := \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

We define $(\zeta'_i) := \Theta(C\langle x_1, \dots, x_n \rangle)$ and it follows that

$$\bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i) \cong C\langle x_1, \dots, x_n \rangle \cong \bigotimes_{i=1}^n (H_i, \zeta'_i)$$

It follows with proposition 2.2.20 that $(\zeta_i) \cong (\zeta'_i)$ and therefore

$$C\langle H_1, \dots, H_n \rangle \cong \bigotimes_{i=1}^n (H_i, \zeta_i)$$

for hypergraphs H_1, \dots, H_n with $\text{card}(H_i) = \text{sort}(x_i) = m_i$.

- We assume that $C\langle x_1, \dots, x_n \rangle, C'\langle x_1, \dots, x_n \rangle$ are contexts with holes of cardinality m_1, \dots, m_n . Both satisfy (2.9) for all H_1, \dots, H_n with $m_i = \text{card}(H_i)$. Then

$$\begin{aligned} C\langle x_1, \dots, x_n \rangle &\cong C\langle \text{var}_{m_1}(x_1), \dots, \text{var}_{m_n}(x_n) \rangle \cong \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i) \\ &\cong C'\langle \text{var}_{m_1}(x_1), \dots, \text{var}_{m_n}(x_n) \rangle \cong C'\langle x_1, \dots, x_n \rangle \end{aligned}$$

- For every context $C\langle x_1, \dots, x_n \rangle$, $(\zeta_i) := \Theta(C\langle x_1, \dots, x_n \rangle)$ is a discrete factorization satisfying:

$$C\langle x_1, \dots, x_n \rangle \cong \bigotimes_{i=1}^n (\text{var}_{m_i}(x_i), \zeta_i)$$

□

Proposition 2.4.3 (Construction of Name-based Graph Terms) *Let H be a hypergraph, where χ_H contains no duplicates, i.e. $\chi_H \in (V_H)_{df}^*$. Then there exists a closed name-based graph term $h[t]$ such that $H \cong \text{val}_n(h[t])$.*

Proof: According to proposition 2.2.20 there is a discrete factorization $\zeta_i : \mathbf{m}_i \rightarrow D, i \in \{1, \dots, n\}$ such that

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

where $z_1, \dots, z_n \in Z, l_1, \dots, l_n \in L$.

Let v_1, \dots, v_r be the string of all nodes of D which are not contained in $\bigcup_{i \in \{1, \dots, n\}} \zeta_i(V_{\mathbf{m}_i})$. Furthermore let $\mu : V_D \rightarrow N$ be any injective mapping. We define:

$$\begin{aligned} h[t] &:= ((\nu \mu(V_D \setminus EXT_D)) \\ &\quad ((z_1, l_1)[\mu(\zeta_1(\chi_{\mathbf{m}_1}))] \mid \dots \mid (z_n, l_n)[\mu(\zeta_n(\chi_{\mathbf{m}_n}))] \\ &\quad \mid [\mu(v_1)] \mid \dots \mid [\mu(v_r)])) \\ &\quad [\mu(\chi_D)] \end{aligned}$$

Let $m := \text{card}(D)$ and we assume that $V_D = \{w_1, \dots, w_k\}$ where $\chi_D = w_1 \dots w_m$.

We can show with proposition 2.2.13 and with definition 2.4.2 that

$$\text{val}_a(((\nu \mu(V_D \setminus EXT_D)h))[\mu(w_1 \dots w_m)]) \cong \otimes(\text{val}_a(h[\mu(w_1 \dots w_k)]), \zeta)$$

where $\zeta : \mathbf{k} \rightarrow D, \zeta(\lfloor \chi_{\mathbf{k}} \rfloor_i) := w_i, t' := \mu(w_1 \dots w_k)$.

We define

$$\mu_i := \mu(\zeta_i(\chi_{\mathbf{m}_i})) \quad t'_i := t' \setminus \left(\bigcup_{j=1, j \neq i}^n \text{fn}(h_j) \setminus \text{fn}(h_i) \right)$$

It follows with definitions 2.4.2 and 2.4.4 that

$$\text{val}_n(h[t]) \cong \otimes \left(\left(\bigotimes_{i=1}^n (\otimes(\text{val}_n((z_i, l_i)[\mu_i]), \zeta_{\mu_i \rightarrow t'_i}), \zeta_{t'_i \rightarrow t'}) \right) \otimes \left(\bigotimes_{i=1}^r (\mathbf{1}, \zeta_{\mu(v_i) \rightarrow t'}) \right), \zeta \right)$$

Corollary 2.2.14 and lemma 2.2.16 then imply that:

$$\text{val}_n(h[t]) \cong \bigotimes_{i=1}^n (\text{val}_n((z_i, l_i)[\mu_i]), \zeta \circ \zeta_{\mu_i \rightarrow t'}) \cong \bigotimes_{i=1}^n ((z_i)_{m_i}(l_i), \zeta \circ \zeta_{\mu_i \rightarrow t'})$$

It is left to show that $\zeta_i = \zeta \circ \zeta_{\mu_i \rightarrow t'}$. We assume that $\zeta_i(\lfloor \chi_{\mathbf{m}_i} \rfloor_l) = w_j$. This implies that $\lfloor \mu_i \rfloor_l = \mu(w_j) = \lfloor t' \rfloor_j$. Thus

$$\zeta(\zeta_{\mu_i \rightarrow t'}(\lfloor \chi_{\mathbf{m}_i} \rfloor_l)) = \zeta(\lfloor \chi_{\mathbf{k}} \rfloor_j) = w_j = \zeta_i(\lfloor \chi_{\mathbf{m}_i} \rfloor_l)$$

□

Proposition 2.4.6 (Normal Form of Name-based Notation)

A name-based graph term is in normal form if it has the form

$$\begin{aligned} &((\nu b_1) \dots (\nu b_n)((z_1, l_1)[a_{11} \dots a_{1n_1}] \mid \dots \mid (z_m, l_m)[a_{m1} \dots a_{mn_m}] \mid \\ &\quad \lfloor c_1 \rfloor \mid \dots \mid \lfloor c_k \rfloor)) [d_1 \dots d_l] \end{aligned} \quad (2.34)$$

and it follows that

$$A \cap C = \emptyset, \quad B \cap D = \emptyset, \quad A \cup C = B \cup D, \quad |B| = n, \quad |C| = k$$

where

$$\begin{aligned} A &:= \{a_{ij} \mid i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}\} \\ B &:= \{b_i \mid i \in \{1, \dots, n\}\} \\ C &:= \{c_i \mid i \in \{1, \dots, k\}\} \\ D &:= \{d_i \mid i \in \{1, \dots, l\}\} \end{aligned}$$

For every graph term $h[t]$ there is a graph term $h'[t']$ in normal form such that $h[t] \simeq_{=} h'[t']$.

Proof: We will first show that

$$(\nu a)h \simeq_R h \text{ if } a \notin fn(h) \quad (\text{A.1})$$

This holds because $(\nu a)h \stackrel{(2.26)}{\simeq_R} (\nu a)(h|0) \stackrel{(2.29)}{\simeq_R} h|(\nu a)0 \stackrel{(2.27)}{\simeq_R} h|0 \stackrel{(2.26)}{\simeq_R} h$

Now let $h[t]$ be an arbitrary name-based graph term. We will transform it into a term in normal form in the following steps:

- First move all hiding operators of the form (νa) to the very left with (2.29). It might be necessary to change a into some other name with (2.30) first.
- This leads to an expression of the form (2.34). We will now ensure that the rest of the conditions is also satisfied.
- $B \cap D = \emptyset$ is true in any case since D is the set of free names in h and all names in B are hidden.
- Then eliminate all $b_i \notin A \cup C$ with (2.28) and (A.1). It follows that $A \cup C = B \cup D$.
- Now eliminate all duplicates in the b_i with (2.28) and $(\nu a)(\nu a)h \simeq_R (\nu a)h$ (because of (A.1)). This leads to $|B| = n$.
- Then eliminate all c_r with $c_r = a_{ij}$ with (2.32) (and of course (2.24), (2.25), i.e. commutativity and associativity of $|$). This implies $A \cap C = \emptyset$.
- Now eliminate all duplicates in the c_i with (2.31) (and (2.24), (2.25)) and it follows that $|C| = k$.

□

Proposition 2.4.5 (Equations for Name-based Graph Terms) *The following equation schemes generate the equivalence on name-based graph terms. Two closed terms $h[t], h'[t']$ are equivalent wrt. to the following equations if and only if $val_n(h[t]) \cong_R val_n(h'[t'])$.*

Let h, h_1, h_2, h_3 be name-based graph terms.

$$(z, l)[t] \simeq (z, l')[t] \text{ if } l R l' \quad (2.23)$$

$$h_1|h_2 \simeq_R h_2|h_1 \quad (2.24)$$

$$h_1|(h_2|h_3) \simeq_R (h_1|h_2)|h_3 \quad (2.25)$$

$$h|0 \simeq_R h \quad (2.26)$$

$$(\nu a)0 \simeq_R 0 \quad (2.27)$$

$$(\nu a)(\nu b)h \simeq_R (\nu b)(\nu a)h \quad (2.28)$$

$$((\nu a)h_1)|h_2 \simeq_R (\nu a)(h_1|h_2) \text{ if } a \notin fn(h_2) \quad (2.29)$$

$$(\nu a)h \simeq_R (\nu b)(h[b/x]) \text{ if } b \notin fn(h) \quad (2.30)$$

$$[a]||[a] \simeq_R [a] \quad (2.31)$$

$$(z, l)[a_1 \dots a_n]||[a_i] \simeq_R (z, l)[a_1 \dots a_n] \quad (2.32)$$

and for closed terms:

$$h[s] \simeq_R (h[s'/s])[s'] \text{ if } |s| = |s'|, s' \text{ is duplicate-free} \quad (2.33)$$

Proof:

Correctness: In order to show correctness we have to prove that $h[t] \simeq_R h'[t']$ implies $val_n(h[t]) \simeq_R val_n(h'[t'])$ by induction on the equation rules.

Completeness:

- We will now show that the proposition holds for graph terms in normal form.

Since every graph term can be converted into normal form (proposition 2.4.6) this implies completeness for arbitrary graph terms. In detail: Let $val_n(h[t]) \simeq_R val_n(h'[t'])$ and let $h_n[t_n]$ respectively $h'_n[t'_n]$ be the normal forms of $h[t]$ and $h'[t']$. Since val_n is correct it follows that

$$val_n(h_n[t_n]) \simeq_R val_n(h[t]) \simeq_R val_n(h'[t']) \simeq_R val_n(h'_n[t'_n])$$

Since the proposition holds for terms in normal form it follows that $h_n[t_n] \simeq_R h'_n[t'_n]$ and thus $h[t] \simeq_R h'[t']$.

- We will now assume that $h[t]$ and $h'[t']$ are in normal form (see proposition 2.4.6). That is they have the form

$$\begin{aligned} h[t] &= ((\nu b_1) \dots (\nu b_n)(\\ &\quad (z_1, l_1)[a_{11} \dots a_{1n_1}] | \dots | (z_m, l_m)[a_{m1} \dots a_{mn_m}] | \\ &\quad [c_1] | \dots | [c_k]))[d_1 \dots d_l] \\ h'[t'] &= ((\nu b'_1) \dots (\nu b'_{n'})(\\ &\quad (z'_1, l'_1)[a'_{11} \dots a'_{1n'_1}] | \dots | (l'_{m'}, z'_{m'}[a'_{m'1} \dots a'_{m'n'_m}] | \\ &\quad [c'_1] | \dots | [c'_{k'}]))[d'_1 \dots d'_{l'}] \end{aligned}$$

Since l is the cardinality of $val_n(h[t])$ and l' is the cardinality of $val_n(h'[t'])$ and both are isomorphic it follows immediately that $l = l'$.

- With definition 2.4.2 and corollary 2.2.14 (compare also with the proof of proposition 2.4.7) we can show that

$$\begin{aligned}
val_n(h[t]) &\cong_R \bigotimes_{i=1}^m ((z_i)_{n_i}(l_i), \zeta \circ \zeta_i) \otimes (\mathbf{k}, \zeta \circ \zeta_0) \\
&\cong_R \bigotimes_{i=1}^m ((z_i)_{n_i}(l_i), \zeta \circ \zeta_i) \quad \text{lemma (2.2.16)} \\
val_n(h'[t']) &\cong_R \bigotimes_{i=1}^{m'} ((z'_i)_{n'_i}(l'_i), \zeta' \circ \zeta'_i) \otimes (\mathbf{k}', \zeta' \circ \zeta'_0) \\
&\cong_R \bigotimes_{i=1}^{m'} ((z'_i)_{n'_i}(l'_i), \zeta' \circ \zeta'_i) \quad \text{lemma (2.2.16)}
\end{aligned}$$

where

$$\begin{aligned}
\zeta_i &:= \zeta_{a_{i1} \dots a_{im_i} \rightsquigarrow d_1 \dots d_l b_1 \dots b_k} \\
\zeta'_i &:= \zeta_{a'_{i1} \dots a'_{im'_i} \rightsquigarrow d'_1 \dots d'_{l'} b'_1 \dots b'_k} \\
\zeta_0 &:= \zeta_{c_1 \dots c_k \rightsquigarrow d_1 \dots d_l b_1 \dots b_k} \\
\zeta'_0 &:= \zeta_{c'_1 \dots c'_k \rightsquigarrow d'_1 \dots d'_{l'} b'_1 \dots b'_k}
\end{aligned}$$

ζ is the projection of $\mathbf{l} + \mathbf{n}$ into $\sigma_{1 \dots l}(\mathbf{l} + \mathbf{n}) =: D$ and ζ' is the projection of $\mathbf{l}' + \mathbf{n}'$ into $\sigma_{1 \dots l'}(\mathbf{l}' + \mathbf{n}') =: D'$.

- Since both co-limits above describe factorizations of isomorphic graphs it follows with proposition 2.2.20 that $m = m'$ and there exists a permutation $\alpha : \{1, \dots, m\} \rightarrow \{1, \dots, m'\}$ such that

$$(\zeta \circ \zeta_i)_{i \in \{1, \dots, m\}} \cong (\zeta' \circ \zeta'_{\alpha(i)})_{i \in \{1, \dots, m\}}$$

This implies that there is an isomorphism $\phi : D \rightarrow D'$ such that $\phi \circ \zeta \circ \zeta_i = \zeta' \circ \zeta'_{\alpha(i)}$.

Since ϕ is an isomorphism and $l = l'$ it follows that also $n = n'$.

Furthermore it follows that $(z_i)_{n_i}(l_i) \cong_R (z'_{\alpha(i)})_{n'_{\alpha(i)}}(l'_{\alpha(i)})$ which implies that $z_i = z'_{\alpha(i)}$, $l_i R l'_{\alpha(i)}$ and $n_i = n'_{\alpha(i)}$.

- We define

$$\begin{aligned}
\mu : V_D \rightarrow N \text{ such that } \mu(\zeta(\lfloor \chi_{\mathbf{l}+\mathbf{n}} \rfloor_i)) &:= \begin{cases} d_i & \text{if } i \in \{1, \dots, l\} \\ b_{i-l} & \text{otherwise} \end{cases} \\
\mu' : V_{D'} \rightarrow N \text{ such that } \mu'(\zeta'(\lfloor \chi_{\mathbf{l}'+\mathbf{n}'} \rfloor_i)) &:= \begin{cases} d'_i & \text{if } i \in \{1, \dots, l'\} \\ b'_{i-l'} & \text{otherwise} \end{cases}
\end{aligned}$$

We define $u := \mu' \circ \phi \circ \mu^{-1}$ which is a bijection from $\{d_1, \dots, d_l, b_1, \dots, b_n\}$ into $\{d'_1, \dots, d'_{l'}, b'_1, \dots, b'_{n'}\}$.

- Furthermore let

$$\beta(i) = j \iff \phi(\zeta(\lfloor \chi_{\mathbf{l}+\mathbf{n}} \rfloor_{l+i})) = \zeta'(\lfloor \chi_{\mathbf{l}'+\mathbf{n}'} \rfloor_{l'+j}) \quad (\text{A.2})$$

Since ϕ is an isomorphism a non-external node $\zeta(\lfloor \chi_{1+\mathbf{n}} \rfloor_{l+i})$ of D is mapped onto a non-external node $\zeta'(\lfloor \chi_{l'+\mathbf{n}'} \rfloor_{l'+j})$ of D' . Thus β is well-defined and bijective.

- Now we will show that $u(d_i) = d'_i$ and that $u(b_i) = b'_{\beta(i)}$.

$$\begin{aligned} u(d_i) &= \mu'(\phi(\zeta(\lfloor \chi_{1+\mathbf{n}} \rfloor_i))) = \mu'(\phi(\lfloor \chi_D \rfloor_i)) = \mu'(\lfloor \chi_{D'} \rfloor_i) \\ &= \mu'(\zeta'(\lfloor \chi_{l'+\mathbf{n}'} \rfloor_i)) = d'_i \\ u(b_i) &= \mu'(\phi(\zeta(\lfloor \chi_{l+n} \rfloor_{l+i}))) = \mu'(\zeta'(\lfloor \chi_{l'+n'} \rfloor_{l+\beta(i)})) = b'_{\beta(i)} \end{aligned}$$

- Furthermore we can show that

$$\begin{aligned} a_{ij} = d_r &\iff a'_{\alpha(i)j} = d'_r \\ a_{ij} = b_r &\iff a'_{\alpha(i)j} = b'_{\beta(r)} \end{aligned}$$

We will show this for the first case only. The proofs for the other cases are rather similar.

$$\begin{aligned} a_{ij} = d_r &\iff \zeta_i(\lfloor \chi_{\mathbf{m}_i} \rfloor_j) = \lfloor \chi_{1+\mathbf{n}} \rfloor_r \\ &\xLeftrightarrow{\zeta \text{ inj.}} \zeta(\zeta_i(\lfloor \chi_{\mathbf{m}_i} \rfloor_j)) = \zeta(\lfloor \chi_{1+\mathbf{n}} \rfloor_r) \\ &\xLeftrightarrow{\phi \text{ bij.}} \zeta'(\zeta'_{\alpha(i)}(\lfloor \chi_{\mathbf{m}_{\alpha(i)}} \rfloor_j)) = \zeta'(\lfloor \chi_{1+\mathbf{n}} \rfloor_r) \\ &\xLeftrightarrow{\zeta' \text{ inj.}} \zeta'_{\alpha(i)}(\lfloor \chi_{\mathbf{m}_{\alpha(i)}} \rfloor_j) = \lfloor \chi_{1+\mathbf{n}} \rfloor_r \iff a'_{\alpha(i)j} = d_r \end{aligned}$$

- Now we will show that $u(a_{ij}) = a'_{\alpha(i)j}$.
 a_{ij} is either d_r or b_r for some index r . It follows that

$$u(a_{ij}) = \left\{ \begin{array}{l} u(d_r) = d'_r \\ u(b_r) = b'_{\beta(r)} \end{array} \right\} = a'_{\alpha(i)j} \quad (\text{A.3})$$

It follows that u is a bijection of the set $A := \{a_{ij} \mid i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}\}$ into the set $A' := \{a'_{ij} \mid i \in \{1, \dots, m'\}, j \in \{1, \dots, n'_i\}\}$.

