What are Polymorphically-Typed Ambients?

Torben Amtoft[⋆], Assaf J. Kfoury[⋆⋆], and Santiago M. Pericas-Geertsen[⋆⋆⋆]

Boston University
{tamtoft,kfoury,santiago}@cs.bu.edu

Abstract. The Ambient Calculus was developed by Cardelli and Gordon as a formal framework to study issues of mobility and migrant code [6]. We consider an Ambient Calculus where ambients transport and exchange programs rather that 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 [7].

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 [8], the Dπ Calculus [16], the Box-Pi Calculus [17], the Seal Calculus [20], among others. 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 paper.

Early type systems for **AC** (see [7, 5] among others) restrict ambients to be *monomorphic*: There can be only one "topic of conversation" (the type of

[⋆] http://www.cs.bu.edu/associates/tamtoft

[⋆⋆] http://www.cs.bu.edu/~kfoury

[⋆⋆⋆] http://cs-people.bu.edu/santiago

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 [23] and [3] 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.

Case 1. Consider a process of the form:

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

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 "$\mathsf{in}\ r$") and, once inside r, both can be dissolved (expressed by the capabilities "$\mathsf{open}\ p$" and "$\mathsf{open}\ q$") in order to unleash their outputs. The type of the input pair (f, x) inside r can be $(\mathsf{int} \to \mathsf{bool}, \mathsf{int})$ or $(\mathsf{bool} \to \mathsf{bool}, \mathsf{bool})$, depending on whether output $\langle \mathsf{even}, 3\rangle$ or output $\langle \mathsf{not}, \mathsf{true}\rangle$ 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:

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

where the operation $\mathsf{mult} : (\mathsf{real}, \mathsf{real}) \to \mathsf{real}$ multiplies two real numbers. Because the type of $\langle 3, 2\rangle$ is $(\mathsf{int}, \mathsf{int})$, which is a subtype of $(\mathsf{real}, \mathsf{real})$, 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, and also incorporated in type systems for concurrent calculi, such as [14, 15] for the π-calculus and [23] for **AC**.

Case 3. Consider now the following process:

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

The outputs are transmitted depending on their arities, here 2 for the output $\langle \mathsf{true}, 5\rangle$ 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, $(\mathsf{bool}, \mathsf{int})$ pairs and $(\mathsf{int}, \mathsf{int}, \mathsf{real})$ 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 $\langle \mathsf{true}, 5\rangle$ and $\langle 5, 6, 3.6\rangle$ have types $(\mathsf{bool}, \mathsf{int})$ and $(\mathsf{int}, \mathsf{int}, \mathsf{real})$, respectively. The polymorphism of n is appropriately called *arity polymorphism*[1].

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:

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

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*[2], and which raises entirely new problems not encountered before in the context of **AC**. The design of a type discipline enforcing it is a delicate matter, and the main focus of this paper.

Orderly communication bears a strong resemblance to what has been called "session types" in the π-calculus [9], originated with the work of Honda and his collaborators [18, 10] whose approach is based on syntax, and more recently developed by Gay and Hole [9] where the session discipline is enforced by a type system for the π-calculus (also integrating subtyping and recursive types).

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 Scope and Contribution of Our Research

The core of our formal calculus is **AC**, augmented with a simply-typed functional language at the level of exchanged data; accordingly we call our calculus **AC+**. 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),

[1] The term "arity polymorphism" was used already by others, e.g. Moggi [12], to describe similar—though different in some respects—situations in functional programming languages.

[2] We thank Benjamin Pierce for suggesting the apt term "orderly communication".

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 [21, 22] for example).

In summary, our main accomplishments 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 [19], we expect its incorporation into our type system to proceed smoothly.

- 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, which is further used to prove that our polymorphically-typed **AC+** satisfies a Subject Reduction property.
- Behavior subsumption and type subsumption are shown to be decidable relations, and this implies the decidability (at least exponential in the worse case) of *type-checking* for type-annotated **AC+** terms.

