

What are Polymorphically-Typed Ambients?

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

Assaf J. Kfoury
Boston University
www.cs.bu.edu/~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, CGG99].

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 others.¹ 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.

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

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.

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 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.

²The expression “arity polymorphism” was used already by others to describe similar situations, though quite different in some respects, in particular in functional programming languages. See, for example, Tullsen’s recent work on the Zip Calculus [Tul00].

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

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 [PS93, PT97] for the π -calculus and [Zim00] for **AC**.

The two last cases by contrast, *arity polymorphism* and *orderly communication*, raise entirely new problems not encountered before in the context of **AC**. The design of a type discipline enforcing them is a delicate matter, especially in the case of orderly communication. 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 Section 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 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 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 may expect its incorporation into our type system to 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

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.

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 semantics.⁵

The meaning of a behavior is a set of traces; this provides a denotational semantics for the mobility and communication parts of **AC+**. A denotational semantics for the embedded part of **AC+** is just a standard CPO-based call-by-value denotational semantics for simply-typed functional programs. The latter is not considered here and its integration with the trace semantics is left to a later report.

- 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 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.

Further material and all missing proofs are included in the technical report [AKPG00], on which this extended abstract is based (this report can be downloaded from the Church Project web site at <http://types.bu.edu/reports/>).

2 Motivating Examples

We give two examples, short but more interesting than the snippets in Section 1.1, to illustrate the expressive power and convenience of a polymorphically typed **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 “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 coopen n (cf. (Red Open) in Fig. 2). Throughout the rest of the report, we 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.

⁵We 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.

⁶This 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).

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

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 the packet communicates a destination (of string type) and receives a path (of capability type).⁸

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

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 **AC** nor, to the best of our knowledge, in any of the type systems for **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 server.⁹

$$\begin{array}{l} \text{server} \triangleq s [! \text{tst} [\text{open } p \mid \\ \quad (x_1, x_2) . \text{tstres} \{ x_1 . \langle \text{prime}(x_2) \rangle \} \mid \\ \quad (x_1, x_2, x_3) . \text{tstres} \{ x_1 . \langle \text{relative-prime}(x_2, x_3) \rangle \}]] \\ \\ \text{client} \triangleq \text{LET} \quad \text{exitPath} = \text{out } c . \text{in } s . \text{in } \text{tst} . \text{coopen } p \\ \quad \text{returnPath} = \text{out } \text{tst} . \text{out } s . \text{in } c \\ \text{IN} \quad c [\text{open } \text{tstres} . (v) . Q \mid p [\text{exitPath} . \langle \text{returnPath}, 1 + 2^{4096} \rangle]] \end{array}$$

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 .in s .in tst .coopen p ”, and *returnPath* is an abbreviation for the path “out tst .out s .in c ”.

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

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

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.

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	
$M \in \text{Exp}$	$::= n \mid c \mid \lambda n : \sigma.M \mid M_1 M_2 \mid \times(M_1, \dots, M_k) \mid \text{if } M_0 \text{ then } M_1 \text{ else } M_2 \mid \dots \mid \quad (k \geq 0)$ $\mid \epsilon \mid M_1.M_2 \mid \text{in } M \mid \text{out } M \mid \text{open } M \mid \text{coopen } M$
Processes	
$P \in \text{Proc}$	$::= \mathbf{0} \mid P_1 \mid P_2 \mid !P \mid (\nu n : \sigma).P \mid M.P \mid M[P] \mid (n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle M \rangle \quad (k \geq 0)$

Figure 1. Syntax of AC+.

de-sugar LET-IN using parallel processes with different input arities, preserving our stated goal of keeping programs completely inside ambients.¹¹ Notice that, in addition to arity polymorphism, *orderly communication* is needed to type the de-sugared term shown above. Both *exitPath* and the result of calling prime are communicated inside the ambient c , and these communications are of the same arity but with different types. \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. 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 .

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” (we have, e.g., that $P \mid \mathbf{0} \equiv P$ and $P \mid Q \equiv Q \mid P$). The definition 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 in one step reduces 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—for instance when communication takes place inside an ambient (Red Amb).

¹¹For this purpose, we require the body of a LET-IN to be of the form $n[Q]$ for some process Q .

