A Calculus of Bounded Capacities*

Franco Barbanera[1], Michele Bugliesi[2],
Mariangiola Dezani-Ciancaglini[3], and Vladimiro Sassone[4]

[1] Università di Catania, Viale A.Doria 6, 95125 Catania (Italy)
`barba@dmi.unict.it`
[2] Università "Cà Foscari", Via Torino 155, 30170 Venezia (Italy)
`michele@dsi.unive.it`
[3] Università di Torino, Corso Svizzera 185, 10149 Torino (Italy)
`dezani@di.unito.it`
[4] University of Sussex, Falmer, Brighton BN1 9RH UK
`vs@susx.ac.uk`

Abstract. Resource control has attracted increasing interest in foundational research on distributed systems. This paper focuses on space control and develops an analysis of space usage in the context of an ambient-like calculus with bounded capacities and weighed processes, where migration and activation require space. A type system complements the dynamics of the calculus by providing static guarantees that the intended capacity bounds are preserved throughout the computation.

Introduction

Emerging computing paradigms, such as Global Computing and Ambient Intelligence, envision scenarios where mobile devices travel across domains and networks boundaries. Current examples include smart cards, embedded devices (e.g. in cars), mobile phones, PDAs, and the list keeps growing. The notion of third-party resource usage will raise to a central role, as roaming entities will need to borrow resources from host networks and, in turn, provide guarantees of bounded resource usage. This is the context of the present paper, which focuses on *space consumption* and *capacity bound* awareness.

Resource control, in diverse incarnations, has recently been the focus of foundational research. Topics considered include the ability to read from and to write to a channel [15], the control of the location of channel names [18], the guarantee that distributed agents will access resources only when allowed to do so [8, 14, 1, 6, 7]. Specific work on the certification of bounds on resource consumption include [9], which introduces a notion of resource type representing an abstract unit of space, and uses a linear type system to guarantee linear space consumption; [4] where quantitative bounds on time usage are enforced using a typed assembly language; and [11], which puts forward a general formulation of resource usage analysis.

* F. Barbanera is partially supported by MIUR project NAPOLI, M. Bugliesi by EU-FET project 'MyThS' IST-2001-32617, and by MIUR project MEFISTO, M. Dezani-Ciancaglini by EU-FET project DART IST-2001-33477, and by MIUR Projects COMETA and McTati, V. Sassone by EU-FET project 'MIKADO' IST-2001-32222. The funding bodies are not responsible for any use that might be made of the results presented here.

We elect to formulate our analysis of space control in an ambient-like calculus, BoCa, because the notion of ambient mobility is a natural vehicle to address the intended application domain. Relevant references to related work in this context include [5], which presents a calculus in which resources may be moved across locations provided suitable space is available at the target location; [17], which uses typing systems to control resource usage and consumption; and [3], which uses static techniques to analyse the behaviour of finite control processes, i.e., those with bounded capabilities for ambient allocation and output creation.

BoCa relies on a physical, yet abstract, notion of "resource unit" defined in terms of a new process constructor, noted ▬ (read "*slot*"), which evolves out of the homonym notion of [5]. A slot may be interpreted as a unit of computation space to be allocated to running processes and migrating ambients. To exemplify, the configuration

$$P \mid \underbrace{\text{▬} \mid \ldots \mid \text{▬}}_{\mathsf{k}\ \text{times}}$$

represents a system which is running process P and which has k resource units available for P to spawn new subprocesses and to accept migrating agents willing to enter. In both cases, the activation of the new components is predicated to the presence of suitable resources: only processes and agents requiring cumulatively no more than k units may be activated on the system. As a consequence, process activation and agent migration involve a protocol to "negotiate" the use of resources with the enclosing, resp. receiving, context (possibly competing with other processes).

For migrating agents this is accounted for by associating each agent with a tag representing the space required for activation at the target context, as in $a^{\mathsf{k}}[\,P\,]$. A notion of well-formedness will ensure that k provides a safe estimate of the space needed by $a[\,P\,]$; namely, the number of resource units allocated to P. Correspondingly, the negotiation protocol for mobility is represented formally by the following reductions (where ▬^{k} is short for $\text{▬} \mid \ldots \mid \text{▬}$, k times):

$$\begin{array}{rcl} a^{\mathsf{k}}[\,\mathbf{in}\, b.P \mid Q\,] \mid b[\,\text{▬}^{\mathsf{k}} \mid R\,] & \searrow & \text{▬}^{\mathsf{k}} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,] \mid R\,] \\ \text{▬}^{\mathsf{k}} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\, b.Q \mid R\,]\,] & \searrow & a^{\mathsf{k}}[\,Q \mid R\,] \mid b[\,P \mid \text{▬}^{\mathsf{k}}\,] \end{array}$$

In both cases, the migrating agent releases the space required for its computation at the source site and gets corresponding space at the target context. Notice that the reductions construe ▬ both as a representation of the physical space available at the locations of the system, and as a particular new kind of co-capability.

