

What are Polymorphically-Typed Ambients?

December 22, 2000*

Torben Amtoft
Boston University
www.cs.bu.edu/associates/tamtoft

Assaf J. Kfoury
Boston University
www.cs.bu.edu/fac/kfoury

Santiago M. Pericas-Geertsen
Boston University
cs-people.bu.edu/santiago

Abstract

The Ambient Calculus was developed by Cardelli and Gordon as a formal framework to study issues of mobility and migrant code [CG98]. We consider an Ambient Calculus where ambients transport and exchange programs rather than just inert data. We propose different senses in which such a calculus can be said to be polymorphically typed, and design accordingly a polymorphic type system for it. Our type system assigns types to embedded programs and what we call behaviors to processes; a denotational semantics of behaviors is then proposed, here called trace semantics, underlying much of the remaining analysis. We state and prove a Subject Reduction property for our polymorphically-typed calculus. Based on techniques borrowed from finite automata theory, type-checking of fully type-annotated processes is shown to be decidable. Our polymorphically-typed calculus is a conservative extension of the typed Ambient Calculus originally proposed by Cardelli and Gordon [CG99].

1 Introduction

1.1 Background and Motivation

With the advent of the Internet a few years ago, considerable effort has gone into the study of *mobile computation* and programming languages that support it. On the theoretical side of this research, several concurrent and distributed calculi have been proposed, such as the Distributed Join Calculus [FGL⁺96], the $D\pi$ Calculus [RH98, RH99], the Box-Pi Calculus [SV99], the Seal Calculus [VC99], among others¹. The *Ambient Calculus* (henceforth, **AC**) is a recent addition to this list and the starting point of our investigation.

Our long-term interest is the design and implementation of a strongly-typed programming language for mobile computation. Part of this effort is an examination of **AC** as a foundation for such a language. An important step in achieving a greater degree of modularity and a more natural style of programming, without sacrificing the benefits of strong typing, is to make ambients *polymorphically typed*. This is the focus of the present report.

*Revised version as of November 12, 2001

¹The proliferation of calculi is mostly the result of different concerns and emphases (mobility, concurrency, security, etc.) brought by different researchers. At this early stage of Internet programming, it is perhaps healthy to have several research agendas based on almost as many different calculi.

Early type systems for **AC** (see [CG99, CGG99, CGG00] among others) restrict ambients to be *monomorphic*: There can be only one “topic of conversation” (the type of exchanged data) in an ambient, initially and throughout its existence as a location of an enclosed process. Below, we identify 4 cases in which ambients can be said to be polymorphically typed. Very recent type systems for **AC** and for an object-oriented version of **AC**, in [Zim00] and [BCC00] respectively, include suitable forms of subtyping, one of the 4 cases below. But none of the other 3 cases has been yet integrated into a polymorphic type system for **AC** or for an extension of it.

We illustrate each of the 4 cases with a very brief example, written in a syntax slightly more general than the original syntax of **AC**, as we allow processes to exchange arbitrary functional expressions (possibly unevaluated for now) rather than just inert data. Our formal presentation in later sections extends the original syntax of **AC** further, in several appropriate ways, and specifies the operational semantics precisely.

Case 1. Consider a process of the form:

$$p[\text{in } r. \langle \text{even}, 3 \rangle] \mid q[\text{in } r. \langle \text{not}, \text{true} \rangle] \mid r[(f, x). n[\langle f x \rangle \mid P] \mid \text{open } p \mid \text{open } q]$$

Here, there are 3 ambients in parallel, named p , q and r , and one ambient named n inside r . Both p and q can move into r (expressed by the capability “in r ”) and, once inside r , both can be dissolved (expressed by the capabilities “open p ” and “open q ”) in order to unleash their outputs. The type of the input pair (f, x) inside r can be $(\text{int} \rightarrow \text{bool}, \text{int})$ or $(\text{bool} \rightarrow \text{bool}, \text{bool})$, depending on whether output $\langle \text{even}, 3 \rangle$ or output $\langle \text{not}, \text{true} \rangle$ is transmitted first, and in either case the type of the application $(f x)$ is bool . We assume the unspecified process P can be executed safely in parallel with the boolean output $\langle f x \rangle$. The polymorphism of r is basically the familiar *parametric polymorphism* of ML.

Case 2. A slight variation of the preceding process is:

$$p[\text{in } r. \langle 3, 2 \rangle] \mid q[\text{in } r. \langle 3.6, 5.1 \rangle] \mid r[(x, y). n[\langle \text{mult}(x, y) \rangle \mid P] \mid \text{open } p \mid \text{open } q]$$

where the operation $\text{mult} : (\text{real}, \text{real}) \rightarrow \text{real}$ multiplies two real numbers. Because the type of $\langle 3, 2 \rangle$ is (int, int) , which is a subtype of $(\text{real}, \text{real})$, it is safe to transmit the output $\langle 3, 2 \rangle$ to the input variables (x, y) . Both ambients p and q can enter the ambient r safely. The polymorphism of r is the familiar *subtype polymorphism* found in many other functional and object-oriented programming languages.

Case 3. Consider now the following process:

$$n[\langle \text{true}, 5 \rangle \mid \langle 5, 6, 3.6 \rangle \mid (x, y). P \mid (x, y, z). Q]$$

The outputs are transmitted depending on their arities, here 2 for the output $\langle \text{true}, 5 \rangle$ and 3 for the output $\langle 5, 6, 3.6 \rangle$. We assume that the unspecified processes $(x, y). P$ and $(x, y, z). Q$ can be executed safely if they input, respectively, $(\text{bool}, \text{int})$ pairs and $(\text{int}, \text{int}, \text{real})$ triples. There is no ambiguity as to which of the two outputs should be transmitted to which of these two processes, i.e., the arity is used as a “switch” to dispatch an output to its appropriate destination. Hence, the execution of the entire process enclosed in the ambient n can proceed safely, provided also that all other outputs of arity 2 and arity 3 in parallel with $\langle \text{true}, 5 \rangle$ and $\langle 5, 6, 3.6 \rangle$ have types $(\text{bool}, \text{int})$ and $(\text{int}, \text{int}, \text{real})$, respectively. The polymorphism of n is appropriately called *arity polymorphism*².

Case 4. A more subtle sense in which the type of exchanged data can change over time, as the computation proceeds inside an ambient, is illustrated by the following:

$$m[\langle 7 \rangle \mid (x). \text{open } n. \langle x = 42 \rangle \mid n[(y). P]]$$

where the type of the equality test “ $x = 42$ ” is bool . Initially, the topic of conversation in the ambient m is int . After the output $\langle 7 \rangle$ is transmitted, the ambient n is opened and the topic of conversation now becomes

² The expression “arity polymorphism” was used already by others, e.g. Moggi [Mog00], to describe similar—though quite different in some respects—situations in functional programming languages (also addressed in Tullsen’s recent work on the Zip Calculus [Tul00]).

bool. Assuming that the unspecified process $(y).P$ can be executed safely whenever it inputs a boolean value, the execution of the entire process enclosed in the ambient m can proceed safely. What takes place in the ambient m is a case of what we shall call *orderly communication*³.

Of the four cases above, perhaps **3** and certainly **4** are arguably excluded from what “polymorphism” has usually meant. Nevertheless, these two cases allow the same ambient to hold different topics of conversation, either *simultaneously* (in **case 3**) or *consecutively* at different times (in **case 4**) — or both simultaneously and consecutively, as illustrated by more interesting examples. Hence, in a wider sense of the word which we here propose, it is appropriate to include **3** and **4** as cases of polymorphic ambients.

1.2 Which Cases to Consider?

The four cases are listed from most studied to least studied. The first case, ML-style *parametric polymorphism*, has been studied for some 25 years. It has given rise to an extensive theory in different representations (usually by universal/existential types, less frequently by intersection/union types) and its incorporation in type systems for programming languages other than pure functional, notably for concurrent calculi, was generally successful. A good example is Turner’s work on the polymorphic π -calculus [Tur95], whose semantic properties are also thoroughly examined by Pierce and Sangiorgi [PS00]; concurrent calculi with a weaker form of parametric polymorphism were also developed in earlier studies [Gay93, VH93].

The second case, *subtype polymorphism*, has been almost as extensively studied. It is well understood in various forms (e.g., “deep subtyping” versus “shallow subtyping” with different tradeoffs) and it was also incorporated in type systems for concurrent calculi. Examples of such work are [PS96, PT97] for the π -calculus and [Zim00] for **AC**.

The last case by contrast, *orderly communication*, raises entirely new problems not encountered before in the context of **AC**. The design of a type discipline enforcing it is a delicate matter. This is one of the challenges we take on in this report.

Orderly communication bears a strong resemblance to what has been called “session types” in the π -calculus [GH99]. Leaving aside differences between the underlying calculi, orderly communication and session types are both motivated by the need to keep track of the order in which communication events take place. There are nevertheless important differences between the two, which are discussed further in Sect. 7.3.

1.3 Scope and Contribution of Our Research

The core of our formal calculus is **AC**, which is augmented with various functionalities at the level of exchanged data. Accordingly, we call our calculus **AC+**. In this report the added functionality is that of a simply-typed functional language. Thus, outputs in processes are now of the form $\langle M \rangle$ where M is a functional program rather than just inert data. But our framework is open-ended and can be adjusted according to needs; in particular, the inserted functional language can be enriched to include such features as *object-orientation*⁴.

Although **AC+** is the result of combining **AC** and a functional language, the two are essentially kept separate in our framework, in the sense that communication between processes is limited to functional programs and cannot include other processes. This is a deliberate decision: We steer clear of a *higher-order AC+*, where processes can exchange other processes (in addition to programs), something that will certainly reproduce many of the challenges already encountered in higher-order versions of the π -calculus (as in the work of Hennessy and his collaborators [YH99, YH00] for example). Our simpler (first-order) version of **AC+** raises many non-trivial problems already and gives us much to investigate; moreover, with an eye to an

³We thank Benjamin Pierce for suggesting the apt expression “orderly communication”.

⁴The approach followed by Bugliesi, Castagna and Crafa in [BCC00] is to put ambients and objects together, in a single integrated calculus. By contrast, following our current approach with functional programs, objects will be embedded inside ambients.

implementation later, there is something to be said in favor of keeping our conceptual framework as simple as possible.

In summary, our main accomplishments in the present report are (highlighted by bullet points):

- We design a type system for **AC+** where embedded programs are assigned *types* and processes are assigned what we call *behaviors*. Our type system smoothly integrates 3 of the 4 cases of polymorphism into a single framework: *subtype polymorphism*, *arity polymorphism* and *orderly communication*.

Our current type system does not include ML-style *parametric polymorphism*. Taking the cue from Turner's work [Tur95], we expect its incorporation into our type system to be proceed smoothly.

The syntax of types and the syntax of behaviors are disjoint, but we refer generically to both by the word "types". Thus, our "type" system assigns both "types and behaviors", "type checking" means "type and behavior checking", and "type inference" means "type and behavior inference". Thus also, subtype polymorphism generically refers to both "type subsumption" (i.e., subtyping) and "behavior subsumption".

The operational semantics of **AC+** has three parts: mobility, communication and embedded execution. *Mobility* consists of reduction rules for capabilities (in, out, open), just as in the original **AC**. *Communication* has a single input/output rule, also in the original **AC**. *Embedded execution* specifies reduction rules for the programs that are transported and exchanged by ambients; there is a choice of rules here, depending on the language of embedded programs and how they are evaluated. For the simply-typed functional programs in this report, we choose the rules of a call-by-value operational semantics.

- We develop a perspicuous denotational semantics of behaviors, which we call their *trace semantics*. Behavior equivalence and behavior subsumption are defined relative to this trace semantics⁵.
- Behavior subsumption and type subsumption are shown to be decidable relations. The deterministic time-complexity of our decision procedures is at least exponential⁶.

The proof of this result is of independent interest; it is a non-trivial adaptation of techniques from finite automata theory where, by contrast, decision procedures typically have low-degree polynomial time complexities.

- Using the trace semantics of behaviors, we prove that our polymorphically-typed **AC+** satisfies a Subject Reduction property.
- Based on the decidability of behavior subsumption and type subsumption, we show that *type-checking* is decidable for fully type-annotated terms of **AC+**. The deterministic time-complexity of the decision procedure is at least exponential in the worst-case.

The more difficult problem of *type-inference* for (un-annotated) terms of **AC+** is left for future work.

- Our polymorphically typed **AC+** is a conservative extension of the typed version of **AC** originally proposed by Cardelli and Gordon [CG99], in the sense that every process typable in the latter is typable in ours (but not the other way around).

Finally, we note that there are several aspects of our polymorphically typed **AC+** that makes it suitable for building programmer-friendly high-level abstractions on top of **AC+**. An illustration of this is given by the macro LET-IN in Example 2.2.

⁵ We are knowingly overloading words that are extensively used, not always with the same meaning, in the literature on concurrency. Such are the words "behavior" and "trace". Although our use is still different (alas!), these words are also suggestive and aptly describe the formal notions they refer to in our work.

⁶ This result in itself is no reason for discomfiture. Hope for efficiency in practice is supported by other similar situations. For example, ML type-inference shows an extreme disparity between worst-case performance in theory (exponential time) and actual performance in practice (very fast).

2 Motivating Examples

We give two examples, short but more interesting than the snippets in Sect. 1.1, to illustrate the expressive power and convenience of a polymorphically typed $\mathbf{AC}+$. The reader is referred to the Appendix for an explanation on how to type the examples presented in this section. Aside from the embedded programs, the syntax of ambients is identical to that first proposed by Cardelli and Gordon [CG98] with the addition of a co-capability “ $\text{coopen } n$ ” akin to a proposal already made by Levi and Sangiorgi [LS00]⁷. For a process to open an ambient n , this ambient must contain a top-level process willing to exercise a $\text{coopen } n$ (cf. (Red Open) in Fig. 2). We shall use $n\{P\}$ as an abbreviation, namely

$$n\{P\} \triangleq n[\text{coopen } n \mid P]$$

for every ambient name n and every process P . Thus, if we write $n\{P\}$, we mean that the ambient n is *openable* without any restriction.

EXAMPLE 2.1 (PACKET ROUTING). This example is representative of a class of processes that can be typed using *orderly communication*. A packet enters a router and requests to be routed to a specific destination. A router reads the destination name (denoted by the string “bu”) and then communicates a path (a sequence of in and out capabilities) back to the packet. The packet uses this path to route itself to the destination. Orderly communication is needed since inside the packet there are *two* topics of conversation: first strings (the destination “bu”), and next capabilities (the path)⁸.

$$\begin{aligned} & \text{router}[\text{!route}\{\text{in packet}.\langle \text{dst} \rangle.\text{open hop}.\langle \text{lookup-route}(\text{dst}) \rangle\} \mid \\ & \quad \text{packet}[\text{in router}.\text{open route}.\langle \text{“bu”} \rangle \mid \text{hop}\{(x).x\}] \end{aligned}$$

Notice that the packet reads and exercises the path by means of its subterm $(x).x$. Despite its simplicity, the term $(x).x$ is not typable in the Cardelli-Gordon type system for \mathbf{AC} nor, to the best of our knowledge, in any of the type systems for \mathbf{AC} available in the literature. At first, it appears that a type derivation for $(x).x$ consists of an instance of the rule (Exp n) followed by (Proc Input) [CG99]. However, this is not the case since $(x).x$ is a shorthand for $(x).x.0$. A close examination of the type derivation for $(x).x.0$ reveals that x requires a type T such that $T = \text{cap}[T]$, but no such type exists in the Cardelli-Gordon system. In that system, the only way to type a process that reads and exercises a capability is by using an extra ambient. Specifically, the process $(x).x$ must be written as $(x).n[x]$ for some ambient name n . \square

EXAMPLE 2.2 (CODE ON DEMAND, DATA-DRIVEN DISPATCH). This example is representative of a class of processes whose typing requires both *arity polymorphism* and *orderly communication*. There is a *server* that delivers programs for high-performance arithmetical tests and functions, here,

$$\text{prime} : \text{int} \rightarrow \text{bool} \quad \text{relative-prime} : \times(\text{int}, \text{int}) \rightarrow \text{bool}$$

Clients request programs for any of these two arithmetical operations. The requested programs are executed locally by the client instead of remotely by the server⁹.

$$\begin{aligned} \text{server} & \triangleq s[\text{!tst}[\text{open } p \mid \\ & \quad (x_1, x_2). \text{tstres}\{x_1.\langle \text{prime}(x_2) \rangle\} \mid \\ & \quad (x_1, x_2, x_3). \text{tstres}\{x_1.\langle \text{relative-prime}(x_2, x_3) \rangle\} \mid] \end{aligned}$$

⁷Levi and Sangiorgi write “ $\overline{\text{open}} \ n$ ” instead of “ $\text{coopen } n$ ”. In the same vein, they also introduce co-capabilities “ $\overline{\text{in}} \ n$ ” and “ $\overline{\text{out}} \ n$ ”, or “ $\text{coin } n$ ” and “ $\text{coout } n$ ” in our notation, which could be easily incorporated into our formal presentation (cf. the discussion in Sect. 7.1.1).

⁸By assumption, the function `lookup-route` takes a string as input and produces a capability path as output.

⁹Data-driven dispatching of code from server to clients is undesirable in many situations in practice, as the data may be prohibitively large. In the present example, the data received by the server takes a few bits to store (one or two integers, very small in size but large in value). The example is for illustrative purposes only, not for prescribing a particular way of programming a COD dispatcher in general.

$$\begin{aligned} \text{client} &\triangleq \text{LET} \quad \text{exitPath} = \text{out } c.\text{in } s.\text{in } \text{tst.coopen } p \\ &\quad \text{returnPath} = \text{out } \text{tst.out } s.\text{in } c \\ &\text{IN} \quad c[\text{open } \text{tstres}.\langle v \rangle.Q \mid p[\text{exitPath}.\langle \text{returnPath}, 1 + 2^{4096} \rangle]] \end{aligned}$$

The process under consideration is $\text{server} \mid \text{client}$ where Q is an unspecified process that makes use of the value of $\text{prime}(1 + 2^{4096})^{10}$.

De-sugaring LET-IN in the definition of client , we obtain the process shown below where exitPath is now an abbreviation (rather than a variable) for the path “out $c.\text{in } s.\text{in } \text{tst.coopen } p$ ”, and returnPath is an abbreviation for the path “out $\text{tst.out } s.\text{in } c$ ”.

$$\begin{aligned} \text{client} &\triangleq c[\langle \text{exitPath} \rangle \mid \\ &\quad (z_1).\langle \text{returnPath} \rangle \mid \\ &\quad (z_2).\langle \text{open } \text{tstres}.\langle v \rangle.Q \mid p[z_1.\langle z_2, 1 + 2^{4096} \rangle] \rangle] \end{aligned}$$

Notice that, *orderly communication* is needed to type the de-sugared term shown above: exitPath , returnPath and the result of calling prime are communicated inside the ambient c , and these communications are of the same arity but of different types.

Observe that the conventional de-sugaring of $\text{LET } z = M \text{ IN } P$ into $((\lambda z.P)M)$ is purposely avoided, because it would nest processes inside programs; specifically in this case, it would place the process P under a λ -abstraction. Instead, we de-sugar LET-IN using parallel processes and nested scopes, preserving our stated goal of keeping programs completely inside ambients. For that to work, we require the body of a LET-IN to be of the form $n[Q]$ for some process Q . More precisely, an expression $\text{LET } z_1 = M_1, \dots, z_k = M_k \text{ IN } m[Q]$, where each z_i can be used not only in Q but also in M_j if $i < j$, is viewed as a convenient shorthand for the **AC+** process $m[\langle M_1 \rangle \mid (z_1).\langle M_2 \rangle \mid (z_2).\langle \dots (z_{k-1}).\langle M_k \rangle \mid (z_k).Q \dots \rangle]$. \square

3 Types and Behaviors

Figure 1 depicts the syntax of our language **AC+**. A process $P \in \text{Proc}$ is basically as in [CG99]: there are constructs for parallel composition ($P_1 \mid P_2$), replication ($!P$), restriction ($(\nu n : \sigma).P$); and there also are constructs for input (where the names $n_1 \dots n_k$ must be distinct) and output. Note that communication is asynchronous, in that an outputting process has no “continuation”; a communication can thus (cf. the metaphor in [Car99]) be viewed as the placement, and subsequent removal, of a Post-It note on a message board that (unlike in [Car99]) has a section for each arity. On the other hand, we believe that our development would carry through (with the obvious modifications) also for an extension of **AC+** allowing synchronous communication.

An expression $M \in \text{Exp}$ denotes a computation over a domain that includes not only simple values (like integers) but also functions, tuples, ambient names, and (paths of) capabilities. Accordingly the set of constants c is open-ended and may include not only numbers and arithmetic operators but also, e.g., a function which given a path and a name n tests whether the capability “in n ” is in the path. Note that for all binding constructs (λ -abstraction, restriction, input) in **AC+**, the name n being bound is annotated with a type σ (to be defined in Sect. 3.2).

The set of names occurring free in P is denoted $\text{fn}(P)$; the set of all names occurring in P is denoted $\text{names}(P)$. We say that a process P is non-conflicting with a set of names X if (i) no name is bound more than once in P , and (ii) a name bound in P does not occur in X . For all P and X , we can clearly find P' such that P' is non-conflicting with X and such that P' and P are equal modulo consistent renaming of bound names. Everything in this paragraph also holds when P is replaced by M .

¹⁰The number $1 + 2^{4096}$ is the so-called 12th Fermat number, because $4096 = 2^{12}$. Among Fermat numbers, $1 + 2^{4096}$ is the smallest whose factorization is currently unknown. However, although its factorization is still unknown, $1 + 2^{4096}$ is known to be composite by algebraic methods.