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	$E ::= \square \mid EM \mid VE \mid \times (V_1, \dots, V_{i-1}, E, M_{i+1}, \dots, M_k) \mid \text{if } E \text{ then } M_1 \text{ else } M_2 \quad (k \geq 0)$ $\mid E.M \mid V.E \mid \text{in } E \mid \text{out } E \mid \text{open } E \mid \text{coopen } E$
Reduction Rules	<p>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 $\text{names}(V)$, similarly for (Red Comm).</p> $\begin{array}{ll} (\lambda n : \sigma.M)V \longrightarrow M[n := V] & \text{(Red Beta)} \\ cV \longrightarrow V' \quad \text{if } V' = \delta(c, V) & \text{(Red Delta)} \\ \text{if true then } M_1 \text{ else } M_2 \longrightarrow M_1 & \text{(Red IfTrue)} \\ \text{if false then } M_1 \text{ else } M_2 \longrightarrow M_2 & \text{(Red IfFalse)} \\ \text{If } M_1 \longrightarrow M_2 \text{ then } E[M_1] \longrightarrow E[M_2] & \text{(Red Context)} \end{array}$ $\begin{array}{ll} n[\text{in } m.P \mid Q] \mid m[R] \xrightarrow{\epsilon} m[n[P \mid Q] \mid R] & \text{(Red In)} \\ m[n[\text{out } m.P \mid Q] \mid R] \xrightarrow{\epsilon} n[P \mid Q] \mid m[R] & \text{(Red Out)} \\ \text{open } n.P \mid n[\text{coopen } n.Q \mid R] \xrightarrow{\epsilon} P \mid Q \mid R & \text{(Red Open)} \\ (n_1 : \sigma_1, \dots, n_k : \sigma_k).P \mid \langle \times (V_1, \dots, V_k) \rangle \xrightarrow{\text{comm}(\tau)} P[n_i := V_i] \text{ if } \tau = \times(\sigma_1, \dots, \sigma_k) & \text{(Red Comm)} \\ !P \xrightarrow{\epsilon} P \mid !P & \text{(Red Repl)} \end{array}$ $\begin{array}{ll} \text{If } M_1 \longrightarrow M_2 \text{ then } M_1[P] \xrightarrow{\epsilon} M_2[P] & \text{(Red Name)} \\ \text{If } M_1 \longrightarrow M_2 \text{ then } M_1.P \xrightarrow{\epsilon} M_2.P & \text{(Red Cap)} \\ \text{If } M_1 \longrightarrow M_2 \text{ then } \langle M_1 \rangle \xrightarrow{\epsilon} \langle M_2 \rangle & \text{(Red Put)} \\ \text{If } P \xrightarrow{\ell} Q \text{ then } (\nu n : \sigma).P \xrightarrow{\ell} (\nu n : \sigma).Q & \text{(Red Res)} \\ \text{If } P \xrightarrow{\ell} Q \text{ then } n[P] \xrightarrow{\epsilon} n[Q] & \text{(Red Amb)} \\ \text{If } P \xrightarrow{\ell} Q \text{ then } P \mid R \xrightarrow{\ell} Q \mid R & \text{(Red Par)} \\ \text{If } P' \equiv P, P \xrightarrow{\ell} Q, Q \equiv Q' \text{ then } P' \xrightarrow{\ell} Q' & \text{(Red } \equiv) \end{array}$
	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.

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. 3. 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 both can be viewed as upper estimates of the behavior of a process “unleashed” by opening n . For instance, a process such as $n[\langle 7 \rangle \mid (x : \text{int}).\text{coopen } n.\langle x = 42 \rangle]$ suggests that n should 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 to be explained in Sect. 3.4 it is convenient to allow for b_0 and b_1 to be distinct.

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

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

Shorthand: $\mathbf{1}$ stands for $\times()$. 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
	$\mid \text{get}(\sigma)$	input of type σ
	$\mid \text{diss}$	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 3. Syntax of Types and Behaviors.

A capability has a type of the form $\text{cap}[B]$ where B is a “behavior with a hole inside”. For instance, if n has type $\text{amb}[b]$ then open n has type $\text{cap}[b \mid \square]$, because a process which executes this capability will subsequently run in parallel with a process of behavior b .