- With definition 2.4.6 it follows that

$$|A| + k = n + l = n' + l' = |A'| + k'$$

And since $|A| = |A'|$ it follows that $k = k'$.

u is a also bijection from $A \cup \{c_1, \dots, c_k\}$ into $A' \cup \{c'_1, \dots, c'_{k'}\}$. And since $u(A) = A'$ it follows that there exists a permutation $\gamma : \{1, \dots, k\} \rightarrow \{1, \dots, k'\}$ with $u(c_i) = c'_{\gamma(i)}$.

- That is we have functions mappings the names of $h[t]$ bijectively into the names of $h'[t']$. With (2.24), (2.25), (2.28) and (2.33) we can show that $h[t] \simeq_R h'[t']$.

□

A.2 Generating Type Systems

A.2.1 A Type System Based on Lattices

Lemma 8.3.7 *Let $E, G[\chi, a] \vdash \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow D$.*

It follows that there is a strong morphism $\psi : D \rightarrow G[\chi]$ such that

$$E, G[\psi(\eta_i(\chi_{\mathbf{m}_i})), a] \vdash H_i$$

Proof: We will first concentrate on the case where all graphs H_i are basic graphs, i.e. either of the form $proc_n(S)$ or $mess_n(H)$. We will proceed by induction of the typing rules and assume that all graphs denoted by the letters B_i or B_{ij} are basic graphs.

(TL-PROC), (TL-MESS) If $E, G[\chi, a] \vdash \bigotimes_{i=1}^n (B_i, \zeta_i)$ follows with typing rules (TL-PROC) or (TL-MESS), we can conclude that $n = 1$, $\zeta_1 : \mathbf{m}_1 \rightarrow \mathbf{m}_1$ and therefore $E, G[\chi, a] \vdash B_1$. Furthermore $\psi : \mathbf{m}_1 \rightarrow G[\chi]$ is the canonical strong morphism.

(TL-CON) We assume that $E, G[\chi, a] \vdash \bigotimes_{i=1}^n (B_i, \varsigma_i) =: H$ with $\varsigma_i : \mathbf{k}_i \rightarrow \hat{D}$.

The typing of H was done with rule (TL-CON) as follows:

$H \cong \bigotimes_{i=1}^m (H_i, \zeta_i)$ with

$$\zeta_i : \mathbf{m}_i \rightarrow D, \phi : D \rightarrow G[\chi], E, G[\phi(\zeta_i(\chi_{\mathbf{m}_i})), a_i] \vdash H_i$$

We assume that $H_i \cong \bigotimes_{j=1}^{n_i} (B_{ij}, \zeta_{ij})$ where the B_{ij} are basic graphs and $\zeta_{ij} : \mathbf{m}_{ij} \rightarrow D_i$. With the induction hypothesis it follows that there are strong morphisms $\psi_i : D_i \rightarrow G[\phi(\zeta_i(\chi_{\mathbf{m}_i}))]$ such that

$$E, G[\psi_i(\zeta_{ij}(\chi_{\mathbf{m}_{ij}})), a] \vdash B_{ij}$$

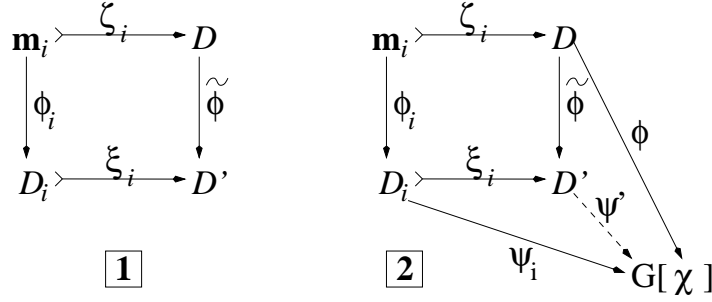
With proposition 2.2.20 it follows that there exists a one-to-one correspondence between the B_i and the B_{ij} , i.e. there are mappings $\alpha_1, \alpha_2 : \mathbb{N} \rightarrow \mathbb{N}$ with

$$\begin{aligned} B_i &\cong B_{\alpha_1(i)\alpha_2(i)} \\ k_i &= m_{\alpha_1(i)\alpha_2(i)} \\ (\varsigma_i)_{i \in \{1, \dots, n\}} &\cong (\xi_{\alpha_1(i)} \circ \zeta_{\alpha_1(i)\alpha_2(i)})_{i \in \{1, \dots, n\}} \end{aligned}$$

where the ξ_i are generated by the co-limit **[1]** in figure A.1 (see also proposition 2.2.13). There exists an isomorphism $\hat{\phi} : \hat{D} \rightarrow D'$ such that $\hat{\phi} \circ \varsigma_i = \xi_{\alpha_1(i)} \circ \zeta_{\alpha_1(i)\alpha_2(i)}$.

Now we regard the situation in figure A.1, **[2]**: Since

$$\phi(\zeta_i(\chi_{\mathbf{m}_i})) = \phi_i(\psi_i(\chi_{\mathbf{m}_i}))$$

Figure A.1: Co-limit constructing the embeddings ξ_i

it follows that there exists a strong morphism $\psi : D' \rightarrow G[\chi]$.

And

$$\begin{aligned} \psi(\hat{\phi}(\varsigma_i(\chi_{\mathbf{k}_i}))) &= \psi(\xi_{\alpha_1(i)}(\zeta_{\alpha_1(i)\alpha_2(i)}(\chi_{\mathbf{m}_{\alpha_1(i)\alpha_2(i)}}))) \\ &= \phi_{\alpha_1(i)}(\zeta_{\alpha_1(i)\alpha_2(i)}(\chi_{\mathbf{m}_{\alpha_1(i)\alpha_2(i)}})) \end{aligned}$$

And it follows that $E, G[(\psi \circ \hat{\phi})(\varsigma_i(\chi_{\mathbf{k}_i})), a] \vdash B_i$.

We will now prove the proposition for decomposition into arbitrary graphs (which are not necessarily basic).

We assume that $E, G[\chi, a] \vdash \bigotimes_{i=1}^m (H_i, \zeta_i) := H$ where $\zeta_i : \mathbf{m}_i \rightarrow D$ and the H_i can be decomposed into basic graphs in the following way: $H_i \cong \bigotimes_{j=1}^{n_i} (B_{ij}, \zeta_{ij})$, i.e.

$$H \cong \bigotimes_{i \in \{1, \dots, m\}, j \in \{1, \dots, n_i\}} (B_{ij}, \xi_i \circ \zeta_{ij})$$

where $\xi_i \circ \zeta_{ij} : \mathbf{m}_{ij} \rightarrow D'$. The ξ_i is computed as described in proposition 2.2.13 (see also figure A.1, **1**).

It follows now that there exists a strong morphism $\psi' : D' \rightarrow G[\chi]$ such that

$$E, G[\psi'(\xi_i(\zeta_{ij}(\chi_{\mathbf{m}_{ij}}))), a] \vdash B_{ij}$$

This implies with (TL-CON) that $E, G[\psi'(\xi_i(\chi_{D_i})), a] \vdash H_i$.

We define $\psi := \psi' \circ \tilde{\phi}$. Then it follows that $\psi(\zeta_i(\chi_{\mathbf{m}_i})) = \psi'(\tilde{\phi}(\zeta_i(\chi_{\mathbf{m}_i}))) = \psi'(\xi_i(\phi_i(\chi_{\mathbf{m}_i}))) = \psi'(\xi_i(\chi_{D_i}))$ and thus

$$E, G[\psi(\zeta_i(\chi_{\mathbf{m}_i})), a] \vdash H_i$$

□

Proposition 8.3.11 (Subject Reduction Property)

Let $E, T \vdash H$ and $H \rightarrow^* H'$. Then $E, T \vdash H'$. Furthermore H contains no bad redexes.

Proof: According to proposition 4.3.5 there are process graphs H_1, H_2 and embeddings ξ_1, ξ_2 such that:

$$\begin{aligned} H &\cong (H_1, \xi_1) \otimes (H_2, \xi_2) \\ H' &\cong (H'_1, \xi_1) \otimes (H_2, \xi_2) \end{aligned}$$

and $H_1 \xrightarrow{(R-REPL)} H'_1$ or $H_1 \xrightarrow{(R-MR)} H'_1$.

Because of proposition 8.3.10 and lemma 8.3.7 it is sufficient to show that $E, G[\chi, a] \vdash H_1$ implies $E, G[\chi, a] \vdash H'_1$.

(R-REPL) In this case we assume that $H_1 = \text{proc}_n(!J)$, $H'_1 = \text{proc}_n(!J) \square J$ and

$$E, G[\chi, a] \vdash \text{proc}_n(!J)$$

By unravelling the typing of $\text{proc}_n(!J)$ we find out that

$$E, G[\chi, a] \vdash J$$

Furthermore there is a strong (F, \leq) -morphism from $A(\text{proc}_n(!J))$ into $G[\chi, a]$. This implies with lemma 8.3.3 that $n = |\chi| = \text{card}(J)$, i.e. replication is defined.

Let $\zeta_1, \zeta_2 : \mathbf{n} \rightarrow \mathbf{n}$ and $\phi : \mathbf{n} \rightarrow G[\chi]$ be the canonical strong morphisms, i.e. $\phi(\zeta_1(\chi_{\mathbf{n}})) = \phi(\zeta_2(\chi_{\mathbf{n}})) = \chi$. It follows with (TL-CON) that

$$E, G[\chi, a \vee a] \vdash (\text{proc}_n(!J), \zeta_1) \otimes (J, \zeta_2) \cong \text{proc}_n(!J) \square J$$

where $a \vee a = a$.

(R-MR) In this case we assume that $H_1 = \text{Red} := \text{Red}_{k,m,n}(\lambda_k x. J_1, J_2)$, $H'_1 = J_1[J_2/x]$ and

$$E, G[\chi, a] \vdash \text{Red}$$

We have

$$\text{Red} \cong (\text{proc}_m(\lambda_k x. J_1), \zeta_1) \otimes (\text{mess}_{n+1}(J_2), \zeta_2)$$

where $\zeta_1 : \mathbf{m} \rightarrow \mathbf{m} + \mathbf{n}$ and $\zeta_2 : \mathbf{n} + \mathbf{1} \rightarrow \mathbf{m} + \mathbf{n}$ with $\zeta_1(\chi_{\mathbf{m}}) = \lfloor \chi_{\mathbf{m}+\mathbf{n}} \rfloor_{1\dots m}$ and $\zeta_2(\chi_{\mathbf{n}+\mathbf{1}}) = \lfloor \chi_{\mathbf{m}+\mathbf{n}} \rfloor_{m+1\dots m+n+k}$.

It follows with lemma 8.3.7 that there are strings $\chi_M, \chi_P \in V_G^*$, $a_1, a_2 \in F(G)$ and $\phi : \mathbf{m} + \mathbf{n} \rightarrow G[\chi]$ such that:

$$\begin{aligned} E, G[\chi_P, a_1] &\vdash \text{proc}_m(\lambda_k x. J_1) \\ E, G[\chi_M, a_2] &\vdash \text{mess}_{n+1}(J_2) \\ \chi_P &:= \phi(\zeta_1(\chi_{\mathbf{m}})) \\ \chi_M &:= \phi(\zeta_2(\chi_{\mathbf{n}+\mathbf{1}})) \\ a_r &= a_1 \vee a_2 \end{aligned}$$

This implies that $\chi = \chi_P \circ \lfloor \chi_M \rfloor_{1\dots n}$, $\lfloor \chi_M \rfloor_{n+1} = \lfloor \chi_P \rfloor_k$.

By further unravelling the typing of the redex we obtain:

$$\frac{\frac{E \setminus x \cup \{x : \eta_x\}, G[\chi_1, a_1] \vdash J_1}{E, G[\lfloor \chi_1 \rfloor_{1\dots m}, a_1] \vdash \lambda_k x. J_1}}{E, G[\lfloor \chi_1 \rfloor_{1\dots m}, a_1] \vdash \text{proc}_m(\lambda_k x. J_1)}$$

where $\lfloor \chi_1 \rfloor_{1\dots m} = \chi_P$.

Furthermore there is a strong morphism $\phi_p : A(\text{proc}_m(\lambda_k x. J_1)) \xrightarrow{F, \leq} G[\chi_p, a_1]$ and there are hyperedges $q \in M_G$, $C \in Co_G$, $p \in P_G$ with $s_G(q) = \lfloor \chi_1 \rfloor_{m+1\dots m+n k}$, $s_G(c) = \eta_x(\chi_{\mathbf{m}_x}) \circ \lfloor \chi_1 \rfloor_k$, $s_G(p) = \lfloor \chi_1 \rfloor_{1\dots m}$.

And it follows that

$$\frac{E, G[\chi_2, a_2] \vdash J_2}{E, G[\chi_M, a_2] \vdash \text{mess}_{n+1}(J_2)}$$

where $\phi_M : A(\text{mess}_{n+1}(J_2)) \xrightarrow{F, \leq} G[\chi_M, a_2]$ is a strong morphism and there exists a $c' \in Co_G$ such that $s_G(c') = \chi_2 \circ \lfloor \chi_M \rfloor_{n+1}$.

We will now show that

$$E \setminus x \cup \{x : \eta_x\}, G[\chi, a] \vdash J_1 \quad (\text{A.4})$$

$$E, G[\eta_x(\chi_{\mathbf{m}_x}), a] \vdash J_2 \quad (\text{A.5})$$

In this case it follows with proposition 8.3.9 that

$$E, G[\chi, a] \vdash J_1[J_2/x]$$

- We will first show that $\chi = \chi_1$. Since it is clear that $\lfloor \chi \rfloor_{1\dots m} = \chi_P = \lfloor \chi_1 \rfloor_{1\dots m}$ it is left to show that $\lfloor \chi \rfloor_{m+1\dots m+n} = \lfloor \chi_1 \rfloor_{m+1\dots m+n}$. Let q' be the one message in $\text{mess}_{n+1}(J_2)$.

$$s_G(\phi_M(q')) = \phi_M(\chi_{\text{mess}_{n+1}(J_2)}) = \chi_M$$

Furthermore $\lfloor s_G(\phi_M(q')) \rfloor_{n+1} = \lfloor \chi_M \rfloor_{n+1} = \lfloor \chi_P \rfloor_k = \lfloor \chi_1 \rfloor_k = \lfloor s_G(q) \rfloor_{n+1}$. This implies that $\text{send}_G(\phi_M(q')) = \text{send}_G(q)$ and therefore $\phi_M(q') = q$ since G is a true graph. And it follows that

$$\begin{aligned} \lfloor \chi \rfloor_{m+1\dots m+n} &= \lfloor \chi_M \rfloor_{1\dots n} = \lfloor s_G(\phi_M(q')) \rfloor_{1\dots n} \\ &= \lfloor s_G(q) \rfloor_{1\dots n} = \lfloor \chi_1 \rfloor_{m+1\dots m+n} \end{aligned}$$

- We will now show that $\eta_x(\chi_{\mathbf{m}_x}) = \chi_2$. This is straightforward since $\chi_2 = \lfloor s_G(c') \rfloor_{1\dots n}$, $\eta_x(\chi_{\mathbf{m}_x}) = \lfloor s_G(c) \rfloor_{1\dots n}$ and $\lfloor s_G(c') \rfloor_{n+1} = \lfloor \chi_M \rfloor_{n+1} = \lfloor \chi_P \rfloor_k = \lfloor \chi_1 \rfloor_k = \lfloor s_G(c) \rfloor_{n+1}$. Since G is a true graph it follows that $c = c'$ and also $\chi_2 = \lfloor s_G(c') \rfloor_{1\dots n} = \lfloor s_G(c) \rfloor_{1\dots n} = \eta_x(\chi_{\mathbf{m}_x})$.
- And we have shown that $\text{card}(J_2) = |\chi_2| = \text{card}(\mathbf{m}_x) = \text{sort}(x)$.
- Furthermore $\text{card}(J_1) = |\chi_1| = |\chi| = m + n$. Since $a = a_1 \vee a_2$ it is obvious that $a_1 \leq a$, $a_2 \leq a$ and thus (A.4) and (A.5) follow with lemma 8.3.4.

□

Proposition 8.3.13 (Folding Type Graphs) *Let $T = G[\chi, a]$ be a type graph. Folding a type graph works as follows: let \approx be the smallest equivalence such that:*

$$\begin{aligned} p_1, p_2 \in P_G, \quad s_G(p_1) = s_G(p_2) &\Rightarrow p_1 \approx p_2 \\ m_1, m_2 \in M_G, \quad \text{send}_G(m_1) = \text{send}_G(m_2) &\Rightarrow m_1 \approx m_2 \\ c_1, c_2 \in Co_G, \quad \text{send}_G(c_1) = \text{send}_G(c_2) &\Rightarrow c_1 \approx c_2 \end{aligned}$$

If \approx is consistent we define

$$FOLD_F(T) := (G/\approx)[FOLD^T(\chi), F_{FOLD^T}(a)]$$

where $FOLD^T = FOLD^G : G \rightarrow G/\approx$ is the projection of G into G/\approx .

(1) $FOLD_F(T)$ is the “smallest” true type graph into which exists a morphism from T . That is, $FOLD_F(T)$ is a true type graph and for every true type graph T' with a strong morphism $\psi : T \xrightarrow{F, \leq} T'$ it follows that $FOLD_F(T)$ is defined and there exists a unique strong morphism $\phi : FOLD_F(T) \xrightarrow{F, \leq} T'$ such that

$$\phi \circ FOLD^T = \psi$$

(2) Let $T = G[\chi, a]$ be a type graph and let \approx be the equivalence defined as above. Furthermore let $\approx' \subseteq \approx$. Let $p : G \rightarrow G/\approx'$ be the projection of G into G/\approx' . It follows that

$$FOLD^G \cong FOLD^{G/\approx'} \circ p$$

(3) Let $T \cong_F C\langle T_1, \dots, T_n \rangle_F$ be a type graph and let $i \in \{1 \dots, n\}$. Furthermore let $T' \cong_F C\langle T_1, \dots, T_{i-1}, FOLD_F(T_i), T_{i+1}, \dots, T_n \rangle_F$. It follows that

$$\begin{aligned} FOLD_F(T) &\cong_F FOLD_F(T') \\ FOLD^T &\cong FOLD^{T'} \circ C\langle id_{T_1}, \dots, id_{T_{i-1}}, FOLD^{T_i}, id_{T_{i+1}}, \dots, id_{T_n} \rangle_F \end{aligned}$$

Proof:

(1) It follows immediately with the definition of \approx that $FOLD_F(T)$ is a type graph. Now let $\psi : T \xrightarrow{F, \leq} T'$ be a strong morphism such that T' is a true type graph.