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. The more difficult problem of *type-inference* for (un-annotated) **AC+** terms 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 [7], in the sense that every process typable in the latter is typable in ours.

Further material and all missing proofs are included in the technical report [1], on which the current paper is based (this report can be downloaded from the Church Project web site at http://types.bu.edu/reports/).

1.3 Motivating Example

We now give an example, short but more interesting than the snippets in Sect. 1.1, to illustrate the expressive power and convenience of a polymorphically typed **AC+**, in particular the use of *orderly communication.* Aside from the embedded programs, the syntax of ambients is identical to that first proposed by Cardelli and Gordon [6] with the addition of a co-capability "$\mathsf{coopen}\ n$" akin to a proposal already made by Levi and Sangiorgi [11]. For a process to open an ambient n, this ambient must contain a top-level process willing to exercise a $\mathsf{coopen}\ n$ (cf. (Red Open) in Fig. 2). We shall use $n\{P\}$ to abbreviate $n[P \mid \mathsf{coopen}\ n]$.

Expressions

$$\begin{aligned} M \in \mathsf{Exp} ::= {} & n \mid c \mid \lambda n : \sigma . M \mid M_1 M_2 \mid \times(M_1, \ldots , M_k) \mid \mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \\ & \mid \epsilon \mid M_1 . M_2 \mid \mathsf{in}\ M \mid \mathsf{out}\ M \mid \mathsf{open}\ M \mid \mathsf{coopen}\ M \qquad (k \geqslant 0) \end{aligned}$$

Processes

$$\begin{aligned} P \in \mathsf{Proc} ::= {} & \mathbf{0} \mid P_1 \mathbin{\boldsymbol{|}} P_2 \mid !P \mid (\nu n : \sigma).P \mid M.P \mid M\mathbf{[}P\mathbf{]} \mid (n_1 : \sigma_1, \ldots , n_k : \sigma_k).P \\ & \mid \langle M \rangle \qquad (k \geqslant 0) \end{aligned}$$

When there is no ambiguity we write M for $M.0$.

Fig. 1. Syntax of **AC+**.

Example 1 (Packet Routing). 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), and next capabilities (the path).

$$\begin{gathered} \mathit{router}\mathbf{[}!\mathit{route}\mathbf{\{}\mathsf{in}\ \mathit{packet}.(\mathit{dst}).\mathsf{open}\ \mathit{hop}.\langle \mathsf{lookup\text{-}route}(\mathit{dst})\rangle \mathbf{\}}\mathbf{]} \mathbin{\boldsymbol{|}} \\ \mathit{packet}\mathbf{[}\mathsf{in}\ \mathit{router}.\mathsf{open}\ \mathit{route}.\langle \text{“bu”} \rangle \mathbin{\boldsymbol{|}} \mathit{hop}\mathbf{\{}(x).x\mathbf{\}}\mathbf{]} \end{gathered}$$

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. In these systems, 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 $(x).n\mathbf{[}x\mathbf{]}$ for some ambient name n. □

2 Types and Behaviors

Figure 1 depicts the syntax of our language **AC+**. A process $P \in \mathsf{Proc}$ is basically as in [7]: there are constructs for parallel composition ($P_1 \mathbin{\boldsymbol{|}} P_2$), replication ($!P$), restriction ($(\nu n : \sigma).P$); and there also are constructs for input and output. Note that communication is asynchronous, in that an outputting process has no "continuation"; a communication can thus (cf. the metaphor in [4]) be viewed as the placement, and subsequent removal, of a Post-It note on a message board that (unlike in [4]) has a section for each arity.

An expression $M \in \mathsf{Exp}$ denotes a computation over a domain that includes not only simple values (like integers) but also functions, tuples, ambient names, and (paths of) capabilities. Note that for all binding constructs in **AC+**, the name n being bound is annotated with a type σ (to be defined in Sect. 2.2).

2.1 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 $M_1 \longrightarrow M_2$.

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)) 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 [7], except that we omit the rule $!P \equiv P \mid !P$ (in the presence of which we do not know whether it will be possible to establish Lemma 2) 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 $\ell = \mathrm{comm}(\tau)$ 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.