The first six behavior constructs alone are sufficient for writing a type system satisfying a subject 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 choices for (approximating) constructs expressing recursion and choice, we have settled for a construct *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].

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.) Example: $\tau_0 = \text{int} \rightarrow \text{int}$ has (minimal) level zero, $b_1 = \text{put}(\text{cap}[\text{put}(\tau_0) \mid \square])$ has (minimal) level one, and $\tau_2 = \text{amb}[b_1, b_1]$ has (minimal) level two.

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). This relation induces a relation on behavior contexts: $B_1 \leq B_2$ holds iff for all level 0 behaviors b we have $B_1[b] \leq B_2[b]$. We shall see (Lemma 4.6) 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 .

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 . On base types, we have $\text{int} \leq \text{real}$. On composite types, the relation is defined using the following polarity rules:

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

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 an ambient in which only values of types τ_1 or τ_2 are allowed to be communicated, and assume that q is 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 $\text{open } n$ 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 processes enclosed by n and the second component expressing what may happen when n is opened.

3.5 The Type System

Figure 4 defines judgements $E \vdash M : \tau$ and $E \vdash P : b$, where E is an environment mapping names into types. We employ a function $\text{type}()$, assigning types to constants.

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 (Beh 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. 4.1; 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 $P = 0$), the process enclosed within m has behavior

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

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 $\text{get}(\text{bool})$ (in our framework, the sequence $\text{get}(\text{int}).\text{put}(\text{int}).\text{get}(\text{bool}).\text{put}(\text{bool})$ is not well-formed, since an input operation cannot be performed when nothing has been output yet).

- 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.

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 $b \rightsquigarrow \text{fromnow } T$.

Non-structural Rules

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

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

Expressions

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

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

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

$$\text{(Exp Abs)} \quad \frac{E, n : \sigma \vdash M : \tau}{E \vdash \lambda n : \sigma. M : \sigma \rightarrow \tau}$$

$$\text{(Exp App)} \quad \frac{E \vdash M_1 : \sigma \rightarrow \tau \quad E \vdash M_2 : \sigma}{E \vdash M_1 M_2 : \tau}$$

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

$$\text{(Exp } \epsilon\text{)} \quad \frac{}{E \vdash \epsilon : \text{cap}[\square]}$$

$$\text{(Exp In)} \quad \frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{in } M : \text{cap}[\square]}$$

$$\text{(Exp Out)} \quad \frac{E \vdash M : \text{amb}[b, b']}{E \vdash \text{out } M : \text{cap}[\square]}$$

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

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

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

Processes

$$\text{(Proc Zero)} \quad \frac{}{E \vdash \mathbf{0} : \varepsilon}$$

$$\text{(Proc Par)} \quad \frac{E \vdash P_1 : b_1 \quad E \vdash P_2 : b_2}{E \vdash P_1 \mid P_2 : b_1 \mid b_2}$$

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

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

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

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

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

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

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

Figure 4. Typing Rules.

4 Trace Semantics of Behaviors

To formally assign meaning to behaviors, we employ the notion 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

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

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.

We view a communication as the placement, and subsequent removal, of a Post-It note on a message board (cf. the metaphor in [Car99]) that has a section for each arity:

Definition 4.2 (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 4.3. In Sect. 3.5 we considered the behavior $b = \text{put}(\text{int}) \mid \text{get}(\text{int}).(\text{put}(\text{bool}) \mid \text{get}(\text{bool})) \mid \varepsilon$. It is easy to see that $\llbracket b \rrbracket$ contains 8 traces, but only one of these belongs to Comm^* :

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

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 $\tau \leq \sigma$ implies $\text{put}(\tau) \leq \text{put}(\sigma)$ and $\text{get}(\sigma) \leq \text{get}(\tau)$;
- the relation \leq on Act extends pointwise to a relation \leq on Trace ;
- $Tr_1 \leq Tr_2$ iff for all $tr_1 \in Tr_1$ there exists $tr_2 \in Tr_2$ such that $tr_1 \leq tr_2$.

Definition 4.4 (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 Sect. 3.4 shows how to define a relation on level 0 types and how a relation on level i behaviors (inducing a relation on level i behavior contexts) gives rise to a relation on level $i + 1$ types.