We define $\phi : FOLD_F(T) \rightarrow T'$ such that

$$\begin{aligned} \phi([v]_\approx) &:= \psi(v) \text{ if } v \in V_T \\ \phi([e]_\approx) &:= \psi(e) \text{ if } e \in E_T \end{aligned}$$

It is straightforward to show that ϕ is well-defined and unique. And it follows immediately from the definition of ϕ that $\phi \circ FOLD^T = \psi$.

It is left to show that ϕ is a (F, \leq) -morphism: let $T = G[\chi, a]$, $T' = G'[\chi', a']$ and $FOLD_F(T) = (G/\approx)[FOLD^T(\chi), F_{FOLD^T}(a)]$

$$F_\phi(F_{FOLD^T}(a)) = F_{\phi \circ FOLD^T}(a) = F_\psi(a) \leq a'$$

- (2) Let $\approx' \subseteq \approx$, $G' := G/\approx'$ and let \sim be the equivalence used to construct $FOLD^{G'}$.

- \sim can be constructed in the following way: \sim_0 is the equivalence on $V_{G'}$ and $E_{G'}$ and

$$\begin{aligned} v_1 \sim_{i+1} v_2 &\iff v_1 \sim_i v_2 \\ &\quad \vee (\exists q_1, q_2 : \lfloor s_{G'}(q_1) \rfloor_j = \lfloor s_{G'}(q_2) \rfloor_j \wedge q_1 \sim_i q_2) \\ &\quad \vee \exists c_1, c_2 : \lfloor s_{G'}(c_1) \rfloor_j = \lfloor s_{G'}(c_2) \rfloor_j \wedge c_1 \sim_i c_2 \\ p_1 \sim_{i+1} p_2 &\iff p_1 \sim_{i+1} p_2 \vee \forall i : \lfloor s_{G'}(p_1) \rfloor_i \sim_i \lfloor s_{G'}(p_2) \rfloor_i \\ q_1 \sim_{i+1} q_2 &\iff q_1 \sim_{i+1} q_2 \vee send_{G'}(q_1) \sim_i send_{G'}(q_2) \\ c_1 \sim_{i+1} c_2 &\iff c_1 \sim_{i+1} c_2 \vee send_{G'}(c_1) \sim_i send_{G'}(c_2) \end{aligned}$$

where $v_1, v_2 \in V_{G'}$, $p_1, p_2 \in P_{G'}$, $q_1, q_2 \in M_{G'}$, $c_1, c_2 \in Co_{G'}$.

And $\sim = \bigcup_{i=0}^{\infty} \sim_i$.

In the same way we can define a sequence $\approx_0, \approx_1, \dots$ of equivalences with $\approx = \bigcup_{i=0}^{\infty} \approx_i$.

- Let $p_1, p_2 \in P_G$, $q_1, q_2 \in M_G$, $c_1, c_2 \in Co_G$. We will now show that

$$p(p_1) \sim p(p_2) \iff p_1 \approx p_2 \quad (\text{A.6})$$

$$p(q_1) \sim p(q_2) \iff q_1 \approx q_2 \quad (\text{A.7})$$

$$p(c_1) \sim p(c_2) \iff c_1 \approx c_2 \quad (\text{A.8})$$

\Rightarrow Induction on \sim_i :

If $p(q_1) \sim_0 p(q_2)$ it follows that $p(q_1) = p(q_2)$ and therefore $q_1 \approx' q_2$ which implies $q_1 \approx q_2$. The same is true for $p(p_1) \sim_0 p(p_2)$ and $p(c_1) \sim_0 p(c_2)$.

If $p(q_1) \sim_{i+1} p(q_2)$ it follows that either $p(q_1) \sim_i p(q_2)$ (induction hypothesis!) or there exist $x'_1, x'_2 \in M_{G'} \cup Co_{G'}$ such that $x'_1 \sim_{i-1} x'_2$ and there is a j such that $\lfloor s_{G'}(x'_i) \rfloor_j = send_{G'}(p(q_i))$, $i \in \{1, 2\}$.

Since p is surjective, there exist $x_1, x_2 \in M_G \cup Co_G$ such that $p(x_i) = x'_i$, $i \in \{1, 2\}$. It follows with the induction hypothesis that $x_1 \approx x_2$.

Furthermore

$$p(\lfloor s_G(x_i) \rfloor_j) = lf s_{G'}(x'_i) \rfloor_j = send_{G'}(p(q_i)) = p(send_G(q_i))$$

and thus $\lfloor s_G(x_i) \rfloor_j \approx' send_G(q_i)$ where $\approx' \subseteq \approx$. This implies $q_1 \approx q_2$ with the definition of \approx

The other cases can be handled in a similar way.

\Leftarrow Induction on \approx_i :

If $q_1 \approx_0 q_2$ it follows that $q_1 = q_2$ and therefore $p(q_1) = p(q_2)$ which implies $p(q_1) \text{sim}(q_2)$. The same is true for $p_1 \approx_0 p_2$ and $c_1 \approx_0 c_2$.

If $q_1 \approx_{i+1} q_2$ it follows that either $q_1 \approx_i q_2$ (induction hypothesis!) or there exist $x_1, x_2 \in M_G \cup Co_G$ such that $x_1 \approx_{i-1} x_2$ and there is a j such that $\lfloor s_G(x_1) \rfloor_j = \text{send}_G(q_i)$, $i \in \{1, 2\}$.

It follows with the induction hypothesis that $p(x_1) \sim p(x_2)$. Furthermore $\lfloor s_{G'}(p(x_i)) \rfloor_j = \text{send}_{G'}(p(q_i))$, $i \in \{1, 2\}$. This implies $p(q_1) \sim p(q_2)$ with the definition of \sim .

The other cases can be handled in a similar way.

- We will now define a morphism $\phi : (G/\approx')/\sim \rightarrow G/\approx$: let $v \in V_G$, $e \in E_G$

$$\phi([p(v)]_\sim) := [v]_\approx \quad \phi([p(e)]_\sim) := [e]_\approx$$

With (A.6), (A.7) and (A.8) it follows that ϕ is well-defined and bijective. Therefore $FOLD^G \cong FOLD^{G'} \circ p$.

(3) Follows immediately with (2).

□

Proposition 8.3.16 (Correctness of the Type Inference Algorithm) Let $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(S, m, E, a)$ where $\hat{\phi} : G \rightarrow \hat{G}$ and let

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

Then

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash S$$

Proof: By induction on S . We will use the notation of algorithm 8.3.14.

Variable: straightforward

Replication: straightforward with the induction hypothesis

Process: $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(\text{proc}_n(S'), n, E, a)$ where

$$\begin{aligned} \hat{\phi} &= FOLD^{\hat{G}} \circ p_{G'} \circ \phi : G \rightarrow \hat{G} \\ \hat{\chi} &= (FOLD^{\hat{G}} \circ p_P)(\chi_P) = (FOLD^{\hat{G}} \circ p_{G'})(\chi') \\ \hat{a} &= F_{FOLD^{\hat{G}}}(F_{p_P}(a_P) \vee F_{p_{G'}}(a')) \end{aligned}$$

Let

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

Since $F_{\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'}}(a') \leq F_{\hat{\phi}' \circ FOLD^{\tilde{G}}}(F_{p_P}(a_P) \vee F_{p_{G'}}(a')) = F_{\hat{\phi}}(\hat{a}) \leq \hat{a}'$ it follows that

$$\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

And the induction hypothesis implies that

$$(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} \circ \phi)(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash S'$$

Since $(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P)(\chi_P) = \hat{\chi}'$ and $F_{\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P}(a_P) \leq \hat{a}'$ it follows that

$$\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P : A(proc_n(S)) \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

That is $\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P$ is the morphism ϕ in rule (TL-PROC), and thus

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash proc_n(S')$$

Message: $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(mess_{n+1}(H), n+1, E, a)$, $m := card(H)$ where

$$\begin{aligned} \hat{\phi} &= FOLD^{\tilde{G}} \circ p_{G'} \circ \phi : G \rightarrow \hat{G} \\ \hat{\chi} &= (FOLD^{\tilde{G}} \circ p_M)(\chi_M) \\ \hat{a} &= F_{FOLD^{\tilde{G}}}(F_{p_M}(a_M) \vee F_{p_{G'}}(a')) \end{aligned}$$

Let

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

We define $\hat{\chi}'_{G'} := (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'})(\chi')$. Since $F_{\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'}}(a') \leq F_{\hat{\phi}' \circ FOLD^{\tilde{G}}}(F_{p_M}(a_M) \vee F_{p_{G'}}(a')) = F_{\hat{\phi}}(\hat{a}) \leq \hat{a}'$ it follows that

$$\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}'_{G'}, \hat{a}']$$

And it follows with the induction hypothesis that

$$(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} \circ \phi)(E), \hat{G}'[\hat{\chi}'_{G'}, \hat{a}'] \vdash H$$

Since $(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M)(\chi_M) = \hat{\chi}'$ and $F_{\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M}(a_M) \leq \hat{a}'$ it follows that

$$\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M : A(mess_{n+1}(H)) \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}, \hat{a}']$$

And it follows with the properties of \approx that

$$\begin{aligned} & \lfloor s_{\hat{G}'}(cont_{\hat{G}'}(\lfloor \hat{\chi}' \rfloor_{n+1})) \rfloor_{1..m} \\ &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M)(\lfloor s_M(cont_M(\lfloor \chi_M \rfloor_{n+1})) \rfloor_{1..m}) \\ &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'})(\chi') = \hat{\phi}'(\hat{\chi}_{G'}) \end{aligned}$$

If we take $\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M$ as the morphism in typing rule (TL-MESS) and $cont_{\hat{G}'}(\lfloor \hat{\chi}' \rfloor_{n+1})$ as the element of $Co_{\hat{G}'}$ it follows that

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash mess_{n+1}(H)$$

Process Graph: $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(H, \text{card}(H), E, a)$ where

$$\begin{aligned}\hat{\phi} &= FOLD^{\tilde{G}} \circ p_{G_n} \circ \phi_n \circ \dots \phi_1 : G_0 \rightarrow \hat{G} \\ \hat{\chi} &= (FOLD^{\tilde{G}} \circ p_C)(\chi_C) \\ \hat{a} &= F_{FOLD^{\tilde{G}} \circ p_{G_n}}(a_n)\end{aligned}$$

Let

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

We define $\psi_i := \hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G_n} \circ \phi_n \circ \dots \phi_{i+1}$, $\hat{\chi}'_i := \psi_i(\chi_i)$. And with lemma 8.3.15 it follows that $F_{\psi_i}(a_i) \leq \hat{a}'$, $i \in \{1, \dots, n\}$. Thus it follows that

$$\psi_i : G_i[\chi_i, a_i] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}'_i, \hat{a}']$$

it follows with the induction hypothesis that

$$(\psi_i \circ \phi_i)(E_i), \hat{G}'[\hat{\chi}'_i, \hat{a}'] \vdash H_i$$

where $(\psi_i \circ \phi_i)(E_i) = (\hat{\phi}' \circ \hat{\phi})(E)$.

We will now show that $\hat{\phi}' \circ FOLD^{G_n} \circ p_C$ can be used as morphism in typing rule (TL-CON). It follows with the properties of \approx that:

$$\begin{aligned}(\hat{\phi}' \circ FOLD^{G_n} \circ p_C \circ \zeta_i)(\chi_{\mathbf{m}_i}) &= (\hat{\phi}' \circ FOLD^{G_n} \circ p_{G_n})(\chi'_i) \\ &= F_{\psi_i}(\chi_i) = \hat{\chi}'_i \\ (\hat{\phi}' \circ FOLD^{G_n} \circ p_C)(\chi_C) &= \hat{\phi}'(\hat{\chi}) = \hat{\chi}'\end{aligned}$$

(TL-CON) implies that

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash H$$

Process Abstraction: $(\hat{\phi}, \hat{\chi}, \hat{a}) := W(\lambda_k x. H, m, E, a)$ where

$$\begin{aligned}\hat{\phi} &= FOLD^{\tilde{G}} \circ p_{G'} \circ \phi \circ p_G : G \rightarrow \hat{G} \\ \hat{\chi} &= [(FOLD^{\tilde{G}} \circ p_{G'})(\chi')]_{1 \dots m} \\ \hat{a} &= F_{FOLD^{\tilde{G}} \circ p_{G'}}(a')\end{aligned}$$

Let

$$\hat{\phi}' : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}', \hat{a}']$$

We define $\hat{\chi}'_{G'} := (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'})(\chi')$. And since $F_{\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'}}(a') \leq \hat{a}'$ it follows that

$$\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}'[\hat{\chi}'_{G'}, \hat{a}']$$

It follows with the induction hypothesis that

$$(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} \circ \phi)(p_G(E) \setminus x \cup \{x : p_{\mathbf{m}_x}\}), \hat{G}'[\hat{\chi}'_{G'}, \hat{a}'] \vdash H$$

and therefore

$$(\hat{\phi}' \circ \hat{\phi})(E) \setminus x \cup \{x : \hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} \circ \phi \circ p_{\mathbf{m}_x}\}, \hat{G}'[\hat{\chi}', \hat{a}'] \vdash H$$

Furthermore there are projections $p_P : P \rightarrow \tilde{G}$, $p_M : M \rightarrow \tilde{G}$, $p_{Co} : Co \rightarrow \tilde{G}$. Let p be the only process in P , q the only message in M and c the only content-edge in Co . It follows with the properties of the quotient graph that

$$\begin{aligned} s_{\hat{G}'}((\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P)(p)) &= \hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P(\chi_P) \\ &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'})([\chi']_{1\dots m}) = \hat{\chi}'_{G'} = \hat{\chi}' \\ s_{\hat{G}'}((\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M)(q)) &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M)(\chi_M) \\ &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'})([\chi']_{m+1\dots m+n k}) = [\hat{\chi}'_{G'}]_{m+1\dots m+n k} \\ s_{\hat{G}'}((\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_C)(c)) &= \hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_C(\chi_{Co}) \\ &= (\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'} \circ \phi \circ p_{\mathbf{m}_x})(\chi_{\mathbf{m}_x}) \circ [\hat{\chi}'_{G'}]_k \end{aligned}$$

If we take $\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{G'}$ as the morphism in typing rule (TL-PA) and use $(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_P)(p)$, $(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_M)(q)$ and $(\hat{\phi}' \circ FOLD^{\tilde{G}} \circ p_{Co})(c)$ as the respective hyperedges it follows that

$$(\hat{\phi}' \circ \hat{\phi})(E), \hat{G}'[\hat{\chi}', \hat{a}'] \vdash \lambda_k x. H$$

□

Proposition 8.3.17 (Soundness of the Type Inference Algorithm)

Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash S$ and let $\hat{\phi} : G \xrightarrow{F, \leq} \hat{G}$ with $\hat{E} = \hat{\phi}(E)$ and $F_{\hat{\phi}}(a) \leq \hat{a}$ then

$$(\phi, \chi', a') := W(S, |\hat{\chi}|, E, a)$$

(where $\phi : G \rightarrow G'$) is defined and there exists a morphism

$$\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

such that $\hat{\phi} = \psi \circ \phi$.

Proof: We will proceed by induction on S :

Variable: $W(x, \text{sort}(x), E, a)$ is always defined and we set $\psi := \hat{\phi}$.

Replication: Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash !H$. It follows that $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash H$ and the induction hypothesis implies that $(\phi, \chi', a') := W(H, \text{card}(H), E, a)$ is defined and that there exists a morphism $\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$ such that $\hat{\phi} = \psi \circ \phi$.

Process: Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash \text{proc}_n(S')$. It follows with (TL-PROC) that $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash S'$ and the induction hypothesis implies that $(\phi, \chi', a') :=$

$W(S', n, E, a)$ is defined and that there exists a morphism $\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$ such that $\hat{\phi} = \psi \circ \phi$.

It is obvious from the algorithm that $n = |\chi'|$ and therefore $\tilde{G} := (PG')/\approx$ is defined.

Since (TL-PROC) implies the existence of a morphism $\phi_P : [\chi_P, a_P] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$ where $P[\chi_P, a_P] := A(proc_n(S'))$, it follows with proposition 2.2.7 that there is a morphism

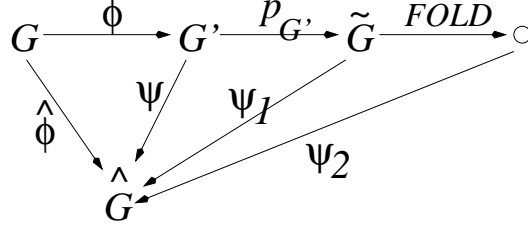
$$\psi_1 : \tilde{G} \rightarrow \hat{G}$$

with $\psi = \psi_1 \circ p_{G'}$ and $\phi_P = \psi_1 \circ p_P$.

And 1 implies that $FOLD_F(\tilde{G})$ is defined and that there exists a morphism

$$\psi_2 : FOLD(\tilde{G}) \rightarrow \hat{G}$$

with $\psi_1 = \psi_2 \circ FOLD^{\tilde{G}}$.



It is left to show that

$$\begin{aligned} \psi_2 : FOLD(\tilde{G})[FOLD^{\tilde{G}}(p_P(\chi_P)), F_{FOLD\tilde{G}}(F_{p_P}(a_P) \vee F_{p_{G'}}(a'))] \\ \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}] \end{aligned}$$

This is true since

$$\psi_2(FOLD^{\tilde{G}}(p_P(\chi_P))) = \psi_1(p_P(\chi_P)) = \phi_P(\chi_P) = \hat{\chi}$$

$$\begin{aligned} F_{\psi_2}(F_{FOLD\tilde{G}}(F_{p_P}(a_P) \vee F_{p_{G'}}(a'))) &= F_{\psi_1}(F_{p_P}(a_P) \vee F_{p_{G'}}(a')) \\ &= F_{\phi_P}(a_P) \vee F_{\psi}(a') \leq \hat{a} \vee \hat{a} = \hat{a} \end{aligned}$$

Message: Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash mess_{n+1}(H)$ and let $m := card(H)$. It follows with (TL-MESS) that $\hat{E}, \hat{G}[\hat{\chi}', \hat{a}] \vdash H$ and the induction hypothesis implies that $(\phi, \chi', a') := W(H, card(H), E, a)$ where $\phi : G \rightarrow G'$ is defined and that there exists a morphism $\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}', \hat{a}]$ such that $\hat{\phi} = \psi \circ \phi$. Furthermore (TL-MESS) implies that there is a morphism

$$\phi_M : M[\chi_M, a_M] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

where $M[\chi_M, a_M] := A(mess_{n+1}(H))$, and that there exists a $c \in Co_{\hat{G}}$ such that $s_{\hat{G}}(c) = \hat{\chi}' \circ [\hat{\chi}]_{n+1}$. That is

$$\begin{aligned} &\phi_M([s_M(cont_M([\chi_M]_{n+1}))]_{1..m}) \\ &= [s_{\hat{G}}(cont_{\hat{G}}([\hat{\chi}]_{n+1}))]_{1..m} = [s_{\hat{G}}(c)]_{1..m} = \hat{\chi}' \end{aligned}$$

Let $\tilde{G} := MG'/\approx$ be the quotient graph. It follows with proposition 2.2.7 and the existence of ψ, ϕ_M that there is a morphism

$$\psi_1 : \tilde{G} \rightarrow \hat{G}$$

with $\psi = \psi_1 \circ p_{G'}$, $\phi_M = \psi_1 \circ p_M$.