Expressions	
$M \in \text{Exp}$	$::= n \mid c \mid \lambda n : \sigma.M \mid M_1 M_2 \mid \times(M_1, \dots, M_k) \mid \text{if } M_0 \text{ then } M_1 \text{ else } M_2 \mid \dots \mid$ $\mid \epsilon \mid M_1.M_2 \mid \text{in } M \mid \text{out } M \mid \text{open } M \mid \text{coopen } M$ $(k \geq 0)$
Processes	
$P \in \text{Proc}$	$::= \mathbf{0} \mid P_1 \mid P_2 \mid !P \mid (\nu n : \sigma).P \mid M.P \mid M[P] \mid (n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle M \rangle$ $(k \geq 0)$
When there is no ambiguity we write M for $M.0$.	

Figure 1. Syntax of AC+.

3.1 Operational Semantics

The semantics of AC+ is presented in Fig. 2. Before an expression M can be passed as an argument to a function or communicated to another process it must be evaluated to a value V , using the evaluation relation $M_1 \longrightarrow M_2$ which is defined using the standard notion of evaluation contexts.

We write $P_1 \equiv P_2$ to denote that P_1 and P_2 are equivalent, modulo consistent renaming of bound names (which may be needed to apply (Red Beta) and (Red Comm), as these rules have side conditions preventing name capture) and modulo “syntactic rearrangement”. The relation is given as the least one satisfying the clauses presented in Fig. 3 and is as in [CG99], except that (for reasons mentioned in Sect. 5) we omit the rule $!P \equiv P \mid !P$ and instead allow this “unfolding” to take place via the rule (Red Repl).

We write $P_1 \xrightarrow{\ell} P_2$ if P_1 reduces in one step to P_2 by performing “an action described by ℓ ”. Here $\ell = \text{comm}(\tau)$ if a value of type τ is communicated at top-level (Red Comm), and $\ell = \epsilon$ otherwise. We use a notion of “process evaluation contexts” to succinctly describe the place in a process where an expression (Red MctxtP) or subprocess (Red PctxtP) is reduced. Reducing inside an ambient is given a special treatment in (Red Amb), as the label “disappears” due to the fact that communications are invisible outside ambients. Note that $P \xrightarrow{\ell} Q$ does not imply that $M.P \xrightarrow{\ell} M.Q$ since M must evaluate to a capability which then is executed before P can be activated; similarly for other constructs.

3.2 Types and Behaviors

The syntax of types ($\tau, \sigma \in \text{Typ}$) and the syntax of behaviors ($b \in \text{Beh}$) are recursively defined in Fig. 4. The first five behavior constructs capture the intuition that we want to keep track of the relationship (sequential or parallel) between occurrences of input and output operations. (See Sect. 7.1.1 for a discussion of whether also to keep track of ambient movements.)

An ambient n has a type of the form $\text{amb}[b_0, b_1]$, where b_0 and b_1 can both be viewed as upper estimates of the behavior of a process “unleashed” by opening n . An example: for $n[\langle 7 \rangle \mid (x : \text{int}).\text{coopen } n.\langle x = 42 \rangle]$ we expect n to have the type $\text{amb}[\text{put}(\text{bool}), \text{put}(\text{bool})]$, reflecting that when n is opened the value 7 has already been communicated—something we would not know if we did not have the explicit occurrence of $\text{coopen } n$, which we keep track of using the behavior diss . The behaviors b_0 and b_1 will often be equal, in which case we may without ambiguity write $\text{amb}[b_0]$ for $\text{amb}[b_0, b_0]$, but as we explain in Sect. 3.4 it is convenient to allow for b_0 and b_1 to be distinct.

A capability has a type of the form $\text{cap}[B]$ where B is a behavior context, that is a “behavior with a hole inside”. To motivate this, consider a process $P = \text{open } n.P'$ where P' has behavior b' and n has type $\text{amb}[b]$. When P is executed, P' will run in parallel with a process of behavior b , so P should be assigned the behavior $b \mid b'$, which can be written as $(b \mid \square) \mid b'$. This is why it makes sense to assign $\text{open } n$ the capability type $\text{cap}[b \mid \square]$, cf. the rules (Exp Open) and (Proc Action) in Fig 6.

The first six behavior constructs in Fig. 4 alone, are sufficient to write a type system satisfying a subject

Values

$$V ::= n \mid c \mid \lambda n : \sigma.M \mid \times(V_1, \dots, V_k) \mid \epsilon \mid V_1.V_2 \mid \text{in } V \mid \text{out } V \mid \text{open } V \mid \text{coopen } V \quad (k \geq 0)$$

Evaluation Contexts

$$\mathcal{E} ::= \square_e \mid \mathcal{E}M \mid V\mathcal{E} \mid \times(V_1, \dots, V_{i-1}, \mathcal{E}, M_{i+1}, \dots, M_k) \mid \text{if } \mathcal{E} \text{ then } M_1 \text{ else } M_2 \quad (k \geq 0) \\ \mid \mathcal{E}.M \mid V.\mathcal{E} \mid \text{in } \mathcal{E} \mid \text{out } \mathcal{E} \mid \text{open } \mathcal{E} \mid \text{coopen } \mathcal{E}$$

Process Evaluation Contexts

$$\mathcal{P} ::= \square_p \mid \mathcal{E}.P \mid \mathcal{E}[P] \mid \langle \mathcal{E} \rangle \mid (\nu n : \sigma).P \mid P \mid P$$

Notation: $\mathcal{E}[M]$ is the expression resulting from replacing \square_e with M in \mathcal{E} ; if the hole in \mathcal{P} is an expression hole then $\mathcal{P}[M]_e$ is the process resulting from replacing this hole with M ; and if the hole in \mathcal{P} is a process hole then $\mathcal{P}[P]_p$ is the process resulting from replacing this hole with P .

Reduction Rules

Let ℓ be a label in $\{\epsilon\} \cup \{\text{comm}(\tau) \mid \tau \in \text{Typ}\}$.

Let $\delta(c, V)$ be a partial function defined for every constant c . For example, $\delta(+, \times(1, 2)) = 3$.

In (Red Beta) we demand that $\lambda n : \sigma.M$ is non-conflicting with $\text{names}(V)$.

In (Red Comm) we demand that $(n_1 : \sigma_1, \dots, n_k : \sigma_k).P$ is non-conflicting with $\text{names}(V_1, \dots, V_k)$.

$$\begin{array}{ll} (\lambda n : \sigma.M)V \longrightarrow M[n := V] & \text{(Red Beta)} \\ cV \longrightarrow V' \text{ where } V' = \delta(c, V) & \text{(Red Delta)} \\ \text{if true then } M_1 \text{ else } M_2 \longrightarrow M_1 & \text{(Red IfTrue)} \\ \text{if false then } M_1 \text{ else } M_2 \longrightarrow M_2 & \text{(Red IfFalse)} \\ \text{If } M_1 \longrightarrow M_2 \text{ then } \mathcal{E}[M_1] \longrightarrow \mathcal{E}[M_2] & \text{(Red MctxtM)} \\ \\ n[\text{in } m.P \mid Q] \mid m[R] \xrightarrow{\epsilon} m[n[P \mid Q] \mid R] & \text{(Red In)} \\ m[n[\text{out } m.P \mid Q] \mid R] \xrightarrow{\epsilon} n[P \mid Q] \mid m[R] & \text{(Red Out)} \\ \text{open } n.P \mid n[\text{coopen } n.Q \mid R] \xrightarrow{\epsilon} P \mid Q \mid R & \text{(Red Open)} \\ (n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle \times(V_1, \dots, V_k) \rangle \xrightarrow{\text{comm}(\tau)} P[n_i := V_i] \text{ where } \tau = \times(\sigma_1, \dots, \sigma_k) & \text{(Red Comm)} \\ !P \xrightarrow{\epsilon} P \mid !P & \text{(Red Repl)} \\ \\ \text{If } M_1 \longrightarrow M_2 \text{ then } \mathcal{P}[M_1]_e \xrightarrow{\epsilon} \mathcal{P}[M_2]_e & \text{(Red MctxtP)} \\ \text{If } P \xrightarrow{\ell} Q \text{ then } \mathcal{P}[P]_p \xrightarrow{\ell} \mathcal{P}[Q]_p & \text{(Red PctxtP)} \\ \text{If } P \xrightarrow{\ell} Q \text{ then } n[P] \xrightarrow{\epsilon} n[Q] & \text{(Red Amb)} \\ \text{If } P \equiv P, P \xrightarrow{\ell} Q, Q \equiv Q' \text{ then } P' \xrightarrow{\ell} Q' & \text{(Red } \equiv \text{)} \end{array}$$

Thus only tuples are communicated, and where there is no ambiguity we may write $\langle M_1, \dots, M_k \rangle$ for $\langle \times(M_1, \dots, M_k) \rangle$

Figure 2. Operational Semantics.

reduction property (Sect. 5), but they do not enable the typing of processes performing (using replication) an unbounded number of input and output operations, and neither do they enable the typing of a conditional where one branch is a capability of type $\text{cap}[\text{put}(\text{int}) \mid \square]$ whereas the other branch is a capability of type $\text{cap}[\text{get}(\text{int}) \mid \square]$. Among many possible options for (approximating) constructs expressing recursion and choice, we in this paper settle for a simple one: the construct $\text{fromnow } T$ with T the “topics of conversation”, which can be thought of as the “union” of all behaviors composed of $\text{put}(\tau)$ and $\text{get}(\tau)$ with $\tau \in T$. As to be demonstrated in Sect. 7.1, this construct actually makes our type system a conservative extension of the one presented in [CG99].

$P \equiv P$	(Struct Refl)
If $P \equiv Q$ then $Q \equiv P$	(Struct Symm)
If $P \equiv Q$ and $Q \equiv R$ then $P \equiv R$	(Struct Trans)
If $P \equiv Q$ then $(\nu n : \tau).P \equiv (\nu n : \tau).Q$	(Struct Res)
If $P \equiv Q$ then $P \mid R \equiv Q \mid R$	(Struct Par)
If $P \equiv Q$ then $!P \equiv !Q$	(Struct Repl)
If $P \equiv Q$ then $M[P] \equiv M[Q]$	(Struct Amb)
If $P \equiv Q$ then $M.P \equiv M.Q$	(Struct Action)
If $P \equiv Q$ then $(n_1 : \tau_1, \dots, n_m : \tau_m).P \equiv (n_1 : \tau_1, \dots, n_m : \tau_m).Q$	(Struct Input)
$P \mid Q \equiv Q \mid P$	(Struct ParComm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct ParAssoc)
$(\nu n_1 : \tau_1).(\nu n_2 : \tau_2).P \equiv (\nu n_2 : \tau_2).(\nu n_1 : \tau_1).P$ if $n_1 \neq n_2$	(Struct ResRes)
$(\nu n : \tau).(P \mid Q) \equiv P \mid (\nu n : \tau).Q$ if $n \notin \text{fn}(P)$	(Struct ResPar)
$(\nu n : \tau).m[P] \equiv m[(\nu n : \tau).P]$ if $n \neq m$	(Struct ResAmb)
$P \mid \mathbf{0} \equiv P$	(Struct ZeroPar)
$(\nu n : \tau).\mathbf{0} \equiv \mathbf{0}$	(Struct ZeroRes)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct ZeroRepl)
$\epsilon.P \equiv P$	(Struct ϵ)
$(M.M').P \equiv M.(M'.P)$	(Struct \cdot)
$P \equiv Q$ if P and Q are equal modulo consistent renaming of bound names	(Struct α – rename)

Figure 3. Structural Congruence.

We shall use the notion of *level*: a type τ has level i if i is an upper bound of the depth of nested occurrences of $\text{amb}[_, _]$ or $\text{cap}[_]$ within τ , similarly for T , b , and B . (We use “_” to stand for an arbitrary entity of the appropriate kind.) Note that an entity that has level i also has level j for all $j > i$. Example: $\tau_0 = \text{int} \rightarrow \text{int}$ has level zero, $b_1 = \text{put}(\text{cap}[\text{put}(\tau_0) \mid \square])$ has level one, and $\tau_2 = \text{amb}[b_1, b_1]$ has level two. Observe, e.g., that if $\text{amb}[b, b']$ has level $i + 1$ then b and b' both have level i , that if $\text{cap}[B]$ has level $i + 1$ then B has level i , that if $B.b$ has level i then B and b both have level i , and that if $\text{get}(\tau)$ has level i then τ has level i .

3.3 Behavior Subsumption

We employ a relation $b_1 \leq b_2$, to be formally defined in Sect. 4.1, with the intuitive interpretation that b_2 is more “permissive” than b_1 . For example, $\text{put}(\text{int}) \leq \text{fromnow} \{\text{int}, (\text{int}, \text{int})\}$, and if integers can be converted into real numbers then also $\text{put}(\text{int}) \leq \text{put}(\text{real})$, since a process that sends an integer thereby also sends a real number, and $\text{get}(\text{real}) \leq \text{get}(\text{int})$, since a process that accepts a real number also will accept an integer. Thus output is covariant and input is contravariant, while in other systems found in the literature it is the other way round—the reason for this discrepancy is that we take a *descriptive* rather than a *prescriptive* point of view. From a prescriptive point of view, a channel that allows the writing of real numbers also allows the writing of integers, and a channel that allows the reading of integers also allows the reading of real numbers.

The relation on behaviors induces a relation on behavior contexts:

Definition 3.1. $B_1 \leq B_2$ holds iff for all level 0 behaviors b we have $B_1[b] \leq B_2[b]$. □

We shall see (Lemma 4.13) that the restriction to level 0 behaviors is not crucial: if $B_1 \leq B_2$ then $B_1[b] \leq B_2[b]$ holds for all b ; moreover for all B_1, B_2 there exists b_0 such that testing $B_1 \leq B_2$ amounts to testing $B_1[b_0] \leq B_2[b_0]$.

Types

$\sigma, \tau \in \text{Typ} ::=$	$\text{bool} \mid \text{int} \mid \text{real} \mid \text{string} \mid \dots$	type constant
	$\mid \sigma \rightarrow \tau$	function type
	$\mid \times(\sigma_1, \dots, \sigma_k)$	tuple with arity $k \geq 0$
	$\mid \dots$	other type constructors according to need
	$\mid \text{amb}[b, b']$	type of ambient name
	$\mid \text{cap}[B]$	type of capability

$$T \in \text{Topics} = \{ \{ \tau_1, \dots, \tau_m \} \mid m \geq 0 \text{ and } \text{arity}(\tau_i) \neq \text{arity}(\tau_j) \text{ for } i \neq j \}$$

When there is no ambiguity, $\times(\sigma)$ is abbreviated to σ and $\times(\sigma_1, \dots, \sigma_k)$ to $(\sigma_1, \dots, \sigma_k)$.

Behaviors

$b \in \text{Beh} ::=$	ε	no traceable action
	$\mid b_1.b_2$	first b_1 then b_2
	$\mid b_1 \mid b_2$	parallel composition
	$\mid \text{put}(\sigma)$	output of type σ (a tuple)
	$\mid \text{get}(\sigma)$	input of type σ (a tuple)
	$\mid \text{diss}$	ambient dissolution
	$\mid \text{fromnow } T$	unordered communication of values with types in T
	$\mid \dots$	other behaviors according to need

$$B \in \text{BehCont} ::= \square \mid b.B \mid B.b \mid b \mid B \mid B \mid b \quad \text{behavior context}$$

Notation: $B[b]$ is the behavior resulting from replacing \square with b in B ; similarly for the behavior context $B[B_1]$.

Figure 4. Syntax of Types and Behaviors.

3.4 Subtyping

We employ a relation $\tau_1 \leq \tau_2$, such that a value of type τ_1 also has type τ_2 . The relation is defined as the least one satisfying the clauses presented in Fig. 5, which can be summarized by stating that the type constructors have the following polarity:

$$\ominus \rightarrow \oplus \quad (\oplus, \dots, \oplus) \quad \text{amb}[\ominus, \oplus] \quad \text{cap}[\oplus]$$

Note that tuples with different arity are incompatible. We now motivate, basically as in [Zim00], the form and polarity of the type $\text{amb}[b_1, b_2]$ where we in addition demand that $b_1 \leq b_2$. For that purpose, consider the process

$$\langle \text{if test}() \text{ then } p \text{ else } q \rangle \mid (n : \text{amb}[b_1, b_2]).Q \quad \text{where } Q \text{ contains subterms } n[P] \text{ and open } n.$$

Assume that p is the name of an ambient in which only values of types τ_1 or τ_2 are allowed to be communicated, and assume that q is the name of an ambient in which only values of types τ_1 or τ_3 are allowed to be communicated; here $\text{arity}(\tau_i) = i$ for $i \in \{1, 2, 3\}$. In order for $n[P]$ to be well-typed, we must demand that in P only values of type τ_1 are communicated (for communicating say values of type τ_2 would be illegal if n happens to be bound to q). On the other hand, the process unleashed by open n may communicate values of any of the types τ_1, τ_2, τ_3 . Therefore we can assign n the type $\text{amb}[\text{fromnow } \{\tau_1\}, \text{fromnow } \{\tau_1, \tau_2, \tau_3\}]$, with the first component expressing what must be required from

$\tau \leq \tau$	(Typ Refl)
If $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$ then $\tau_1 \leq \tau_3$	(Typ Trans)
$\text{int} \leq \text{real}$	(Typ Base)
If $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$ then $\sigma_1 \rightarrow \tau_1 \leq \sigma_2 \rightarrow \tau_2$	(Typ Fun)
If $\forall i \in \{1 \dots k\} : \tau_i \leq \sigma_i$ then $\times(\tau_1, \dots, \tau_k) \leq \times(\sigma_1, \dots, \sigma_k)$	(Typ Tuple)
If $b_0 \leq b_1$ and $b_2 \leq b_3$ then $\text{amb}[b_1, b_2] \leq \text{amb}[b_0, b_3]$	(Typ Amb)
If $B_1 \leq B_2$ then $\text{cap}[B_1] \leq \text{cap}[B_2]$	(Typ Cap)

Figure 5. The Subtyping Relation

processes enclosed by n and the second component expressing what may happen when n is opened. Note that the type of p , $\text{amb}[\text{fromnow } \{\tau_1, \tau_2\}, \text{fromnow } \{\tau_1, \tau_2\}]$, can by subtyping be converted to this type; similarly for q .

3.5 The Type System

Figure 6 defines judgements $E \vdash M : \tau$ and $E \vdash P : b$, where E is an environment mapping names into types (we write $E, n : \sigma$ for the environment E' that behaves as E except that it maps n into σ). We employ a function `type()`, assigning types to constants. Note that if $E \vdash M : \tau$ then there exists $\tau^- \leq \tau$ such that $E \vdash M : \tau^-$ is derived by a structural inference rule, and if $E \vdash P : b$ then there exists $b^- \leq b$ such that $E \vdash P : b^-$ is derived by a structural inference rule.

The side condition in (Proc Repl) prevents us from assigning $!\langle 7 \rangle$ the incorrect behavior $\text{put}(\text{int})$ (but instead we can use (Proc Subsumption) and assign it the behavior $\text{fromnow } \{\text{int}\}$).

The side conditions for (Proc Amb) employ a couple of notions which will be formally defined in Sect. 4.2; below we shall convey the intuition by providing a few examples. First we address the notion of being *safe*.

- The behavior $\text{put}(\text{int}) \mid \text{get}(\text{bool})$ is not safe, since a process which expects a boolean may receive an integer.
- Referring back to “Case 4” from Sect. 1.1 (with $P = \mathbf{0}$), the process enclosed within m has behavior

$$\text{put}(\text{int}) \mid \text{get}(\text{int}).(\text{get}(\text{bool}) \mid \text{put}(\text{bool}))$$

which is safe, since no matter how the parallel behaviors are interleaved in a “well-formed” way then (i) $\text{put}(\text{bool})$ cannot precede $\text{get}(\text{int})$; and (ii) $\text{put}(\text{int})$ cannot immediately precede $\text{get}(\text{bool})$.

- Perhaps surprisingly, the behavior $\text{diss}.\text{put}(\text{int}) \mid \text{get}(\text{bool})$ is considered safe, since nothing bad happens as long as no one attempts to open the enclosing ambient (a process doing that would not be safe).

Concerning the relation $b \rightsquigarrow b_0$, the idea is that b_0 denotes “what remains” of b after its first occurrence of diss . (If b contains no diss we can pick $b_0 = \varepsilon$; see Sect. 7.1.1 for an alternative approach.) For example, with $b = \text{get}(\text{int}).\text{diss} \mid \text{put}(\text{int})$ we have $b \rightsquigarrow \varepsilon$ (since we can infer that $\text{put}(\text{int})$ is performed before diss). And with $b = \text{fromnow } T \mid \text{diss}$, we have $b \rightsquigarrow \text{fromnow } T$.