Lemma 4.5. *The operators “ \mid ” and “ \cdot ” on behaviors respect the relation \leq ; thus the equivalence relation \equiv induced by \leq is a congruence on behaviors wrt. these operators. Moreover, modulo \equiv it holds that “ \mid ” is associative and commutative and “ \cdot ” is associative, both with ε as neutral element. Also note that $\varepsilon \equiv \text{fromnow } \emptyset$. \square*

For type checking (Sect. 6) we have a convenient result:

Lemma 4.6. *Given B_1 and B_2 , we can construct a level zero behavior test (for example $\text{put}(\times(\tau_1, \dots, \tau_{m+1}))$) where each τ_i is int and where m is the maximal arity of a tuple occurring inside B_1 or B_2) 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) $B_1[\text{test.test}] \leq B_2[\text{test.test}]$. □

Definition 4.7 (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 $tr_1 \in \text{Comm} \setminus \text{WtComm}$. □

As an example of an unsafe behavior, consider $b = \text{put}(\text{int}) \mid \text{get}(\text{int}).\text{get}(\text{bool}) \mid \text{put}(\text{int})$ (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 the trace $\text{put}(\text{int}) \text{ get}(\text{int})$ is in Comm^* and the trace $\text{put}(\text{int}) \text{ get}(\text{bool})$ is in Comm but not in WtComm , yet their concatenation belongs to $\llbracket b \rrbracket$.

Definition 4.8 ($b \rightsquigarrow b_0$). The relation $b \rightsquigarrow b_0$ 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 $tr_0 \in \llbracket b_0 \rrbracket$ with $tr \leq tr_0$. □

5 Subject Reduction

Lemma 5.1 (Subject reduction for expressions). Suppose $M_1 \longrightarrow M_2$. If $E \vdash M_1 : \tau$ then also $E \vdash M_2 : \tau$. □

The proof follows standard techniques, together with assumptions about a suitable relationship between δ (evaluating constants) and $\text{type}()$ (typing constants).

Lemma 5.2 (Subject congruence). Suppose that $P \equiv Q$. Then $E \vdash P : b$ if and only if $E \vdash Q : b$. □

We do not know if Lemma 5.2 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 $!P \longrightarrow P \mid !P$ instead.

The formulation of subject reduction for processes will make it explicit that “well-typed processes communicate according to their behavior”. This property is expressed using a relation $\ell \sim b$, defined by stipulating that

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

Theorem 5.3 (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 $b_0.b' \leq b$. □

The proof is by induction on the derivation of $P \xrightarrow{\ell} Q$, making use of the previous lemmas (and some other auxiliary results). It is easy to see that if we can prove $b_0.b' \leq b$ then b' will be safe.

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 Figure 4. For this purpose we need to show: (i) that we can decide the side-conditions for all the rules in Figure 4, (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 (Beh Subsumption), (Exp Subsumption), (Proc Repl) and (Proc Amb) are also decidable.

For the purpose of checking that a given type derivation is valid, we employ “nested” automata. The nesting of automata corresponds to the notion of leveled types introduced in Section 3.2. For every $i \geq 0$ define $W_i = \{\text{diss}\} \cup \{\text{put}(\sigma) \mid \sigma \text{ is of level } i\} \cup \{\text{get}(\sigma) \mid \sigma \text{ is of level } i\}$. For a given derivation \mathcal{D} , clearly we need to consider only finitely many W_i 's and for each such W_i only a finite number of elements will be of interest.

A (non-deterministic) automaton 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, u, q_2) with $q_1, q_2 \in \mathcal{Q}$ and $u \in W_i \cup \{\epsilon\}$. Given a behavior b , we say that an automaton G implements b if $\llbracket b \rrbracket = \{tr \mid G \text{ accepts } tr\}$.