And (1) implies that $FOLD(\tilde{G})$ is defined and that there exists a morphism

$$\psi_2 : FOLD(\tilde{G}) \rightarrow \hat{G}$$

with $\psi_1 = \psi_2 \circ FOLD^{\tilde{G}}$ (see figure above). It is left to show that

$$\begin{aligned} \psi_2 : FOLD(\tilde{G})[FOLD^{\tilde{G}}(p_M(\chi_M)), F_{FOLD\tilde{G}}(F_{p_M}(a_M) \vee F_{p_{G'}}(a'))] \\ \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}] \end{aligned}$$

This is true since

$$\begin{aligned} \psi_2(FOLD^{\tilde{G}}(p_M(\chi_M))) &= \hat{\chi} = \psi_1(p_M(\chi_M)) \\ &= \phi_M(\chi_M) = \hat{\chi} \\ F_{\psi_2}(F_{FOLD\tilde{G}}(F_{p_M}(a_M) \vee F_{p_{G'}}(a'))) &= F_{\psi_1}(F_{p_M}(a_M) \vee F_{p_{G'}}(a')) \\ &= F_{\phi_M}(a_M) \vee F_{\psi}(a') \leq \hat{a} \vee \hat{a} = \hat{a} \end{aligned}$$

Process Graph: Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow D$ and let $\phi_D : D \rightarrow \hat{G}[\hat{\chi}]$. It follows that $\hat{E}, \hat{G}[\phi_D(\zeta_i(\chi_{\mathbf{m}_i})), \hat{a}] \vdash H_i$.

We define $\psi_{-1} := \hat{\phi}$ and $\phi_0 := id_G$ and will now show by induction on i that $(\phi_i, s_i, a_i) := W(H_i, card(H_i), E_{i-1}, a_{i-1})$ is defined and that there exist morphisms

$$\psi_i : G_i[\chi_i, a_i] \xrightarrow{F, \leq} \hat{G}[\phi_D(\zeta_i(\chi_{\mathbf{m}_i})), \hat{a}]$$

such that $\psi_{i-1} = \psi_i \circ \phi_i$ for $1 \leq i \leq n$.

$i = 0$: Define $\psi_0 := \hat{\phi}$

$i \rightarrow i + 1$: Because of the existence of ψ_{i-1} and since

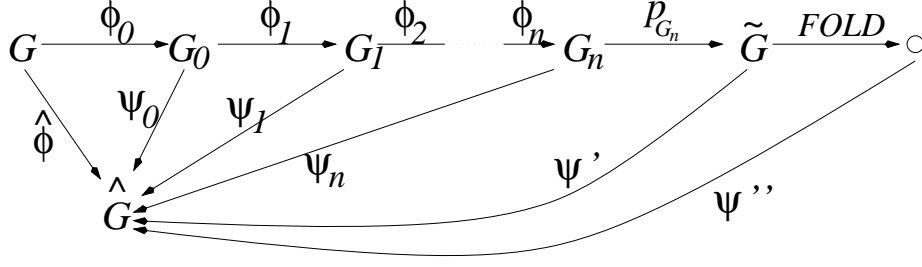
$$\psi_{i-1}(E_{i-1}) = \psi_{i-1}(\phi_{i-1} \circ \dots \circ \phi_0)(E) = \hat{\phi}(E) = \hat{E}$$

it follows with the outer induction hypothesis that (ϕ_i, χ_i, a_i) is defined and that there exists a morphism

$$\psi_i : G_i[\chi_i, a_i] \xrightarrow{F, \leq} \hat{G}[\phi_D(\zeta_i(\chi_{\mathbf{m}_i})), \hat{a}]$$

such that $\psi_{i-1} = \psi_i \circ \phi_i$.

Furthermore $\psi_n(\chi'_i) = (\psi_n \circ \phi_n \circ \dots \circ \phi_{i+1})(\chi_i) = \psi_i(\chi_i) = \phi_D(\zeta_i(\chi_{\mathbf{m}_i}))$. This implies with 2.2.7 that $\tilde{G}[\tilde{\chi}, \tilde{a}] := (D^0 G_n)/\approx$ is defined and that there exists a morphism $\psi' : \tilde{G} \rightarrow \hat{G}$ such that $\psi' \circ p_{G_n} = \psi_n$ and $\psi' \circ p_D = \phi_D$.



And (1) implies that $FOLD(\tilde{G})$ is defined and that there exists a morphism

$$\psi'' : FOLD(\tilde{G}) \rightarrow \hat{G}$$

with $\psi' = \psi'' \circ FOLD^{\tilde{G}}$. It is left to show that

$$\psi'' : FOLD(\tilde{G})[FOLD^{\tilde{G}}(p_D(\chi_D)), F_{FOLD\tilde{G}}(F_{p_{G_n}}(a_n))] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

This is true since

$$\begin{aligned} \psi''(FOLD^{\tilde{G}}(p_D(\chi_D))) &= \psi'(p_D(\chi_D)) = \phi_D(\chi_D) = \hat{\chi} \\ F_{\psi''}(F_{FOLD\tilde{G}}(F_{p_{G_n}}(a_n))) &= F_{\psi'}(F_{p_{G_n}}(a_n)) \\ &= F_{\psi_n}(a_n) \leq \hat{a} \end{aligned}$$

Process Abstraction: Let $\hat{E}, \hat{G}[\hat{\chi}, \hat{a}] \vdash \lambda_k x. H$. It follows that $\hat{E} \setminus x \cup \{x : \eta_x\}, \hat{G}[\hat{\chi} \circ \hat{\chi}_0, \hat{a}] \vdash H$.

Furthermore there are $q \in M_{\hat{G}}, c \in Co_{\hat{G}}, p \in P_{\hat{G}}$ such that $s_{\hat{G}}(q) = \hat{\chi}_0 \circ [\hat{\chi}]_k, s_{\hat{G}}(c) = \eta_x(\chi_{\mathbf{m}_x}) \circ [\hat{\chi}]_k, s_{\hat{G}}(p) = \hat{\chi}$.

We can show with the properties of graph construction (proposition 2.2.7) that there exists a morphism $\bar{\psi} : \bar{G} \rightarrow \hat{G}$ such that $\bar{\psi} \circ p_G = \hat{\phi}$ and $\bar{\psi} \circ p_{\mathbf{m}_x} = \eta_x$.

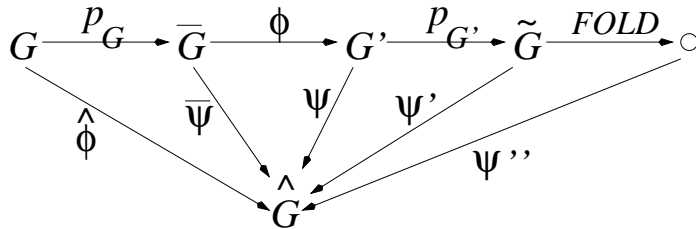
Since $\bar{\psi}(p_G(E) \setminus x \cup \{x : p_{\mathbf{m}_x}\}) = \hat{E} \setminus x \cup \{x : \eta_x\}$, it follows with the induction hypothesis that $(\phi, \chi', a') := W(H, card(H), p_G(E) \setminus x \cup \{x : p_{\mathbf{m}_x}\}, F_{p_G}(a))$ is defined and that there exists a morphism

$$\psi : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi} \circ \hat{\chi}_0, \hat{a}]$$

Let \tilde{G} be the quotient graph. It follows with the the existence of the hyperedges q, c, p above and with proposition 2.2.7 that there is a morphism

$$\psi' : \tilde{G} \rightarrow \hat{G}$$

with $\psi = \psi' \circ p_{G'}$, $s_{\hat{G}}(p) = \psi'(\chi_P)$, $s_{\hat{G}}(q) = \psi'(\chi_M)$ and $s_{\hat{G}}(c) = \psi'(\chi_{Co})$.



And (1) implies that $FOLD(\tilde{G})$ is defined and that there exists a morphism

$$\psi'' : FOLD(\tilde{G}) \rightarrow \hat{G}$$

with $\psi' = \psi'' \circ FOLD^{\tilde{G}}$.

It is left to show that

$$\psi'' : FOLD(\tilde{G}[\lfloor FOLD^{\tilde{G}}(p_{G'}(\chi')) \rfloor_{1\dots m}, F_{FOLD^{\tilde{G}}}(F_{p_{G'}}(a')) \rfloor) \xrightarrow{F, \leq} \hat{G}[\hat{\chi}\hat{\chi}_0, \hat{a}]$$

This is true since

$$\begin{aligned} \psi''(\lfloor FOLD^{\tilde{G}}(p_{G'}(\chi')) \rfloor_{1\dots m}) &= \psi''(FOLD^{\tilde{G}}(\chi_P)) = \psi'(\chi_P) \\ &= s_{\hat{G}}(p) = \hat{\chi} \\ F_{\psi''}(F_{FOLD^{\tilde{G}}}(F_{p_{G'}}(a'))) &= F_{\psi'}(F_{p_{G'}}(a')) = F_{\psi}(a') \leq \hat{a} \end{aligned}$$

□

Proposition 8.3.19 *Let $E, T \vdash H$. This implies that there is a morphism*

$$\psi : A(H) \xrightarrow{F, \leq} T$$

Proof: We will proceed by induction on the typing of H :

(TL-PROC), (TL-MESS) immediate

(TL-CON) Let

$$E, G[\chi, a] \vdash \bigotimes_{i=1}^n (H_i, \zeta_i)$$

such that there exists a strong morphism $\phi : D \rightarrow G[\chi]$ and

$$E, G[\phi(\zeta_i(\chi_{\mathbf{m}_i})), a_i] \vdash H_i$$

It follows with the induction hypothesis that there are strong morphisms

$$\psi_i : A(H_i) \xrightarrow{F, \leq} G[\phi(\zeta_i(\chi_{\mathbf{m}_i})), a_i]$$

$\hat{G}[\hat{\chi}, \hat{a}] := A(H) \cong \bigotimes_{i=1}^n (A(H_i), \zeta_i)$ (see figure). Because of the existence of the ψ_i and ϕ it follows with the properties of the co-limit that there exists a strong morphism $\psi : \hat{G}[\hat{\chi}] \rightarrow G[\chi]$ with $\psi \circ \eta_i = \psi_i$ and $\psi \circ \tilde{\phi} = \phi$.

Let $\hat{G}_i[\hat{\chi}_i, \hat{a}_i] := A(H_i)$. It follows that

$$F_\psi(\hat{a}) = F_\psi\left(\bigvee_{i=1}^n F_{\eta_i}(\hat{a}_i)\right) = \bigvee_{i=1}^n F_{\psi_i}(\hat{a}_i) \leq \bigvee_{i=1}^n a = a$$

and therefore

$$\psi : A(H) \cong_F \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} G[\chi, a]$$

□

A.2.2 A Type System Based on Monoids

Proposition 8.4.6 (Equivalence) *Let $H_1, H_2 \in \mathcal{S}$ with $H_1 \equiv H_2$.*

$T \Vdash H_1 \iff T \Vdash H_2$ for any type graph T .

This implies immediately $TT \vdash H_1 \iff TT \vdash H_2$ for any true type graph TT .

Proof: We proceed by induction on the rules of structural equivalence. We have to take into account application of rule (TL- \leq).

(C-PROC) Let $H_i := \text{proc}_n(S_i)$, $i = 1, 2$ and let $H_1 \equiv H_2$. Furthermore let $T \Vdash H_1$ where $A(H_1) \square_F T' \lesssim T$ and $T' \Vdash S_1$. Since $S_1 \equiv S_2$ it follows with the induction hypothesis that $T' \Vdash S_2$.

According to condition (1) of the linear mapping it follows that $A(H_1) \cong A(H_2)$. And lemma 8.2.12 implies that $A(H_2) \square_F T' \cong A(H_1) \square_F T' \lesssim_F T$.

And therefore $T \Vdash H_2$

(C-MESS) Let $H_i := \text{mess}_n(\mathbf{0})$, $i = 1, 2$ and let $T \Vdash H_1$ where $A(H_1) \lesssim T$. Since $H_1 \equiv H_2$ it follows that $A(H_1) \cong A(H_2)$ and typing rule (TM- \leq) implies $T \Vdash H_2$.

(C-PA) Let $\lambda_k.H_1 \equiv \lambda_k.H_2$ and let $T \Vdash \lambda_k.H_1$. It follows that

$$T \gtrsim_F \sigma_{1\dots m}(T' \square A(\text{Red}_{k,m,n}(\lambda_k.H_1))^-)$$

with $T' \Vdash H_1$. Since $H_1 \equiv H_2$ it follows with the induction hypothesis that $T' \Vdash H_2$.

Since $\text{Red}_{k,m,n}(\lambda_k.H_1) \equiv \text{Red}_{k,m,n}(\lambda_k.H_2)$ it follows that

$$A(\text{Red}_{k,m,n}(\lambda_k.H_1)) \cong_F A(\text{Red}_{k,m,n}(\lambda_k.H_2))$$

It follows with lemma 8.2.12 that

$$\begin{aligned} & \sigma_{1\dots m}(T' \square A(\text{Red}_{k,m,n}(\lambda_k.H_2))^-) \\ & \cong_F \sigma_{1\dots m}(T' \square A(\text{Red}_{k,m,n}(\lambda_k.H_1))^-) \lesssim_F T \end{aligned}$$

and therefore $T \Vdash \lambda_k.H_2$.

(C-REPL) Let $!H_1 \equiv !H_2$ with $T \Vdash !H_1$ where $T \sqcap_F T \xrightarrow{J, \leq} T$. Typing rules (TM-REPL), (TM- \leq) imply that $T' \Vdash H_1$ where $T' \lesssim_F T$. The induction hypothesis implies that $T' \Vdash H_2$ and therefore $T \Vdash !H_2$.

(C-CON) Let $H_i \equiv J_i$, $i \in \{1, \dots, n\}$ and let C be a context with holes of cardinality $\text{card}(H_1), \dots, \text{card}(H_n)$. Let $T \Vdash C\langle H_1, \dots, H_n \rangle$.

With lemma 8.4.5 it follows that $T_i \Vdash H_i$ and $C\langle T_1, \dots, T_n \rangle_F \lesssim_F T$. The induction hypothesis implies that $T_i \Vdash J_i$ and therefore

$$C\langle T_1, \dots, T_n \rangle_F \Vdash C\langle J_1, \dots, J_n \rangle$$

With (TL- \leq) it follows that $T \Vdash C\langle J_1, \dots, J_n \rangle$.

Rule (C- α) can be omitted since it has no meaning in the restricted calculus. \square

Proposition 8.4.8 (Message Reception) *Let $TT \vdash H$. If H contains a redex*

$$\text{Red} := \text{Red}_{k,m,n}(\lambda_k x. K)$$

then message-reception is defined, i.e. $m + n = \text{card}(K)$.

Let $H \xrightarrow{(R-MR)} H'$. It follows that H' has the same type as H , i.e. $TT \vdash H'$.

Proof: Let $T \Vdash H$ such that there is a (J, \leq) -morphism $\phi : T \rightarrow TT$.

If H contains a subgraph of the form $\text{Red} := \text{Red}_{k,m,n}(\lambda_k x. K)$ then H has the form $C\langle \text{Red}, \hat{H} \rangle$ where C is a discrete context.

Thus $H' \cong_F C\langle K, \hat{H} \rangle$.

- By unravelling the typing of H it follows with lemma 8.4.5 that $T \gtrsim_F C\langle T_R, \hat{T} \rangle_F$ where $T_R \Vdash \text{Red}$ and $\hat{T} \Vdash \hat{H}$.

By further unravelling the typing it follows that

$$T_R \gtrsim R\langle A(\text{proc}_m(\lambda_k x. K)) \sqcap_F \sigma_{1\dots m}(T_K \sqcap_F A(\text{Red})^-), A(\text{mess}_{n+1}(\mathbf{0})) \rangle_F$$

where $T_K \Vdash K$ and

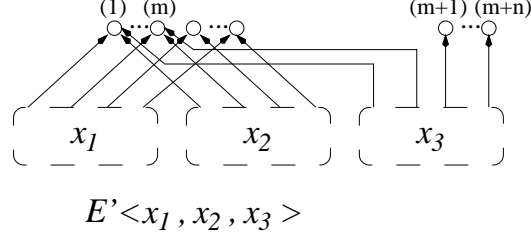
$$R\langle x_1, x_2 \rangle := \begin{array}{c} \begin{array}{ccccccc} (1) & \dots & (k) & \dots & (m) & & (m+1) & \dots & (m+n) \\ \circ & & \circ & & \circ & & \circ & & \circ \\ \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ \text{---} & & \text{---} & & \text{---} & & \text{---} & & \text{---} \\ \text{---} & & \text{---} & & \text{---} & & \text{---} & & \text{---} \end{array} \\ \left[\begin{array}{ccc} & & \\ & x_1 & \\ & & \end{array} \right] & \left[\begin{array}{ccc} & & \\ & x_2 & \\ & & \end{array} \right] \end{array}$$

- It follows that $C\langle T_K, \hat{T} \rangle_F \Vdash H'$. In the rest of this proof we will show that there exists a morphism $C\langle T_K, \hat{T} \rangle_F \xrightarrow{J, \leq} TT$.

- After some transformations (see proposition 2.2.13) it follows that

$$T_R \stackrel{\sim}{>}_F E' \langle T_K, A(Red)^-, A(Red) \rangle_F =: T'_R$$

where



- We will now show that there exists a (F, \geq) -morphism (!)

$$\psi : T_R \rightarrow T_K \square_F A(Red)[0]$$

It is sufficient to show that there is a (F, \geq) -morphism from T'_R into $T_K \square_F A(Red)[\perp]$.