Making the weight of an ambient depend explicitly on its contents allows a clean and simple treatment of the open capability: opening does not require resources, as those needed to allocate the contents are exactly those taken by the ambient as such.

$$\mathbf{opn}\, a.P \mid a[\,\overline{\mathbf{opn}}.Q \mid R\,] \quad \searrow \quad P \mid Q \mid R$$

Notice that in order for these reductions to provide the intended semantics of resource negotiation, it is crucial that the redexes are well-formed. Accordingly, the dynamics of ambient mobility is inherently dependent on the assumption that all migrating agents are well-formed. As we shall discuss, this assumption is central to the definition of behavioural equivalence as well.

Resource management and consumption does not concern exclusively mobility, as *all* processes need and use space. It is natural then to expect that "spawning" (activating) processes requires resources, and that unbounded replication of processes is controlled so as to guard against processes that may consume an *infinite* amount of resources. The action of spawning a new process is made explicit in BoCa by introducing a new process construct, $\mathsf{k}\triangleright$, whose semantics is defined by the following reduction:

$$\mathsf{k} \triangleright P \mid \blacksquare^{\mathsf{k}} \quad \searrow \quad P$$

Here $\mathsf{k} \triangleright P$ is a "spawner" which launches P provided that the local context is ready to allocate enough fresh resources for the activation. The tag k represents the "activation cost" for process P, viz. its weight, while $\mathsf{k} \triangleright P$, the "frozen code" of P, weighs 0: again here the hypothesis of well-formedness of terms is critical to make sense of the spawning protocol. The adoption of an explicit spawning operator allows us to delegate to the "spawner" the responsibility of resource control in the mechanism for process replication. In particular, we restrict the replication primitive "**!**" to 0-weight processes only. We can then rely on the usual congruence rule that identifies $!P$ with $!P \mid P$, and use $!(\mathsf{k} \triangleright P)$ to realise a resource-aware version of replication. This results in a system which separates process *duplication* and *activation*, and so allows a fine analysis of resource consumption in computation.

BoCa is completed by two constructs that provide for dynamic allocation of resources. In our approach resources are not "created" from the void, but rather acquired dynamically – in fact, transferred – from the context, again as a result of a negotiation.

$$\begin{array}{rcl} a^{\mathsf{k}+1}[\, \mathbf{put}.P \mid \blacksquare \mid Q\,] \mid b^{\mathsf{h}}[\, \mathbf{get}\; a.R \mid S\,] & \searrow & a^{\mathsf{k}}[\, P \mid Q\,] \mid b^{\mathsf{h}+1}[\, R \mid \blacksquare \mid S\,] \\ \mathbf{put}^{\downarrow}.P \mid \blacksquare \mid a^{\mathsf{k}}[\, \mathbf{get}^{\uparrow}.Q \mid R\,] & \searrow & P \mid a^{\mathsf{k}+1}[\, \blacksquare \mid Q \mid R\,] \end{array}$$

Resource transfer is realised as a two-way synchronisation in which a context offers some of its resource units to any enclosed or sibling ambient that makes a corresponding request. The effect of the transfer is reflected in the tags that describe the resources allocated to the receiving ambients. We formalise slot transfers only between siblings and from father to child. As we shall see, transfers across siblings make it possible to encode a notion of private resource, while transfer from child to parent can easily be encoded in terms of the existing constructs.

The semantic theory of BoCa is supported by a labelled transition systems which gives rise to a bisimulation congruence adequate with respect to barbed congruence. Besides enabling powerful co-inductive characterizations of process equivalences, the labelled transition system yields an effective tool for contextual reasoning on process behavior. More specifically, it enables a formal representation of *open systems*, in which processes may acquire resources and space from their enclosing context. Due to the lack of space, here we only discuss the notion of barb, leaving the presentation of the transition system to the forthcoming full version of the paper. We will focus, instead, on BoCa's capacity types, a system of types that guarantees capacity bounds on computational ambients. Precisely, given lower and upper bounds for ambients capacities, the system enables us to certify statically the absence of under/over-flows, potentially arising from an uncontrolled use of dynamic space allocation capabilities.

We remark that our approach is typical of a way to couple language design with type analysis very useful in frameworks like Global Computing, where it is ultimately unrealistic to assume acquaintance with all the entities which may in the future interact with us, as it is usually done for standard type systems. The openness of the network and its very dynamic nature deny us any substantial form of global knowledge. Therefore, syntactic constructs must be introduced to support the static analysis, as e.g., our "negotiation" protocols. In our system, the possibility of dynamically checking particular space constraints is a consequence of the explicit presence of the primitive $\blacksquare$. A further reason to avoid resource control mechanisms in ambient-like calculi mainly based on static typing systems is that they tend, as perfectly illustrated in [17], to require 3-way synchronisations which, as explained in [16], make the calculus cumbersome.

Structure of the paper. In §1 we give the formal description of BoCa and its operational semantics, and we illustrate it with a few examples. The type system for the calculus is illustrated in §2. In §3 we discuss the issue of resource interference, and we extend the calculus to deal with private resources in the form of named slots.