4 Trace Semantics of Behaviors

In this section we shall define several relations on behaviors, in particular, an ordering relation. We have taken a semantic rather than an axiomatic approach, motivated by the observation that choosing the “right”

Non-structural Rules

(Proc Subsumption)

$$\frac{E \vdash P : b}{E \vdash P : b'} \quad (b \leq b')$$

(Exp Subsumption)

$$\frac{E \vdash M : \sigma}{E \vdash M : \sigma'} \quad (\sigma \leq \sigma')$$

Expressions

(Exp n)

$$\frac{E(n) = \sigma}{E \vdash n : \sigma}$$

(Exp c)

$$\frac{\text{type}(c) = \sigma}{E \vdash c : \sigma}$$

(Exp If)

$$\frac{E \vdash M_1 : \text{bool} \quad E \vdash M_2 : \tau \quad E \vdash M_3 : \tau}{E \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau}$$

(Exp Abs)

$$\frac{E, n : \sigma \vdash M : \tau}{E \vdash \lambda n : \sigma. M : \sigma \rightarrow \tau}$$

(Exp App)

$$\frac{E \vdash M_1 : \sigma \rightarrow \tau \quad E \vdash M_2 : \sigma}{E \vdash M_1 M_2 : \tau}$$

(Exp Tuple)

$$\frac{E \vdash M_1 : \tau_1 \quad \dots \quad E \vdash M_k : \tau_k}{E \vdash \times(M_1, \dots, M_k) : \times(\tau_1, \dots, \tau_k)}$$

(Exp ϵ)

$$\frac{}{E \vdash \epsilon : \text{cap}[\square]}$$

(Exp In)

$$\frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{in } M : \text{cap}[\square]}$$

(Exp Out)

$$\frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{out } M : \text{cap}[\square]}$$

(Exp Open)

$$\frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{open } M : \text{cap}[b' \mid \square]}$$

(Exp Coopen)

$$\frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{coopen } M : \text{cap}[\text{diss.}\square]}$$

(Exp Action)

$$\frac{E \vdash M_1 : \text{cap}[B_1] \quad E \vdash M_2 : \text{cap}[B_2]}{E \vdash M_1.M_2 : \text{cap}[B_1[B_2]]}$$

Processes

(Proc Zero)

$$\frac{}{E \vdash \mathbf{0} : \varepsilon}$$

(Proc Par)

$$\frac{E \vdash P_1 : b_1 \quad E \vdash P_2 : b_2}{E \vdash P_1 \mid P_2 : b_1 \mid b_2}$$

(Proc Repl)

$$\frac{E \vdash P : b}{E \vdash !P : b} \quad (\text{if } (b \mid b) \leq b)$$

(Proc Res)

$$\frac{E, n : \text{amb}[b, b'] \vdash P : b_1}{E \vdash (\nu n : \text{amb}[b, b']).P : b_1}$$

(Proc Amb)

$$\frac{E \vdash M : \text{amb}[b, b'] \quad E \vdash P : b_1}{E \vdash M[P] : \varepsilon} \quad (\text{if } b_1 \text{ safe and } b_1 \rightsquigarrow b \text{ and } b \leq b')$$

(Proc Action)

$$\frac{E \vdash M : \text{cap}[B] \quad E \vdash P : b}{E \vdash M.P : B[b]}$$

(Proc Input)

$$\frac{E, n_1 : \tau_1, \dots, n_k : \tau_k \vdash P : b}{E \vdash (n_1 : \tau_1, \dots, n_k : \tau_k).P : \text{get}(\sigma).b}$$

(Proc Output)

$$\frac{E \vdash M : \times(\tau_1, \dots, \tau_k)}{E \vdash \langle M \rangle : \text{put}(\sigma)}$$

In (Proc Input) and (Proc Output), $\sigma = \times(\tau_1, \dots, \tau_k)$ and $k \geq 0$.

Figure 6. Typing Rules.

set of axioms is often a somewhat ad-hoc exercise. An added advantage of the semantic approach is that in our case it considerably facilitates type checking, as illustrated by the analysis in Sect. 6.

The semantics of behaviors is expressed using the notion of *traces*, defined below. (In fact, it might be possible to formulate the type system in terms of traces; then behaviors would be used “only” as finite representations of certain sets of traces.)

Definition 4.1 (Traces). A trace $tr \in \text{Trace}$ is a finite sequence of actions, where an action $a \in \text{Act}$ is a behavior that is either $\text{put}(\tau)$, $\text{get}(\tau)$, or diss . \square

The semantics $\llbracket b \rrbracket$ of a behavior b belongs to the powerset $\mathcal{P}(\text{Trace})$, and is given by

$$\begin{aligned} \llbracket \varepsilon \rrbracket &= \{\bullet\} & \llbracket \text{diss} \rrbracket &= \{\text{diss}\} \\ \llbracket b_1.b_2 \rrbracket &= \llbracket b_1 \rrbracket \diamond \llbracket b_2 \rrbracket & \llbracket b_1 \mid b_2 \rrbracket &= \llbracket b_1 \rrbracket \parallel \llbracket b_2 \rrbracket \\ \llbracket \text{put}(\tau) \rrbracket &= \{\text{put}(\tau)\} & \llbracket \text{get}(\tau) \rrbracket &= \{\text{get}(\tau)\} \\ \llbracket \text{fromnow } T \rrbracket &= \{tr \mid \forall a \text{ occurring in } tr : \exists \tau \in T : a \in \{\text{put}(\tau), \text{get}(\tau)\}\} \end{aligned}$$

Here \bullet denotes the empty sequence, $tr_1 \diamond tr_2$ denotes the concatenation of tr_1 and tr_2 which trivially lifts to sets of traces (Tr ranges over such), and $Tr_1 \parallel Tr_2$ denotes all traces that can be formed by arbitrarily interleaving a trace in Tr_1 with a trace in Tr_2 . Note that \diamond and \parallel are both associative operators (and the latter even commutative) on sets of traces, with $\{\bullet\}$ as neutral element.

Since a simple structural induction shows that $\llbracket b \rrbracket \neq \emptyset$ for all b , we trivially have

Lemma 4.2. *If $E \vdash P : b$ then $\llbracket b \rrbracket \neq \emptyset$.* \square

Consider the run-time behavior of a process not interacting with other processes. Each input must necessarily be preceded by an output with the same arity, and an error occurs if the type of the value being output is not a subtype of what the inputting process expects. This observation motivates the following definition:

Definition 4.3 (Comm). A trace tr belongs to Comm if $tr = \text{put}(\tau) \text{get}(\sigma)$ with $\text{arity}(\tau) = \text{arity}(\sigma)$. If in addition it holds that $\tau \leq \sigma$ we say that $tr \in \text{WtComm}$, the set of *well-typed communications*. \square

EXAMPLE 4.4. In Sect. 3.5 we considered the behavior $b = \text{put}(\text{int}) \mid \text{get}(\text{int}).(\text{get}(\text{bool}) \mid \text{put}(\text{bool}))$. It is easy to see that $\llbracket b \rrbracket$ consists of the 8 traces depicted below:

```

put(int) get(int) put(bool) get(bool)
put(int) get(int) get(bool) put(bool)
get(int) put(int) put(bool) get(bool)
get(int) put(int) get(bool) put(bool)
get(int) put(bool) put(int) get(bool)
get(int) get(bool) put(int) put(bool)
get(int) put(bool) get(bool) put(int)
get(int) get(bool) put(bool) put(int)

```

Only the first of these traces belongs to Comm^* (and even to WtComm^*). The other traces, however, are still relevant if b is the behavior of a process placed in a non-empty context. \square

4.1 Ordering on Traces and Behaviors

In order to define the relation \leq on Beh (cf. Sect. 3.3), we now define relations \leq on Act , Trace , and $\mathcal{P}(\text{Trace})$:

- on Act , \leq is the least reflexive and transitive relation satisfying that if $\tau \leq \sigma$ then $\text{put}(\tau) \leq \text{put}(\sigma)$ and $\text{get}(\sigma) \leq \text{get}(\tau)$;
- the relation \leq on Act extends pointwise to a relation \leq on Trace (note that if $tr_1 \leq tr_2$ then tr_1 and tr_2 have the same length);
- $Tr_1 \leq Tr_2$ iff for all $tr_1 \in Tr_1$ there exists $tr_2 \in Tr_2$ such that $tr_1 \leq tr_2$.

Note that $_ \parallel _$ is a monotone operator on sets of traces. That is, if $Tr_1 \leq Tr'_1$ and $Tr_2 \leq Tr'_2$ then $(Tr_1 \parallel Tr_2) \leq (Tr'_1 \parallel Tr'_2)$.

Lemma 4.5. *Assume that $tr \leq tr'$. Then $tr \in \text{Comm}$ if and only if $tr' \in \text{Comm}$. Moreover, if $tr' \in \text{WtComm}$ then $tr \in \text{WtComm}$. \square*

Proof. We can assume that $tr = \text{put}(\tau) \text{get}(\sigma)$ and $tr' = \text{put}(\tau') \text{get}(\sigma')$ (as otherwise the claims vacuously hold). Since $tr \leq tr'$ then $\tau \leq \tau'$ and $\sigma' \leq \sigma$, thus $\text{arity}(\tau) = \text{arity}(\tau')$ and $\text{arity}(\sigma) = \text{arity}(\sigma')$. Therefore $\text{arity}(\tau) = \text{arity}(\sigma)$ if and only if $\text{arity}(\tau') = \text{arity}(\sigma')$, implying the first claim. The last claim follows from the fact that if $\tau' \leq \sigma'$ then also $\tau \leq \sigma$. \square

Definition 4.6 (Behavior subsumption). $b_1 \leq b_2$ iff $\llbracket b_1 \rrbracket \leq \llbracket b_2 \rrbracket$. \square

Our definition of the relations $b_1 \leq b_2$ and $\tau \leq \sigma$ may seem circular, but is not: the development in this section shows how a relation on level i types gives rise to a relation on level i behaviors, whereas Fig. 5 shows how to define a relation on level 0 types, and how a relation on level i behaviors gives rise to a relation on level $i + 1$ types (since, thanks to the restriction to level 0 behaviors in Def. 3.1, it induces a relation on level i behavior contexts). To be more precise: inductively we define for each $i \geq 0$ first a relation $\tau \leq_i \sigma$ and then a relation $b \leq_i b'$; finally define \leq as $\bigcup_{i \geq 0} \leq_i$. By induction on i we can prove that these relations behave as expected: if τ and σ are of level i and $\tau \leq \sigma$ then even $\tau \leq_i \sigma$, and if b and b' are of level i and $b \leq b'$ then even $b \leq_i b'$.

It is straightforward to verify that the relations \leq are reflexive and transitive. We write \equiv for the equivalence relation induced by \leq . That is, we write $b_1 \equiv b_2$ whenever $b_1 \leq b_2$ and $b_2 \leq b_1$.

Lemma 4.7. *The operators “|” and “.” on behaviors respect the relation \leq ; thus \equiv is a congruence on behaviors wrt. these operators. Moreover, modulo \equiv it holds that “|” is associative and commutative and that “.” is associative, both with ε as neutral element. Also note that $\varepsilon \equiv \text{fromnow } \emptyset$. \square*

The proof for Lemma 4.7 is conducted by decomposing the lemma into easily verifiable claims about relations on sets of traces.

Lemma 4.8. *Given the behaviors b_0, b_1 and b_2 . Then $b_0.(b_1 | b_2) \leq (b_0.b_1) | b_2$. \square*

Proof. Assume that $tr \in \llbracket b_0.(b_1 | b_2) \rrbracket$. Then there exists $tr_0 \in \llbracket b_0 \rrbracket$ and $tr_{12} \in \llbracket b_1 | b_2 \rrbracket$ such that $tr = tr_0 \diamond tr_{12}$, and there also exists $tr_1 \in \llbracket b_1 \rrbracket$ and $tr_2 \in \llbracket b_2 \rrbracket$ such that $tr_{12} \in \{tr_1\} | \{tr_2\}$. But then $tr = tr_0 \diamond tr_{12} \in \{tr_0 \diamond tr_1\} | \{tr_2\}$, showing that $tr \in \llbracket (b_0.b_1) | b_2 \rrbracket$ as desired. \square

Lemma 4.9. *Given the behavior contexts B_1, B_2 , and the behavior b . Then $B_1[B_2[b]] = (B_1[B_2])[b]$. \square*

Proof. Structural induction in B_1 . If $B_1 = \square$ the claim is trivial. Otherwise, let $B_1 = b_0.B_0$ (the other cases are similar). Then the induction hypothesis enables us to derive the desired relation $B_1[B_2[b]] = b_0.(B_0[B_2[b]]) = b_0.((B_0[B_2])[b]) = (b_0.B_0[B_2])[b] = (B_1[B_2])[b]$. \square

The next two lemmas are used in the proof of Lemma 4.13, which is needed in Sect. 6 to show the decidability of type checking. Lemma 4.13 states that we can determine if two behavior contexts are in subtype relation by constructing a single behavior, of the form test.test , and then checking if the behaviors that result by filling the holes with test.test are in subtype relation. Here test is an action (and thus also a behavior) that *tests* the behavior contexts in question, according to the following

Definition 4.10. Given a behavior context B . We say that an action test *tests* B if test is incompatible with all behaviors occurring as part of B . \square

Assume that test tests B . Then $\llbracket \text{test} \rrbracket = \{\text{test}\}$, and test also tests all subcontexts of B . Furthermore, it is easy to see by induction in B that all $tr \in \llbracket B[\text{test.test}] \rrbracket$ can uniquely be written on the form $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$, where all actions occurring in tr_1, tr_2 , or tr_3 are incompatible with test .

Lemma 4.11. *Given B a behavior context and b a behavior, and assume that test tests B . Let $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ belong to $\llbracket B[\text{test.test}] \rrbracket$, and let tr_0 belong to $\{tr_2\} | \llbracket b \rrbracket$. Then $tr_1 \diamond tr_0 \diamond tr_3$ belongs to $\llbracket B[b] \rrbracket$. \square*

Proof. See Appendix B. □

Lemma 4.12. *Given B a behavior context and b a behavior, and assume that test tests B . Let tr belong to $\llbracket B[b] \rrbracket$. Then there exists $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ in $\llbracket B[\text{test.test}] \rrbracket$ such that we can write $tr = tr_1 \diamond tr_0 \diamond tr_3$ with $tr_0 \in \{tr_2\} \parallel \llbracket b \rrbracket$. □*

Proof. See Appendix B. □

Lemma 4.13. *Given B_1 and B_2 behavior contexts, we can construct a level zero behavior test such that the following conditions are equivalent:*

(a) $B_1 \leq B_2$

(b) $B_1[b] \leq B_2[b]$ for all b (regardless of level)

(c) $B_1[\text{test.test}] \leq B_2[\text{test.test}]$. □

Proof. With m the maximal arity of a tuple occurring inside B_1 or B_2 , we choose test to be a behavior of the form $\text{put}(\times(\tau_1, \dots, \tau_{m+1}))$ where $\tau_i = \text{int}$ for $i \in \{1, \dots, m+1\}$. Then clearly test tests B_1 as well as B_2 .

Trivially, (b) implies (a) which further implies (c), so it suffices to show that (c) implies (b). For this purpose, assume that b has been given. Let tr belong to $\llbracket B_1[b] \rrbracket$, then our task is to find $tr^+ \in \llbracket B_2[b] \rrbracket$ with $tr^+ \geq tr$. By Lemma 4.12, there exists $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ in $\llbracket B_1[\text{test.test}] \rrbracket$ such that we can write $tr = tr_1 \diamond tr_0 \diamond tr_3$ with $tr_0 \in \{tr_2\} \parallel \llbracket b \rrbracket$. Using our assumption (c), there exists a trace $tr_1^+ \diamond \text{test} \diamond tr_2^+ \diamond \text{test} \diamond tr_3^+$ in $\llbracket B_2[\text{test.test}] \rrbracket$ with $tr_1^+ \diamond \text{test} \diamond tr_2^+ \diamond \text{test} \diamond tr_3^+ \geq tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$. Since test is incompatible with every tr_i and every tr_i^+ ($i = 1, 2, 3$), we deduce that $tr_i^+ \geq tr_i$ for all $i \in \{1, 2, 3\}$, implying that there exists $tr_0^+ \in \{tr_2^+\} \parallel \llbracket b \rrbracket$ with $tr_0^+ \geq tr_0$. We can then use $tr^+ = tr_1^+ \diamond tr_0^+ \diamond tr_3^+$, since by Lemma 4.11 we infer that tr^+ belongs to $\llbracket B_2[b] \rrbracket$, and clearly $tr^+ \geq tr$. □

4.2 Safety and Related Notions

The following definition captures the intuition that if P can be assigned a *safe* behavior then all communications performed by P will be well-typed—at least until the ambient enclosing P is dissolved.

Definition 4.14 (Safe behavior). A behavior b is safe if for all traces $tr \in \llbracket b \rrbracket$ it is impossible to write $tr = tr_0 \diamond tr_1 \diamond tr_2$ with $tr_0 \in \text{Comm}^*$ and $tr_1 \in \text{Comm} \setminus \text{WtComm}$. □

EXAMPLE 4.15. Referring back to Example 4.4, where the traces of a behavior b were listed, we can now demonstrate that b is in fact safe (as claimed in Sec. 3.5). For the first trace in b belongs to WtComm^* ; the second trace can be written as $tr_0 \diamond tr$ with $tr_0 \in \text{WtComm}$ and tr not of the form $tr_1 \diamond tr_2$ for any $tr_1 \in \text{Comm}$; and none of the remaining traces are of the form $tr_1 \diamond tr_2$ with $tr_1 \in \text{Comm}$. □

As an example of an unsafe behavior, consider $b = \text{put}(\text{int}) \mid \text{get}(\text{int}).\text{get}(\text{bool}).\varepsilon \mid \text{put}(\text{int})$ (which might be assigned to the process $\langle 7 \rangle \mid (x : \text{int}).(y : \text{bool}).\text{if } y \text{ then in } n \text{ else in } m \mid \langle 8 \rangle$). For $\llbracket b \rrbracket$ contains the trace $\text{put}(\text{int}) \text{get}(\text{int}) \text{put}(\text{int}) \text{get}(\text{bool})$ which belongs to $\text{Comm} \diamond (\text{Comm} \setminus \text{WtComm})$.

Lemma 4.16. *Suppose $b \leq b^+$ with b^+ safe. Then also b is safe.* □

Proof. Assume b is not safe, that is there exists $tr \in \llbracket b \rrbracket$ such that we can write $tr = tr_0 \diamond tr_1 \diamond tr_2$ with $tr_0 \in \text{Comm}^*$ and $tr_1 \in \text{Comm} \setminus \text{WtComm}$. Since $b \leq b^+$ there exists $tr^+ \in \llbracket b^+ \rrbracket$ with $tr^+ \geq tr$, and we can thus write $tr^+ = tr_0^+ \diamond tr_1^+ \diamond tr_2^+$ with $tr_0^+ \geq tr_0$, $tr_1^+ \geq tr_1$, and $tr_2^+ \geq tr_2$. By Lemma 4.5 we infer that $tr_0^+ \in \text{Comm}^*$, that $tr_1^+ \in \text{Comm}$, and that $tr_1^+ \notin \text{WtComm}$. But this conflicts with b^+ being safe. □

Lemma 4.17. *Suppose that $\llbracket b' \rrbracket \cap \text{Comm}^* \neq \emptyset$. If $b'.b$ is safe then also b is safe.* □

Proof. Assume b is not safe, that is there exists $tr \in \llbracket b \rrbracket$ such that we can write $tr = tr_0 \diamond tr_1 \diamond tr_2$ with $tr_0 \in \text{Comm}^*$ and $tr_1 \in \text{Comm} \setminus \text{WtComm}$. Let tr' belong to $\llbracket b' \rrbracket$ and to Comm^* , and let $tr'' = tr' \diamond tr$. Then $tr'' \in \llbracket b'.b \rrbracket$, and $tr'' = tr' \diamond tr_0 \diamond tr_1 \diamond tr_2$ with $tr' \diamond tr_0 \in \text{Comm}^*$. But this conflicts with $b'.b$ being safe. \square

Definition 4.18 ($b \rightsquigarrow b'$). The relation $b \rightsquigarrow b'$ amounts to the following property: whenever there exists $tr_1 \in \text{Comm}^*$ and tr such that $tr_1 \diamond \text{diss} \diamond tr \in \llbracket b \rrbracket$, then there exists $tr' \in \llbracket b' \rrbracket$ with $tr \leq tr'$. \square

Lemma 4.19. Assume that $b_1^- \leq b_1$ and that $b_0 \leq b_0^+$. If $b_1 \rightsquigarrow b_0$ then also $b_1^- \rightsquigarrow b_0^+$. \square

Proof. Assume that $tr_1^- \diamond \text{diss} \diamond tr^- \in \llbracket b_1^- \rrbracket$ with $tr_1^- \in \text{Comm}^*$. Since $b_1^- \leq b_1$, there exists $tr_1 \geq tr_1^-$ and $tr \geq tr^-$ such that $tr_1 \diamond \text{diss} \diamond tr \in \llbracket b_1 \rrbracket$. By Lemma 4.5, $tr_1 \in \text{Comm}^*$, so since $b_1 \rightsquigarrow b_0$ there exists $tr_0 \in \llbracket b_0 \rrbracket$ such that $tr \leq tr_0$. Since $b_0 \leq b_0^+$, there exists $tr_0^+ \in \llbracket b_0^+ \rrbracket$ such that $tr_0 \leq tr_0^+$. This is as desired, since $tr^- \leq tr_0^+$. \square

Lemma 4.20. Suppose that $\llbracket b_0 \rrbracket \cap \text{Comm}^* \neq \emptyset$. If $b_0.b \rightsquigarrow b'$ then also $b \rightsquigarrow b'$. \square

Proof. Assume that $tr_1 \diamond \text{diss} \diamond tr \in \llbracket b \rrbracket$ with $tr_1 \in \text{Comm}^*$. Let tr_0 belong to $\llbracket b_0 \rrbracket$ and to Comm^* . Then $tr_0 \diamond tr_1 \diamond \text{diss} \diamond tr \in \llbracket b_0.b \rrbracket$ with $tr_0 \diamond tr_1 \in \text{Comm}^*$, so since $b_0.b \rightsquigarrow b'$ there as desired exists $tr' \in \llbracket b' \rrbracket$ with $tr \leq tr'$. \square

5 Subject Reduction

In this section we shall show that our type system is semantically sound. This property is formulated as a subject reduction result (Theorem 5.8), intuitively stating that “well-typed processes communicate according to their behavior” and also stating that “well-typed safe processes never evolve into ill-typed processes”. (The latter “safety” property demonstrates that the process $((n : \text{int}).\langle n + 7 \rangle) \mid \langle \text{true} \rangle$, though typeable, cannot be assigned a safe behavior, since it evolves into $\text{true} + 7$ which clearly cannot be typed.) As a preparation for showing this result, we state a few standard lemmas. Also, we need an assumption about the relationship between evaluating constants and typing constants:

if $\delta(c, V) = V'$ and $\text{type}(c) = \tau \rightarrow \tau'$ and $E \vdash V : \tau$ then $E \vdash V' : \tau'$.