Let

$$\begin{aligned} T'_R &= G_R[\chi'_R, a'_R] \\ T_K &= G_K[\chi_K, a_K] \\ A(Red) &= G_{Red}[\chi_{Red}, a_{Red}] \\ T_I &= G_I[\chi_I, a_I] := T_K \square_F A(Red)[0] \end{aligned}$$

Let $\zeta_i : \mathbf{m} + \mathbf{n} \rightarrow \mathbf{m} + \mathbf{n} + \bar{\mathbf{n}} =: D$ be the embeddings corresponding to the context E' such that

$$T'_R \cong_F (T_K, \zeta_1) \otimes (A(Red)^-, \zeta_2) \otimes (A(Red), \zeta_3)$$

That is $V_D := \{v_1, \dots, v_{m+n}, w_1, \dots, w_n\}$, $\chi_D := v_1 \dots v_{m+n}$, $\zeta_1(\chi_K) = \zeta_2(\chi_{Red}) = v_1 \dots v_m w_1 \dots w_n$, $\zeta_3(\chi_{Red}) := \chi_D$.

Let

$$\begin{aligned} \eta_1 : G_K[\chi_K] &\rightarrow G_R[\chi'_R] \\ \eta_2 : G_{Red}[\chi_{Red}] &\rightarrow G_R[\chi'_R] \\ \eta_3 : G_{Red}[\chi_{Red}] &\rightarrow G_R[\chi'_R] \end{aligned}$$

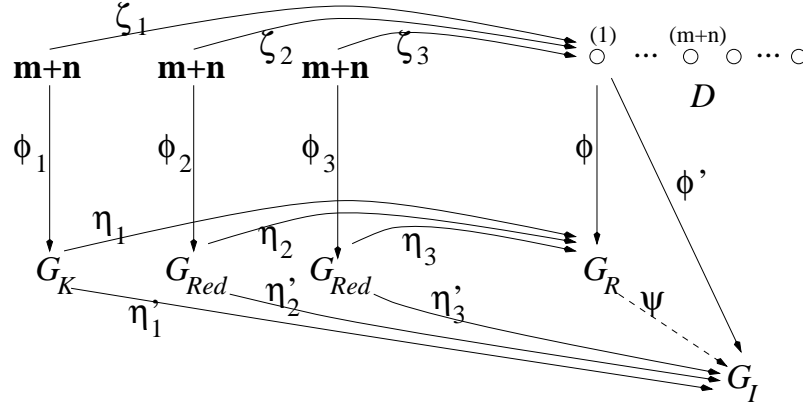
be the corresponding embeddings into $G_R[\chi'_R]$, let ϕ_i be the canonical strong morphisms and let $\phi : D \rightarrow G_R[\chi_R]$.

We will now show that there exists a strong morphism

$$\psi : G'_R[\chi'_R] \rightarrow G_K[\chi_K] \square_F G_{Red}[\chi_{Red}]$$

such that $F_\psi(a'_R) \geq a_I$.

There are obviously embeddings $\eta'_1 : G_K[\chi_K] \rightarrow G_I[\chi_I]$ and $\eta'_2 = \eta'_3 : G_{Red}[\chi_{Red}] \rightarrow G_I[\chi_I]$. Furthermore there is a strong morphism $\phi' : D \rightarrow G_I[\chi_I]$ with $\phi'(v_1 \dots v_{m+n}) := \chi_I$, $\phi'(w_1 \dots w_n) := \chi_I]_{m+1 \dots m+n}$.



Since $\eta'_i \circ \phi_i = \phi' \circ \zeta_i$ it follows with the properties of a co-limit that there exists a strong morphism $\psi : G_R[\chi_R] \rightarrow G_I[\chi_I]$ with $\psi \circ \phi = \phi'$ and $\psi \circ \eta_i = \eta'_i$.

Furthermore

$$\begin{aligned}
 F_\psi(a_R) &= F_\psi(F_{\eta_1}(a_K) + F_{\eta_2}(0 - a_R) + F_{\eta_3}(a_R)) \\
 &= F_{\eta'_1}(a_K) + F_{\eta'_2}(0 - a_R) + F_{\eta'_2}(a_R) \\
 &= F_{\eta'_1}(a_K) + F_{\eta'_2}((0 - a_R) + a_R) \stackrel{(8.4)}{\geq} F_{\eta'_1}(a_K) + F_{\eta'_2}(0) = a_I
 \end{aligned}$$

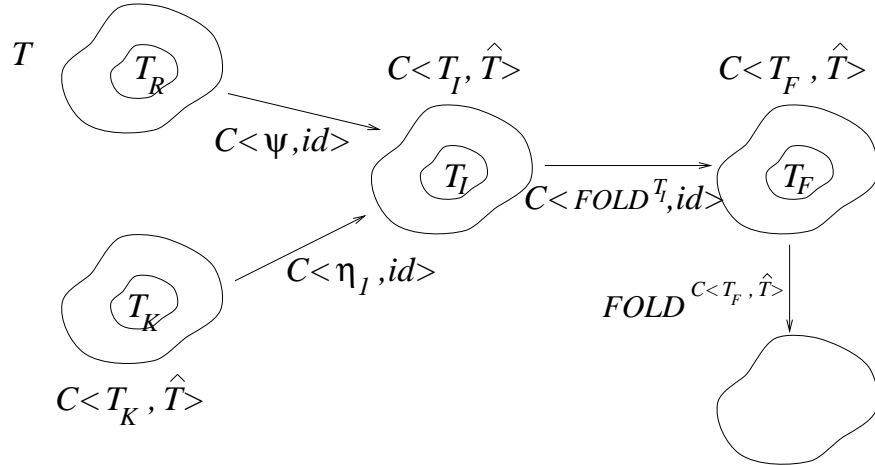
Since ψ is a (F, \geq) -morphisms it follows immediately with lemma 8.2.11 that $C\langle\psi, id\rangle_F$ is also a (F, \geq) -morphisms.

Since $C\langle\psi, id\rangle_F$ is surjective and therefore $J_{C\langle\psi, id\rangle_F} \geq F_{C\langle\psi, id\rangle_F}$, it is also a (J, \geq) -morphism.

- $\eta_1 : T_K \rightarrow T_K \square_F A(Red)[0] =: T_I$ is a $(F, =)$ -morphism. It follows with lemma 8.2.11 that $C\langle\eta_1, id\rangle_F$ is also a $(F, =)$ -morphism.

And since $C\langle\eta_1, id\rangle_F$ is injective and therefore $G_{C\langle\eta_1, id\rangle_F} \leq F_{C\langle\eta_1, id\rangle_F}$ it is also a (J, \leq) -morphism.

- We now have the following situation:



where $T_F := FOLD_J(T_I)$.

Let \approx be the equivalence used to construct $FOLD^{T_R}$. The surjective morphism $\psi : T_R \rightarrow T_I$ can be characterized by the equivalence \approx' and it follows with conditions (4) and (4) of the linear mapping that $\approx' \subseteq \approx$. It follows with condition (2) in proposition 8.3.13 that $FOLD^{T_R} = FOLD^{T_I} \circ \psi$.

And it follows with condition (3) in proposition 8.3.13 that

$$\begin{aligned} FOLD^{C\langle T_R, \hat{T} \rangle_F} &= FOLD^{C\langle T_F, \hat{T} \rangle_F} \circ C\langle FOLD^{T_R}, id \rangle_F \\ &= FOLD^{C\langle T_F, \hat{T} \rangle_F} \circ C\langle FOLD^{T_I}, id \rangle_F \circ C\langle \psi, id \rangle_F \end{aligned}$$

- It is left to show that

$$\rho := FOLD^{C\langle T_F, \hat{T} \rangle_F} \circ C\langle FOLD^{T_I}, id \rangle_F : C\langle T_I, \hat{T} \rangle_F \rightarrow FOLD_J(T)$$

is a (J, \leq) -morphism.

Let $T = G[\chi, a]$ and $C\langle T_I, \hat{T} \rangle_F = G'_I[\chi'_I, a'_I]$. Since $C\langle \psi, id \rangle_F$ is a (J, \geq) morphism it follows that

$$J_\rho(a'_I) \leq J_\rho(J_{C\langle \psi, id \rangle_F}(a)) = J_{FOLD^{C\langle T_R, \hat{T} \rangle_F}}(a) = J_{FOLD^T}(a)$$

And $\rho \circ C\langle \eta_1, id \rangle$ is the morphism folding $C\langle T_K, \hat{T} \rangle_F$ into the true type graph $FOLD_J(T)$. And since there is a (J, \leq) -morphism from $FOLD_J(T)$ into TT (proposition 8.3.13, condition (1)) it follows that $TT \vdash C\langle K, \hat{H} \rangle$. \square

A.2.3 Transformation of Type Systems

Proposition 8.5.1 (Lattice \rightarrow Monoid) *Let S be a process description or process graph in \mathcal{S}_n without variables whose messages are all labelled $\mathbf{0}$.*

If $E, G[\chi, a] \vdash_{A,F}^L S$ it follows that $G[\chi, a] \vdash_{A,F,F}^M S$.

Proof: Let $E, G[\chi, a] \vdash_{A,F}^L S$. We will show by induction on S that there exists a type graph $\hat{G}[\hat{\chi}, \hat{a}]$ such that $\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{F,F,A}^M S$ and there exists a morphism

$$\hat{\phi} : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} G[\chi, a]$$

Replication: If $E, G[\chi, a] \vdash_{A,F}^L !H$ it follows with (TL-REPL) that $E, G[\chi, a] \vdash_{A,F}^L H$. And the induction hypothesis implies that there exists a type graph $\hat{G}[\hat{\chi}, \hat{a}]$ such that $\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M H$ and a morphism

$$\hat{\phi} : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} G[\chi, a]$$

Let $\hat{T} := \hat{G}[\hat{\chi}, \hat{a}]$. We will now show that there is a (F, \leq) -morphism $\phi : \hat{T} \square \hat{T} \rightarrow \hat{T}$: Let $\zeta : \mathbf{n} \rightarrow \mathbf{n}$ be a strong morphism. Since $\hat{T} \square \hat{T} := (\hat{T}, \zeta) \otimes (\hat{T}, \zeta)$ we obtain the following commutative diagram:

$$\begin{array}{ccccc}
& & \zeta & & \zeta \\
& & \rightarrow & & \rightarrow \\
\mathbf{n} & & \mathbf{n} & & \mathbf{n} \\
\downarrow & & \downarrow & & \downarrow \\
\hat{T} & \xrightarrow{\eta_1} & \hat{T} & \square & \hat{T} \xleftarrow{\eta_2} \hat{T} \\
& \searrow & \downarrow \phi & \nearrow & \\
& id & \hat{T} & id &
\end{array}$$

It follows that

$$\begin{aligned}
F_\phi(F_{\eta_1}(\hat{a}) \vee F_{\eta_2}(\hat{a})) &= F_{\phi \circ \eta_1}(\hat{a}) \vee F_{\phi \circ \eta_2}(\hat{a}) \\
&= F_{id}(\hat{a}) \vee F_{id}(\hat{a}) = \hat{a}
\end{aligned}$$

And it follows with (TM-REPL) that

$$\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M !H$$

Process: If $E, G[\chi, a] \vdash_{A,F}^L proc_n(S)$ it follows with (TL-PROC) that $E, G[\chi, a] \vdash_{A,F}^L S$. And the induction hypothesis implies that there exists a type graph $\hat{G}[\hat{\chi}, \hat{a}]$ such that $\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M S$ and a morphism

$$\hat{\phi} : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} G[\chi, a]$$

(TM-PROC) implies that

$$\tilde{G}[\tilde{\chi}, \tilde{a}] := A(proc_n(S)) \square_F \hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M proc_n(S)$$

We define $P[\chi_P, a_P] := A(proc_n(S))$. Let $p_P : P[\chi_P] \rightarrow \tilde{G}[\tilde{\chi}]$ and $p_{\hat{G}} : \hat{G}[\hat{\chi}] \rightarrow \tilde{G}[\tilde{\chi}]$ be the projections into \tilde{G} .

Since $E, G[\chi, a] \vdash_{A,F,F}^L proc_n(S)$ it follows that there is a morphism

$$\phi_P : A(proc_n(S)) \xrightarrow{F, \leq} G[\chi, a]$$

The existence of $\hat{\phi}$ and ϕ_P implies with proposition 2.2.7 that there exists a morphism $\psi : \tilde{G}[\tilde{\chi}] \rightarrow G[\chi]$ such that $\hat{\phi} = \psi \circ p_{\hat{G}}$ and $\phi_P = \psi \circ p_P$.

It is left to show that ψ is a (F, \leq) -morphism:

$$F_\psi(\tilde{a}) = F_\psi(F_{p_P}(a_P) \vee F_{p_{\hat{G}}}(\hat{a})) = F_{\phi_P}(a_P) \vee F_{\hat{\phi}}(\hat{a}) \leq a \vee a = a$$

Message: If $E, G[\chi, a] \vdash_{A,F}^L mess_n(\mathbf{0})$ it follows with (TL-MESS) that there exists a morphism

$$\phi : A(mess_{n+1}(\mathbf{0})) \xrightarrow{F, \leq} G[\chi, a]$$

And (TM-MESS) implies that $A(mess_{n+1}(\mathbf{0})) \vdash_{A,F,F}^M mess_{n+1}(\mathbf{0}) :$

Process Graph: If $E, G[\chi, a] \vdash_{A,F}^L \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow C$.

It follows with lemma 8.3.7 that there exists a morphism $\phi_C : C \rightarrow G[\chi]$ such that $E, G[\phi_C(\eta_i(\chi_{\mathbf{m}_i})), a] \vdash_{A,F}^L H_i$ and $\chi = \phi_C(\chi_C)$.

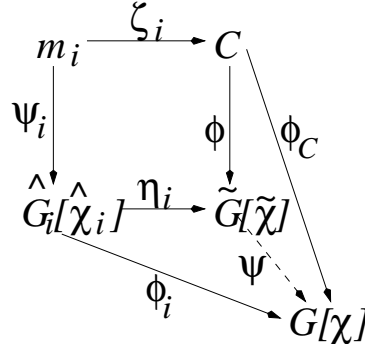
The induction hypothesis implies that $\hat{G}_i[\hat{\chi}_i, \hat{a}_i] \Vdash_{A,F,F}^M H_i$ and there exist morphisms

$$\phi_i : \hat{G}_i[\hat{\chi}_i, \hat{a}_i] \xrightarrow{F, \leq} G[\phi_C(\eta_i(\chi_{\mathbf{m}_i})), a]$$

(TM-CON) implies that

$$\tilde{G}[\tilde{\chi}, \tilde{a}] := \bigotimes_{i=1}^n (\hat{G}_i[\hat{\chi}_i, \hat{a}_i], \zeta_i) \Vdash_{A,F,F}^M \bigotimes_{i=1}^n (H_i, \zeta_i)$$

It is now left to show that there exists a morphism from $\tilde{G}[\tilde{\chi}]$ into $G[\chi]$: Let $\phi : C \rightarrow \tilde{G}[\tilde{\chi}]$ and let $\eta_i : \hat{G}_i[\hat{\chi}_i] \rightarrow \tilde{G}[\tilde{\chi}]$ be the embeddings into $\tilde{G}[\tilde{\chi}]$. Furthermore let $\psi_i : \mathbf{m}_i \rightarrow \hat{G}_i[\hat{\chi}_i]$ be the canonical strong morphism. It follows that



$$\phi_i(\psi_i(\chi_{\mathbf{m}_i})) = \phi_i(\hat{\chi}_i) = \phi_C(\zeta_i(\chi_{\mathbf{m}_i}))$$

Thus $\phi_i \circ \psi_i = \phi_C \circ \zeta_i$ and it follows with the definition of graph construction (definition 2.2.9) that there exists a morphism

$$\psi : \tilde{G}[\tilde{\chi}] \rightarrow G[\chi]$$

with $\psi \circ \eta_i = \phi_i$, $\psi \circ \phi = \phi_C$.

Furthermore $F_\psi(\tilde{a}) = F_\psi(\bigvee_{i=1}^n F_{\eta_i}(a_i)) = \bigvee_{i=1}^n F_{\phi_i}(\hat{a}_i) \leq \bigvee_{i=1}^n a = a$, i.e.

$$\psi : \tilde{G}[\tilde{\chi}, \tilde{a}] \xrightarrow{F, \leq} G[\chi, a]$$

Process Abstraction: If $E, G[\chi, a] \vdash \lambda_k x. H$, it follows with (TL-PA) that

$$E \setminus x \cup \{x : \eta_x\}, G[\chi \circ \chi_0, a] \vdash_{A,F}^L H$$

Let $n := |\chi|$, $m := |\chi_0|$. Since (TL-PA) demands the existence of a message, a process, a content-edge and because of the restrictions on A in this section it follows that there exists a morphism

$$\phi_R : R[\chi_R] \rightarrow G[\chi \circ \chi_0]$$

where $R[\chi_R, a_R] := A(\text{Red}_{k,m,n}(\lambda_k. H))$ and $s_G(\text{cont}_G(\lfloor \chi \rfloor_k)) = \eta_x(\chi_{\mathbf{m}_x})$.

The induction hypothesis implies that $\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M H$ and that there exists a morphism

$$\hat{\phi} : \hat{G}[\hat{\chi}, \hat{a}] \xrightarrow{F, \leq} G[\chi \circ \chi_0, a]$$

It follows with (TM-PA) that

$$\tilde{G}[\tilde{\chi}, \tilde{a}] := \sigma_{1,\dots,m}(\hat{G}[\hat{\chi}, \hat{a}] \square (R[\chi_R, a_R])^-) \Vdash_{A,F,F}^M \lambda_k.H$$

Let $\eta_1 : \hat{G}[\hat{\chi}] \hookrightarrow \tilde{G}[\tilde{\chi}]$, $\eta_2 : R[\chi_R] \hookrightarrow \tilde{G}[\tilde{\chi}]$ be the canonical embeddings into $\tilde{G}[\tilde{\chi}]$. It follows with the definition of graph construction (definition 2.2.9) that there exists a morphism

$$\psi : \tilde{G}[\tilde{\chi}] \rightarrow G[\chi]$$

with $\psi \circ \eta_1 = \hat{\phi}$, $\psi \circ \eta_2 = \phi_R$. It is now left to show that ψ is a (F, \leq) -morphism:

$$F_\psi(\tilde{a}) = F_\psi(F_{\eta_1}(\hat{a}) \vee F_{\eta_1}(\perp - a_R)) = F_{\phi_1}(\hat{a}) \vee F_{\phi_2}(\perp - a_R) = F_{\phi_1}(\hat{a}) \vee \perp \leq a$$

In a lattice, it holds that $\perp - x := \min\{y \mid x \vee y \geq \perp\} = \perp$.

□

Proposition 8.5.2 (Monoid \rightarrow Lattice) *Let S be a process description or process graph in \mathcal{S}_n without variables whose messages are all labelled $\mathbf{0}$.*

We consider a type system where the linear mapping A satisfies the constraints of the type system based on lattices and where the monoid operations in $F(G) = (I, +, \leq)$ coincide with the supremum, i.e. $+$ $=$ \vee $=$ \oplus .

If $\hat{G}[\hat{\chi}, \hat{a}] \Vdash_{A,F,F}^M S$ it follows that for any environment E for \hat{G} :

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L S$$

Proof: Because of $+$ $=$ \vee $=$ \oplus for every $F(G) = (I, +, \leq)$ it follows that F is compatible with itself.