2.2 Types and Behaviors

The syntax of types ($\tau, \sigma \in \mathsf{Typ}$) and the syntax of behaviors ($b \in \mathsf{Beh}$) are recursively defined in Fig. 3. The first five behavior constructs capture the intuition (cf. [2]) that we want to keep track of the relationship (sequential or parallel) between occurrences of input and output operations.

An ambient n has a type of the form $\mathsf{amb}[b_0, b_1]$, 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 $n[\langle 7\rangle \mid (x : \mathsf{int}).\mathsf{coopen}\ n.\langle x = 42\rangle]$ we expect n to have the type $\mathsf{amb}[\mathsf{put}(\mathsf{bool}), \mathsf{put}(\mathsf{bool})]$, 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 $\mathsf{coopen}\ n$, 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 write $\mathsf{amb}[b_0]$ for $\mathsf{amb}[b_0, b_0]$; but as in [23] the possibility of them being distinct facilitates a smooth integration of subtyping.

A capability has a type of the form $\mathsf{cap}[B]$ where B is a behavior context, that is a "behavior with a hole inside". To motivate this, consider a process $P = \mathsf{open}\ n.P'$ where P' has behavior b' and n has type $\mathsf{amb}[b]$. 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 $(b \mid \Box)\lfloor b' \rfloor$. This is why it makes sense to assign $\mathsf{open}\ n$ the capability type $\mathsf{cap}[b \mid \Box]$, cf. the rules (Exp Open) and (Proc Action) in Fig 4.

The first six behavior constructs in Fig. 3 alone, are sufficient to write a type system satisfying a subject reduction property (Sect. 4), but they do not enable

Values $V ::= \cdots$ (omitted, as standard)

Evaluation Contexts and **Process Evaluation Contexts**

$$\begin{aligned}
\mathcal{E} ::=\ & \Box_e \mid \mathcal{E}M \mid V\mathcal{E} \mid \times(V_1,..,V_{i-1},\mathcal{E},M_{i+1},..,M_k) \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \\
& \mid \mathcal{E}.M \mid V.\mathcal{E} \mid \mathsf{in}\ \mathcal{E} \mid \mathsf{out}\ \mathcal{E} \mid \mathsf{open}\ \mathcal{E} \mid \mathsf{coopen}\ \mathcal{E} \qquad (k \geqslant 0) \\
\mathcal{P} ::=\ & \Box_p \mid \mathcal{E}.P \mid \mathcal{E}[P] \mid \langle\mathcal{E}\rangle \mid (\nu n:\sigma).\mathcal{P} \mid \mathcal{P} \mid P
\end{aligned}$$

$\mathcal{E}\lfloor M \rfloor$ is the expression resulting from replacing $\Box_e$ with M in E.
$\mathcal{P}\lfloor M \rfloor_e$ is the process resulting from replacing $\Box_e$ with M in $\mathcal{P}$.
$\mathcal{P}\lfloor P \rfloor_p$ is the process resulting from replacing $\Box_p$ with P in $\mathcal{P}$.

Reduction Rules

Let ℓ be a label in $\{\epsilon\} \cup \{\text{comm}(\tau) \mid \tau \in \mathsf{Typ}\}$.
Let $\delta(c,V)$ be a partial function defined for every constant c.
In (Red Beta) and (Red Comm), we demand that there is no name capture.

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

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

Fig. 2. Operational semantics.

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 $\mathsf{cap}[\mathsf{put}(\mathsf{int}) \mid \Box]$ whereas the other branch is a capability of type $\mathsf{cap}[\mathsf{get}(\mathsf{int}) \mid \Box]$. Among many possible options for (approximating) constructs expressing recursion and choice, we in this paper settle for a simple one: the construct $\mathsf{fromnow}\ T$ with T the "topics of conversation", which can be thought of as the "union" of all behaviors composed of $\mathsf{put}(\tau)$ and $\mathsf{get}(\tau)$ with $\tau \in T$.