For an example, this property holds non-vacuously for $c = +$, $V = \times(1, 2)$, $V' = 3$, $\tau = \times(\text{int}, \text{int})$, $\tau' = \text{int}$.

Lemma 5.1 (Swapping). Assume that $E_1, n_1 : \sigma_1, n_2 : \sigma_2, E_2 \vdash M : \tau$ with $n_1 \neq n_2$. Then it follows that $E_1, n_2 : \sigma_2, n_1 : \sigma_1, E_2 \vdash M : \tau$, with a derivation of the same shape. Similarly for $E_1, n_1 : \sigma_1, n_2 : \sigma_2, E_2 \vdash P : b$. \square

Proof. By induction on derivations. \square

Lemma 5.2 (Weakening). Assume that $E \vdash M : \tau$, and that n is a name not in $\text{names}(M)$. Then for all σ it follows that $E, n : \sigma \vdash M : \tau$, with a derivation of the same shape. Similarly for a judgment $E \vdash P : b$ (with $n \notin \text{names}(P)$). \square

Proof. The proof is by induction on derivations, applying Lemma 5.1 to deal with the cases (Exp Abs), (Proc Res), (Proc Input). \square

Lemma 5.3 (Strengthening). Assume that $E, n : \sigma \vdash M : \tau$, with n not in $\text{names}(M)$. Then also $E \vdash M : \tau$ holds, and with a derivation of the same shape. Similarly for $E, n : \sigma \vdash P : b$ with $n \notin \text{names}(P)$. \square

Proof. Similar to the proof of Lemma 5.2. \square

Lemma 5.4 (Substitution). Assume that $E, n_1 : \tau_1, \dots, n_k : \tau_k \vdash M : \sigma$ (with $n_1 \dots n_k$ distinct), and that there exists $V_1 \dots V_k$ such that for all $i \in \{1 \dots k\}$ it holds that $E \vdash V_i : \tau_i$ and that M is non-conflicting with $\{n_i\} \cup \text{names}(V_i)$. Then $E \vdash M[n_i := V_i] : \sigma$. Similarly, if $E, n_1 : \tau_1, \dots, n_k : \tau_k \vdash P : b$ and for all i it holds that $E \vdash V_i : \tau_i$ and that P is non-conflicting with $\{n_i\} \cup \text{names}(V_i)$ then $E \vdash P[n_i := V_i] : b$. \square

Proof. The proof is by induction in the size of the typing derivation for M (resp. P). We do a case analysis on the inference rule applied, but only list two of the cases; the remaining follow by straightforward applications of the induction hypothesis, except for (Exp Abs) and (Proc Input) which are handled as (Proc Res) below.

(Exp n). Here M is a name n . If $n = n_i$ for some $i \in \{1 \dots k\}$, we infer that $\sigma = \tau_i$. The claim is then $E \vdash V_i : \tau_i$, which is among our assumptions. If $n \neq n_i$ for all $i \in \{1 \dots k\}$, we infer that $\sigma = E(n)$. But then we have $E \vdash n : \sigma$, as desired.

(Proc Res). Here P takes the form $(\nu n : \tau).P_0$, and $E, n_1 : \tau_1, \dots, n_k : \tau_k \vdash P : b$ holds because

$$E, n_1 : \tau_1, \dots, n_k : \tau_k, n : \tau \vdash P_0 : b. \quad (1)$$

Note that for all $i \in \{1 \dots k\}$ it holds (since P is non-conflicting with $\{n_i\} \cup \text{names}(V_i)$) that $n_i \neq n$ and $n \notin \text{names}(V_i)$ and that P_0 is non-conflicting with $\{n_i\} \cup \text{names}(V_i)$. We can thus apply Lemma 5.2 to infer that $E, n : \tau \vdash V_i : \tau_i$, and (repeatedly) apply Lemma 5.1 to infer that

$$E, n : \tau, n_1 : \tau_1, \dots, n_k : \tau_k \vdash P_0 : b \quad (2)$$

by a derivation of the same shape as the one for (1). We can thus apply the induction hypothesis on (2) to infer that $E, n : \tau \vdash P_0[n_i := V_i] : b$. But then an application of (Proc Res) yields $E \vdash (\nu n : \tau).P_0[n_i := V_i] : b$, which is as desired since $P[n_i := V_i] = (\nu n : \tau).P_0[n_i := V_i]$. \square

Lemma 5.5 (Reduction of subprocess). Assume that with b safe it holds that $E \vdash \mathcal{P}[P]_p : b$. Then there exists E_1 and safe b_1 such that $E_1 \vdash P : b_1$. Moreover, if there exists b_0 and b'_1 with $b_0.b'_1 \leq b_1$ such that $E_1 \vdash Q : b'_1$, then there exists b' with $b_0.b' \leq b$ such that $E \vdash \mathcal{P}[Q]_p : b'$. \square

Proof. The proof is by induction in \mathcal{P} . We do a case analysis on \mathcal{P} , where only three cases are possible (as \mathcal{P} expects a process, not an expression, in its hole):

$\mathcal{P} = \square_p$. Choose $E_1 = E$, $b_1 = b$, and $b' = b'_1$. Then the claim clearly holds.

$\mathcal{P} = (\nu n : \sigma).P_0$. We assume $E \vdash (\nu n : \sigma).P_0[P]_p : b$, implying $E, n : \sigma \vdash P_0[P]_p : b$. Inductively there exists E_1 and safe b_1 such that $E_1 \vdash P : b_1$. Now assume that $E_1 \vdash Q : b'_1$ with $b_0.b'_1 \leq b_1$. Inductively we can find b' with $b_0.b' \leq b$ such that $E, n : \sigma \vdash P_0[Q]_p : b'$. But then also $E \vdash \mathcal{P}[Q]_p : b'$.

$\mathcal{P} = \mathcal{P}_2 \mid R$. Since $E \vdash \mathcal{P}_2[P]_p \mid R : b$, there exists b_2 and b_3 with $b_2 \mid b_3 \leq b$ such that $E \vdash \mathcal{P}_2[P]_p : b_2$ and $E \vdash R : b_3$.

We now prove that b_2 is safe. If this is not the case, there exists $tr_0 \diamond tr_1 \diamond tr_2 \in \llbracket b_2 \rrbracket$ with $tr_0 \in \text{Comm}^*$ and $tr_1 \in \text{Comm} \setminus \text{WtComm}$. By Lemma 4.2 we can find $tr_3 \in \llbracket b_3 \rrbracket$, thus $tr_0 \diamond tr_1 \diamond tr_2 \diamond tr_3 \in \llbracket b_2 \mid b_3 \rrbracket$ which is a contradiction since $b_2 \mid b_3$ is safe (by Lemma 4.16).

We can thus apply the induction hypothesis to find E_1 and safe b_1 such that $E_1 \vdash P : b_1$. Now assume that $E_1 \vdash Q : b'_1$ with $b_0.b'_1 \leq b_1$. Inductively, there exists b'_2 with $b_0.b'_2 \leq b_2$ such that $E \vdash \mathcal{P}_2[Q]_p : b'_2$. Define $b' = b'_2 \mid b_3$. Then $E \vdash \mathcal{P}[Q]_p : b'$, and by Lemma 4.8 also $b_0.b' \leq (b_0.b'_2) \mid b_3 \leq b_2 \mid b_3 \leq b$. \square

Lemma 5.6 (Subject reduction for expressions). Suppose $M_1 \longrightarrow M_2$. If $E \vdash M_1 : \tau$ then also $E \vdash M_2 : \tau$. \square

Proof. Induction in the derivation of $M_1 \longrightarrow M_2$, where the only non-trivial cases are (Red Delta) and (Red Beta). In both cases we assume that $E \vdash MV : \tau$, which must be derived using (Exp App) and zero or more applications of (Exp Subsumption) from judgments

$$E \vdash M : \sigma \rightarrow \tau^- \text{ and } E \vdash V : \sigma$$

where $\tau^- \leq \tau$. There exists $\sigma^+ \geq \sigma$ and $\tau^{--} \leq \tau^-$ such that the former judgment is derived using zero or more occurrences of (Exp Subsumption) from a judgment

$$E \vdash M : \sigma^+ \rightarrow \tau^{--} \quad (3)$$

which is derived using (Exp c) or (Exp Abs). Note that $E \vdash V : \sigma^+$, and that $\tau^{--} \leq \tau$.

In the case (Red Delta), M is a constant c with $\text{type}(c) = \sigma^+ \rightarrow \tau^{--}$. Our initial assumptions about the relationship between δ and $\text{type}()$ imply that $E \vdash \delta(c, V) : \tau^{--}$. But then also $E \vdash \delta(c, V) : \tau$, as desired.

In the case (Red Beta), M takes the form $\lambda n : \sigma^+. M_1$ with M non-conflicting with $\text{names}(V)$, and the premise of (3) takes the form $E, n : \sigma^+ \vdash M_1 : \tau^{--}$. We infer that M_1 is non-conflicting with $\text{names}(V)$ and with n , so we can apply Lemma 5.4 to deduce that $E \vdash M_1[n := V] : \tau^{--}$. But then also $E \vdash M_1[n := V] : \tau$, as desired. \square

Lemma 5.7 (Subject congruence). *Suppose that $P \equiv Q$. Then $E \vdash P : b$ if and only if $E \vdash Q : b$.* \square

The proof is given in Appendix B. We do not know if Lemma 5.7 still holds if the equivalence rule $!P \equiv P \mid !P$ is included in the definition of structural equivalence. This is why our operational semantics includes the reduction rule $!P \rightarrow P \mid !P$ instead.

The formulation of subject reduction for processes states that if a process having behavior b performs a step labeled ℓ then the resulting process can be assigned a behavior that denotes “what remains of b after ℓ ”. To formalize this, we employ a relation $\ell \sim b_0$ that is defined by stipulating that

$$\begin{aligned} \epsilon &\sim \epsilon \\ \text{comm}(\tau) &\sim \text{put}(\tau^-). \text{get}(\tau^+) \text{ if } \tau^- \leq \tau \leq \tau^+ \end{aligned}$$

Note that $\ell \sim b_0$ implies that $\llbracket b_0 \rrbracket \cap \text{Comm}^* \neq \emptyset$ (preconditions for Lemmas 4.17 and 4.20).

Theorem 5.8 (Subject reduction for processes). *Suppose that $P \xrightarrow{\ell} Q$. If with b safe it holds that $E \vdash P : b$ then there exists b_0 with $\ell \sim b_0$ and safe b' such that $E \vdash Q : b'$ and $b_0.b' \leq b$.* \square

Proof. The proof is by induction on the derivation of $P \xrightarrow{\ell} Q$, with a case analysis on the last rule used. By using Lemmas 4.16 and 4.17, we infer that if $\ell \sim b_0$ and $b_0.b' \leq b$ then b' will be safe, and that wlog. we can assume that the last rule used to derive $E \vdash P : b$ is a structural one (i.e., not (Proc Subsumption)).

(Red In). Our assumptions are that $n[\text{in } m.P \mid Q \mid m[R] \xrightarrow{\epsilon} m[n[P \mid Q] \mid R]$ and $E \vdash n[\text{in } m.P \mid Q \mid m[R] : \bar{b}$. The two top-level parallel processes have been typed using instantiations of the rule (Proc Amb) of the form

$$\frac{E \vdash n : \text{amb}[b_n, b_n^+] \quad E \vdash \text{in } m.P \mid Q : b_{12}}{E \vdash n[\text{in } m.P \mid Q] : \epsilon} \text{ and } \frac{E \vdash m : \text{amb}[b_m, b_m^+] \quad E \vdash R : b_3}{E \vdash m[R] : \epsilon}$$

where b_{12} is safe and satisfies $b_{12} \rightsquigarrow b_n$ with $b_n \leq b_n^+$, and where b_3 is safe and satisfies $b_3 \rightsquigarrow b_m$ with $b_m \leq b_m^+$. Clearly $\epsilon \mid \epsilon \leq b$ holds, implying $\epsilon \leq b$. From the rightmost premise of the leftmost inference we infer that there exists B_0, b_1, b_2 such that $E \vdash \text{in } m : \text{cap}[B_0]$, $E \vdash P : b_1$, and $E \vdash Q : b_2$, with $\text{cap}[\square] \leq \text{cap}[B_0]$ and with $B_0[b_1] \mid b_2 \leq b_{12}$. Thus $\square \leq B_0$ which by Lemma 4.13 entails $b_1 \leq B_0[b_1]$, implying $b_1 \mid b_2 \leq b_{12}$.

From the above we can derive $E \vdash P \mid Q : b_{12}$, enabling us to apply (Proc Amb) to get $E \vdash n[P \mid Q] : \epsilon$. Then we can derive $E \vdash n[P \mid Q] \mid R : b_3$, enabling us to apply (Proc Amb) to get, after also applying (Proc Subsumption), $E \vdash m[n[P \mid Q] \mid R] : b$. This yields the claim, with $b' = b$ and $b_0 = \epsilon$.

(Red Out). Our assumptions are $m[n[\text{out } m.P \mid Q] \mid R] \xrightarrow{\epsilon} n[P \mid Q] \mid m[R]$ and $E \vdash m[n[\text{out } m.P \mid Q] \mid R] : \bar{b}$. The topmost ambient has been typed using an instantiation of the rule (Proc Amb) of the form

$$\frac{E \vdash m : \text{amb}[b_m, b_m^+] \quad E \vdash n[\text{out } m.P \mid Q] \mid R : b_3}{E \vdash m[n[\text{out } m.P \mid Q] \mid R] : \epsilon}$$

where b_3 is safe and satisfies $b_3 \rightsquigarrow b_m$ with $b_m \leq b_m^+$, and where $\varepsilon \leq b$. We infer that the ambient in parallel with R has been typed using an instantiation of the rule (Proc Amb) of the form

$$\frac{E \vdash n : \text{amb}[b_n, b_n^+] \quad E \vdash \text{out } m.P \mid Q : b_{12}}{E \vdash n[\text{out } m.P \mid Q] : \varepsilon}$$

where b_{12} is safe and satisfies $b_{12} \rightsquigarrow b_n$ with $b_n \leq b_n^+$. From the rightmost premise of this inference we infer that there exists B_0, b_1, b_2 such that $E \vdash \text{out } m : \text{cap}[B_0]$, $E \vdash P : b_1$, and $E \vdash Q : b_2$, with $\text{cap}[\square] \leq \text{cap}[B_0]$ and with $B_0[b_1] \mid b_2 \leq b_{12}$. Thus $\square \leq B_0$ which by Lemma 4.13 entails $b_1 \leq B_0[b_1]$, implying $b_1 \mid b_2 \leq b_{12}$. Additionally, there exists b'_3 such that $E \vdash R : b'_3$ and $\varepsilon \mid b'_3 \leq b_3$, implying $b'_3 \leq b_3$ and thus also $E \vdash R : b_3$.

From the above we can derive $E \vdash P \mid Q : b_{12}$, enabling us to apply (Proc Amb) to get $E \vdash n[P \mid Q] : \varepsilon$, and we can also apply (Proc Amb) to get $E \vdash m[R] : \varepsilon$. Thus we can arrive at $E \vdash n[P \mid Q] \mid m[R] : b$, which yields the claim with $b' = b$ and $b_0 = \varepsilon$.

(Red Open). Our assumptions are that the process $\text{open } n.P \mid n[\text{coopen } n.Q \mid R] \xrightarrow{\varepsilon} P \mid Q \mid R$, and that $\bar{E} \vdash \text{open } n.P \mid n[\text{coopen } n.Q \mid R] : b$. Let $E(n) = \text{amb}[b_4, b_5]$ with $b_4 \leq b_5$. The rightmost parallel process has been typed using an instantiation of (Proc Amb) of the form

$$\frac{E \vdash n : \text{amb}[b'_4, _] \quad E \vdash \text{coopen } n.Q \mid R : b_{23}}{E \vdash n[\text{coopen } n.Q \mid R] : \varepsilon}$$

where $b_{23} \rightsquigarrow b'_4$. Since the left premise has been derived using (Exp n) followed by zero or more applications of (Exp Subsumption), from the polarity rule for ambient types we have that $b'_4 \leq b_4$. Concerning the right premise, we deduce that there exists b_2 and b_3 with

$$E \vdash Q : b_2 \text{ and } E \vdash R : b_3 \tag{4}$$

such that $E \vdash \text{coopen } n.Q : B_2[b_2]$ where $B_2[b_2] \mid b_3 \leq b_{23}$ and where $\text{diss}.\square \leq B_2$, implying (using Lemma 4.13) that $\text{diss}.b_2 \leq B_2[b_2]$ and therefore $\text{diss}.b_2 \mid b_3 \leq b_{23}$. By combining these results, we deduce by Lemma 4.19 that

$$\text{diss}.b_2 \mid b_3 \rightsquigarrow b_5. \tag{5}$$

The leftmost parallel process $\text{open } n.P$ has been typed using an instantiation of (Proc Action) of the form

$$\frac{E \vdash \text{open } n : \text{cap}[B_1] \quad E \vdash P : b_1}{E \vdash \text{open } n.P : B_1[b_1]} \tag{6}$$

where $B_1[b_1] \leq b$. The derivation of the left premise contains an instance of (Exp Open) with premise $E \vdash n : \text{amb}[_, b'_5]$, where $b_5 \leq b'_5$ (by the polarity rule for ambient types) and where $\text{cap}[b'_5 \mid \square] \leq \text{cap}[B_1]$. Therefore $(b'_5 \mid \square) \leq B_1$ which by Lemma 4.13 implies $b'_5 \mid b_1 \leq B_1[b_1]$, enabling us to derive

$$b_5 \mid b_1 \leq b. \tag{7}$$

We want to prove that $E \vdash P \mid Q \mid R : b$, as then our claim will follow with $b' = b$ and $b_0 = \varepsilon$, and due to (6) and (4) this can be accomplished by showing $b_1 \mid b_2 \mid b_3 \leq b$. So let $tr \in \llbracket b_1 \mid b_2 \mid b_3 \rrbracket$ be given. There must exist tr_1, tr_2, tr_3 such that $tr_i \in \llbracket b_i \rrbracket$ for $i \in \{1, 2, 3\}$ and such that $tr \in \{tr_1\} \parallel \{tr_{23}\}$ for some $tr_{23} \in \{tr_2\} \parallel \{tr_3\}$. Since $\text{diss} \diamond tr_{23}$ belongs to $\llbracket (\text{diss}.b_2) \mid b_3 \rrbracket$ we infer from (5) that there exists $tr_5 \in \llbracket b_5 \rrbracket$ with $tr_5 \geq tr_{23}$. We can then clearly find $tr^+ \geq tr$ such that $tr^+ \in \{tr_1\} \parallel \{tr_5\}$. This shows that $tr^+ \in \llbracket b_1 \rrbracket \parallel \llbracket b_5 \rrbracket = \llbracket b_1 \mid b_5 \rrbracket$, so by (7) there exists $tr^{++} \in \llbracket b \rrbracket$ with $tr^{++} \geq tr^+$. As then $tr \leq tr^{++}$, this is as desired.

(Red Comm). Our assumptions are that $(n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle \times(V_1, \dots, V_k) \rangle \xrightarrow{\text{comm}(\tau)} P[n_i := V_i]$ with $\tau = \times(\sigma_1, \dots, \sigma_k)$, and that $E \vdash (n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle \times(V_1, \dots, V_k) \rangle : b$. The two top-level parallel processes have been typed using instantiations of the rules (Proc Input) and (Proc Output) of the form

$$\frac{E, n_1 : \sigma_1, \dots, n_k : \sigma_k \vdash P : b'}{E \vdash (n_1 : \sigma_1, \dots, n_k : \sigma_k).P : \text{get}(\tau).b'} \text{ and } \frac{E \vdash \times(V_1, \dots, V_k) : \times(\sigma_1^-, \dots, \sigma_k^-)}{E \vdash \langle \times(V_1, \dots, V_k) \rangle : \text{put}(\tau^-)}$$

where $\tau^- = \times(\sigma_1^-, \dots, \sigma_k^-)$ and where $\text{get}(\tau).b' \mid \text{put}(\tau^-) \leq b$ and where for $i \in \{1 \dots k\}$ we have $E \vdash V_i : \sigma_i^-$. By Lemma 4.2 there exists $tr \in \llbracket b' \rrbracket$, and thus $\text{put}(\tau^-) \text{get}(\tau) \diamond tr$ belongs to $\llbracket \text{get}(\tau).b' \mid \text{put}(\tau^-) \rrbracket$ which by Lemma 4.16 is safe. So since $\text{arity}(\tau^-) = \text{arity}(\tau) = k$ this shows that $\tau^- \leq \tau$. Thus for all $i \in \{1 \dots k\}$ we have $\sigma_i^- \leq \sigma_i$ and therefore also $E \vdash V_i : \sigma_i$. By assumption, $(n_1 : \sigma_1, \dots, n_k : \sigma_k).P$ is non-conflicting with $\text{names}(V_i)$ and therefore P is non-conflicting with $\{n_i\} \cup \text{names}(V_i)$. We can thus apply Lemma 5.4 to infer that $E \vdash P[n_i := V_i] : b'$, and by defining $b_0 = \text{put}(\tau^-).\text{get}(\tau)$ we obtain $\text{comm}(\tau) \sim b_0$. We also have $b_0.b' \leq \text{put}(\tau^-) \mid (\text{get}(\tau).b') \leq b$. This yields the claim.

(Red Repl). Our assumptions are that $!P \xrightarrow{\epsilon} P \mid !P$, and that $E \vdash !P : b$ which must have been derived from $\overline{E} \vdash P : b$ with $b \mid b \leq b$. But then we can clearly derive, by (Proc Par) and (Proc Subsumption), $E \vdash P \mid !P : b$, from which the claim follows with $b' = b$ and $b_0 = \epsilon$.