We will show by induction on S that if $G[\chi, a] \Vdash_{A,F,F}^M S$ and there is a morphism

$$\hat{\phi} : G[\chi, a] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

where \hat{G} is a true type graph it follows that there exists an environment E for \hat{G} with

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L S$$

Replication: If $T \Vdash_{A,F,F}^M !H$ it follows with (TM-REPL) that $G[\chi, a] \Vdash_{A,F,F}^M !H$

where $G[\chi, a] \lesssim T$. If there exists a morphism $\hat{\phi} : T \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$ such that \hat{G} is a true type graph, $\hat{\phi}$ is also a (F, \leq) -morphism from $G[\chi, a]$ into $\hat{G}[\hat{\chi}, \hat{a}]$.

The induction hypothesis implies that $E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L H$ and thus (TL-REPL) implies that

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L !H$$

Process: If $G[\chi, a] \Vdash_{A,F,F}^M \text{proc}_n(S)$ it follows with (TM-PROC) that $G'[\chi', a'] \Vdash_{A,F,F}^M S$ with

$$T \cong A(\text{proc}_n(S)) \sqcap G'[\chi', a'] \text{ and } T \lesssim G[\chi, a]$$

Let $p_P : A(\text{proc}_n(S)) \xrightarrow{F, \leq} G[\chi, a]$ and $p_{G'} : G'[\chi', a'] \xrightarrow{F, \leq} G[\chi, a]$ be the corresponding projections.

If there is a morphism $\hat{\phi} : G[\chi, a] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$ such that \hat{G} is a true type graph, it follows that $\hat{\phi} \circ p_{G'} : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$.

Then the induction hypothesis implies that $E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L S$. And with (TL-PROC) and the existence of p_P it follows that

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L \text{proc}_n(S)$$

Message: Let $A(\text{mess}_{n+1}(\mathbf{0})) \Vdash_{A,F,F}^M \text{mess}_{n+1}(\mathbf{0})$ and there is a morphism

$$\hat{\phi} : A(\text{mess}_{n+1}(\mathbf{0})) \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$$

such that \hat{G} is a true type graph and where $M[\chi_M, a_M] := A(\text{mess}_{n+1}(\mathbf{0}))$.

It follows that $s_{\hat{G}}(\text{cont}_{\hat{G}}(\lfloor \hat{\chi} \rfloor_{n+1})) = \hat{\phi}(s_M(\text{cont}_M(\lfloor \chi_M \rfloor_{n+1}))) = \hat{\phi}(\varepsilon) = \varepsilon$.

Since $E, \hat{G}[\varepsilon, \hat{a}] \vdash_{A,F,F}^L \mathbf{0}$ (see (TL-CON)) it follows with (TL-MESS) that

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash \text{mess}_{n+1}(\mathbf{0})$$

Process Graph: If $G[\chi, a] \Vdash_{A,F,F}^M \bigotimes_{i=1}^n (H_i, \zeta_i)$ where $\zeta_i : \mathbf{m}_i \rightarrow C$ and $\phi_C : C \rightarrow G[\chi]$ it follows with lemma 8.4.5 that there exist type graphs $G_i[\chi_i, a_i]$ with $G_i[\chi_i, a_i] \Vdash_{A,F,F}^M H_i$ and $\bigotimes_{i=1}^n (G_i[\chi_i, a_i], \zeta_i) \lesssim G[\chi, a]$.

Let $\phi_i : G_i[\chi_i, a_i] \xrightarrow{F, \leq} G[\chi'_i, a]$ where $\chi'_i := \phi_i(\chi_i)$ be the projections into $G[\chi, a]$ and let $\psi_i : \mathbf{m}_i \rightarrow G_i[\chi_i]$ the canonical strong morphism.

Furthermore let $\hat{\phi} : G[\chi, a] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$. It follows that $\hat{\phi} \circ \phi_i : G_i[\chi_i, a_i] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}_i, \hat{a}]$ where $\hat{\chi}_i := \hat{\phi}(\phi_i(\chi_i))$. The induction hypothesis implies that $E, \hat{G}[\hat{\chi}_i, \hat{a}] \vdash_{A,F}^L H_i$.

We conclude that

$$\begin{aligned} \hat{\phi}(\phi_C(\chi_C)) &= \hat{\phi}(\chi) = \hat{\chi} \\ \hat{\phi}(\phi_C(\zeta_i(\chi_{\mathbf{m}_i}))) &= \hat{\phi}(\phi_i(\psi_i(\chi_{\mathbf{m}_i}))) = \hat{\phi}(\phi_i(\chi_i)) = \hat{\chi}_i \end{aligned}$$

With rule (TL-CON) and the morphism $\hat{\phi} \circ \phi_C : C \rightarrow \hat{G}[\hat{\chi}]$ it follows that

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L \bigotimes_{i=1}^n (H_i, \zeta_i)$$

Process Abstraction: If $G[\chi, a] \Vdash_{A,F}^M \lambda_k x. H$ it follows that $G'[\chi', a'] \Vdash_{A,F}^M H$ such that

$$\sigma_{1,\dots,m}(G'[\chi', a'] \sqcap A(\text{Red}_{k,m,n}(\lambda_k.H)))^- \lesssim G[\chi, a]$$

where $R[\chi_R, a_R] := A(\text{Red}_{k,m,n}(\lambda_k.H))$. Let

$$\begin{aligned} \phi_P : G'[\chi', a'] &\xrightarrow{F, \leq} G[\chi \circ \chi_0, a] \\ \phi_R : R[\chi_R] &\xrightarrow{F, \leq} G[\chi \circ \chi_0] \end{aligned}$$

Let $\hat{\phi} : G[\chi, a] \xrightarrow{F, \leq} \hat{G}[\hat{\chi}, \hat{a}]$. It follows that $\hat{\phi} \circ \phi_P : G'[\chi', a'] \xrightarrow{F, \leq} \hat{G}[\hat{\chi} \circ \hat{\chi}_0, \hat{a}]$ where $\hat{\chi}_0 := \hat{\phi}(\chi_0)$. It follows with the induction hypothesis that

$$E \setminus x \cup \{x : \eta_x\}, \hat{G}[\hat{\chi} \circ \hat{\chi}_0, \hat{a}] \vdash H$$

where $\eta_x : \mathbf{0} \mapsto \hat{G}$.

There is a morphism

$$\hat{\phi} \circ \phi_R : R[\chi_R] \xrightarrow{F, \leq} \hat{G}[\hat{\chi} \circ \hat{\chi}_0]$$

The properties of A ensure that R contains a process $p' \in P_R$, a message $q' \in M_R$ and a content $c' \in Co_R$ such that

$$\begin{aligned} s_R(p') &= \lfloor \chi_R \rfloor_{1\dots m} \\ s_R(q') &= \lfloor \chi_R \rfloor_{m+1\dots m+n k} \\ s_R(c') &= \text{cont}_R(\lfloor \chi \rfloor_k) \circ \lfloor s_R \rfloor_k = \varepsilon \circ \lfloor \chi_R \rfloor_k \end{aligned}$$

We will now set $p := \hat{\phi}(\phi_R(p'))$, $q := \hat{\phi}(\phi_R(q'))$, $c := \hat{\phi}(\phi_R(c'))$ and show that these hyperedges of \hat{G} satisfy the conditions of typing rule (TL-PA):

$$\begin{aligned} s_{\hat{G}}(p) &= s_{\hat{G}}(\hat{\phi}(\phi_R(p'))) = \hat{\phi}(\phi_R(\lfloor \chi_R \rfloor_{1\dots m})) = \hat{\chi} \\ s_{\hat{G}}(q) &= s_{\hat{G}}(\hat{\phi}(\phi_R(q'))) = \hat{\phi}(\phi_R(\lfloor \chi_R \rfloor_{m+1\dots m+n k})) = \hat{\chi}_0 \circ \lfloor \hat{\chi} \rfloor_k \\ s_{\hat{G}}(c) &= s_{\hat{G}}(\hat{\phi}(\phi_R(c'))) = \hat{\phi}(\phi_R(\varepsilon \circ \lfloor \chi_R \rfloor_k)) = \varepsilon \circ \lfloor \hat{\chi} \rfloor_k = \eta_x(\chi_{\mathbf{m}_x}) \circ \lfloor \hat{\chi} \rfloor_k \end{aligned}$$

This implies that

$$E, \hat{G}[\hat{\chi}, \hat{a}] \vdash_{A,F}^L H$$

□

A.2.4 Comparison of Type Systems: SPIDER \leftrightarrow π -Calculus

Lemma A.2.1

$$\begin{aligned} &E, G[\chi, a] \vdash \Theta_N(((S)_x[t])[t])) \\ \Rightarrow &\lfloor t \rfloor_1 : \text{Tree}_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : \text{Tree}_\sigma^G(\lfloor \chi \rfloor_n) \vdash \Delta'_\pi((S)_x[t]) \end{aligned}$$

implies

$$\begin{aligned} &E, G[\chi', a'] \vdash \Theta_N(((S)_x[t])[t'])) \\ \Rightarrow &\lfloor t' \rfloor_1 : \text{Tree}_\sigma^G(\lfloor \chi' \rfloor_1), \dots, \lfloor t' \rfloor_n : \text{Tree}_\sigma^G(\lfloor \chi' \rfloor_n) \vdash \Delta'_\pi((S)_x[t]) \end{aligned}$$

if $n := |t| = |t'|$, $\text{Set}(t) = \text{Set}(t')$.

Proof: Let $E, G[\chi', a'] \vdash \Theta_N(((S)_x[t])[t'])$.

We have

$$\Theta_N(((S)_x[t])[t']) \equiv \otimes(\Theta_N(((S)_x[t])[t]), \zeta_{t \rightarrow t'})$$

And lemma 8.3.7 implies that there exist $\chi \in V_G^*$, $a \in F(G)$ such that

$$E, G[\chi, a] \vdash \Theta_N(((S)_x[t])[t])$$

Furthermore there exists a strong morphism $\phi : \mathbf{n} \rightarrow G[\chi']$ such that $\chi = \phi(\zeta_{t \rightarrow t'}(\chi_{\mathbf{m}}))$. It follows that $[\chi]_i = [\chi']_j \iff [t]_i = [t']_j$.

The precondition implies that

$$[t]_1 : Tree_\sigma^G([\chi]_1), \dots, [t]_m : Tree_\sigma^G([\chi]_m) \vdash \Delta'_\pi((S)_x[t])$$

and by reordering the type assignments it follows that

$$[t']_1 : Tree_\sigma^G([\chi']_1), \dots, [t']_n : Tree_\sigma^G([\chi']_n) \vdash \Delta'_\pi((S)_x[t])$$

□

Proposition 8.6.2 *Let $E, G[\chi, a] \vdash \Theta_N(h[t])$ and $n := |\chi|$. It follows that*

$$[u]_1 : Tree_\sigma^G([\chi]_1), \dots, [u]_n : Tree_\sigma^G([\chi]_n) \vdash \Delta_\pi^u(S)$$

where u is a duplicate-free string with $|u| = n$.

Proof: We will now show by induction on h that

$$\begin{aligned} & E, G[\chi, a] \vdash \Theta_N(h[t]) \\ \Rightarrow & [t]_1 : Tree_\sigma^G([\chi]_1), \dots, [t]_n : Tree_\sigma^G([\chi]_n) \vdash \Delta'_\pi(h) \end{aligned} \quad (\text{A.9})$$

If we can show this, the proposition follows immediately with $\Delta_\pi^u(h[t]) = \Delta'_\pi(h)[u/t]$:

- If $E, G[\chi, a] \vdash h[t]$ it follows that

$$[t]_1 : Tree_\sigma^G([\chi]_1), \dots, [t]_n : Tree_\sigma^G([\chi]_n) \vdash \Delta'_\pi(h)$$

And with the substitution law **(3)** it follows that

$$[u]_1 : Tree_\sigma^G([\chi]_1), \dots, [u]_n : Tree_\sigma^G([\chi]_n) \vdash \Delta'_\pi(h)[u/t]$$

We will now prove (A.9) by induction on h :

Empty Graph: Let $h := 0$. Then $\Delta'_\pi(h) := 0$, $\Theta_N(0[\varepsilon]) \equiv \mathbf{0}$.

Let $E, G[\varepsilon, a] \vdash \mathbf{0}$. And it follows with (T π -NIL) that

$$\Gamma \vdash 0$$

where Γ is the empty type assignment set, since 0 is always typable.

Node: Let $h := \lceil a \rceil$. Then $\Delta'_\pi(h) := 0$, $\Theta_N(\lceil a \rceil[a]) \equiv \mathbf{1}$.

Let $E, G[\chi, a] \vdash \mathbf{1}$ with $|s| = 1$. And it follows with (T π -NIL) that

$$a : Tree_\sigma^G(\lfloor \chi \rfloor_1) \vdash 0$$

since 0 is always typable.

Message: Let $h = (0)_M[a_1 \dots a_{n+1}]$. Then

$$\Delta'_\pi(h) := \overline{a_{n+1}}(a_1, \dots, a_n)$$

$$\Theta_N(h[a_1 \dots a_{n+1}]) \equiv mess_{n+1}(\mathbf{0})$$

Let $E, G[\chi, a] \vdash mess_{n+1}(\mathbf{0})$. (TL-MeSS) implies that there exists a morphism

$$A(mess_{n+1}(\mathbf{0})) \xrightarrow{F, \leq} G[\chi, a]$$

We will now show that $Tree_\sigma^G(\lfloor \chi \rfloor_{n+1}) = [Tree_\sigma^G(\lfloor s \rfloor_1), \dots, Tree_\sigma^G(\lfloor \chi \rfloor_n)]$:

Since $M[\chi_M, a_M] := A(mess_{n+1}(\mathbf{0}))$ contains a message q with $s_M(q) = \chi_M$ it follows that $s_T(\phi(q)) = \chi$ which implies

$$Tree_\sigma^G(\lfloor \chi \rfloor_{n+1}) = [Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, Tree_\sigma^G(\lfloor \chi \rfloor_n)]$$

If we define $\Gamma := a_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, a_{n+1} : Tree_\sigma^G(\lfloor \chi \rfloor_{n+1})$ it follows that $\Gamma(a_{n+1}) = [\Gamma(a_1), \dots, \Gamma(a_n)]$ and with (T π -OUT) we conclude that

$$\Gamma \vdash \overline{a_{n+1}}a_1 \dots a_n$$

The rest follows with lemma A.2.1

Replication: Let $h = (!h'[t'])[a_1 \dots a_m]$. Then

$$\Delta'_\pi(h) := !\Delta_\pi^{a_1 \dots a_m}(h'[t'])$$

$$\Theta_N(h[a_1 \dots a_m]) \equiv proc_m(!\Theta_N(h'[t']))$$

Let $E, G[\chi, a] \vdash proc_m(!\Theta_N(h'[t']))$. It follows with typing rules (TL-PROC) and (TL-REPL) that $E, G[\chi, a] \vdash \Theta_N(h'[t'])$.

Now the induction hypothesis implies that

$$a_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, a_m : Tree_\sigma^G(\lfloor \chi \rfloor_m) \vdash \Delta_\pi^{a_1 \dots a_m}(h'[t'])$$

And with (T π -REPL) it follows that

$$a_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, a_m : Tree_\sigma^G(\lfloor \chi \rfloor_m) \vdash !\Delta_\pi^{a_1 \dots a_m}(h'[t'])$$

The rest follows with lemma A.2.1

Process Abstraction: Let $h = (\lambda_k.h'[t'])_P[a_1 \dots a_m]$. Then

$$\Delta'_\pi(h) := a_k(x_1 \dots x_n). \Delta_\pi^{a_1 \dots a_m x_1 \dots x_n}(h(t))$$

where $n := |t| - m$

$$\Theta_N(h[a_1 \dots a_m]) \equiv \text{proc}_m(\lambda_k. \Theta_N(h'[t']))$$

Let

$$E, G[\chi, a] \vdash \text{proc}_m(\lambda_k. \Theta_N(h'[t']))$$

It follows with typing rules (TL-PROC) and (TL-PA) that

$$E \setminus x \cup \{x : \eta_x\}, G[\chi \circ \chi_0, a] \vdash \Theta_N(h'[t'])$$

and there exists a message $q \in M_G$ with $s_G(q) = \chi_0 \circ [\chi]_k$.

Now the induction hypothesis implies that

$$\begin{aligned} a_1 : \text{Tree}_\sigma^G([\chi]_1), \dots, a_m : \text{Tree}_\sigma^G([\chi]_m), \\ x_1 : \text{Tree}_\sigma^G([\chi_0]_1), \dots, x_n : \text{Tree}_\sigma^G([\chi_0]_n) \vdash \Delta_\pi^{a_1 \dots a_m x_1 \dots x_n}(h'[t']) \end{aligned}$$

Let

$$\begin{aligned} \Gamma := & a_1 : \text{Tree}_\sigma^G([\chi]_1), \dots, a_m : \text{Tree}_\sigma^G([s]_m), \\ & x_1 : \text{Tree}_\sigma^G([\chi_0]_1), \dots, x_n : \text{Tree}_\sigma^G([\chi_0]_n) \end{aligned}$$

And since $s_G(q) = \chi_0 \circ [\chi]_k$ it follows that

$$\text{Tree}_\sigma^G([\chi]_k) = [\text{Tree}_\sigma^G([\chi_0]_1), \dots, \text{Tree}_\sigma^G([\chi_0]_n)]$$

and therefore $\Gamma(a_k) = [\Gamma(x_1), \dots \Gamma(x_n)]$.

It follows with (T π -IN) that

$$\begin{aligned} a_1 : \text{Tree}_\sigma^G([\chi]_1), \dots, a_m : \text{Tree}_\sigma^G([\chi]_m) \\ \vdash a_k(x_1, \dots, x_n). \Delta_\pi^{a_1 \dots a_m x_1 \dots x_n}(h'[t']) \end{aligned}$$

The rest follows with lemma A.2.1

Parallel Composition: Let $h = h_1|h_2$. Then

$$\Delta'_\pi(h) := \Delta'_\pi(h_1) | \Delta'_\pi(h_2)$$

$$\Theta_N(h[t]) \equiv (\Theta_N(h_1[t_1]), \zeta_{t_1 \rightarrow t}) \otimes (\Theta_N(h_2[t_2]), \zeta_{t_2 \rightarrow t}) =: H$$

where $t_1 := t \setminus (\text{Set}(t_2) \setminus \text{Set}(t_1))$ and $t_2 := t \setminus (\text{Set}(t_1) \setminus \text{Set}(t_2))$.

Let $n := |t|$, $n_i := |t_i|$.

Let $E, G[\chi, a] \vdash H$. It follows with lemma 8.3.7 that $E, G[\chi_i, a_i] \vdash \Theta_N(h_1[t_1])$ where $\phi : \mathbf{n} \rightarrow G[\chi]$ is a strong morphism such that $\chi_i = \phi(\zeta_{t_i \rightarrow t}(\chi_{\mathbf{n}}))$.