Types

$$\begin{array}{rcll}
\sigma, \tau \in \mathsf{Typ} & ::= & \mathsf{bool} \mid \mathsf{int} \mid \mathsf{real} \mid \mathsf{string} \mid \cdots & \text{type constant} \\
 & & \mid \sigma \rightarrow \tau & \text{function type} \\
 & & \mid \times(\sigma_1, \ldots, \sigma_k) & \text{tuple with arity } k \geqslant 0 \\
 & & \mid \mathsf{amb}[b, b'] & \text{type of ambient name} \\
 & & \mid \mathsf{cap}[B] & \text{type of capability}
\end{array}$$

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

When there is no ambiguity, we write σ for $\times(\sigma)$ and $(\sigma_1, \ldots, \sigma_k)$ for $\times(\sigma_1, \ldots, \sigma_k)$.

Behaviors

$$\begin{array}{rcll}
b \in \mathsf{Beh} & ::= & \varepsilon & \text{no traceable action} \\
 & & \mid b_1.b_2 & \text{first } b_1 \text{ then } b_2 \\
 & & \mid b_1 \boldsymbol{|} b_2 & \text{parallel composition} \\
 & & \mid \mathsf{put}(\sigma) & \text{output of type } \sigma \text{ (a tuple)} \\
 & & \mid \mathsf{get}(\sigma) & \text{input of type } \sigma \text{ (a tuple)} \\
 & & \mid \mathsf{diss} & \text{ambient dissolution} \\
 & & \mid \mathsf{fromnow}\ T & \text{unordered communication} \\
 & & & \text{of values with types in } T \\
B \in \mathsf{BehCont} & ::= & \Box \mid b.B \mid B.b \mid b \boldsymbol{|} B \mid B \boldsymbol{|} b & \text{behavior context}
\end{array}$$

Notation: $B\lfloor b \rfloor$ is the behavior resulting from replacing $\Box$ with b in B; similarly for the behavior context $B\lfloor B_1 \rfloor$.

Fig. 3. Syntax of types and behaviors.

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 $\mathsf{amb}[\_, \_]$ or $\mathsf{cap}[\_]$ within τ, similarly for T, b, and B. Example: $\tau_0 = \mathsf{int} \rightarrow \mathsf{int}$ has level zero, $b_1 = \mathsf{put}(\mathsf{cap}[\mathsf{put}(\tau_0) \boldsymbol{|} \Box])$ has level one, and $\tau_2 = \mathsf{amb}[b_1, b_1]$ has level two (as well as any higher level).

2.3 Behavior Subsumption

We employ a relation $b_1 \leqslant b_2$, to be formally defined in Sect. 3, with the intuitive interpretation that b_2 is more "permissive" than b_1. For example, $\mathsf{put}(\mathsf{int}) \leqslant \mathsf{fromnow}\ \{\mathsf{int}, (\mathsf{int}, \mathsf{int})\}$, and if integers can be converted into real numbers then also $\mathsf{put}(\mathsf{int}) \leqslant \mathsf{put}(\mathsf{real})$, since a process that sends an integer thereby also sends a real number, and $\mathsf{get}(\mathsf{real}) \leqslant \mathsf{get}(\mathsf{int})$, 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 1. *$B_1 \leqslant B_2$ holds iff for all level 0 behaviors b: $B_1\lfloor b \rfloor \leqslant B_2\lfloor b \rfloor$.*

2.4 Subtyping

We employ a relation $\tau_1 \leqslant \tau_2$, such that a value of type τ_1 also has type τ_2. On base types, we have $\mathsf{int} \leqslant \mathsf{real}$. On composite types, the relation is defined using the following polarity rules (tuples with different arity are incompatible):

$$\ominus \to \oplus \qquad (\oplus, \ldots, \oplus) \qquad \mathsf{amb}[\ominus, \oplus] \qquad \mathsf{cap}[\oplus]$$

2.5 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. The function $\mathsf{type}()$ assigns types to constants.