(Red MctxtP). Our assumptions are that $\mathcal{P}[M_1]_e \xrightarrow{\epsilon} \mathcal{P}[M_2]_e$ because $M_1 \longrightarrow M_2$, and that $E \vdash \mathcal{P}[M_1]_e : b$ with b safe. Clearly there exists E_1 and σ such that $E_1 \vdash M_1 : \sigma$. By Lemma 5.6, this implies $E_1 \vdash M_2 : \sigma$. It is easy to see that then also $E \vdash \mathcal{P}[M_2]_e : b$. The claim thus follows, with $b' = b$ and $b_0 = \epsilon$.

(Red PctxtP). Our assumptions are that $\mathcal{P}[P]_p \xrightarrow{\ell} \mathcal{P}[Q]_p$ because $P \xrightarrow{\ell} Q$, and that $E \vdash \mathcal{P}[P]_p : b$ with b safe. By Lemma 5.5 there exists E_1 and safe b_1 such that $E_1 \vdash P : b_1$. We can thus apply the induction hypothesis on $P \xrightarrow{\ell} Q$ to find b_0 with $\ell \sim b_0$ and b'_1 with $b_0.b'_1 \leq b_1$ such that $E_1 \vdash Q : b'_1$. The claim now follows, since by Lemma 5.5 there exists b' with $b_0.b' \leq b$ such that $E \vdash \mathcal{P}[Q]_p : b'$.

(Red Amb). Our assumptions are that $n[P] \xrightarrow{\epsilon} n[Q]$ because $P \xrightarrow{\ell} Q$, and that $E \vdash n[P] : b$ which must have been derived from $E \vdash n : \text{amb}[b_n, b_n^+]$ and $E \vdash P : b_1$ where b_1 is safe and $b_1 \rightsquigarrow b_n$ and $b_n \leq b_n^+$ and $\epsilon \leq b$. We can thus apply the induction hypothesis to find b'_1 and b'_0 with b'_1 safe and $\ell \sim b'_0$ and $b'_0.b'_1 \leq b_1$ such that $E \vdash Q : b'_1$. By Lemmas 4.19 and 4.20 we infer that $b'_1 \rightsquigarrow b_n$, showing that we can apply (Proc Amb) and (Proc Subsumption) to derive $E \vdash n[Q] : b$. This yields the claim, with $b' = b$ and $b_0 = \epsilon$.

(Red \equiv). Our assumptions are that $P' \xrightarrow{\ell} Q'$ because $P' \equiv P$ and $P \xrightarrow{\ell} Q$ and $Q \equiv Q'$, and that $E \vdash P' : b$. By Lemma 5.7 we infer that $E \vdash P : b$, and by applying the induction hypothesis we find b' and b_0 with $\ell \sim b_0$ and $b_0.b' \leq b$ such that $E \vdash Q : b'$. By one more application of Lemma 5.7 this yields the desired judgment $E \vdash Q' : b'$. \square

6 Type Checking

In this section we show that given a complete type derivation for some process P (or an expression M), we can check its validity according to the rules from Fig. 6. For this purpose we need to show: (i) that we can decide the side-conditions for all the rules in Fig. 6, (ii) that we can determine if a certain behavior b (resp. behavior context B) can be obtained by filling in the hole of some behavior context B' with some behavior b' (resp. behavior context B'') and, (iii) that we can check if two behaviors (types) are syntactically identical. Clearly, conditions (ii) and (iii) are decidable. In what follows, we prove that the side conditions for the rules (Proc Subsumption), (Exp Subsumption), (Proc Repl) and (Proc Amb) are also decidable.

For that purpose, we use techniques from the theory of finite non-deterministic automata. We have chosen to disallow ϵ -transitions. Instead some constructs will have features of the well-known “subset construction” inlined, thus gearing our exposition towards an actual implementation. In fact, we believe that this choice makes certain correctness proofs easier (as a string can be uniquely decomposed into a sequence of automaton labels).

Definition 6.1 (Automata). An automaton G of level i is a quadruple $G = (\mathcal{Q}, \iota, F, \delta)$ with \mathcal{Q} a finite set of states, $\iota \in \mathcal{Q}$ the initial state, $F \subseteq \mathcal{Q}$ the set of acceptance states, and δ a transition relation: a finite set of triples of the form (q_1, a, q_2) with $q_1, q_2 \in \mathcal{Q}$ and a an action of level i . \square

We write δ^* for the reflexive and transitive closure of δ . Thus $(q, \bullet, q) \in \delta^*$ for all $q \in \mathcal{Q}$, and $(q, tr_1, q_1) \in \delta^*$ and $(q_1, tr_2, q_2) \in \delta^*$ implies $(q, tr_1 \diamond tr_2, q_2) \in \delta^*$.

Definition 6.2 (Acceptance). The set of strings accepted by G , to be denoted $\text{Acc}(G)$, is given by $\{tr \mid \exists q \in F : (\iota, tr, q) \in \delta^*\}$. \square

The following result is immediate:

Lemma 6.3. Given G , it is decidable whether $\text{Acc}(G) = \emptyset$. \square

Definition 6.4 (Implementation). We say that an automaton G implements a behavior b if $\llbracket b \rrbracket = \text{Acc}(G)$. \square

Lemma 6.5. Given b of level i , we can construct G of level i implementing b . \square

The proof is given in Appendix B.

Lemma 6.6. Let G_1 and G_2 be automata of level i . Further, assume that for all τ and σ of level i it is decidable whether $\tau \leq \sigma$. Then we can construct an automaton $G_1 \setminus G_2$ of level i such that for all traces tr the following property holds: tr belongs to $\text{Acc}(G_1 \setminus G_2)$ if and only if

- tr belongs to $\text{Acc}(G_1)$, and
- no trace tr^+ , such that $tr \leq tr^+$, belongs to $\text{Acc}(G_2)$. \square

With $G_j = (\mathcal{Q}_j, \iota_j, F_j, \delta_j)$ for $j = 1, 2$, we construct $G_1 \setminus G_2 = (\mathcal{Q}, \iota, F, \delta)$ as follows:

$$\begin{aligned} \mathcal{Q} &= \mathcal{Q}_1 \times \mathcal{P}(\mathcal{Q}_2) \\ \iota &= (\iota_1, \{\iota_2\}) \\ F &= \{(q_1, Q_2) \mid q_1 \in F_1 \text{ and } Q_2 \cap F_2 = \emptyset\} \\ \delta &= \{((q_1, Q_2), a, (q'_1, Q'_2)) \mid (q_1, a, q'_1) \in \delta_1 \wedge Q'_2 = \{q'_2 \in \mathcal{Q}_2 \mid \exists q_2 \in Q_2, a^+ \geq a : (q_2, a^+, q'_2) \in \delta_2\}\} \end{aligned}$$

Note that our assumption entails that $a \leq a^+$ is decidable; hence it is in fact possible to construct $G_1 \setminus G_2$. To prove Lemma 6.6 we need a couple of additional lemmas:

Lemma 6.7. Suppose that $((q_1, Q_2), tr, (q'_1, Q'_2)) \in \delta^*$. Then $(q_1, tr, q'_1) \in \delta_1^*$, and if $(q_2, tr^+, q'_2) \in \delta_2^*$ with $q_2 \in Q_2$ and $tr^+ \geq tr$ then $q'_2 \in Q'_2$. \square

Proof. We employ induction on the length of tr . If $tr = \bullet$, then $q'_1 = q_1$ and $Q'_2 = Q_2$, and $tr^+ \geq tr$ implies $tr^+ = \bullet$. Thus the claim is clear.

If tr takes the form $a \diamond tr_0$, there exists q''_1 and Q''_2 such that

$$((q_1, Q_2), a, (q''_1, Q''_2)) \in \delta \text{ and} \tag{1}$$

$$((q''_1, Q''_2), tr_0, (q'_1, Q'_2)) \in \delta^*. \tag{2}$$

By (1) we infer that $(q_1, a, q''_1) \in \delta_1$, and by applying the induction hypothesis to (2) we further infer that $(q''_1, tr_0, q'_1) \in \delta_1^*$, thus implying the desired relation $(q_1, tr, q'_1) \in \delta_1^*$. For the remaining part of the lemma, assume that $(q_2, tr^+, q'_2) \in \delta_2^*$ with $q_2 \in Q_2$ and $tr^+ \geq tr$. We can clearly write tr^+ on the form $a^+ \diamond tr_0^+$ with $a^+ \geq a$ and $tr_0^+ \geq tr_0$, and there thus exists q''_2 such that $(q_2, a^+, q''_2) \in \delta_2$ and $(q''_2, tr_0^+, q'_2) \in \delta_2^*$. From this we infer using (1) that $q''_2 \in Q''_2$, and therefore by applying the induction hypothesis to (2) we infer the desired relation $q'_2 \in Q'_2$. \square

Lemma 6.8. Suppose that $(q_1, tr, q'_1) \in \delta_1^*$, and that $Q'_2 = \{q'_2 \mid \exists tr^+ \geq tr, q_2 \in Q_2 : (q_2, tr^+, q'_2) \in \delta_2^*\}$. Then $((q_1, Q_2), tr, (q'_1, Q'_2)) \in \delta^*$. \square

Proof. We employ induction on the length of tr . If $tr = \bullet$, then $q'_1 = q_1$ and it is easy to see that $Q'_2 = Q_2$, yielding the claim. Now assume that tr takes the form $a \diamond tr_0$. First define

$$Q_2'' = \{q_2'' \mid \exists a^+ \geq a \text{ and } q_2 \in Q_2 : (q_2, a^+, q_2'') \in \delta_2\}$$

and note that

$$Q_2' = \{q_2' \mid \exists tr_0^+ \geq tr_0 \text{ and } q_2'' \in Q_2'' : (q_2'', tr_0^+, q_2') \in \delta_2^*\}. \quad (3)$$

For suppose $q_2' \in Q_2'$. Then there exists $tr^+ \geq tr$ and $q_2 \in Q_2$ such that $(q_2, tr^+, q_2') \in \delta_2^*$. There thus exists q_2'' and $a^+ \geq a$ and $tr_0^+ \geq tr_0$ such that $(q_2, a^+, q_2'') \in \delta_2$, implying $q_2'' \in Q_2''$, and $(q_2'', tr_0^+, q_2') \in \delta_2^*$. Conversely, assume that there exists $tr_0^+ \geq tr_0$ and $q_2'' \in Q_2''$ such that $(q_2'', tr_0^+, q_2') \in \delta_2^*$. Since $q_2'' \in Q_2''$ there exists $a^+ \geq a$ and $q_2 \in Q_2$ such that $(q_2, a^+, q_2'') \in \delta_2$. But this shows that $(q_2, a^+ \diamond tr_0^+, q_2') \in \delta_2^*$ with $a^+ \diamond tr_0^+ \geq tr$, implying that $q_2' \in Q_2'$. We have thus established (3).

Next observe that there exists q_1'' such that $(q_1, a, q_1'') \in \delta_1$ and $(q_1'', tr_0, q_1') \in \delta_1^*$. By the construction of δ , the former relation implies that $((q_1, Q_2), a, (q_1'', Q_2'')) \in \delta$. By the induction hypothesis, the latter relation together with (3) implies that $((q_1'', Q_2''), tr_0, (q_1', Q_2')) \in \delta^*$. Therefore $((q_1, Q_2), tr, (q_1', Q_2')) \in \delta^*$, as desired. \square

We are now ready to prove Lemma 6.6. For “only if”, our assumptions are that $((\iota_1, \{\iota_2\}), tr, (q_1, Q_2)) \in \delta^*$ with $q_1 \in F_1$ and $Q_2 \cap F_2 = \emptyset$. By Lemma 6.7 we infer that $(\iota_1, tr, q_1) \in \delta_1^*$, implying that $tr \in \text{Acc}(G_1)$, and that for $tr^+ \geq tr$ it holds that whenever $(\iota_2, tr^+, q_2) \in \delta_2^*$ then $q_2 \in Q_2$ and thus $q_2 \notin F_2$, implying that $tr^+ \notin \text{Acc}(G_2)$.

For “if”, our assumptions imply that there exists $q_1 \in F_1$ such that $(\iota_1, tr, q_1) \in \delta_1^*$, and that with $Q_2 = \{q_2 \mid \exists tr^+ \geq tr : (\iota_2, tr^+, q_2) \in \delta_2^*\}$ we have $Q_2 \cap F_2 = \emptyset$. By Lemma 6.8, we infer that $((\iota_1, \{\iota_2\}), tr, (q_1, Q_2)) \in \delta^*$ which amounts to $tr \in \text{Acc}(G_1 \setminus G_2)$ since $(q_1, Q_2) \in F$. This concludes the proof of Lemma 6.6.

Lemma 6.9. *Assume that for all τ, σ of level i it is decidable whether $\tau \leq \sigma$. Let b_1, b_2 be of level i . Then it is decidable whether $b_1 \leq b_2$.* \square

Proof. By Lemma 6.5, we can construct G_1 and G_2 of level i implementing b_1 and b_2 . By Lemma 6.6, we can then construct $G_1 \setminus G_2$ such that for all traces tr the following property holds: tr belongs to $\text{Acc}(G_1 \setminus G_2)$ if and only if (i) tr belongs to $\text{Acc}(G_1)$, and (ii) no trace tr^+ with $tr \leq tr^+$ belongs to $\text{Acc}(G_2)$.

We shall prove that deciding $b_1 \leq b_2$ amounts to deciding whether $\text{Acc}(G_1 \setminus G_2) = \emptyset$, a property which is decidable (Lemma 6.3).

So first assume that $b_1 \leq b_2$ is false. Thus there exists $tr \in \llbracket b_1 \rrbracket = \text{Acc}(G_1)$ such that for no tr^+ with $tr \leq tr^+$ it holds that $tr^+ \in \llbracket b_2 \rrbracket = \text{Acc}(G_2)$. But this shows that $tr \in \text{Acc}(G_1 \setminus G_2)$. Hence, $\text{Acc}(G_1 \setminus G_2) \neq \emptyset$.

Conversely, assume that $\text{Acc}(G_1 \setminus G_2) \neq \emptyset$. Let $tr \in \text{Acc}(G_1 \setminus G_2)$. Then $tr \in \text{Acc}(G_1) = \llbracket b_1 \rrbracket$, and for no tr^+ with $tr^+ \geq tr$ it holds that $tr^+ \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$. Hence, $b_1 \leq b_2$ is false. \square

Lemma 6.10. *Let τ, σ be of level i , and assume that for all b_1, b_2 of level j with $j < i$ it is decidable whether $b_1 \leq b_2$. Then it is decidable whether $\tau \leq \sigma$.* \square

Proof. An easy induction on the combined size of τ and σ . Below we list a few of the most interesting cases:

$\tau = \tau_1 \rightarrow \tau_2$ and $\sigma = \sigma_1 \rightarrow \sigma_2$. Here $\tau \leq \sigma$ holds iff $\sigma_1 \leq \tau_1$ and $\tau_2 \leq \sigma_2$. But since τ_1, τ_2, σ_1 , and σ_2 are all of level i , the induction hypothesis tells us that this is decidable.

$\tau = \text{amb}[b_1, b_1']$ and $\sigma = \text{amb}[b_2, b_2']$. Here $\tau \leq \sigma$ holds iff $b_2 \leq b_1$ and $b_1' \leq b_2'$. But since b_1, b_1', b_2 , and b_2' are all of level $i-1$, the assumptions of the lemma tell us that this is decidable.

$\tau = \text{cap}[B_1]$ and $\sigma = \text{cap}[B_2]$. Here $\tau \leq \sigma$ holds iff $B_1 \leq B_2$, and by Lemma 4.13 we can thus construct a level zero behavior test such that deciding $\tau \leq \sigma$ amounts to deciding $B_1[\text{test.test}] \leq B_2[\text{test.test}]$. But since B_1 and B_2 are of level $i-1$, and hence also the behaviors $B_1[\text{test.test}]$ and $B_2[\text{test.test}]$ are of level $i-1$, this is decidable by the assumptions of the lemma. \square

The following Theorem is an immediate consequence (proved by induction) of Lemmas 6.9 and 6.10.

Theorem 6.11. *Given b_1 and b_2 , it is decidable whether $b_1 \leq b_2$. Given τ and σ , it is decidable whether $\tau \leq \sigma$.* \square

Corollary 6.12. *Given a behavior b , it is decidable whether $b \mid b \leq b$.* \square

We are still left with deciding whether a given b is safe and whether $b \rightsquigarrow b_0$ holds for given b and b_0 . For that purpose, we introduce a couple of auxiliary concepts.

Definition 6.13. Let \mathcal{L} be a language, that is a member of $\mathcal{P}(\text{Trace})$, and let $G = (\mathcal{Q}, \iota, F, \delta)$ be an automaton. We then define

$$G\#\mathcal{L} = \{q \in \mathcal{Q} \mid \exists tr \in \mathcal{L} : (\iota, tr, q) \in \delta^*\}.$$

We say that \mathcal{L} is *testable* if for every automaton G there exists a procedure for computing $G\#\mathcal{L}$. \square

Lemma 6.14. *Let \mathcal{L} be testable, and let G be an automaton. Then we can construct an automaton $G - \mathcal{L}$ such that for all traces tr the following property holds: tr belongs to $\text{Acc}(G - \mathcal{L})$ if and only if there exists $tr_0 \in \mathcal{L}$ such that $tr_0 \diamond tr \in \text{Acc}(G)$.* \square

Proof. With $G = (\mathcal{Q}, \iota, F, \delta)$ and with ι_0 a symbol not in \mathcal{Q} , we construct $G - \mathcal{L} = (\mathcal{Q}_0, \iota_0, F_0, \delta_0)$ as follows:

$$\begin{aligned} \mathcal{Q}_0 &= \mathcal{Q} \cup \{\iota_0\} \\ F_0 &= F \cup (\text{if } (G\#\mathcal{L}) \cap F = \emptyset \text{ then } \emptyset \text{ else } \{\iota_0\}) \\ \delta_0 &= \delta \cup \{(\iota_0, a, q) \mid \exists q_0 \in G\#\mathcal{L} \text{ with } (q_0, a, q) \in \delta\}. \end{aligned}$$

Concerning the claim of the lemma, we first address the “only if” part. The assumption is that $tr \in \text{Acc}(G - \mathcal{L})$, that is there exists $q_0 \in F_0$ such that $(\iota_0, tr, q_0) \in \delta_0^*$. We must consider two cases:

- If $tr = \bullet$ then $\iota_0 \in F_0$, so by the construction of F_0 there exists q_1 in $(G\#\mathcal{L}) \cap F$. There thus exists $tr_0 \in \mathcal{L}$ with $(\iota, tr_0, q_1) \in \delta^*$ where $q_1 \in F$, implying that $tr_0 \diamond tr = tr_0 \in \text{Acc}(G)$.
- If tr takes the form $a \diamond tr_1$, we infer that there exists $q \in \mathcal{Q}_0$ such that $(\iota_0, a, q) \in \delta_0$ and such that $(q, tr_1, q_0) \in \delta_0^*$. The former relation implies the existence of $q_1 \in G\#\mathcal{L}$ with $(q_1, a, q) \in \delta$, and thus the existence of $tr_0 \in \mathcal{L}$ with $(\iota, tr_0, q_1) \in \delta^*$; the latter relation implies (since clearly $q \neq \iota_0$) that $(q, tr_1, q_0) \in \delta^*$ and $q_0 \neq \iota_0$ and thus (as $q_0 \in F_0$) also $q_0 \in F$. Since combining the abovementioned properties of δ^* yields $(\iota, tr_0 \diamond a \diamond tr_1, q_0) \in \delta^*$, this demonstrates that $tr_0 \diamond tr \in \text{Acc}(G)$, as desired.

Next we address the “if” part, where our assumptions are that there exists $tr_0 \in \mathcal{L}$ such that $tr_0 \diamond tr \in \text{Acc}(G)$, that is there exists $q \in F$ such that $(\iota, tr_0 \diamond tr, q) \in \delta^*$. We must consider two cases:

- If $tr = \bullet$ then $q \in G\#\mathcal{L}$, which by the construction of F_0 implies that $\iota_0 \in F_0$ and therefore $\bullet \in \text{Acc}(G - \mathcal{L})$, as desired.
- If tr takes the form $a \diamond tr_1$, there exists q_1 and q_2 such that $(\iota, tr_0, q_1) \in \delta^*$, implying $q_1 \in G\#\mathcal{L}$, such that $(q_1, a, q_2) \in \delta$, thus implying $(\iota_0, a, q_2) \in \delta_0$, and such that $(q_2, tr_1, q) \in \delta^*$, implying that also $(q_2, tr_1, q) \in \delta_0^*$. Therefore $(\iota_0, tr, q) \in \delta_0^*$, and since $q \in F_0$ this demonstrates the desired relation $tr \in \text{Acc}(G - \mathcal{L})$. \square

Lemma 6.15. *Given a behavior b , it is decidable whether b is safe.* \square

Proof. The language $\mathcal{L}_0 = \text{Comm}^*$ is clearly testable, and using Theorem 6.11 we infer that also the language $\mathcal{L}_1 = \text{Comm} \setminus \text{WtComm}$ is testable.

By Lemma 6.5 there exists an automaton G implementing b . Using Lemma 6.14 twice we then construct the automaton $G' = (G - \mathcal{L}_0) - \mathcal{L}_1$. We shall prove that b is safe if and only if $\text{Acc}(G') = \emptyset$, a property which is decidable (Lemma 6.3).

First assume that b is not safe. Then there exists $tr_0 \in \mathcal{L}_0$ and $tr_1 \in \mathcal{L}_1$ and tr_2 such that $tr_0 \diamond tr_1 \diamond tr_2 \in \llbracket b \rrbracket = \text{Acc}(G)$. By Lemma 6.14 we infer first that $tr_1 \diamond tr_2 \in \text{Acc}(G - \mathcal{L}_0)$ and then that $tr_2 \in \text{Acc}(G')$. Hence, $\text{Acc}(G') \neq \emptyset$.