The induction hypothesis implies that

$$\lfloor t_i \rfloor_1 : Tree_\sigma^G(\lfloor \chi_i \rfloor_1), \dots, \lfloor t_i \rfloor_{n_i} : Tree_\sigma^G(\lfloor \chi_i \rfloor_{n_i}) \vdash \Delta'_\pi(h_i)$$

Furthermore

$$\begin{aligned} \lfloor \chi_i \rfloor_j = \lfloor \chi \rfloor_k &\iff \lfloor \phi(\zeta_{t_i \mapsto t}(\chi_{\mathbf{n}_i})) \rfloor_j = \lfloor \phi(\chi_{\mathbf{n}}) \rfloor_k \\ \xLeftrightarrow[\phi \text{ inj.}] \lfloor \zeta_{t_i \mapsto t}(\chi_{\mathbf{n}_i}) \rfloor_j = \lfloor \chi_{\mathbf{n}} \rfloor_k &\iff \lfloor t_i \rfloor_j = \lfloor t \rfloor_k \end{aligned}$$

And with the weakening law (4) it follows that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n) \vdash \Delta'_\pi(h_i)$$

And (T π -PAR) implies that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n) \vdash \Delta'_\pi(h_1) | \Delta'_\pi(h_2) = \Delta'_\pi(h_1 | h_2)$$

Hiding: Let $h = (\nu b)h'$. Then

$$\begin{aligned} \Delta'_\pi(h) &:= (\nu b)\Delta'_\pi(h') \\ \Theta_N(h[t]) &\equiv \begin{cases} \otimes(val_n(h'[t \circ b]), \zeta_{t \mapsto tob}) & \text{if } b \in fn(h') \\ val_n(h'[t]) & \text{otherwise} \end{cases} \end{aligned}$$

Let $E, G[\chi, a] \vdash \Theta_N(h[t])$. We will now distinguish the following two cases:

- $b \in fn(h')$, $n := |t|$.

In this case $\Theta_N(h[t]) \equiv \otimes(\Theta_N(h'[t \circ b]), \zeta)$ where ζ is the projection of $\mathbf{n} + \mathbf{1}$ into $\sigma_{1..n}(\mathbf{n} + \mathbf{1})$.

It follows with lemma 8.3.7 that $E, G[\chi', a'] \vdash \Theta_N(h'[t \circ b])$ where $\phi : \mathbf{n} \rightarrow G[\chi]$ is a strong morphism such that $\chi' = \phi(\zeta(\chi_{\mathbf{n}+1}))$ which implies $\lfloor \chi \rfloor_i = \lfloor \chi' \rfloor_i$ if $i \in \{1, \dots, n\}$.

The induction hypothesis implies that

$$\begin{aligned} \lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^T(\lfloor \chi \rfloor_n), b : Tree_\sigma^G(\lfloor \chi' \rfloor_{n+1}) \\ \vdash \Delta'_\pi(h') \end{aligned}$$

It follows with (T π -RESTR) that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n) \vdash (\nu b)\Delta'_\pi(h')$$

- $b \notin fn(h')$, $n := |t|$.

In this case $\Theta_N(h[t]) \equiv \Theta_N(h'[t])$ and therefore $E, G[\chi, a] \vdash \Theta_N(h'[t])$.

The induction hypothesis implies that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n) \vdash \Delta'_\pi(h')$$

With the weakening law (4) it follows that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n), b : \beta \vdash \Delta'_\pi(h')$$

where β is a new variable.

And it follows with (T π -RESTR) that

$$\lfloor t \rfloor_1 : Tree_\sigma^G(\lfloor \chi \rfloor_1), \dots, \lfloor t \rfloor_n : Tree_\sigma^G(\lfloor \chi \rfloor_n) \vdash (\nu b)\Delta'_\pi(h')$$

□

Lemma 8.6.3 *Let G_1, \dots, G_n be true graphs (where all content-edges have cardinality 0), let \sim be an equivalence on the nodes of the type graphs and let the σ_i be mappings satisfying*

$$v_i \in V_{G_i}, v_j \in V_{G_j}, (v_i, i) \sim (v_j, j) \Rightarrow Tree_{\sigma_i}^{G_i}(v_i) = Tree_{\sigma_j}^{G_j}(v_j) \quad (8.22)$$

Then there is a smallest consistent equivalence \approx containing \sim such that $G_1 \dots G_n / \approx$ is a true graph.

Let $p_i : G_i \rightarrow G_1 \dots G_n / \approx$ be the i -th projection into the quotient graph. We define $\sigma(p_i(v_i)) := \sigma_i(v_i)$. Then σ is well-defined and:

$$Tree_{\sigma}^{G_1 \dots G_n / \approx}(p_i(v_i)) = Tree_{\sigma_i}^{G_i}(v_i)$$

for all $v_i \in V_{G_i}$.

Proof:

- We will first show that $(v_i, i) \approx (v_j, j)$ implies $Tree_{\sigma_i}^{G_i}(v_i) = Tree_{\sigma_j}^{G_j}(v_j)$:
Let $\approx_0 := \sim$ and let $\approx_{k+1} \supseteq \approx_k$ be the smallest equivalence satisfying

$$\begin{aligned} \forall k \in \{1, \dots, \text{card}(p_i)\} : \\ (\lfloor s_{G_i}(p_i) \rfloor_k, i) \approx_k (\lfloor s_{G_j}(p_j) \rfloor_k, j) &\Rightarrow (p_i, i) \approx_{k+1} (p_j, j) \\ (\text{send}_{G_i}(q_i), i) \approx_k (\text{send}_{G_j}(q_j), j) &\Rightarrow (q_i, i) \approx_{k+1} (q_j, j) \\ (\text{send}_{G_i}(c_i), i) \approx_k (\text{send}_{G_j}(c_j), j) &\Rightarrow (c_i, i) \approx_{k+1} (c_j, j) \\ (q_i, i) \approx_k (q_j, j) &\Rightarrow (\lfloor s_{G_i}(q_i) \rfloor_l, i) \approx_{k+1} (\lfloor s_{G_j}(q_j) \rfloor_l, j) \end{aligned}$$

where $p_i, q_i, c_i \in E_{G_i}$, $z_{G_i}(p_i) = \text{proc}$, $z_{G_i}(q_i) = \text{mess}$, $z_{G_i}(c_i) = \text{cont}$.

It follows that $\approx = \bigcup_{i=0}^{\infty} \approx_i$.

We can easily show by induction on k that $(v_i, i) \approx_k (v_j, j)$ implies $Tree_{\sigma_i}^{G_i}(v_i) = Tree_{\sigma_j}^{G_j}(v_j)$.

Furthermore if $(q_i, i) \approx_k (q_j, j)$ it follows that

$$\begin{aligned} \text{card}(q_i) &= \text{card}(Tree_{\sigma_i}^{G_i}(\text{send}_{G_i}(q_i))) \\ &= \text{card}(Tree_{\sigma_j}^{G_j}(\text{send}_{G_j}(q_j))) = \text{card}(q_j) \end{aligned}$$

- We will now define the operation cut_k where $k \in \mathbb{N}$.

$$\begin{aligned} cut_0(tr) &:= [] \\ cut_{k+1}([tr_1, \dots, tr_n]) &:= [cut_k(tr_1), \dots, cut_k(tr_n)] \\ cut_{k+1}(\alpha) &:= \alpha \end{aligned}$$

That is cut_k reduces a type tree to its upper k levels.

By induction on k we will show that

$$cut_k(Tree_{\sigma}^G(p_i(v_i))) = cut_k(Tree_{\sigma_i}^{G_i}(v_i))$$

Then the equality holds also for entire type trees.

$k = 0$: obvious

$k \rightarrow k + 1$: there are two cases:

- If $Tree_\sigma^G(p_i(v_i)) = \alpha$ it follows that $\sigma(p_i(v_i)) = \alpha$ which implies that $\sigma_i(v_i) = \alpha$. And thus $Tree_{\sigma_i}^{G_i}(v_i) = \alpha$.
- If $cut_k(Tree_\sigma^G(p_i(v_i))) = [tr_1, \dots, tr_n]$ it follows that there exists a message q in G with $s_G(q) = s \circ p_i(v_i)$. This implies that there exists a message q_j in G_j with $s_{G_j} = s_j \circ v_j$, $p_j(s_j \circ v_j) = s \circ p_i(v_i)$ and $p_j(q_j) = q$. $|s_j| = |s| =: n$.

Since $p_i(v_i) = p_j(v_j)$ and therefore $(v_i, i) \approx (v_j, j)$ and

$$\begin{aligned} Tree_\sigma^G(p_i(v_i)) &= Tree_\sigma^G(p_j(v_j)) \\ &= [Tree_\sigma^G(p_j(\lfloor s_j \rfloor_1)), \dots, Tree_\sigma^G(p_j(\lfloor s_j \rfloor_n))] \end{aligned}$$

it follows that

$$\begin{aligned} & cut_k(Tree_\sigma^G(p_i(v_i))) \\ &= [cut_{k-1}(Tree_\sigma^G(p_j(\lfloor s_j \rfloor_1))), \dots, cut_{k-1}(Tree_\sigma^G(p_j(\lfloor s_j \rfloor_n)))] \\ \text{Ind.hyp} \quad &= [cut_{k-1}(Tree_{\sigma_j}^{G_j}(\lfloor s_j \rfloor_1)), \dots, cut_{k-1}(Tree_{\sigma_j}^{G_j}(\lfloor s_j \rfloor_n))] \\ &= cut_k(Tree_{\sigma_j}^{G_j}(v_j)) = cut_k(Tree_{\sigma_i}^{G_i}(v_i)) \end{aligned}$$

□

Lemma A.2.2 Let $\Gamma \vdash p$ where $\Theta_\pi(p) = (S)_x[t]$ with $n := |t|$. Let t' be a duplicate-free string such that $Set(t') = Set(t) \subseteq Set(\Gamma)$.

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^t(p)) \wedge \forall i \in \{1, \dots, n\} : Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor t \rfloor_i)$$

implies that there exists a type graph $G'[\chi', a']$ and a type environment E' with

$$E', G'[\chi', a'] \vdash \Theta_N(\Theta_\pi^{t'}(p)) \wedge \forall i \in \{1, \dots, n\} : Tree_{\sigma'}^{G'}(\lfloor \chi' \rfloor_i) = \Gamma(\lfloor t' \rfloor_i)$$

Proof: We can show by induction on p that $\Theta_N(\Theta_\pi^{t'}(p)) \equiv \otimes(\Theta_N(\Theta_\pi^t(p)), \zeta_{t \rightarrow t'})$.

We define $G' := G$, $a' := a$ and χ' is defined such that

$$\lfloor \chi' \rfloor_i = \lfloor \chi \rfloor_j \iff \lfloor t' \rfloor_i = \lfloor t \rfloor_j$$

Since $\phi : \mathbf{n} \rightarrow G[\chi']$ and $\phi(\zeta_{t \rightarrow t'}(\chi_{\mathbf{n}})) = \chi$ it follows with (TL-CON) that

$$E, G'[\chi', a'] \vdash \Theta_N(\Theta_\pi^{t'}(p))$$

Let i be a fixed natural number. Then there exists a natural number j with $\lfloor t' \rfloor_i = \lfloor t \rfloor_j$. Therefore

$$Tree_{\sigma'}^{G'}(\lfloor \chi' \rfloor_i) = Tree_\sigma^G(\lfloor \chi \rfloor_j) = \Gamma(\lfloor t \rfloor_j) = \Gamma(\lfloor t' \rfloor_i)$$

□

Proposition 8.6.4 *Let $\Gamma \vdash p$ and let t be a duplicate-free string such that $fn(p) = Set(t) \subseteq Set(\Gamma)$.*

Then there exists a true type graph $G[\chi, a]$, a type environment E and a mapping σ such that

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^t(p))$$

and $Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor t \rfloor_i)$.

Proof: The environment E will contain only embeddings of the form $\eta_x : \mathbf{0} \mapsto G$ and all content-edges in G have cardinality 0. We proceed by induction on p :

Dead Process: $\Gamma \vdash 0$. Then $t = \varepsilon$ and $\Theta_N(\Theta_\pi^\varepsilon(p)) = \mathbf{0}$.

We define $G[\chi, a] := \mathbf{0}[\perp]$, $E := \emptyset$.

It follows with (TL-CON) that $E, G[\chi, a] \vdash \mathbf{0}$.

Parallel Composition: $\Gamma \vdash p_1 | p_2$ which implies that $\Gamma \vdash p_i$.

$$\Theta_N(\Theta_\pi^t(p_1 | p_2)) \equiv (\Theta_\pi^{t_1}(p_1), \zeta_{t_1 \mapsto t}) \otimes (\Theta_\pi^{t_2}(p_2), \zeta_{t_2 \mapsto t})$$

where $t_1 := t \setminus (Set(t_2) \setminus Set(t_1))$, $t_2 := t \setminus (Set(t_1) \setminus Set(t_2))$, $n_i := |t_i|$, $n := |t|$.

It follows with the induction hypothesis that there exist $E_i, G_i[\chi_i, a_i]$ such that

$$E_i, G_i[\chi_i, a_i] \vdash \Theta_N(\Theta_\pi^{t_i}(p_i))$$

and $Tree_{\sigma_i}^{G_i}(\lfloor \chi_i \rfloor_j) = \Gamma(\lfloor t_i \rfloor_j)$.

We will define an equivalence \sim on G_1, G_2 with

$$\begin{aligned} (\lfloor \chi_1 \rfloor_i, 0) \sim (\lfloor \chi_2 \rfloor_j, 1) &\iff \lfloor t_1 \rfloor_i = \lfloor t_2 \rfloor_j \\ &\iff \zeta_{t_1 \mapsto t}(\lfloor \chi_{\mathbf{n}_1} \rfloor_i) = \zeta_{t_2 \mapsto t}(\lfloor \chi_{\mathbf{n}_2} \rfloor_j) \end{aligned}$$

Since $\lfloor t_1 \rfloor_i = \lfloor t_2 \rfloor_j$ implies $\Gamma(\lfloor t_1 \rfloor_i) = \Gamma(\lfloor t_2 \rfloor_j)$ it follows that $\lfloor \chi_1 \rfloor_i \sim \lfloor \chi_2 \rfloor_j$ implies $Tree_{\sigma_1}^{G_1}(\lfloor \chi_1 \rfloor_i) = \Gamma(\lfloor t_1 \rfloor_i) = \Gamma(\lfloor t_2 \rfloor_j) = Tree_{\sigma_2}^{G_2}(\lfloor \chi_2 \rfloor_j)$.

It follows with lemma 8.6.3 that $G := G_1 G_2 / \approx$ is defined and G is a true graph. Let $p_i : G_i \rightarrow G$ be the projections into the quotient graph.

Let $\chi \in V_G^*$ with $|\chi| = n$ and $\lfloor \chi \rfloor_j = p_i(\lfloor \chi_i \rfloor_k) \iff \lfloor t \rfloor_j = \lfloor t_i \rfloor_k$.

It follows that

$$Tree_\sigma^G(\lfloor \chi \rfloor_j) = Tree_\sigma^G(p_i(\lfloor \chi_i \rfloor_k)) = Tree_{\sigma_i}^{G_i}(\lfloor \chi_i \rfloor_k) = \Gamma(\lfloor t_i \rfloor_k) = \Gamma(\lfloor t \rfloor_j)$$

It follows with proposition 8.3.18 that

$$p_i(E_i), G[p_i(\chi_i), F_{p_i}(a_i)] \vdash \Theta_N(\Theta_\pi^t(p_i))$$

Because of the special form of the embeddings in E_i (only embeddings of the form $p_x : \mathbf{0} \mapsto G$) $E := p_1(E_1) \cup \zeta_2(E_2)$ is always defined.

If $\phi : \mathbf{n} \rightarrow G[\chi]$ is the canonical strong morphism it follows that $\phi(\zeta_{t_i \rightarrow t}(\chi_{\mathbf{n}_i})) = \chi_i$. With the weakening law (lemma 8.3.5) and (TL-CON) it follows that

$$E, G[\chi, \bigvee_{i=1}^2 F_{\zeta_{t_i \rightarrow t}}(a_i)] \vdash \Theta_N(\Theta_\pi^t(p_1|p_2))$$

Since G_1, G_2 contain only content edges of cardinality 0, the same is true for G .

Replication: $\Gamma \vdash !p$ which implies $\Gamma \vdash p$.

$$\Theta_N(\Theta_\pi^{fn_p}(!p)) \equiv proc_n(!\Theta_N(\Theta_\pi^{fn_p}(p)))$$

With the induction hypothesis it follows that

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^{fn_p}(p))$$

where $Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor fn_p \rfloor_i)$. (TL-REP) implies that

$$E, G[\chi, a] \vdash !\Theta_N(\Theta_{fn_p}(p))$$

Let $P[\chi_P, a_P] := A(proc_n(!\Theta_N(\Theta_{fn_p}(p))))$. Let \sim be the smallest equivalence with $(\lfloor \chi \rfloor_j, 0) \sim (\lfloor \chi_P \rfloor_j, 1)$ for $j \in \{1, \dots, |\chi|\}$.

$P[\chi_P]$ is equivalent to $proc_n$ and we define $\sigma_P(\lfloor \chi_P \rfloor_i) := Tree_\sigma^G(\lfloor \chi \rfloor_i)$. Therefore the preconditions of lemma 8.6.3 are satisfied and it follows that $G' := GP/\approx$ is defined and that G' is a true graph. Furthermore let σ' be the resulting function.

Let $p_G : G \rightarrow G'$ and $p_P : P \rightarrow G'$ be the projections into the quotient graph. We define $\chi' := p_G(\chi) = p_P(\chi_P)$.

And

$$Tree_{\sigma'}^{G'}(\lfloor \chi' \rfloor_i) = Tree_{\sigma'}^{G'}(p_G(\lfloor \chi \rfloor_i)) = Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor fn_p \rfloor_i)$$

It follows with proposition 8.3.18 that

$$p_G(E), G'[\chi', F_{p_G}(a)] \vdash !\Theta_N(\Theta_\pi^{fn_p}(p))$$

And because of the existence of p_P it follows with (TL-PROC) that

$$p_G(E), G'[\chi', F_{p_G}(a) \vee F_{p_P}(a_P)] \vdash proc_n(!\Theta_N(\Theta_\pi^{fn_p}(p)))$$

The rest follows with lemma A.2.2.

Restriction: Let $\Gamma \vdash (\nu b)p$ which implies $\Gamma, b : tr \vdash p$. We will now distinguish the following two cases:

- $b \in fn(p)$. Then

$$\Theta_N(\Theta_\pi^t((\nu b)p)) \equiv \otimes(\Theta_N(\Theta_\pi^{tob}(p)), \zeta)$$

where ζ is the projection of $\mathbf{n} + \mathbf{1}$ into $\sigma_{1\dots n}(\mathbf{n} + \mathbf{1})$ and $n := |t|$.