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

The side conditions for ($\mathsf{Proc\ Amb}$) employ a couple of notions which will be formally defined in Sect. 3; below we shall convey the intuition by providing a few examples. First we address the notion of being *safe*.

- The behavior $\mathsf{put}(\mathsf{int}) \mid \mathsf{get}(\mathsf{bool})$ 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
$$b = \mathsf{put}(\mathsf{int}) \mid \mathsf{get}(\mathsf{int}).(\mathsf{get}(\mathsf{bool}) \mid \mathsf{put}(\mathsf{bool})) \qquad (1)$$
which *is* safe, since no matter how the parallel behaviors are interleaved in a "well-formed" way then *(i)* $\mathsf{put}(\mathsf{bool})$ cannot precede $\mathsf{get}(\mathsf{int})$; and *(ii)* $\mathsf{put}(\mathsf{int})$ cannot immediately precede $\mathsf{get}(\mathsf{bool})$.
- Perhaps surprisingly, the behavior $\mathsf{diss}.(\mathsf{put}(\mathsf{int}) \mid \mathsf{get}(\mathsf{bool}))$ 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. For example, with $b = \mathsf{get}(\mathsf{int}).\mathsf{diss} \mid \mathsf{put}(\mathsf{int})$ we have $b \rightsquigarrow \varepsilon$ (since we can infer that $\mathsf{put}(\mathsf{int})$ is performed before diss). And with $b = \mathsf{fromnow}\ T \mid \mathsf{diss}$, we have $b \rightsquigarrow \mathsf{fromnow}\ T$.

Example 2. With $b = \mathsf{get}(\mathsf{string}).(\mathsf{get}(\mathsf{cap}[\square]).\varepsilon \mid \mathsf{put}(\mathsf{cap}[\square]))$, we can construct a typing for Example 1 as follows: assign the behavior $\mathsf{get}(\mathsf{cap}[\square]).\varepsilon \mid \mathsf{diss}$ to the body of *hop* (which can then be given the type $\mathsf{amb}[\mathsf{get}(\mathsf{cap}[\square]).\varepsilon]$), assign the (safe) behavior $b \mid \mathsf{diss}$ to the body of *route* (which can then be given the type $\mathsf{amb}[b]$), and assign $b \mid \mathsf{put}(\mathsf{string})$ (which is clearly safe) to the body of *packet*. □

Non-structural Rules

(Beh Subsumption)

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

(Exp Subsumption)

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

Expressions (selected rules only)

(Exp App)

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

(Exp Action)

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

(Exp ϵ)

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

(Exp In)

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

(Exp Out)

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

(Exp Open)

$$\frac{E \vdash M : \mathsf{amb}[b, b']}{E \vdash \mathsf{open}\ M : \mathsf{cap}[b' \mathbin{\boldsymbol{|}} \square]}$$

(Exp Coopen)

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

Processes

(Proc Zero)

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

(Proc Par)

$$\frac{E \vdash P_1 : b_1 \qquad E \vdash P_2 : b_2}{E \vdash P_1 \mathbin{\boldsymbol{|}} P_2 : b_1 \mathbin{\boldsymbol{|}} b_2}$$

(Proc Repl)

$$\frac{E \vdash P : b}{E \vdash !P : b} \quad (\text{if } (b \mathbin{\boldsymbol{|}} b) \leqslant b)$$

(Proc Action)

$$\frac{E \vdash M : \mathsf{cap}[B] \qquad E \vdash P : b}{E \vdash M.P : B \lfloor b \rfloor}$$

(Proc Res)

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

(Proc Amb)

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

(Proc Input)

$$\frac{E, n_1 : \tau_1, \cdots, n_k : \tau_k \vdash P : b}{E \vdash (n_1 : \tau_1, \ldots, n_k : \tau_k).P : \mathsf{get}(\tau_1, \cdots, \tau_k).b}$$

(Proc Output)

$$\frac{E \vdash M : \times(\tau_1, \ldots, \tau_k)}{E \vdash \langle M \rangle : \mathsf{put}(\tau_1, \cdots, \tau_k)}$$

Fig. 4. Typing rules.