Lemma 6.1. *Given b of level i , we can construct G of level i implementing b .* \square

Proof. (Sketch) By induction on the structure of b . The cases in which b is ε , diss , $\text{put}(\sigma)$ or $\text{get}(\sigma)$ are immediate. If $b = \text{fromnow } T$ then construct an automaton where the only state is the initial state ι which is also an acceptance state; for each $\sigma \in T$ there are transitions $(\iota, \text{get}(\sigma), \iota)$ and $(\iota, \text{put}(\sigma), \iota)$ in δ . If $b = b_1.b_2$ or $b = b_1 \mid b_2$ we can, by the induction hypothesis, construct automata G_1 implementing b_1 and G_2 implementing b_2 . Let $G_i = (\mathcal{Q}_i, \iota_i, F_i, \delta_i)$ (for $i = 1, 2$); in both cases we can then clearly construct an automaton $G = (\mathcal{Q}, \iota, F, \delta)$ implementing b . If $b = b_1.b_2$ the key step is to let δ contain transitions $(q, \varepsilon, \iota_2)$ for all $q \in F_1$, and to let $\iota = \iota_1$ and $F = F_2$. If $b = b_1 \mid b_2$ we let G be the product automaton of G_1 and G_2 , in particular ι is given as the pair (ι_1, ι_2) and F is given as the set of pairs in $F_1 \times F_2$. \square

Lemma 6.2. *Assume that for 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. (Sketch) We need to show that we can decide if $\llbracket b_1 \rrbracket \leq \llbracket b_2 \rrbracket$, which amounts to deciding whether for every $tr_1 \in \llbracket b_1 \rrbracket$ there exists $tr_2 \in \llbracket b_2 \rrbracket$ such that $tr_1 \leq tr_2$, which again amounts to deciding whether the language $\llbracket b_1 \rrbracket \setminus \llbracket b_2 \rrbracket$ is empty where we define $\llbracket b_1 \rrbracket \setminus \llbracket b_2 \rrbracket = \{tr_1 \in \llbracket b_1 \rrbracket \mid \nexists tr_2 \in \llbracket b_2 \rrbracket \text{ for which } tr_1 \leq tr_2\}$. The language $\llbracket b_1 \rrbracket \setminus \llbracket b_2 \rrbracket$ is clearly decidable, since in the absence of subtyping, it is equivalent to $\llbracket b_1 \rrbracket - \llbracket b_2 \rrbracket = \overline{\llbracket b_2 \rrbracket} \cap \llbracket b_1 \rrbracket$, a language that can be accepted by an automaton constructed from those for b_1 and b_2 (cf. Lemma 6.1). The inclusion of subtyping over the elements of $\llbracket b_1 \rrbracket$ and $\llbracket b_2 \rrbracket$ does not pose substantial additional complications since, by assumption, we can decide if two types of level i are in subtype relation. We can thus construct an automaton¹² accepting the language $\llbracket b_1 \rrbracket \setminus \llbracket b_2 \rrbracket$, and we have seen that $\llbracket b_1 \rrbracket \leq \llbracket b_2 \rrbracket$ iff the language accepted by this automaton is empty—a property that is trivially decidable. \square

Lemma 6.3. *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. (Sketch) By induction on the structure of τ and σ . Whenever $\tau = \text{cap}[B]$ and $\sigma = \text{cap}[B']$, we use Lemma 4.6 to test whether $B \leq B'$. \square

The following Theorem is a consequence of Lemmas 6.2 and 6.3.

Theorem 6.4. *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 6.5. *Given a behavior b , it is decidable whether $b \mid b \leq b$.* \square

Lemma 6.6. *Given a behavior b , it is decidable whether b is safe.* \square

Proof. (Sketch) By Lemma 6.1 we can construct an automaton G that accepts the language $\llbracket b \rrbracket$. Clearly, by an inspection of the transition relation of G we can determine if a string (trace) accepted by G contains a prefix in the set Comm^* where a suffix is in $\text{Comm} \setminus \text{WtComm}$ (we use Theorem 6.4 to test membership of WtComm). The behavior b is safe iff no such trace is found. \square

Lemma 6.7. *Given b_1 and b_2 , it is decidable whether $b_1 \rightsquigarrow b_2$.* \square

Proof. (Sketch) Construct G_1 and G_2 implementing b_1 and b_2 . Construct G_0 such that G_0 accepts tr iff there exists $tr_1 \in \text{Comm}^*$ such that G_1 accepts $tr_1 \diamond \text{diss} \diamond tr$. Then, as in the proof of Lemma 6.2, construct the automaton $G = G_0 \setminus G_2$. Finally, by definition of the \rightsquigarrow relation, we have $b_1 \rightsquigarrow b_2$ iff the language accepted by G is empty. \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 Figure 4.