With the induction hypothesis it follows that

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^{tob}(p))$$

where $Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor t \rfloor_i)$ for $i \in \{1, \dots, n\}$ and $Tree_\sigma^G(\lfloor \chi \rfloor_{n+1}) = \Gamma(b)$.

We define $G'[\chi', a'] := \otimes(G[\chi, a], \zeta)$ and let $p : G[\chi] \rightarrow G'[\chi']$ be the embedding of G into G' and let $\phi : \sigma_{1\dots n}(\mathbf{n} + \mathbf{1}) \rightarrow G'[\chi']$ be the strong morphism created by the co-limit.

It follows with proposition 8.3.18 that

$$p(E), G'[p(\chi), F_p(a)] \vdash \Theta_N(\Theta_\pi^{tob}(p))$$

Since $p(\chi) = \phi(\zeta(\chi_{\mathbf{n}+1}))$ it follows with (TL-CON) that

$$p(E), G'[\chi', F_p(a)] \vdash \otimes(\Theta_N(\Theta_\pi^{tob}(p)), \zeta)$$

and $Tree_\sigma^{G'}(\lfloor \chi' \rfloor_i) = Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor t \rfloor_i)$

- $b \notin fn(p)$. Then

$$\Theta_N(\Theta_\pi^t((\nu b)p)) \equiv \Theta_N(\Theta_\pi^t(p))$$

And with the induction hypothesis it follows that

$$E, G[\chi, a] \vdash \Theta_N(\Theta_\pi^t(p))$$

where $Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor t \rfloor_i)$ for $i \in \{1, \dots, n\}$.

Input Prefix: Let $\Gamma \vdash b(x_1 \dots x_n).p$ which implies that

$$\Gamma, x_1 : tr_1, \dots, x_n : tr_n \vdash p$$

and $\Gamma(b) = [tr_1, \dots, tr_n]$

Let $m := |fn_p|$. And let k be a natural number such that $\lfloor fn_p \rfloor_k = b$.

$$\Theta_N(\Theta_\pi^{fn_p}(b(x_1 \dots x_n).p)) \equiv proc_m(\lambda_k. \Theta_N(\Theta_\pi^{fn_p x_1 \dots x_n}(p)))$$

It follows with the induction hypothesis that

$$E, G[\chi \circ \chi_0, a] \vdash \Theta_N(\Theta_\pi^{fn_p x_1 \dots x_n}(p))$$

where $|\chi| = m$, $|\chi_0| = n$, $Tree_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor fn_p \rfloor_i)$ for $i \in \{1, \dots, m\}$ and $Tree_\sigma^G(\lfloor \chi_0 \rfloor_i) = tr_i$ for $i \in \{1, \dots, n\}$.

We define

$$\begin{aligned} P[\chi_P, a_P] &:= \text{proc}_m[\perp] \\ M[\chi_M, a_M] &:= \text{mess}_{n+1}[\perp] \\ Co[\chi_{Co}, a_{Co}] &:= \text{cont}_1[\perp] \end{aligned}$$

Let \sim be the smallest equivalence on G, P, M, Co satisfying

$$\begin{aligned} (\lfloor \chi \rfloor_i, 0) &\sim (\lfloor \chi_P \rfloor_i, 1) \quad i \in \{1, \dots, m\} \\ (\lfloor \chi_0 \rfloor_i, 0) &\sim (\lfloor \chi_M \rfloor_i, 2) \quad i \in \{1, \dots, n\} \\ (\lfloor \chi \rfloor_k, 0) &\sim (\lfloor \chi_M \rfloor_{n+1}, 2) \\ (\lfloor \chi \rfloor_k, 0) &\sim (\lfloor \chi_{Co} \rfloor_1, 3) \end{aligned}$$

and let

$$\begin{aligned} \sigma_P(\lfloor \chi_P \rfloor_i) &:= \text{Tree}_\sigma^G(\lfloor \chi \rfloor_i) \text{ if } i \in \{1, \dots, m\} \\ \sigma_M(\lfloor \chi_M \rfloor_i) &:= \text{tr}_i \text{ if } i \in \{1, \dots, n\} \end{aligned}$$

The preconditions of lemma 8.6.3 are satisfied and it follows that $G' := GPMC o / \approx$ is defined and that G' is a true graph. Furthermore let σ' be the resulting function.

Let $p_G : G \rightarrow G'$, $p_P : P \rightarrow G$, $p_M : M \rightarrow G$, $p_{Co} : Co \rightarrow G$ be the projections into the quotient graph. We define $\chi' := p_G(\chi)$, $\chi'_0 := p_G(\chi_0)$.

It follows with proposition 8.3.18 that

$$p_G(E), G'[\chi' \circ \chi'_0, F_{p_G}(a)] \vdash \Theta_N(\Theta_\pi^{fn_p x_1 \dots x_n}(p))$$

The projections of the process, message and content-edge in P, M respectively Co satisfy the conditions of rule (TL-PA). Therefore

$$p_G(E), G'[\chi', F_{p_G}(a) \vee F_{p_P}(a_P)] \vdash \lambda_k. \Theta_N(\Theta_\pi^{fn_p x_1 \dots x_n}(p))$$

And because of p_P it follows with (TL-PROC) that

$$p_G(E), G'[\chi', F_{p_G}(a)] \vdash \text{proc}_m(\lambda_k. \Theta_N(\Theta_\pi^{fn_p x_1 \dots x_n}(p)))$$

Furthermore lemma 8.6.3 implies that

$$\text{Tree}_{\sigma'}^{G'}(\lfloor \chi' \rfloor_i) = \text{Tree}_\sigma^G(\lfloor \chi \rfloor_i) = \text{Tree}_\sigma^G(\lfloor \chi \rfloor_i) = \Gamma(\lfloor fn_p \rfloor_i)$$

The rest follows with lemma A.2.2.

Output Prefix: Let $\Gamma \vdash \bar{b}a_1 \dots a_n$ which implies that $\Gamma(b) = [\Gamma(a_1), \dots, \Gamma(a_n)]$.

$$\Theta_N(\Theta_\pi^{a_1 \dots a_n b}(\bar{b}a_1 \dots a_n)) \equiv \text{mess}_{n+1}(\mathbf{0})$$

We define

$$G[\chi, a] := A(\text{mess}_{n+1}(\mathbf{0}))$$

Let \sim be the identity on G , i.e. $(v, 0) \sim (v', 0) \iff v = v'$.

$$\sigma(\lfloor \chi \rfloor_i) := \Gamma(a_i) \text{ if } i \in \{1, \dots, n\}$$

The the preconditions of lemma 8.6.3 are satisfied and it follows that $G' := G/\approx$ is defined and that G' is a true graph. Furthermore let σ' be the resulting function.

Let $p_G : G \rightarrow G$, be the projection into the quotient graph. We define $\chi' := p_G(\chi)$. It follows with (TL-MESS) that

$$E, G'[\chi', F_{p_G}(a)] \vdash \text{mess}_{n+1}(\mathbf{0})$$

for any type environment E .

Furthermore lemma 8.6.3 implies that

$$\begin{aligned} & \text{Tree}_{\sigma'}^{G'}(\lfloor \chi' \rfloor_i) = \text{Tree}_{\sigma}^G(\lfloor \chi \rfloor_i) \\ = & \begin{cases} \Gamma(a_i) & \text{if } i \in \{1, \dots, n\} \\ [\Gamma(a_1), \dots, \Gamma(a_n)] = \Gamma(b) & \text{if } i = n + 1 \end{cases} \end{aligned}$$

The rest follows with lemma A.2.2.

□

Bibliography

- [Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, pages 611–638. Springer-Verlag, 1997.
- [AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
- [ACS96] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π -calculus. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 147–162. Springer-Verlag, 1996. LNCS 1119.
- [AG97] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *CONCUR'97*, pages 59–73. Springer-Verlag, jul 1997.
- [AGN95] S. Abramsky, S.J. Gay, and R. Nagarajan. Interaction categories and foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School, NATO ASI Series F: Computer and System Sciences*. Springer-Verlag, 1995.
- [Apt90] K.R. Apt. Logic programming. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 493–573. Elsevier, 1990.
- [Bar84] Henk P. Barendregt. *The Lambda Calculus—its Syntax and Semantics*, volume 103 of *Studies in Logic and Foundations of Mathematics*. North-Holland, 1984.
- [Bar90] Henk P. Barendregt. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 321–364. Elsevier, 1990.
- [BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
- [BC87] Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20:83–127, 1987.

- [BDG88] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1988.
- [Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, third edition, 1967.
- [Bou89] G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings of TAPSOFT '89*, pages 149–161. Springer-Verlag, 1989. LNCS 351.
- [Bou97] Gérard Boudol. The pi-calculus in direct style. In *POPL '97*, pages 228–241. ACM Press, 1997.
- [BS93] K. Barthelmann and G. Schied. Graph-grammar semantics of a higher-order programming language for distributed systems. In H.J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 71–85. Springer-Verlag, 1993. LNCS 776.
- [Cro93] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
- [Ehr79] 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.
- [FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, 1996.
- [FGL⁺96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 406–421. Springer-Verlag, 1996. LNCS 1119.
- [Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π -calculus. In *ACM Symposium on Principles of Programming Languages '93*, 1993.
- [Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

- [HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ -Calculus*. London Mathematical Society, 1986. Student Texts 1.
- [JR90] D. Janssens and G. Rozenberg. Graph grammar-based description of object-based systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 341–404. Springer-Verlag, 1990. LNCS 489.
- [KLG93] Simon M. Kaplan, Joseph P. Loyall, and Steven K. Goering. Specifying concurrent languages and systems with Δ -grammars. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 9. MIT Press, Cambridge, Massachusetts, 1993.
- [Kob97] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Twelfth Annual Symposium on Logic in Computer Science (LICS) (Warsaw, Poland)*, pages 128–139. IEEE, Computer Society Press, 1997. Full version as as Technical Report 97-02, University of Tokyo.
- [Laf90] Yves Lafont. Interaction nets. In *POPL '90*, pages 95–108. ACM Press, 1990.
- [Laf97] Yves Lafont. Interaction combinators. *Information and Computation*, 1997. Accepted for publication in *Information and Computation*. Final manuscript received for publication March 26, 1997.
- [Loy92] Joseph Patrick Loyall. *Specification of Concurrent Systems Using Graph Grammars*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1992. Report No. UIUCDCS-R-92-1752.
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. LNCS 92.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Mil90] 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.
- [Mil91] Robin Milner. The polyadic π -calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, Laboratory for Foundations of Computer Science, 1991.
- [Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

- [Mil93] Robin Milner. Higher-order action calculi. In *Computer Science Logic*, pages 238–260. Springer-Verlag, 1993. LNCS 832.
- [Mil94] Robin Milner. Pi-nets: a graphical form of pi-calculus. In *European Symposium on Programming*, pages 26–42. Springer-Verlag, 1994. LNCS 788.
- [Mil96] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.
- [MPW89a] 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.
- [MPW89b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part II. Tech. Rep. ECS-LFCS-89-86, University of Edinburgh, Laboratory for Foundations of Computer Science, 1989.
- [MS92a] R. Milner and D. Sangiorgi. Techniques of weak bisimulation up-to. In *CONCUR '92*. Springer-Verlag, 1992. LNCS 630, (Revised Version).
- [MS92b] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. of 19-the International Colloquium on Automata, Languages and Programming (ICALP '92)*. Springer Verlag, 1992. LNCS 623.
- [NS97] Uwe Nestmann and Martin Steffen. Typing confluence. In *Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems (Cesena, Italy, July 4–5, 1997)*, pages 77–101, 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.
- [Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of Logics in Computer Science, LICS '93*, pages 376–385, 1993.
- [PV98] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS '98*, 1998.
- [Rei80] W. Reisig. A graph grammar representation of nonsequential processes. In H. Noltemeier, editor, *Graphtheoretic Concepts in Computer Science*, pages 318–325. Springer-Verlag, 1980. LNCS 100.

- [Rei85] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.
- [Rei98] Boris Reichel. *Verteilte Auswertung attributierter Graphersetzungssysteme zur Verarbeitung massiver, graphartig strukturierter Daten*. PhD thesis, Technische Universität München, 1998.
- [RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Annual International Colloquium on Automata, Languages and Programming, all*, volume 24, 1997.
- [RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *25th Annual Symposium on Principles of Programming Languages (POPL) (San Diego, CA)*. ACM, 1998.
- [Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
- [RV97] Antonio Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Euro-Par '97*. Springer-Verlag, 1997.
- [San92] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992. CST-99-93.
- [San96] Davide Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.
- [Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massachusetts, 1977.
- [SW98] Davide Sangiorgi and David Walker. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, INRIA, 1998.
- [Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, Technische Universität Berlin, 1996.
- [Tho89a] Bent Thomsen. A calculus of higher order communicating systems. *Proceedings 16th Annual Symposium on Principles of Programming Languages*, pages 143–154, 1989.
- [Tho89b] Bent Thomsen. Plain CHOCS. Tech. Rep. DOC 89/4, Department of Computing, Imperial College London, 1989.
- [Tho95] Bent Thomsen. A theory of higher order communicating systems. *Information and Computation*, 116:38–57, 1995.

- [Tur95] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. ECS-LFCS-96-345.
- [Urz95] P. Urzyczyn. Positive recursive type assignment. In J. Wiedermann and P. Hájek, editors, *Mathematical Foundations of Computer Science 1995*. Springer-Verlag, 1995. LNCS 969.
- [Vas94] Vasco T. Vasconcelos. Predicative polymorphism in π -calculus. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *Proceedings of PARLE '94*, pages 425–439. Springer-Verlag, 1994. LNCS 817.
- [VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π -calculus. In *Proceedings of CONCUR '93*, pages 524–538. Springer-Verlag, 1993. LNCS 715.
- [Yos94] Nobuko Yoshida. Graph notation for concurrent combinators. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming, International Workshop TPPP '94*, pages 393–412. Springer-Verlag, 1994. LNCS 907.
- [Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Proceedings of FST/TCS'16*, pages 371–386. Springer-Verlag, 1996.

Index

- $\bigotimes_{i=1}^n (H_i, \zeta_i)$, 26
- $\bigotimes_{i=1}^n (T_i, \zeta_i)_F$, 120
- \oplus , 36, 117
- σ_α , 36
- $\sigma_{i_1 \dots i_n}$, 40
- θ_δ , 36
- $\theta_{i,j}$, 41
- $\zeta_{s \rightarrow t}$, 42
- \wedge , 116
- \vee , 116
- $+$, 116
- \perp , 116
- \simeq , 44
- \cong , 16
- \cong_F , 121
- \sim_F , 121
- \Vdash , 140
- \vdash , 126, 141
- $\xrightarrow{F, \theta}$, 119
- A_{df}^* , 10
- $C\langle \phi_1, \dots, \phi_n \rangle$, 36
- $C\langle T_1, \dots, T_n \rangle_F$, 120
- $C\langle x_1, \dots, x_n \rangle$, 20
- Co_T , 124
- $f|^{Y'}$, 10
- $f|_{X'}$, 10
- $FOLD^T$, 130
- $FOLD_F(T)$, 130
- $\mathcal{G}(Z, L)$, 13
- $\mathcal{H}(Z, L)$, 13
- Int_∞ , 117
- \mathbf{m} , 16
- $\overline{\mathbf{m}}$, 16
- M_H , 60
- M_T , 124
- P_H , 60
- P_T , 124
- $\lfloor s \rfloor_{i_1 \dots i_n}$, 10
- \mathcal{S}_n , 100
- \mathcal{S}_n , 59
- val_e , 37
- val_n , 42
- $z_n(l)$, 16
- active input port, 76
- active output port, 76
- algebraic approach, 32
- bad redex, 63
- barbed bisimulation, 55
- barbed congruence, 55, 77
- basic graph, 16
- bisimulation, 54
- canonical projection, 19
- canonical strong morphism, 26
- cardinality of a hyperedge, 14
- cardinality of a hypergraph, 14
- categorical approach, 22
- category, 22
- closed name-based graph term, 42
- closure under inverse morphisms, 133, 144
- co-limit, 24, 26
- compatible type functors, 139, 145
- consistent equivalence relation, 19
- context graph, 20
- diagram, 24
- discrete context, 21
- discrete hypergraph, 16
- double-pushout approach, 32
- edge, 13
- embedding, 17
- external node, 13

- factorization, 17
- folding type graphs, 130
- free variable, 60
- free name, 104
- full abstraction, 88
- functor, 23
- garbage, 79
- generic type system, 108
- graph expression, 36
- hyperedge, 13
- hypergraph, 13
- hypergraph epimorphism, 15
- hypergraph isomorphism, 15
- hypergraph monomorphism, 15
- hypergraph morphism, 15
- idempotent, 116
- internal node, 14
- isolated node, 14
- isomorphic hypergraphs, 16
- isomorphic type graphs, 121
- isomorphism in a category, 23
- join-morphism, 118
- l-monoid, 116
- l-monoid morphism, 118
- label of a hyperedge, 13
- λ -calculus, 90
- lattice, 116
- lattice-ordered commutative monoid, 116
- linear mapping, 123, 125, 139, 159
- message, 59
- message reception, 63
- monoid morphism, 118
- morphism in a category, 22
- multi-pointed hypergraph, 13
- name in the π -calculus, 104
- name-based context, 45
- name-based graph term, 41
- name-based notation, 41
- nice morphism, 16
- node, 13
- nondeterminism, 92
- normal bisimulation, 80
- normal form of name-based notation, 44
- object in a category, 22
- π -calculus, 52, 104
- port, 60
- principal type, 107, 129, 133
- process with a variable, 60
- process with process abstraction, 60
- process with replication, 59
- process description, 60
- pushout, 24
- quotient graph, 19, 25
- reduction semantics of SPIDER, 62
- replication, 63
- residual, 116
- residuated l-monoid, 116
- send-node, 60
- set-based approach, 13
- simple hypergraph, 13
- sort of a hyperedge, 13
- source node, 13
- SPIDER expression, 59
- SPIDER, 59
- SPIDER based on graph expressions, 97
- SPIDER based on the name-based notation, 99
- strong morphism, 15
- strong simulation, 87
- structural congruence in SPIDER, 61
- subject reduction property, 107, 129, 142
- substitution in SPIDER, 61
- true graph, 124
- true type, 141
- true type graph, 124
- type, 126, 140
- type assignment, 126, 160
- type environment, 126
- type functor, 119
- type graph, 119

- type graph morphism, 119
- type inference, 107, 130, 143
- type system based on lattices, 124
- type system based on monoids, 138
- type system for the π -calculus, 160
- type tree, 160
- typing rules, 126, 140

- weak barbed congruence for arbitrary
 calculi, 89
- weak barbed congruence, 77
- weak simulation, 87