3 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" 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.

Definition 2 (Traces). *A trace $tr \in \mathsf{Trace}$ is a finite sequence of actions, where an action $a \in \mathsf{Act}$ is a behavior that is either $\mathsf{put}(\tau)$, $\mathsf{get}(\tau)$, or diss.*

The semantics $[\![b]\!]$ of a behavior b belongs to the powerset $\mathcal{P}(\mathsf{Trace})$:

$$\begin{aligned} [\![\varepsilon]\!] &= \{\bullet\} & [\![\mathsf{diss}]\!] &= \{\mathsf{diss}\} \\ [\![b_1.b_2]\!] &= [\![b_1]\!] \diamond [\![b_2]\!] & [\![b_1 \mid b_2]\!] &= [\![b_1]\!] \parallel [\![b_2]\!] \\ [\![\mathsf{put}(\tau)]\!] &= \{\mathsf{put}(\tau)\} & [\![\mathsf{get}(\tau)]\!] &= \{\mathsf{get}(\tau)\} \end{aligned}$$

$$[\![\mathsf{fromnow}\ T]\!] = \{tr \mid \forall a \text{ occurring in } tr : \exists \tau \in T : a \in \{\mathsf{put}(\tau), \mathsf{get}(\tau)\}\}$$

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.

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 the type of the value being input. This motivates the following definition:

Definition 3 (Comm). *A trace tr belongs to Comm if $tr = \mathsf{put}(\tau)\,\mathsf{get}(\sigma)$ with $\mathsf{arity}(\tau) = \mathsf{arity}(\sigma)$. If in addition it holds that $\tau \leqslant \sigma$ we say that $tr \in \mathsf{WtComm}$, the set of* well-typed communications.

Example 3. With b as in (1), it is easy to see that $[\![b]\!]$ is given by the 8 traces

$$\begin{array}{ll} \mathsf{put(int)\ get(int)\ put(bool)\ get(bool)} & \mathsf{put(int)\ get(int)\ get(bool)\ put(bool)} \\ \mathsf{get(int)\ put(int)\ put(bool)\ get(bool)} & \mathsf{get(int)\ put(int)\ get(bool)\ put(bool)} \\ \mathsf{get(int)\ put(bool)\ put(int)\ get(bool)} & \mathsf{get(int)\ get(bool)\ put(int)\ put(bool)} \\ \mathsf{get(int)\ put(bool)\ get(bool)\ put(int)} & \mathsf{get(int)\ get(bool)\ put(bool)\ put(int)} \end{array}$$

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. □

Definition 4 (Behavior subsumption). *$b_1 \leqslant b_2$ iff $[\![b_1]\!] \leqslant [\![b_2]\!]$, where the relations $\leqslant$ on Act, Trace, and $\mathcal{P}(\mathsf{Trace})$ are given by:*

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

Our definition of the relations $b_1 \leqslant b_2$ and $\tau \leqslant \sigma$ 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. 2.4 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. 1, it induces a relation on level i behavior contexts).

The operators "$\mid$" and "." on behaviors respect the relation $\leqslant$; thus the equivalence relation $\equiv$ induced by $\leqslant$ is a congruence on behaviors wrt. these operators. Modulo $\equiv$ it holds that "$\mid$" is associative and commutative and that "." is associative, both with ε as neutral element. Note that $\varepsilon \equiv \mathsf{fromnow}\ \emptyset$.

The result below plays an important part in type checking:

Lemma 1. *Given B_1 and B_2 behavior contexts, we can construct a level zero behavior* test *such that the following conditions are equivalent:*

(a) $B_1 \leqslant B_2$
(b) $B_1\lfloor b \rfloor \leqslant B_2\lfloor b \rfloor$ *for all b (regardless of level)*
(c) $B_1\lfloor \mathsf{test.test} \rfloor \leqslant B_2\lfloor \mathsf{test.test} \rfloor$.