Next assume that $\text{Acc}(G') \neq \emptyset$. Let $tr \in \text{Acc}(G')$. By Lemma 6.14 we infer first that there exists $tr_1 \in \mathcal{L}_1$ such that $tr_1 \diamond tr \in \text{Acc}(G - \mathcal{L}_0)$, and next that there exists $tr_0 \in \mathcal{L}_0$ such that $tr_0 \diamond tr_1 \diamond tr \in \text{Acc}(G) = \llbracket b \rrbracket$. Hence, b is not safe. \square

Lemma 6.16. *Given b_1 and b_2 , it is decidable whether $b_1 \rightsquigarrow b_2$.* \square

Proof. By Lemma 6.5 there exists automata G_1 and G_2 implementing b_1 and b_2 . The language $\mathcal{L} = \text{Comm}^* \diamond \text{diss}$ is clearly testable, so using Lemma 6.14 we can construct an automaton $G_0 = G_1 - \mathcal{L}$. By Lemma 6.6 (employing Theorem 6.11) we can construct an automaton $G = G_0 \setminus G_2$. We shall prove that $b_1 \rightsquigarrow b_2$ holds if and only if $\text{Acc}(G) = \emptyset$, a property which is decidable (Lemma 6.3).

For “if”, we must establish $b_1 \rightsquigarrow b_2$ and therefore consider the situation where there exists $tr_1 \in \text{Comm}^*$ and tr such that $tr_1 \diamond \text{diss} \diamond tr \in \llbracket b_1 \rrbracket = \text{Acc}(G_1)$. Since $tr_1 \diamond \text{diss} \in \mathcal{L}$, we infer by Lemma 6.14 that $tr \in \text{Acc}(G_0)$. Since by assumption $tr \notin \text{Acc}(G)$, Lemma 6.6 tells us that there exists tr^+ with $tr \leq tr^+$ such that $tr^+ \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$, which is just as desired.

For “only if”, we assume in order to arrive at a contradiction that there exists $tr \in \text{Acc}(G)$. By Lemma 6.6 we infer that $tr \in \text{Acc}(G_0)$ and that for all tr^+ with $tr^+ \geq tr$ it does not hold that $tr^+ \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$. By Lemma 6.14 we infer that there exists $tr_1 \in \text{Comm}^* \diamond \text{diss}$ such that $tr_1 \diamond tr \in \text{Acc}(G_1) = \llbracket b_1 \rrbracket$. But since $b_1 \rightsquigarrow b_2$, this yields the desired contradiction. \square

We are now ready to state the main result of this section. Namely, that given a complete derivation for an expression M or a process P , we can verify that the derivation is valid according to the rules from Fig. 6.

Theorem 6.17 (Decidability of type checking). *Given a purported derivation of $E \vdash M : \tau$ or $E \vdash P : b$, we can check its validity.* \square

It is clear, however, that the worst-case complexity of type checking is at least exponential.

7 Comparison with Other Systems

7.1 Type Systems for the Ambient Calculus

We start by establishing that our type system is a conservative extension of the type system for **AC** presented in [CG99, Sect. 3]; for that purpose we employ a function *Plus* translating entities in the latter system into entities in the former: expressions M^C into expressions, processes P^C into processes, “message types” W^C into types, “exchange types” T^C into behaviors, and environments E^C into environments. *Plus* is defined recursively on the structure of its argument; most clauses are straightforward homomorphisms except for

$$\begin{aligned} \text{Plus}(M^C [P^C]) &= \text{Plus}(M^C) [\text{Plus}(P^C) \mid \text{coopen } \text{Plus}(M^C).0] \\ \text{Plus}(\text{amb}[T^C]) &= \text{amb}[\text{Plus}(T^C), \text{Plus}(T^C)] \\ \text{Plus}(\text{cap}[T^C]) &= \text{cap}[\text{Plus}(T^C) \mid \square] \\ \text{Plus}(\text{Shh}) &= \varepsilon \\ \text{Plus}(W_1^C \times \dots \times W_n^C) &= \text{fromnow } \{\times(\text{Plus}(W_1^C), \dots, \text{Plus}(W_n^C))\} \end{aligned}$$

We then have the following result, proved in Appendix B:

Theorem 7.1. *Suppose that $E^C \vdash P^C : T^C$, respectively $E^C \vdash M^C : W^C$, is derivable in the system of [CG99, Sect. 3]. Then $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$, respectively $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{Plus}(W^C)$, is derivable in our system.* \square

It is also easy to show that if $P_1^C \longrightarrow P_2^C$ holds in the system of [CG99], after replacing the rule $!P \equiv P \mid !P$ by the rule $!P \longrightarrow P \mid !P$, then $\text{Plus}(P_1^C) \longrightarrow \text{Plus}(P_2^C)$ holds in our system. A proof of this claim is given as Theorem B.2 in Appendix B.

7.1.1 Possible Extensions of the Polymorphically-Typed AC+.

It is relatively straightforward to extend our system to record ambient movements: we augment Act with actions enter and exit, and augment Beh with behaviors that are suitable abstractions of sets of traces containing these actions. (In fact, the type system of [Zim00] can be viewed as such an abstraction where, e.g., $[\forall O \rightsquigarrow I]$ is the set of traces containing actions $\text{put}(\tau)$ with τ described by O , actions $\text{get}(\tau)$ with τ described by I , but no enter or exit actions.) As in [CGG99] we can then express that an ambient is immobile. Thanks to diss and the relation $b \rightsquigarrow b_0$, we are able to declare ambients immobile even though they open packets that have moved, thus overcoming (as also [LS00] does) the problem faced in [CGG99]. Another application might be to predict the shape of ambients, as done in [NN00] using tree grammars.

We might also consider introducing a special “void” behavior $*$ such that $[\ast] = \emptyset$. Then for b not containing diss we would have $b \rightsquigarrow *$, enabling us to assign the type $\text{amb}[\ast, \ast]$ to an ambient which does not allow itself to be opened. For subject reduction to carry through, however, we must ensure (by suitable side conditions) that Lemma 4.2 still holds (that is, if $E \vdash P : b$ then $[\![b]\!] \neq \emptyset$). To see why, consider the process $P = (x : \text{int}).(\text{open } n.\langle x + 7 \rangle) \mid \langle \text{true} \rangle$ where n has type $\text{amb}[\ast, \ast]$. If we do not put any restrictions on our type system, we can derive $E \vdash P : *$ (since, e.g., $\ast \equiv b.\ast$ for all b), even though with $\ell = \text{comm}(\text{int})$ it holds that $P \xrightarrow{\ell} Q$ where $Q = \text{open } n.\langle \text{true} + 7 \rangle$ which clearly cannot be typed. In order to type a process that attempts to open a locked ambient, we must thus assign it a non-void behavior such as, e.g., ε .

Besides the tasks mentioned in Sect. 1 (in particular type inference), future work includes investigating the relationship to the system proposed by Levi & Sangiorgi [LS00] which—using the notion of single-threadedness—made a first attempt to rule out so-called “grave” interferences (a notion that is not precisely defined in [LS00]). For that purpose we must extend our poly-typed AC+ with coin and coout expressions, recorded also in the traces.

We then expect that a process is free from grave interferences if it can be assigned a behavior b such that $[\![b]\!]$ contains not more than one “well-formed” trace. For a suitable definition of “well-formed”, this does not hold for the process $P = \langle 7 \rangle \mid (x).\text{in } n \mid (x).\text{out } m$, which exhibits what, in our view, should be considered a “grave” interference in that quite different actions are taken depending on which inputting process receives the output. By contrast, since only one subprocess carries the “thread” at any time, the system of Levi & Sangiorgi will assign P a single-threaded type, and accordingly they consider this kind of interference to be “plain”.

7.2 Type and Effect Systems

Our initial development was partly inspired by type and effect systems, in particular those developed by the first author, together with H. R. Nielson and F. Nielson, for Concurrent ML [PR97] and reported in, e.g., [ANN99, ANN98, NN94]. We use NNA when referring to the common features of this body of work.

In NNA, as in the type system for AC+, there are “atomic” behaviors recording input and output operations, and also constructs for sequential as well as parallel composition (which in NNA is expressed using the SPAWN construct) of two behaviors. In NNA there is explicit recursion, and a choice operator $b_1 + b_2$ to express approximation, whereas in our system the construct $\text{fromnow } T$ covers both recursion and approximation — note that the behavior $\text{fromnow } \{\tau_1, \dots, \tau_n\}$ can be expressed using recursion as $\text{rec } \beta.((\text{get}(\tau_1) + \text{put}(\tau_1) + \dots + \text{get}(\tau_n) + \text{put}(\tau_n)).\beta + \varepsilon)$.

However, the conceptual differences between Concurrent ML and AC+ show up in several places. In NNA there are types $\tau \text{ event } b$ and $\tau \text{ chan}$, and an atomic behavior $\tau \text{ CHAN}$ recording the creation of a channel carrying values of type τ , whereas there are no counterparts to our constructs $\text{amb}[b, b']$, $\text{cap}[B]$, or diss .

The subject reduction property is stated in similar terms in the two systems. In both cases the basic content is that if P rewrites to Q by some action described by b_0 , and P can be assigned behavior b , then Q can be assigned behavior b' with $b_0.b' \leq b$. Note that, in contrast to AC+ the system for NNA is one-tiered, in that an expression is assigned both a type and a behavior.

When it comes to the ordering on behaviors, NNA pursues an axiomatic approach (though a notion of traces is briefly mentioned in [ANN99, Sect. 2.7.1]), whereas we have taken a semantic approach.

A key distinguishing feature of type-and-effect systems is that function types are annotated with “latent” behaviors, introduced by (Exp Abs) and eliminated by (Exp App). This feature is absent from our system. One may still argue that the behavior b inside $\text{amb}[b, b]$ can be considered “latent”, in which case the rule (Proc Amb) for $n[P]$ is viewed as an introduction rule and the rule for $\text{open } n.P$, derived from (Exp Open) and (Proc Action), is viewed as an elimination rule. However, this correspondence seems rather far-fetched, as the entire development of our system for **AC+** shares the conceptual framework of [CG99] (and the systems spawned by this paper) rather than that of type-and-effect systems.

7.3 Type Systems for the Pi-Calculus

There are many points of contact between the π -calculus and **AC**. Other researchers have already related the two calculi and discussed their respective merits. In the Introduction, we briefly discussed parametric polymorphism and subtyping, which both allow a richer type discipline for both the π -calculus and **AC+**: A considerably larger class of potentially useful processes can be typed and, as a result, executed safely.

In this subsection we give a brief presentation of *session types* in the π -calculus, because they were motivated by considerations similar to ours in relation to *orderly communication* in **AC+**. Before we do this, we need to briefly recall the system of simple types for the π -calculus; along the way we mention several extensions of simple types that have been proposed in the literature. Our presentation is a very brief adaptation of parts in [Gay99] and [GH99].

7.3.1 Simple Types and Extensions.

The syntax of processes P in the π -calculus can be given by the following grammar:

$$P ::= \mathbf{0} \mid P \mid Q \mid x?[y].P \mid x![y].P \mid (\nu \tilde{x})P \mid !P$$

where x and y range over names, and \tilde{x} and \tilde{y} range over lists of names. This is the syntax of the plain π -calculus, the simplest form that captures the idea of communication between concurrent processes using channels. There are many variations and extensions of the plain π -calculus which we here ignore.

Several type systems for the π -calculus have been proposed, starting with a system of *simple types*. Using channels to only carry messages each consisting of a tuple of channel names, the purest form of simple types can be defined by the grammar:

$$T ::= \hat{\ }[T_1, \dots, T_n]$$

where $n \geq 0$. In this syntax there are no ground types, with the empty tuple $\hat{\ }[\]$ playing the role of a single ground type. Several systems of simple types for the π -calculus in the literature add ground types $\text{bool}, \text{int}, \dots$, as well as types built from them using appropriate constructors and selectors. The typing rules for the system of simple types is presented in Fig. 7.

Recursive types are easily added to the simple types above, by introducing type variables and a recursion constructor μ into their syntax. Calling them *sorts* instead of *types*, Milner presented the essence of such a system in [Mil91]. Milner’s presentation did not use an environment-based system of typing rules; instead of an explicit environment that uniquely assigns types to a finite set of names, he used a function from a set of sorts to tuples of sorts, also implicitly supporting recursive sorts (types).

Pierce and Sangiorgi were probably the first to propose an environment-based system of typing rules, which explicitly supported recursive types as well as *input/output subtyping*, in [PS96]. The latter is accomplished by making every channel type specify the direction in which messages may flow, with 3 possibilities: a channel is used for input only (designated by a type of the form $?[\tilde{T}]$), or for output only (type $![\tilde{T}]$), or for both input and output (type $\hat{\ }[\tilde{T}]$). A notion of subtyping is now easily added, whereby a channel assigned an input-and-output type can be used wherever a channel of input-only or output-only type is required.

Further extensions of the system of simple types for the π -calculus include *linear types* in [KPT96], *types for partial deadlock-freedom* in [Kob98], and, as already noted, *parametric polymorphism* in [Tur95].

$\text{(Nil)} \quad \frac{}{\Gamma \vdash \mathbf{0}}$	$\text{(Par)} \quad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P Q}$
$\text{(Res)} \quad \frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\nu x : T)P}$	$\text{(Repl)} \quad \frac{\Gamma \vdash P}{\Gamma \vdash !P}$
$\text{(Out)} \quad \frac{\Gamma \vdash P \quad \Gamma \vdash x : \hat{[T]} \quad \Gamma \vdash \tilde{y} : \tilde{T}}{\Gamma \vdash x![\tilde{y}].P}$	$\text{(In)} \quad \frac{\Gamma, \tilde{y} : \tilde{T} \vdash P \quad \Gamma \vdash x : \hat{[T]}}{\Gamma \vdash x?[\tilde{y} : \tilde{T}].P}$

Figure 7. Simple types for the π -calculus.

7.3.2 Session Types in the Pi-Calculus.

Honda and his co-workers proposed a variant of the π -calculus where some channels are designated to be *session channels*, in [THK94] and [HVK98]. Such a channel is allowed to carry a sequence of different message types over time, by contrast to a channel that is restricted to a single type of message throughout its lifetime, as in the simply-typed π -calculus. More recently, Gay and Hole in [GH99] have developed a type system for the π -calculus which combines *session types*, subtyping and recursive types. Whereas session channels form a distinct syntactic category in [THK94] and [HVK98], Gay and Hole enforce this distinction, perhaps more elegantly, by means of their type system.

Consider a process P , which receives an integer along a channel x before it sends a boolean along the same x . In such a situation, x can be assigned the session type $?[\text{int}].![\text{bool}].\text{end}$ and the corresponding typing derivation for P will produce a judgement of the form

$$x : ?[\text{int}].![\text{bool}].\text{end} \vdash P \quad \text{where } P = x?[a : \text{int}].x![\text{prime}(a)].P'$$

for some continuation process P' . A process Q that communicates with P uses channel x with a complementary session-type. For example, a typing derivation for Q should produce a judgement of the form

$$x : ![\text{int}].?[\text{bool}].\text{end} \vdash Q \quad \text{where } Q = x![1 + 2^{2048}].x?[b : \text{bool}].Q'$$

for some continuation Q' . Note how the session types of x are dual of each other in the two judgements for P and Q . The typing rules allow P and Q to be put in parallel, to obtain the judgement

$$x : ![\text{int}].?[\text{bool}].\text{end}^2 \vdash P | Q$$

where the superscript 2 on the delimiter end indicates that the type of x is obtained by the parallel composition of two complementary session types.

So far, session types can be introduced with no change in the syntax of the underlying π -calculus. More interestingly, session types allow a form of branching, which requires an extension of the underlying calculus. Suppose P is a server which offers a choice between $\text{prime}() : \text{int} \rightarrow \text{bool}$ and $\text{relative-prime}() : \text{int} \rightarrow \text{bool}$. An appropriate session-type for x is now

$$x : \&\langle \text{test1} : ?[\text{int}].![\text{bool}].\text{end}, \text{test2} : ?[\text{int}].?[\text{int}].![\text{bool}].\text{end} \rangle$$

The complementary session-type for x , from the side of a process Q communicating with P , is

$$x : \oplus\langle \text{test1} : ![\text{int}].?[\text{bool}].\text{end}, \text{test2} : ![\text{int}].![\text{int}].?[\text{bool}].\text{end} \rangle$$

The type constructor $\&$ specifies that P offers a choice between test1 and test2 , and \oplus specifies that Q selects between test1 and test2 . The syntax of the underlying π -calculus is extended so that now P is written as

$$P = x \triangleright \{ \text{test1} : x?[a : \text{int}].x![\text{prime}(a)].P', \\ \text{test2} : x?[a : \text{int}].x?[b : \text{int}].x![\text{relative-prime}(a, b)].P' \}$$

while Q may be

$$Q = x \triangleleft test1 : x![1 + 2^{2048}].x?[b : bool].Q'$$

indicating that Q has selected option $test1$. P and Q can now be safely put in parallel.

The system of session types as just described can be augmented to include subtyping and recursive types. A full account is given in [GH99].

Despite the common underlying motivation and the many similarities, there are also many differences between sessions types in the π -calculus and orderly communication in **AC+**. Technical issues regarding the former do not apply to the latter and vice-versa; this is best illustrated by some of the problems we have solved in relation to orderly communication which have no counterpart (or have not been raised) in relation to session types. However, a final assessment of their respective merits awaits a more systematic comparison, probably to be based on a translation from the π -calculus to **AC+**, or vice-versa, which is also type preserving. By “type preserving” we mean that if P is a process of the π -calculus and Q is its translation into **AC+**, then P is typable in the system with session types if and only if Q is typable in our type system for **AC+** with orderly communication. It is a question whether such a type-preserving translation, in either direction, is possible at all; we have not tried to carry it out.

References

- [ANN98] T. Amtoft, H. R. Nielson, and F. Nielson. Behaviour analysis for validating communication patterns. *Springer International Journal on Software Tools for Technology Transfer*, 2(1):13–28, 1998. 25
- [ANN99] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999. 25, 25
- [BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *CONCUR 2000*, vol. 1877 of *LNCS*, pp. 504–520, 2000. 2, 3
- [Car99] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, eds., *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *LNCS*, pp. 51–94. Springer-Verlag, 1999. 6, 6
- [CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, ed., *FoSSaCS’98*, vol. 1378 of *LNCS*, pp. 140–155. Springer-Verlag, 1998. 1, 5
- [CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL’99, San Antonio, Texas*, pp. 79–92. ACM Press, Jan. 1999. 1, 2, 4, 5, 6, 7, 8, 24, 24, 24, 26, 32, 34, 34, 36, 36
- [CGG99] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, eds., *ICALP’99*, vol. 1644 of *LNCS*, pp. 230–239. Springer-Verlag, July 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999. 2, 25, 25
- [CGG00] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, Tohoku University, Sendai, Japan, vol. 1872 of *LNCS*, pp. 333–347. Springer-Verlag, Aug. 2000. 2
- [FGL⁺96] C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *CONCUR 1996*, vol. 1119 of *LNCS*, pp. 406–421. Springer-Verlag, 1996. 1
- [Gay93] S. J. Gay. A sort inference algorithm for the polyadic pi-calculus. In *POPL 1993*, pp. 429–438, 1993. 3

- [Gay99] S. J. Gay. Some type systems for the pi calculus. Technical report, Dept of Comp. Sci., Royal Holloway, University of London, 1999. 26
- [GH99] S. Gay and M. Hole. Types and subtypes for client-server interactions. In *Proc. European Symp. on Programming*, vol. 1576 of *LNCS*, pp. 74–90. Springer-Verlag, 1999. 3, 26, 27, 28
- [HVK98] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, vol. 1381 of *LNCS*, pp. 122–138. Springer-Verlag, 1998. 27, 27
- [Kob98] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. on Prog. Langs. & Sys.*, 20(2):436–482, 1998. A preliminary summary appeared at LICS'1997. 26
- [KPT96] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. In *POPL'96, St. Petersburg Beach, Florida*, pp. 358–371. ACM Press, Jan. 1996. 26
- [LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*, pp. 352–364. ACM Press, Jan. 2000. 5, 25, 25, 25
- [Mil91] R. Milner. The polyadic π -calculus: A tutorial. Technical Report 91-180, University of Edinburgh – Lab for Foundations of Computer Science, 1991. 26
- [Mog00] E. Moggi. Arity polymorphism and dependent types. In *Subtyping & Dependent Types in Programming, Ponte de Lima, Portugal, 2000*. Proceedings online at <http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html>. 2
- [NN94] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 84–97, 1994. 25
- [NN00] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL'00, Boston, Massachusetts*, pp. 142–154. ACM Press, 2000. 25
- [PR97] P. Panangaden and J. H. Reppy. The essence of concurrent ML. In F. Nielson, ed., *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science. Springer-Verlag, 1997. 25
- [PS96] B. C. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. A revised and extended version of a paper appearing at LICS'93. 3, 26
- [PS00] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, May 2000. 3
- [PT97] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report, IU, 1997. 3
- [RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *POPL 1998*, pp. 378–390. ACM Press, 1998. 1
- [RH99] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL'99, San Antonio, Texas*, pp. 93–104. ACM Press, 1999. 1
- [SV99] P. Sewell and J. Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop (CSFW-12), Mordano, Italy, June 1999*. 1
- [THK94] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, vol. 817 of *LNCS*, pp. 398–413. Springer-Verlag, 1994. 27, 27

- [Tul00] M. Tullsen. The zip calculus. In *Fifth International Conference on Mathematics of Program Construction (MPC 2000)*. Springer-Verlag, July 2000. 2
- [Tur95] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. Ph.D. thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345. 3, 4, 26
- [VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of LNCS. Springer-Verlag, 1999. 1
- [VH93] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic pi-calculus. In *CONCUR 1993*, vol. 715 of LNCS, pp. 524–538. Springer-Verlag, 1993. 3
- [YH99] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order mobile processes. In *CONCUR 1999*, vol. 1664 of LNCS, pp. 557–573. Springer-Verlag, 1999. 3
- [YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In *LICS 2000*, pp. 334–345, 2000. 3
- [Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *FOSSACS 2000, Berlin*, vol. 1784 of LNCS, pp. 375–390. Springer-Verlag, 2000. 2, 3, 10, 25

A Typing of Examples

In this Appendix, we explain how to type the examples presented in the first two sections.