Theorem 6.8 (Decidability of type checking). *Given a purported derivation of $E \vdash M : \tau$ or $E \vdash P : b$, we can check its validity.* \square

Proof. As discussed earlier, it suffices to show that the side-conditions for all the rules in Figure 4 are decidable. This follows from Theorem 6.4 (and the following Corollary), together with Lemmas 6.6 and 6.7. \square

¹²With $G_i = (\mathcal{Q}_i, \iota_i, F_i, \delta_i)$ implementing b_i for $i = 1, 2$, we define $G = (\mathcal{Q}, \iota, F, \delta)$ implementing $\llbracket b_1 \rrbracket \setminus \llbracket b_2 \rrbracket$ as follows: $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{P}(\mathcal{Q}_2)$, $\iota = (\iota_1, \{\iota_2\})$, $F = \{(q_1, Q_2) \mid q_1 \in F_1 \text{ and } Q_2 \cap F_2 = \emptyset\}$, and $((q_1, Q_2), u, (q'_1, Q'_2))$ belongs to δ iff $(q_1, u, q'_1) \in \delta_1$ and $Q'_2 = \{q'_2 \in \mathcal{Q}_2 \mid \exists u^+ \geq u \text{ and } q_2 \in \mathcal{Q}_2 : (q_2, u^+, q'_2) \in \delta_2\}$.

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 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

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

By induction in the derivation we can now prove

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 $Plus(E^C) \vdash Plus(P^C) : Plus(T^C)$, respectively $Plus(E^C) \vdash Plus(M^C) : Plus(W^C)$, is derivable in our system. \square*

It is also easy to show that if $P_1^C \longrightarrow P_2^C$ holds in a system that is as in [CG99], except that the rule $!P \equiv P \mid !P$ has been replaced by $!P \longrightarrow P \mid !P$, then $Plus(P_1^C) \longrightarrow Plus(P_2^C)$ holds in our system.

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., $\llbracket \forall O \rightsquigarrow I \rrbracket$ 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** nor **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 $\llbracket * \rrbracket = \emptyset$. Then for b not containing **diss** we would have $b \rightsquigarrow *$, enabling us to assign the type $\text{amb}[*, *]$ to an ambient which cannot be opened. For subject reduction to carry through, however, we must ensure (by suitable side conditions) that if $E \vdash P : b$ then $\llbracket b \rrbracket \neq \emptyset$. We would thus not be able to type a process that attempts to open a locked ambient.

Besides the several tasks enumerated in Sect. 1, 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 $\llbracket b \rrbracket$ 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 [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 \mathbf{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 `fromnow` T covers both recursion and approximation — note that `fromnow` $\{\tau_1, \dots, \tau_n\}$ can be expressed as $\text{rec } \beta.((\text{get}(\tau_1) + \text{put}(\tau_1) + \dots + \text{get}(\tau_n) + \text{put}(t_n)).\beta + \varepsilon)$.

However, the conceptual differences between Concurrent ML and \mathbf{AC}^+ show up in several places. In NNA there are types τ `event` b and τ `chan`, and an atomic behavior τ `CHAN` recording the creation of a channel carrying values of type τ , whereas there are no counterparts to our constructs `amb` $[b, b']$, `cap` $[B]$, or `diss`. 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. We believe the latter to be in general the right choice: Without a set-theoretic semantic interpretation, choosing the “right” set of axioms is a somewhat ad-hoc exercise. An added advantage of the semantic approach is that it has considerably facilitated type checking, as illustrated by the analysis in Sect. 6.

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 `amb` $[b, b]$ can be considered “latent”, in which case the rule for `n` $[P]$ (`Proc Amb`) is viewed as an introduction rule and the rule for `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 \mathbf{AC}^+ shares the conceptual framework of [CG99] (and the systems spawned by this paper) rather than that of type-and-effect systems.

7.3 Session Types versus Orderly Communication

Session types in the π -calculus and *orderly communication* in \mathbf{AC}^+ are motivated by similar considerations. Session types originated with the work of Honda and his collaborators, who 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 a system of simple types for the 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 by means of their type system.