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 5 (Safety). *A behavior b is safe if no trace $tr \in [\![b]\!]$ can be written $tr = tr_0 \diamond tr_1 \diamond tr_2$ with $tr_0 \in \mathsf{Comm}^*$ and $tr_1 \in \mathsf{Comm} \setminus \mathsf{WtComm}$.*

Example 4. Referring back to Example 3, where the traces of a behavior b were listed, we can now demonstrate that b is in fact safe (as claimed in Sec. 2.5). For the first trace in b belongs to WtComm^*; the second trace can be written as $tr_0 \diamond tr$ with $tr_0 \in \mathsf{WtComm}$ and tr not of the form $tr_1 \diamond tr_2$ for any $tr_1 \in \mathsf{Comm}$; and none of the remaining traces are of the form $tr_1 \diamond tr_2$ with $tr_1 \in \mathsf{Comm}$. □

Definition 6 ($b \rightsquigarrow b'$). *The relation $b \rightsquigarrow b'$ amounts to the following property: whenever there exists $tr_1 \in \mathsf{Comm}^*$ and tr such that $tr_1 \diamond \mathsf{diss} \diamond tr \in [\![b]\!]$, then there exists $tr' \in [\![b']\!]$ with $tr \leqslant tr'$.*

4 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 1), 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" shows that the process $((n : \mathsf{int}).\langle n + 7\rangle) \mid \langle \mathsf{true} \rangle$, though typeable, cannot be assigned a safe behavior since it evolves into the process $\langle \mathsf{true} + 7\rangle$, which clearly cannot be typed.

Lemma 2 (Subject congruence). *Suppose that $P \equiv Q$. Then $E \vdash P : b$ if and only if $E \vdash Q : b$.*

Assuming a suitable relationship between δ and $\mathsf{type}()$, we have

Lemma 3 (Subject reduction for expressions). *Suppose $M_1 \longrightarrow M_2$. If $E \vdash M_1 : \tau$ then also $E \vdash M_2 : \tau$.*

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

$$\epsilon \sim \varepsilon$$
$$\mathrm{comm}(\tau) \sim \mathsf{put}(\tau^-).\mathsf{get}(\tau^+) \text{ if } \tau^- \leqslant \tau \leqslant \tau^+$$

Theorem 1 (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 $b_0.b' \leqslant b$.*

5 Type Checking

In this section we show that given a complete type derivation for some process P, we can check its validity according to the rules from Fig. 4. For that purpose, we use techniques from the theory of finite non-deterministic automata.

By induction we can show that for all b it is possible to construct an automaton G that *implements* b, i.e. an automaton G such that $[\![b]\!] = \{tr \mid G \text{ accepts } tr\}$.

Lemma 4. *Assume that for τ, σ of level i it is decidable whether $\tau \leqslant \sigma$. Let b_1, b_2 be of level i. Then it is decidable whether $b_1 \leqslant b_2$.*

The method is to first construct G_1 and G_2 implementing b_1 and b_2, then construct their "difference automaton" $G_1 \setminus G_2$, and finally to check whether the latter rejects all inputs.

Lemma 5. *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 $b_1 \leqslant b_2$. Then it is decidable whether $\tau \leqslant \sigma$.*

The proof is by induction on the structure of τ and σ. Whenever $\tau = \mathsf{cap}[B]$ and $\sigma = \mathsf{cap}[B']$, we use Lemma 1 to test whether $B \leqslant B'$.

Theorem 2. *Given b_1 and b_2, it is decidable whether $b_1 \leqslant b_2$. Given τ and σ, it is decidable whether $\tau \leqslant \sigma$.*

This follows from Lemmas 4 and 5. We also have

Lemma 6. *Given behaviors b and b', it is decidable whether b is safe, and it is decidable whether $b_1 \rightsquigarrow b_2$.*

These results show that the side conditions for the rules (Beh Subsumption), (Exp Subsumption), (Proc Repl) and (Proc Amb) are decidable, yielding

Theorem 3 (Decidability of type checking). *Given a purported derivation of $E \vdash M : \tau$ or $E \vdash P : b$, we can check its validity.*