Case 2 By assigning x and y the type real , p the type $\text{amb}[\text{put}(\text{int}, \text{int})]$, and q the type $\text{amb}[\text{put}(\text{real}, \text{real})]$, we can construct a type derivation for

$$p\{ \text{in } r.\langle 3, 2 \rangle \} \mid q\{ \text{in } r.\langle 3.6, 5.1 \rangle \} \mid r[(x, y).n[\langle \text{mult}(x, y) \rangle \mid P] \mid \text{open } p \mid \text{open } q]$$

where the body of ambient r has behavior $\text{get}(\text{real}, \text{real}).\varepsilon \mid \text{put}(\text{int}, \text{int}) \mid \text{put}(\text{real}, \text{real})$ which is clearly safe.

Case 3 With b and b' the behaviors of P and Q , we can construct a type derivation for

$$n[\langle \text{true}, 5 \rangle \mid \langle 5, 6, 3.6 \rangle \mid (x, y).P \mid (x, y, z).Q]$$

where the behavior $\text{put}(\text{bool}, \text{int}) \mid \text{put}(\text{int}, \text{int}, \text{real}) \mid \text{get}(\text{bool}, \text{int}).b \mid \text{get}(\text{int}, \text{int}, \text{real}).b'$ (which is safe under suitable assumptions on b and b') has been assigned to the body of ambient n .

Case 4 By assigning n the type $\text{amb}[\text{get}(\text{bool}).b]$ (with b the behavior of P), which is possible since the body of ambient n has behavior $b_n = \text{get}(\text{bool}).b \mid \text{diss}$ where b_n is safe and $b_n \rightsquigarrow \text{get}(\text{bool}).b$, we can construct a type derivation for

$$m[\langle 7 \rangle \mid (x).\text{open } n.\langle x = 42 \rangle \mid n\{ (y).P \}]$$

where the body of ambient m has behavior $b' = \text{put}(\text{int}) \mid \text{get}(\text{int}).(\text{get}(\text{bool}).b \mid \text{put}(\text{bool})) \mid \varepsilon$. Under suitable assumptions on b , this behavior is safe, as shown in Example 4.15 for the simple case $b = \varepsilon$.

Example 2.1 Let $b = \text{get}(\text{string}).(\text{get}(\text{cap}[\square]).\varepsilon \mid \text{put}(\text{cap}[\square]))$. By assigning the behavior $\text{get}(\text{cap}[\square]).\varepsilon \mid \text{diss}$ to the body of hop (which can then be given the type $\text{amb}[\text{get}(\text{cap}[\square]).\varepsilon]$), by assigning the (safe) behavior $b \mid \text{diss}$ to the body of route (which can then be given the type $\text{amb}[b]$), and by assigning $b \mid \text{put}(\text{string})$ (which is clearly safe) to the body of packet , we can construct a type derivation for

$$\text{router}[\text{!route}\{\text{in } \text{packet}.\langle \text{dst} \rangle.\text{open } \text{hop}.\langle \text{lookup-route}(\text{dst}) \rangle\}] \mid \\ \text{packet}[\text{in } \text{router}.\text{open } \text{route}.\langle \text{“bu”} \rangle \mid \text{hop}\{(x).x\}]$$

Example 2.2 By assigning z_1 the type $\text{cap}[\text{diss}[\square]]$, z_2 and x_1 the type $\text{cap}[\square]$, $tstres$ the type $\text{amb}[\text{put}(\text{bool})]$, and by assigning the behavior $\text{diss.put}(\text{cap}[\square], \text{int})$ to the body of p (which can then be given type $\text{amb}[\text{put}(\text{cap}[\square], \text{int})]$), we can construct a type derivation for

$$\begin{aligned} \text{server} &\triangleq s[!tst[\text{open } p \mid \\ &\quad (x_1, x_2). \text{tstres}\{ x_1.\langle \text{prime}(x_2) \rangle \} \mid \\ &\quad (x_1, x_2, x_3). \text{tstres}\{ x_1.\langle \text{relative-prime}(x_2, x_3) \rangle \}]] \\ \text{client} &\triangleq c[\langle \text{out } c.\text{in } s.\text{in } \text{tst.coopen } p \rangle \mid \\ &\quad (z_1).(\langle \text{out } \text{tst.out } s.\text{in } c \rangle \mid \\ &\quad (z_2).(\text{open } \text{tstres}.(v).Q \mid p[z_1.\langle z_2, 1 + 2^{4096} \rangle]))] \end{aligned}$$

since the body of tst has the safe behavior $\text{put}(\text{cap}[\square], \text{int}) \mid \text{get}(\text{cap}[\square], \text{int}) \mid \text{get}(\text{cap}[\square], \text{int}, \text{int})$ and the body of c has behavior (with b the behavior of Q)

$$\text{put}(\text{cap}[\text{diss}[\square]]) \mid \text{get}(\text{cap}[\text{diss}[\square]]).(\text{put}(\text{cap}[\square]) \mid \text{get}(\text{cap}[\square]).(\text{put}(\text{bool}) \mid \text{get}(\text{bool}).b \mid \varepsilon))$$

which is clearly safe (under suitable assumptions on b).

B Proofs of Results in Main Text

Lemma 4.11 Given B a behavior context and b a behavior, and assume that test tests B . Let $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ belong to $\llbracket B[\text{test.test}] \rrbracket$, and let tr_0 belong to $\{tr_2\} \parallel \llbracket b \rrbracket$. Then $tr_1 \diamond tr_0 \diamond tr_3$ belongs to $\llbracket B[b] \rrbracket$.

Proof. The proof is by induction in B , where we perform a case analysis on the form of B (omitting two symmetric cases).

$B = \square$. We infer that $tr_1 = tr_2 = tr_3 = \bullet$, and that $tr_0 \in \llbracket b \rrbracket$. Then the claim clearly holds.

$B = b'.B'$. With the assumptions of the lemma given, we can (since $B[\text{test.test}] = b'.B'[\text{test.test}]$) write $tr_1 = tr' \diamond tr'_1$ where $tr' \in \llbracket b' \rrbracket$ and where $tr'_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ belongs to $\llbracket B'[\text{test.test}] \rrbracket$. Using the induction hypothesis, we infer that $tr'_1 \diamond tr_0 \diamond tr_3$ belongs to $\llbracket B'[b] \rrbracket$. But then $tr_1 \diamond tr_0 \diamond tr_3$ belongs to $\llbracket b' \rrbracket \diamond \llbracket B'[b] \rrbracket = \llbracket b'.B'[b] \rrbracket = \llbracket B[b] \rrbracket$, as desired.

$B = b' \mid B'$. With the assumptions of the lemma given, we infer (since $B[\text{test.test}] = b' \parallel B'[\text{test.test}]$) that there exists $tr'_1, tr'_2, tr'_3, tr''_1, tr''_2, tr''_3$ such that for $i \in \{1, 2, 3\}$ we have $tr_i \in \{tr'_i\} \parallel \{tr''_i\}$ and such that $tr'_1 \diamond tr'_2 \diamond tr'_3 \in \llbracket b' \rrbracket$ and such that $tr''_1 \diamond \text{test} \diamond tr''_2 \diamond \text{test} \diamond tr''_3$ belongs to $\llbracket B'[\text{test.test}] \rrbracket$. By associativity of interleaving, there exists $tr'_0 \in \{tr''_2\} \parallel \llbracket b \rrbracket$ such that $tr_0 \in \{tr'_2\} \parallel \{tr'_0\}$. Using the induction hypothesis on B' , we infer that $tr''_1 \diamond tr'_0 \diamond tr''_3 \in \llbracket B'[b] \rrbracket$. But then clearly $tr_1 \diamond tr_0 \diamond tr_3 \in \llbracket b' \rrbracket \parallel \llbracket B'[b] \rrbracket = \llbracket B[b] \rrbracket$, as desired. \square

Lemma 4.12 Given B a behavior context and b a behavior, and assume that test tests B . Let tr belong to $\llbracket B[b] \rrbracket$. Then there exists $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ in $\llbracket B[\text{test.test}] \rrbracket$ such that we can write $tr = tr_1 \diamond tr_0 \diamond tr_3$ with $tr_0 \in \{tr_2\} \parallel \llbracket b \rrbracket$.

Proof. The proof is by induction in B , where we perform a case analysis on the form of B (omitting two symmetric cases).

$B = \square$. Now $tr \in \llbracket b \rrbracket$, so choosing $tr_0 = tr$ and $tr_1 = tr_2 = tr_3 = \bullet$ clearly yields the claim.

$B = b'.B'$. We can write $tr = tr' \diamond tr''$ with $tr' \in \llbracket b' \rrbracket$ and with $tr'' \in \llbracket B'[b] \rrbracket$. Inductively, there exists $tr'_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3$ in $\llbracket B'[\text{test.test}] \rrbracket$ such that we can write $tr'' = tr'_1 \diamond tr_0 \diamond tr_3$ with $tr_0 \in \{tr_2\} \parallel \llbracket b \rrbracket$. Let $tr_1 = tr' \diamond tr'_1$, then we have the desired relations $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3 \in \llbracket b' \rrbracket \diamond \llbracket B'[\text{test.test}] \rrbracket = \llbracket B[\text{test.test}] \rrbracket$ and $tr = tr_1 \diamond tr_0 \diamond tr_3$.

$B = b' \mid B'$. There exists $tr' \in \llbracket b' \rrbracket$ and $tr'' \in \llbracket B'[b] \rrbracket$ such that $tr \in \{tr'\} \parallel \{tr''\}$. The induction hypothesis gives us $tr''_1 \diamond \text{test} \diamond tr''_2 \diamond \text{test} \diamond tr''_3$ in $\llbracket B'[\text{test.test}] \rrbracket$ such that $tr'' = tr''_1 \diamond tr''_0 \diamond tr''_3$ with $tr''_0 \in \{tr''_2\} \parallel \llbracket b \rrbracket$. Since $tr \in \{tr'\} \parallel \{tr''_1 \diamond tr''_0 \diamond tr''_3\}$ there clearly exists $tr'_1, tr_1, tr'_3, tr_3, tr'_0, tr_0$ such that $tr' = tr'_1 \diamond tr'_0 \diamond tr'_3$, $tr = tr_1 \diamond tr_0 \diamond tr_3$, and for $i \in \{0, 1, 3\}$ also $tr_i \in \{tr'_i\} \parallel \{tr''_i\}$. Since tr_0 belongs to $\{tr'_0\} \parallel (\{tr''_2\} \parallel \llbracket b \rrbracket)$, associativity of interleaving enables us to find $tr_2 \in \{tr'_0\} \parallel \{tr''_2\}$ such that $tr_0 \in \{tr_2\} \parallel \llbracket b \rrbracket$. This establishes the desired relation $tr_1 \diamond \text{test} \diamond tr_2 \diamond \text{test} \diamond tr_3 \in \{tr'_1 \diamond tr'_0 \diamond tr'_3\} \parallel \{tr''_1 \diamond \text{test} \diamond tr''_2 \diamond \text{test} \diamond tr''_3\} \subseteq \llbracket b' \rrbracket \parallel \llbracket B'[\text{test.test}] \rrbracket = \llbracket B[\text{test.test}] \rrbracket$. \square

Lemma 5.7 Suppose that $P \equiv Q$. Then $E \vdash P : b$ if and only if $E \vdash Q : b$.

Proof. The proof is by induction on the derivation of $P \equiv Q$, much as the similar result in [CG99]. Below we list the most interesting cases.

(Struct ParComm). Follows from commutativity of “ \mid ”.

(Struct ParAssoc). Assume that $E \vdash (P \mid Q) \mid R : b$ (the other direction is similar). Then there exists b_{12} and b_3 with $b_{12} \mid b_3 \leq b$ such that $E \vdash P \mid Q : b_{12}$ and $E \vdash R : b_3$. Therefore there exists b_1 and b_2 with $b_1 \mid b_2 \leq b_{12}$ such that $E \vdash P : b_1$ and $E \vdash Q : b_2$. Now $E \vdash P \mid (Q \mid R) : b_1 \mid (b_2 \mid b_3)$, and by Lemma 4.7 we infer that $b_1 \mid (b_2 \mid b_3) \equiv (b_1 \mid b_2) \mid b_3 \leq b_{12} \mid b_3 \leq b$. This implies the desired judgment $E \vdash P \mid (Q \mid R) : b$.

(Struct ResRes). Follows easily from Lemma 5.1.

(Struct ResPar). Assume that $E \vdash (\nu n : \tau).(P \mid Q) : b$. There exists b_1 and b_2 with $b_1 \mid b_2 \leq b$ such that $E, n : \tau \vdash P : b_1$ and $E, n : \tau \vdash Q : b_2$. Since $n \notin \text{fn}(P)$, we can α -rename P into a P' such that $n \notin \text{names}(P')$. Clearly $E, n : \tau \vdash P' : b_1$ (this claim amounts to the case (Struct α - rename)), so by Lemma 5.3 we have $E \vdash P' : b_1$ and therefore also $E \vdash P : b_1$. We can thus infer first $E \vdash (\nu n : \tau).Q : b_2$ and then (since $b_1 \mid b_2 \leq b$) the desired judgment $E \vdash P \mid (\nu n : \tau).Q : b$.

The other direction is quite similar, employing Lemma 5.2 rather than Lemma 5.3.

(Struct ResAmb). Let $E(m) = (E, n : \tau)(m) = \text{amb}[b_1, b_2]$. Assume that $E \vdash (\nu n : \tau).m[P] : b$ (the other direction is similar). We infer that $\varepsilon \leq b$, and that there exists safe b_0 such that $E, n : \tau \vdash P : b_0$ and $b_0 \rightsquigarrow b_1$ and $b_1 \leq b_2$. This implies that $E \vdash (\nu n : \tau).P : b_0$, clearly enabling us to infer the desired judgment $E \vdash m[(\nu n : \tau).P] : b$.

(Struct ZeroPar). Assume that $E \vdash P \mid \mathbf{0} : b$. There exists b_1 and b_2 with $b_1 \mid b_2 \leq b$ such that $E \vdash P : b_1$ and $E \vdash \mathbf{0} : b_2$. We infer that $\varepsilon \leq b_2$ and by Lemma 4.7 therefore $b_1 \equiv b_1 \mid \varepsilon \leq b_1 \mid b_2 \leq b$. This implies the desired judgment $E \vdash P : b$.

The other direction is trivial.

(Struct ZeroRepl). First assume that $E \vdash !\mathbf{0} : b$. We infer that there exists $b_0 \leq b$ such that $E \vdash \mathbf{0} : b_0$. But then also $E \vdash \mathbf{0} : b$, as desired.

Conversely, assume that $E \vdash \mathbf{0} : b$ from which we infer that $\varepsilon \leq b$. Since $\varepsilon \mid \varepsilon \leq \varepsilon$ we can apply (Proc Zero) and (Proc Repl) to derive $E \vdash !\mathbf{0} : \varepsilon$, and therefore also the desired $E \vdash !\mathbf{0} : b$.

(Struct ε). Assume that $E \vdash P : b$. Since $E \vdash \varepsilon : \text{cap}[\square]$, we can by (Proc Action) infer $E \vdash \varepsilon.P : b$.

Conversely, assume that $E \vdash \varepsilon.P : b$. Then there exists B and b_0 with $B[b_0] \leq b$ such that $E \vdash \varepsilon : \text{cap}[B]$ and $E \vdash P : b_0$. As $\text{cap}[\square] \leq \text{cap}[B]$ we deduce that $\square \leq B$, implying (by Lemma 4.13) that $b_0 = \square[b_0] \leq B[b_0] \leq b$. Thus we can derive the desired judgment $E \vdash P : b$.

(Struct \cdot). Assume that $E \vdash (M_1.M_2).P : b$. There exists B_0 and b_0 with $B_0[b_0] \leq b$ such that $E \vdash M_1.M_2 : \text{cap}[B_0]$ and $E \vdash P : b_0$. There thus exists B_1 and B_2 with $\text{cap}[B_1[B_2]] \leq \text{cap}[B_0]$, implying $B_1[B_2] \leq B_0$,

such that $E \vdash M_1 : \text{cap}[B_1]$ and $E \vdash M_2 : \text{cap}[B_2]$. We can thus infer first $E \vdash M_2.P : B_2[b_0]$ and then $E \vdash M_1.(M_2.P) : B_1[B_2[b_0]]$. But by employing Lemmas 4.9 and 4.13 we deduce that $B_1[B_2[b_0]] = (B_1[B_2])[b_0] \leq B_0[b_0] \leq b$. This shows that we can derive the desired judgment $E \vdash M_1.(M_2.P) : b$.

Conversely, assume that $E \vdash M_1.(M_2.P) : b$. There exists B_1 and b_1 with $B_1[b_1] \leq b$ such that $E \vdash M_1 : \text{cap}[B_1]$ and $E \vdash M_2.P : b_1$, and therefore there exists B_2 and b_0 with $B_2[b_0] \leq b_1$ such that $E \vdash M_2 : \text{cap}[B_2]$ and $E \vdash P : b_0$. We can thus infer first $E \vdash M_1.M_2 : \text{cap}[B_1[B_2]]$ and then $E \vdash (M_1.M_2).P : (B_1[B_2])[b_0]$. But by employing Lemma 4.9 and the fact that “ $|$ ” and “ \cdot ” respect \leq (Lemma 4.7), we deduce that $(B_1[B_2])[b_0] = B_1[B_2[b_0]] \leq B_1[b_1] \leq b$. This shows that we can derive the desired judgment $E \vdash (M_1.M_2).P : b$. \square

Lemma 6.5 Given b of level i , we can construct G of level i implementing b .

Proof. The proof is by structural induction in b , where we perform a case analysis:

$b = \varepsilon$. Choose some ι and let $G = (\{\iota\}, \iota, \{\iota\}, \emptyset)$; then $\text{Acc}(G) = \{\bullet\} = \llbracket \varepsilon \rrbracket$.

$b = \text{put}(\sigma)$. (The cases $b = \text{get}(\sigma)$ and $b = \text{diss}$ are similar.) Choose distinct ι and q and let

$$G = (\{\iota, q\}, \iota, \{q\}, \{(\iota, \text{put}(\sigma), q)\}).$$

Then $\text{Acc}(G) = \{\text{put}(\sigma)\} = \llbracket b \rrbracket$, and since b is of level i also G is of level i .

$b = \text{fromnow } T$. Choose some ι and let $G = (\{\iota\}, \iota, \{\iota\}, \delta)$ where $\delta = \bigcup_{\sigma \in T} \{(\iota, \text{put}(\sigma), \iota), (\iota, \text{get}(\sigma), \iota)\}$. This clearly does the job.

$b = b_1.b_2$. Since b_1 as well as b_2 are of level i , we can apply the induction hypothesis to construct level i automata G_1 implementing b_1 and G_2 implementing b_2 . Let $G_j = (\mathcal{Q}_j, \iota_j, F_j, \delta_j)$ for $j = 1, 2$; wlog. we can assume that \mathcal{Q}_1 and \mathcal{Q}_2 are disjoint. We now construct $G = (\mathcal{Q}, \iota, F, \delta)$ as follows:

$$\begin{aligned} \mathcal{Q} &= \mathcal{Q}_1 \cup \mathcal{Q}_2 \\ \iota &= \iota_1 \\ F &= F_2 \cup (\text{if } \iota_2 \in F_2 \text{ then } F_1 \text{ else } \emptyset) \\ \delta &= \delta_1 \cup \delta_2 \cup \{(q_1, a, q_2) \mid q_1 \in F_1 \text{ and } (\iota_2, a, q_2) \in \delta_2\} \end{aligned}$$

We first prove that $\llbracket b \rrbracket$ is a subset of $\text{Acc}(G)$. So let $tr \in \llbracket b \rrbracket$ be given. We can write $tr = tr_1 \diamond tr_2$ with $tr_1 \in \llbracket b_1 \rrbracket$ and $tr_2 \in \llbracket b_2 \rrbracket$. Therefore $tr_1 \in \text{Acc}(G_1)$ and $tr_2 \in \text{Acc}(G_2)$, implying that there exists $q_1 \in F_1$ and $q_2 \in F_2$ such that $(\iota_1, tr_1, q_1) \in \delta_1^*$ and $(\iota_2, tr_2, q_2) \in \delta_2^*$. We do a case analysis on tr_2 :

- If $tr_2 = \bullet$, then $\iota_2 = q_2 \in F_2$ implying $F_1 \subseteq F$ and therefore $q_1 \in F$. This shows the desired relation $tr \in \text{Acc}(G)$, since from $\iota = \iota_1$ and $tr = tr_1$ and $\delta_1 \subseteq \delta$ we infer that $(\iota, tr, q_1) \in \delta^*$.
- If tr_2 takes the form $a \diamond tr'_2$, there exists q'_2 such that $(\iota_2, a, q'_2) \in \delta_2$ and $(q'_2, tr'_2, q_2) \in \delta_2^*$. By construction of δ , the former relation implies that $(q_1, a, q'_2) \in \delta$. Moreover, it clearly holds that $(\iota_1, tr_1, q_1) \in \delta_1^*$ and that $(q'_2, tr'_2, q_2) \in \delta_2^*$. Therefore $(\iota_1, tr_1 \diamond (a \diamond tr'_2), q_2) = (\iota, tr, q_2)$ belongs to δ^* , which since $q_2 \in F_2 \subseteq F$ implies the desired relation $tr \in \text{Acc}(G)$.