Despite the many similarities, there are also many differences between sessions types in the π -calculus and orderly communication in \mathbf{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 \mathbf{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 in \mathbf{AC}^+ , then P is typable in the system with session types if and only if Q is typable in our type system for \mathbf{AC}^+ with orderly communication.

Further discussion of session types and possible connections with orderly communication are included in the full technical report [AKPG00].

References

- [AKPG00] T. Amtoft, A. J. Kfoury, and S. Pericas-Geersten. What are polymorphically-typed ambients? Technical report, Comp. Sci. Dept., Boston Univ., Oct. 2000.

- [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.
- [ANN99] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
- [BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *CONCUR 2000*, vol. 1877 of *LNCS*, pp. 504–520, 2000.
- [Car99] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, eds., *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, *LNCS*, pp. 51–94. Springer-Verlag, 1999.
- [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.
- [CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL'99, San Antonio, Texas*, pp. 79–92. ACM Press, Jan. 1999.
- [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.
- [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*, Aug. 2000.
- [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.
- [Gay93] S. J. Gay. A sort inference algorithm for the polyadic pi-calculus. In *POPL 1993*, pp. 429–438, 1993.
- [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.
- [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.
- [LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*, pp. 352–364. ACM Press, Jan. 2000.
- [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.
- [NN00] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL'00, Boston, Massachusetts*, pp. 142–154. ACM Press, 2000.
- [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.
- [PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *8th annual IEEE Symposium on Logic in Computer Science (LICS '93)*. IEEE Computer Society Press, 1993. A revised and extended version has appeared in journal of *MCS* 6(5), 409–454, 1996.
- [PS00] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, May 2000.
- [PT97] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report, IU, 1997.
- [RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *POPL 1998*, pp. 378–390. ACM Press, 1998.
- [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.
- [SV99] P. Sewell and J. Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop (CSFW-12)*, *Mordano, Italy*, June 1999.
- [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.
- [Tul00] M. Tullsen. The zip calculus. In *Fifth International Conference on Mathematics of Program Construction (MPC 2000)*. Springer-Verlag, July 2000.

- [Tur95] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. Ph.D. thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.
- [VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of *LNCS*. Springer-Verlag, 1999.
- [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.
- [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.
- [YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In *LICS 2000*, pp. 334–345, 2000.
- [Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *FOSSACS 2000, Berlin*, vol. 1784 of *LNCS*, pp. 375–390. Springer-Verlag, 2000.

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 `amb[put(int, int)]`, and q the type `amb[put(real, 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 `get(real, real).ε | put(int, int) | put(real, 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 body of ambient n has behavior `put(bool, int) | put(int, int, real) | get(bool, int).b | get(int, int, real).b'` which is safe under suitable assumptions on b and b' .

Case 4 By assigning n the type `amb[get(bool).b]` (with b the behavior of P), which is possible since the body of ambient n has behavior $b_n = \text{get(bool).b} \mid \text{diss}$ where b_n is safe and $b_n \rightsquigarrow \text{get(bool).b}$, 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 `put(int) | get(int).(put(bool) | get(bool).b) | ε` which is safe (cf. Sect 3.5) under suitable assumptions on b .

Example 2.1 Let $b = \text{get(string).(get(cap[□]).ε | put(cap[□]))}$. By assigning the behavior `get(cap[□]).ε | diss` to the body of *hop* (which can then be given the type `amb[get(cap[□]).ε]`), by assigning the (safe) behavior $b \mid \text{diss}$ to the body of *route* (which can then be given the type `amb[b]`), and by assigning $b \mid \text{put(string)}$ (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 `cap[diss.□]`, z_2 and x_1 the type `cap[□]`, *tstres* the type `amb[put(bool)]`, and by assigning the behavior `diss.put(cap[□], int)` to the body of p (which can then be given type `amb[put(cap[□], int)]`), we can construct a type derivation for

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

$$\text{client} \triangleq c[\langle \text{out } c.\text{in } s.\text{in } tst.\text{coopen } p \rangle \mid \\ (z_1).\langle z_1, \text{out } tst.\text{out } s.\text{in } c \rangle \mid \\ (z_1, z_2).(\text{open } tstres.(v).Q \mid p[z_1.\langle z_2, 1 + 2^{4096} \rangle])]$$

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*)

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

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