6 Discussion

Our type system is a conservative extension of the type system for **AC** presented in [7, Sect. 3]. To see this, we employ a function *Plus* translating entities in the latter system into entities in the former; in particular "message types" W^C into types, and "exchange types" T^C into behaviors. *Plus* is defined recursively on the structure of its argument; most clauses are "homomorphisms" except for

$$\begin{aligned}
\mathit{Plus}(M^C\boldsymbol{[}P^C\boldsymbol{]}) &= \mathit{Plus}(M^C)\boldsymbol{[}\mathit{Plus}(P^C) \mid \mathsf{coopen}\ \mathit{Plus}(M^C).\mathbf{0}\boldsymbol{]}\\
\mathit{Plus}(\mathsf{cap}[T^C]) &= \mathsf{cap}[\mathit{Plus}(T^C) \mid \Box]\\
\mathit{Plus}(\mathit{Shh}) &= \varepsilon\\
\mathit{Plus}(W_1^C \times \ldots \times W_n^C) &= \mathsf{fromnow}\ \{\times(\mathit{Plus}(W_1^C),\ldots,\mathit{Plus}(W_n^C))\}
\end{aligned}$$

Theorem 4. *Suppose that $E^C \vdash P^C : T^C$, respectively $E^C \vdash M^C : W^C$, is derivable in the system of [7, Sect. 3]. Then $\mathit{Plus}(E^C) \vdash \mathit{Plus}(P^C) : \mathit{Plus}(T^C)$, respectively $\mathit{Plus}(E^C) \vdash \mathit{Plus}(M^C) : \mathit{Plus}(W^C)$, is derivable in our system.*

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[3]. As in [5] 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 [11] does) the problem faced in [5]. Another application might be to predict the shape of ambients, as done in [13] using tree grammars.

Besides the tasks mentioned in Sect. 1 (in particular type inference), future work includes investigating the relationship to the system proposed by Levi & Sangiorgi [11] 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 [11]). For that purpose we must extend our poly-typed **AC+** with coin and coout expressions, recorded also in the traces.

References

[1] Torben Amtoft, Assaf J. Kfoury, and Santiago Pericas-Geersten. What are polymorphically-typed ambients? Technical Report BUCS-TR-2000-021, Comp. Sci. Dept., Boston Univ., December 2000.

[2] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[3] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Typed mobile objects. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 504–520, 2000.

[4] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 51–94. Springer-Verlag, 1999.

[3] In fact, the type system of [23] can be viewed as such an abstraction where, e.g., $[\![\veebar O \rightsquigarrow I]\!]$ is the set of traces containing actions $\mathsf{put}(\tau)$ with τ described by O, actions $\mathsf{get}(\tau)$ with τ described by I, but no enter or exit actions.

[5] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, July 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.

[6] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[7] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *POPL'99, San Antonio, Texas*, pages 79–92. ACM Press, January 1999.

[8] Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In *CONCUR 1996*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[9] Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Proc. European Symp. on Programming*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.

[10] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[11] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*, pages 352–364. ACM Press, January 2000.

[12] Eugenio 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.

[13] Hanne Riis Nielson and Flemming Nielson. Shape analysis for mobile ambients. In *POPL'00, Boston, Massachusetts*, pages 142–154. ACM Press, 2000.

[14] Benjamin C. Pierce and Davide 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.

[15] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, IU, 1997.

[16] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL'99, San Antonio, Texas*, pages 93–104. ACM Press, 1999.

[17] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop (CSFW-12), Mordano, Italy*, June 1999.

[18] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[19] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.

[20] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, volume 1686 of *LNCS*. Springer-Verlag, 1999.

[21] Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order mobile processes. In *CONCUR 1999*, volume 1664 of *LNCS*, pages 557–573. Springer-Verlag, 1999.

[22] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. In *LICS 2000*, pages 334–345, 2000.

[23] Pascal Zimmer. Subtyping and typing algorithms for mobile ambients. In *FOSSACS 2000, Berlin*, volume 1784 of *LNCS*, pages 375–390. Springer-Verlag, 2000.