Next we prove that $\text{Acc}(G)$ is a subset of $\llbracket b \rrbracket$. So let $tr \in \text{Acc}(G)$ be given. That is, there exists $q \in F$ such that $(\iota, tr, q) \in \delta^*$. There are now two cases to consider:

- If $q \in F_1$ then it is easy to see that $(\iota, tr, q) \in \delta_1^*$. Thus $tr \in \text{Acc}(G_1) = \llbracket b_1 \rrbracket$. Moreover, from the construction of F we infer that $\iota_2 \in F_2$, showing that $\bullet \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$. Therefore $tr = tr \diamond \bullet$ belongs to $\llbracket b \rrbracket$, as desired.
- If $q \in F_2$ then it is easy to see that there exists $q_1 \in \mathcal{Q}_1$ and $q_2 \in \mathcal{Q}_2$ such that we can write $tr = tr_1 \diamond (a \diamond tr_2)$ with $(\iota_1, tr_1, q_1) \in \delta_1^*$ and $(q_2, tr_2, q) \in \delta_2^*$ and $(q_1, a, q_2) \in \delta$, implying that $q_1 \in F_1$ and that $(\iota_2, a, q_2) \in \delta_2$. This shows that $tr_1 \in \text{Acc}(G_1) = \llbracket b_1 \rrbracket$, and that $(\iota_2, a \diamond tr_2, q) \in \delta_2^*$ which amounts to $a \diamond tr_2 \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$. This establishes the desired relation $tr \in \llbracket b \rrbracket$.

$b = b_1 \mid b_2$. Since b_1 as well as b_2 are of level i , we can apply the induction hypothesis to construct level i automata G_1 implementing b_1 and G_2 implementing b_2 . Let $G_j = (\mathcal{Q}_j, \iota_j, F_j, \delta_j)$ for $j = 1, 2$. We now construct $G = (\mathcal{Q}, \iota, F, \delta)$ as follows:

$$\begin{aligned} \mathcal{Q} &= \mathcal{Q}_1 \times \mathcal{Q}_2 \\ \iota &= (\iota_1, \iota_2) \\ F &= F_1 \times F_2 \\ \delta &= \{((q_1, q_2), a, (q'_1, q_2)) \mid (q_1, a, q'_1) \in \delta_1\} \\ &\quad \cup \{((q_1, q_2), a, (q_1, q'_2)) \mid (q_2, a, q'_2) \in \delta_2\} \end{aligned}$$

To reason about G , we first establish

$$(q_1, tr_1, q'_1) \in \delta_1^* \wedge (q_2, tr_2, q'_2) \in \delta_2^* \wedge tr \in \{tr_1\} \parallel \{tr_2\} \implies ((q_1, q_2), tr, (q'_1, q'_2)) \in \delta^*. \quad (1)$$

We prove (1) by induction on the length of tr . If $tr = \bullet$, then also $tr_1 = tr_2 = \bullet$ so $q'_1 = q_1$ and $q'_2 = q_2$ and the claim is trivial.

If tr takes the form $a \diamond tr'$, we can wlog. assume that there exists tr'_1 such that $tr_1 = a \diamond tr'_1$ and $tr' \in \{tr'_1\} \parallel \{tr_2\}$. Thus there exists $q'_1 \in \mathcal{Q}_1$ such that $(q_1, a, q'_1) \in \delta_1$, implying $((q_1, q_2), a, (q'_1, q_2)) \in \delta$, and $(q'_1, tr'_1, q'_1) \in \delta_1^*$. The induction hypothesis tells us that $((q'_1, q_2), tr', (q'_1, q'_2)) \in \delta^*$, enabling us to arrive at the desired relation $((q_1, q_2), tr, (q'_1, q'_2)) \in \delta^*$.

Next we establish, also by induction on the length of tr , that

$$((q_1, q_2), tr, (q'_1, q'_2)) \in \delta^* \implies \exists tr_1, tr_2: tr \in \{tr_1\} \parallel \{tr_2\} \wedge (q_1, tr_1, q'_1) \in \delta_1^* \wedge (q_2, tr_2, q'_2) \in \delta_2^*. \quad (2)$$

If $tr = \bullet$, then $q'_1 = q_1$ and $q'_2 = q_2$. We can thus choose $tr_1 = tr_2 = \bullet$.

If tr takes the form $a \diamond tr'$, then there exists q such that $((q_1, q_2), a, q) \in \delta$ and $(q, tr', (q'_1, q'_2)) \in \delta^*$. Wlog. we can assume that there exists q'_2 such that $q = (q_1, q'_2)$ and $(q_2, a, q'_2) \in \delta_2$. The induction hypothesis tells us that there exists tr_1 and tr'_2 with $tr' \in \{tr_1\} \parallel \{tr'_2\}$ such that $(q_1, tr_1, q'_1) \in \delta_1^*$ and $(q'_2, tr'_2, q'_2) \in \delta_2^*$. Let $tr_2 = a \diamond tr'_2$. Then clearly $tr \in \{tr_1\} \parallel \{tr_2\}$ and $(q_2, tr_2, q'_2) \in \delta_2^*$, as desired.

We are now ready to embark on the proof that $\llbracket b \rrbracket = \text{Acc}(G)$. If $tr \in \llbracket b \rrbracket$ there exists tr_1 and tr_2 with $tr \in \{tr_1\} \parallel \{tr_2\}$ such that $tr_1 \in \llbracket b_1 \rrbracket = \text{Acc}(G_1)$ and $tr_2 \in \llbracket b_2 \rrbracket = \text{Acc}(G_2)$. There thus exists $q_1 \in F_1$ and $q_2 \in F_2$ such that $(\iota_1, tr_1, q_1) \in \delta_1^*$ and $(\iota_2, tr_2, q_2) \in \delta_2^*$. By (1) we infer that $(\iota, tr, (q_1, q_2)) \in \delta^*$. Since $(q_1, q_2) \in F$, this shows that $tr \in \text{Acc}(G)$. Conversely, assume that $tr \in \text{Acc}(G)$. Thus there exists $(q_1, q_2) \in F$ such that $((\iota_1, \iota_2), tr, (q_1, q_2)) \in \delta^*$. By (2) we infer that there exists tr_1, tr_2 with $tr \in \{tr_1\} \parallel \{tr_2\}$ such that $(\iota_1, tr_1, q_1) \in \delta_1^*$ and $(\iota_2, tr_2, q_2) \in \delta_2^*$. Since $q_1 \in F_1$ and $q_2 \in F_2$, this shows that $tr_1 \in \text{Acc}(G_1) = \llbracket b_1 \rrbracket$ and that $tr_2 \in \text{Acc}(G_2) = \llbracket b_2 \rrbracket$, implying $tr \in \llbracket b \rrbracket$ as desired. \square

Theorem 7.1 Suppose that $E^C \vdash P^C : T^C$, respectively $E^C \vdash M^C : W^C$, is derivable in the system of [CG99, Sect. 3]. Then $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$, respectively $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{Plus}(W^C)$, is derivable in our system.

Proof. The proof is by induction in the derivation (in the system of [CG99]), where we perform a case analysis on the last rule applied and where we employ the following simple observations:

$$\forall T^C : \varepsilon \leq \text{Plus}(T^C) \quad (3)$$

$$\forall T^C : \text{Plus}(T^C) \mid \text{Plus}(T^C) \leq \text{Plus}(T^C) \quad (4)$$

$$\forall T^C : \text{Plus}(T^C) \mid \text{diss is safe} \quad (5)$$

$$\forall T^C : (\text{Plus}(T^C) \mid \text{diss}) \rightsquigarrow \text{Plus}(T^C). \quad (6)$$

(Exp n). Our assumption is that $E^C \vdash n : W^C$ because $E^C(n) = W^C$. But then $(\text{Plus}(E^C))(n) = \text{Plus}(W^C)$, implying the desired judgment $\text{Plus}(E^C) \vdash \text{Plus}(n) : \text{Plus}(W^C)$.

(Exp ϵ). Our assumption is that $E^C \vdash \epsilon : \text{cap}[T^C]$. We must prove that $\text{Plus}(E^C) \vdash \epsilon : \text{cap}[\text{Plus}(T^C) \mid \square]$, and, since by (Exp ϵ) we have $\text{Plus}(E^C) \vdash \epsilon : \text{cap}[\square]$, it is sufficient to show that $\text{cap}[\square] \leq \text{cap}[\text{Plus}(T^C) \mid \square]$ which amounts to establishing $\square \leq \text{Plus}(T^C) \mid \square$. But this follows since for all b we have, employing (3), that $\square[b] = b \equiv \varepsilon \mid b \leq \text{Plus}(T^C) \mid b = (\text{Plus}(T^C) \mid \square)[b]$.

(Exp .). Our assumption is that $E^C \vdash M_1^C.M_2^C : \text{cap}[T^C]$ because $E^C \vdash M_1^C : \text{cap}[T^C]$ and $E^C \vdash M_2^C : \text{cap}[T^C]$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(M_1^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$ and $\text{Plus}(E^C) \vdash \text{Plus}(M_2^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$, which by (Exp Action) yields $\text{Plus}(E^C) \vdash \text{Plus}(M_1^C).\text{Plus}(M_2^C) : \text{cap}[\text{Plus}(T^C) \mid (\text{Plus}(T^C) \mid \square)]$. An application of (Exp Subsumption) yields the judgment $\text{Plus}(E^C) \vdash \text{Plus}(M_1^C.M_2^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$ provided we can establish $\text{Plus}(T^C) \mid (\text{Plus}(T^C) \mid \square) \leq \text{Plus}(T^C) \mid \square$ which amounts to showing that for all b we have $\text{Plus}(T^C) \mid \text{Plus}(T^C) \mid b \leq \text{Plus}(T^C) \mid b$. But this follows from (4).

(Exp In). Our assumption is that $E^C \vdash \text{in } M^C : \text{cap}[T^C]$ because $E^C \vdash M^C : \text{amb}[S^C]$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{amb}[\text{Plus}(S^C), \text{Plus}(S^C)]$ which by (Exp In) yields $\text{Plus}(E^C) \vdash \text{in } \text{Plus}(M^C) : \text{cap}[\square]$. An application of (Exp Subsumption) yields the desired judgment $\text{Plus}(E^C) \vdash \text{Plus}(\text{in } M^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$, provided we can show that $\square \leq \text{Plus}(T^C) \mid \square$. But this follows as in the case (Exp ϵ).

(Exp Out). Similar to the case (Exp In).

(Exp Open). Our assumption is that $E^C \vdash \text{open } M^C : \text{cap}[T^C]$ because $E^C \vdash M^C : \text{amb}[T^C]$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{amb}[\text{Plus}(T^C), \text{Plus}(T^C)]$ which by (Exp Open) yields the desired judgment $\text{Plus}(E^C) \vdash \text{Plus}(\text{open } M^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$.

(Proc Action). Our assumption is that $E^C \vdash M^C.P^C : T^C$ because $E^C \vdash M^C : \text{cap}[T^C]$ and $E^C \vdash P^C : T^C$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{cap}[\text{Plus}(T^C) \mid \square]$ and $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$, which by (Proc Action) yields $\text{Plus}(E^C) \vdash \text{Plus}(M^C).\text{Plus}(P^C) : \text{Plus}(T^C) \mid \text{Plus}(T^C)$. Using (4) and the rule (Proc Subsumption), this yields the judgment $\text{Plus}(E^C) \vdash \text{Plus}(M^C.P^C) : \text{Plus}(T^C)$.

(Proc Amb). Our assumption is that $E^C \vdash M^C[P^C] : S^C$ because $E^C \vdash M^C : \text{amb}[T^C]$ and $E^C \vdash P^C : T^C$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(M^C) : \text{amb}[\text{Plus}(T^C), \text{Plus}(T^C)]$ and $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$, and therefore also $\text{Plus}(E^C) \vdash \text{Plus}(P^C) \mid \text{coopen } \text{Plus}(M^C) : \text{Plus}(T^C) \mid \text{diss}$. Using (5) and (6) we can thus apply (Proc Amb) to get the judgement $\text{Plus}(E^C) \vdash \text{Plus}(M^C)[\text{Plus}(P^C) \mid \text{coopen } \text{Plus}(M^C)] : \varepsilon$ which by (3) yields $\text{Plus}(E^C) \vdash \text{Plus}(M^C[P^C]) : \text{Plus}(S^C)$.

(Proc Res). Our assumption is that $E^C \vdash (\nu n : \text{amb}[T^C]).P^C : S^C$ because $E^C, n : \text{amb}[T^C] \vdash P^C : S^C$. By applying the induction hypothesis we infer that $\text{Plus}(E^C), n : \text{amb}[\text{Plus}(T^C), \text{Plus}(T^C)] \vdash \text{Plus}(P^C) : \text{Plus}(S^C)$ which by (Proc Res) yields $\text{Plus}(E^C) \vdash (\nu n : \text{amb}[\text{Plus}(T^C), \text{Plus}(T^C)]).\text{Plus}(P^C) : \text{Plus}(S^C)$.

(Proc Zero). Our assumption is that $E^C \vdash \mathbf{0} : T^C$. We must prove that $\text{Plus}(E^C) \vdash \text{Plus}(\mathbf{0}) : \text{Plus}(T^C)$, but since $\text{Plus}(E^C) \vdash \mathbf{0} : \varepsilon$ this follows from (3).

(Proc Par). Our assumption is that $E^C \vdash P^C \mid Q^C : T^C$ because $E^C \vdash P^C : T^C$ and $E^C \vdash Q^C : T^C$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$ and that $\text{Plus}(E^C) \vdash \text{Plus}(Q^C) : \text{Plus}(T^C)$, which by (Proc Par) yields $\text{Plus}(E^C) \vdash \text{Plus}(P^C) \mid \text{Plus}(Q^C) : \text{Plus}(T^C) \mid \text{Plus}(T^C)$. Using (4) and the rule (Proc Subsumption), this yields the judgment $\text{Plus}(E^C) \vdash \text{Plus}(P^C \mid Q^C) : \text{Plus}(T^C)$.

(Proc Repl). Our assumption is that $E^C \vdash !P^C : T^C$ because $E^C \vdash P^C : T^C$. By applying the induction hypothesis we infer that $\text{Plus}(E^C) \vdash \text{Plus}(P^C) : \text{Plus}(T^C)$, and due to (4) we can apply (Proc Repl) to infer the desired judgment $\text{Plus}(E^C) \vdash !\text{Plus}(P^C) : \text{Plus}(T^C)$.

(Proc Input). Our assumption is that $E^C \vdash (n_1 : W_1^C, \dots, n_k : W_k^C).P^C : \times(W_1^C, \dots, W_k^C)$ because

$$E^C, n_1 : W_1^C, \dots, n_k : W_k^C \vdash P^C : \times(W_1^C, \dots, W_k^C).$$

Let $\tau = \times(Plus(W_1^C), \dots, Plus(W_k^C))$, thus $Plus(\times(W_1^C, \dots, W_k^C)) = \text{fromnow } \{\tau\}$. By applying the induction hypothesis we infer that $Plus(E^C), n_1 : Plus(W_1^C), \dots, n_k : Plus(W_k^C) \vdash Plus(P^C) : \text{fromnow } \{\tau\}$ which by (Proc Input) yields the judgement $Plus(E^C) \vdash (n_1 : Plus(W_1^C), \dots, n_k : Plus(W_k^C)).Plus(P^C) : \text{get}(\tau).\text{fromnow } \{\tau\}$. Since clearly $\text{get}(\tau).\text{fromnow } \{\tau\} \leq \text{fromnow } \{\tau\}$, we can apply (Proc Subsumption) to arrive at the desired judgment

$$Plus(E^C) \vdash Plus((n_1 : W_1^C, \dots, n_k : W_k^C).P^C) : Plus(\times(W_1^C, \dots, W_k^C)).$$

(Proc Output). Our assumption is that $E^C \vdash \langle M_1^C, \dots, M_k^C \rangle : \times(W_1^C, \dots, W_k^C)$ because for all $i \in \{1 \dots k\}$ we have $E^C \vdash M_i^C : W_i^C$. Let $\tau = \times(Plus(W_1^C), \dots, Plus(W_k^C))$, so that $Plus(\times(W_1^C, \dots, W_k^C)) = \text{fromnow } \{\tau\}$. By applying the induction hypothesis we infer that for all $i \in \{1 \dots k\}$ it holds that $Plus(E^C) \vdash Plus(M_i^C) : Plus(W_i^C)$ which by (Exp Tuple) and (Proc Output) yields the judgement

$$Plus(E^C) \vdash \langle \times(Plus(M_1^C), \dots, Plus(M_k^C)) \rangle : \text{put}(\tau).$$

Since clearly $\text{put}(\tau) \leq \text{fromnow } \{\tau\}$ we can apply (Proc Subsumption) to arrive at the desired judgment $Plus(E^C) \vdash Plus(\langle M_1^C, \dots, M_k^C \rangle) : Plus(\times(W_1^C, \dots, W_k^C))$. \square

Lemma B.1. *Suppose that $P_1^C \equiv P_2^C$ holds by a derivation in the system of [CG99] where the rule $!P \equiv P \mid !P$ has not been applied. Then $Plus(P_1^C) \equiv Plus(P_2^C)$ holds in our system.* \square

Proof. An easy induction in the derivation; below we list the non-trivial cases.

(Struct Amb). Here $M^C[P^C] \equiv M^C[Q^C]$ because $P^C \equiv Q^C$. By applying the induction hypothesis we infer that $Plus(P^C) \equiv Plus(Q^C)$, implying that $Plus(P^C) \mid \text{coopen } Plus(M^C) \equiv Plus(Q^C) \mid \text{coopen } Plus(M^C)$. Therefore we have the desired relation

$$\begin{aligned} Plus(M^C[P^C]) &= Plus(M^C)[Plus(P^C) \mid \text{coopen } Plus(M^C)] \\ &\equiv Plus(M^C)[Plus(Q^C) \mid \text{coopen } Plus(M^C)] = Plus(M^C[Q^C]). \end{aligned}$$

(Struct ResPar). Here $(\nu n : W^C).(P^C \mid Q^C) \equiv P^C \mid (\nu n : W^C).Q^C$ because $n \notin \text{fn}(P^C)$. It is easy to establish (by structural induction in P^C) that then $n \notin \text{fn}(Plus(P^C))$, and therefore we have the desired relation

$$(\nu n : Plus(W^C)).(Plus(P^C) \mid Plus(Q^C)) \equiv Plus(P^C) \mid (\nu n : Plus(W^C)).Plus(Q^C).$$

(Struct ResAmb). Here $(\nu n : W^C).m[P^C] \equiv m[(\nu n : W^C).P^C]$ because $n \neq m$. Using (Struct ResAmb) and (Struct ResPar), and that $n \notin \text{fn}(\text{coopen } m)$, we obtain the desired relation

$$\begin{aligned} (\nu n : Plus(W^C)).m[Plus(P^C) \mid \text{coopen } m] &\equiv m[(\nu n : Plus(W^C)).(Plus(P^C) \mid \text{coopen } m)] \\ &\equiv m[(\nu n : Plus(W^C)).Plus(P^C) \mid \text{coopen } m], \end{aligned}$$

where we have used the fact that $Plus(m) = m$. \square

Theorem B.2. *If $Q^C \longrightarrow R^C$ holds in the system of [CG99], except that the rule $!P \equiv P \mid !P$ has been replaced by $!P \longrightarrow P \mid !P$, then $Plus(Q^C) \xrightarrow{\ell} Plus(R^C)$ holds in our system for some ℓ .* \square

Proof. An easy induction in the derivation of $Q^C \longrightarrow R^C$, using Lemma B.1. Below we list the non-trivial cases:

(Red In). The situation is that

$$n[\text{in } m.P^C \mid Q^C] \mid m[R^C] \longrightarrow m[n[P^C \mid Q^C] \mid R^C].$$

Let $P = Plus(P^C)$, $Q = Plus(Q^C)$, and $R = Plus(R^C)$. In our system we then have the desired relation

$$\begin{aligned} & n[(in\ m.P \mid Q) \mid coopen\ n] \mid m[R \mid coopen\ m] \\ \equiv & n[in\ m.P \mid (Q \mid coopen\ n)] \mid m[R \mid coopen\ m] \\ \xrightarrow{\epsilon} & m[n[P \mid (Q \mid coopen\ n)] \mid (R \mid coopen\ m)] \equiv m[(n[(P \mid Q) \mid coopen\ n] \mid R) \mid coopen\ m] \end{aligned}$$

(Red Out). The situation is that

$$m[n[out\ m.P^C \mid Q^C] \mid R^C] \longrightarrow n[P^C \mid Q^C] \mid m[R^C].$$

Let $P = Plus(P^C)$, $Q = Plus(Q^C)$, and $R = Plus(R^C)$. In our system we then have the desired relation

$$\begin{aligned} & m[(n[(out\ m.P \mid Q) \mid coopen\ n] \mid R) \mid coopen\ m] \\ \equiv & m[n[out\ m.P \mid (Q \mid coopen\ n)] \mid (R \mid coopen\ m)] \\ \xrightarrow{\epsilon} & n[P \mid (Q \mid coopen\ n)] \mid m[R \mid coopen\ m] \equiv n[(P \mid Q) \mid coopen\ n] \mid m[R \mid coopen\ m] \end{aligned}$$

(Red Open). The situation is that

$$open\ n.P^C \mid n[Q^C] \longrightarrow P^C \mid Q^C.$$

Let $P = Plus(P^C)$, and $Q = Plus(Q^C)$. In our system we then have the desired relation

$$open\ n.P \mid n[Q \mid coopen\ n] \xrightarrow{\epsilon} P \mid (Q \mid \mathbf{0}) \equiv P \mid Q$$

□