1 The Calculus BoCa

The calculus is a conservative extension of the Ambient Calculus. We presuppose two mutually disjoint sets: $\mathcal{N}$ of names, and $\mathcal{V}$ of variables. The set $\mathcal{V}$ is ranged over by letters at the end of the alphabet, typically x, y, z, while a, b, c, d, n, m range over $\mathcal{N}$. Finally, h, k and other letters in the same font denote integers. The syntax of the calculus is defined below, with π and W types as introduced in §2.

Definition 1 (Preterms and Terms). The set of process *preterms* is defined by the following productions (where we assume $\mathsf{k} \geq 0$):

$$\begin{array}{lll} \textit{Processes} & P & ::= \blacksquare \mid \mathbf{0} \mid M.P \mid P|P \mid M^{\mathsf{k}}[\,P\,] \mid \mathbf{!}P \mid (\boldsymbol{\nu} a:\pi)P \mid \mathsf{k} \triangleright P \mid (x:W)P \mid \langle M \rangle P \\ \textit{Capabilities} & C & ::= \mathbf{in}\ M \mid \mathbf{out}\ M \mid \mathbf{opn}\ M \mid \mathbf{get}\ M \mid \mathbf{get}^{\uparrow} \mid \overline{\mathbf{opn}} \mid \mathbf{put} \mid \mathbf{put}^{\downarrow} \\ \textit{Messages} & M & ::= nil \mid a \in \mathcal{N} \mid x \in \mathcal{V} \mid C \mid M.M \end{array}$$

A (well-formed) *term* P is a preterm such that $w(P) \neq \bot$, where $w : \textit{Processes} \rightharpoonup \omega$ is the partial *weight* function defined as follows:

$$w(\mathbf{0}) = 0 \qquad w(\blacksquare) = 1 \qquad w(P \,|\, Q) = w(P) + w(Q)$$

$$w(M.P) = w((x:\chi)P) = w(\langle M \rangle P) = w((\boldsymbol{\nu} a:\pi)P) = w(P)$$

$$w(a^{\mathsf{k}}[\,P\,]) = \text{if } w(P) \text{ is } \mathsf{k} \text{ then } \mathsf{k} \text{ else } \bot$$

$$w(\mathsf{k} \triangleright P) = \text{if } w(P) \text{ is } \mathsf{k} \text{ then } 0 \text{ else } \bot$$

$$w(\mathbf{!}P) = \text{if } w(P) \text{ is } 0 \text{ then } 0 \text{ else } \bot$$

We use the standard notational conventions for ambient calculi. We omit types when not relevant; we write $a[\,P\,]$ instead of $a^{\mathsf{k}}[\,P\,]$ when the value of k does not matter; we use $\blacksquare^{\mathsf{k}}$ as a shorthand for $\underbrace{\blacksquare \,|\, \ldots \,|\, \blacksquare}_{\mathsf{k}}$ and similarly C^{k} as a shorthand for $\underbrace{C.\ldots.C}_{\mathsf{k}}$

1.1 Reduction

The dynamics of the calculus is defined as usual in terms of structural congruence and reduction (cf. Figure 1). Unlike other calculi, however, in BoCa both relations are only defined for proper terms, a fact we will leave implicit in the rest of the presentation.

Structural Congruence: $(|,\mathbf{0})$ is a commutative monoid.

$$\begin{array}{lllll}
(\nu a)(P \mid Q) \equiv (\nu a)P \mid Q \ \ (a \notin \mathrm{fn}(Q)) & & (\nu a)a^{0}[\,\mathbf{0}\,] \equiv \mathbf{0} \\
(\nu a)\mathbf{0} \equiv \mathbf{0} & & (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \\
!P \equiv P \mid !P & & a[\,(\nu b)P\,] \equiv (\nu b)a[\,P\,] \ \ (a \neq b)
\end{array}$$

Reduction: $E \ ::= \ \{\cdot\} \mid E \mid P \ \mid \ (\nu m)E \mid m^{\mathsf{k}}[\,E\,]$ is an evaluation context

$$\begin{array}{lrcl}
\text{(ENTER)} & a^{\mathsf{k}}[\,\mathbf{in}\,b.P \mid Q\,] \mid b[\,\blacksquare^{\mathsf{k}} \mid R\,] & \searrow & \blacksquare^{\mathsf{k}} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,] \mid R\,] \\
\text{(EXIT)} & \blacksquare^{\mathsf{k}} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\,b.Q \mid R\,]\,] & \searrow & a^{\mathsf{k}}[\,Q \mid R\,] \mid b[\,P \mid \blacksquare^{\mathsf{k}}\,] \\
\text{(OPEN)} & \mathbf{opn}\,a.P \mid a[\,\overline{\mathbf{opn}}.Q \mid R\,] & \searrow & P \mid Q \mid R \\
\text{(GETS)} & a^{\mathsf{k}+1}[\,\mathbf{put}.P \mid \blacksquare \mid Q\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\,a.R \mid S\,] & \searrow & a^{\mathsf{k}}[\,P \mid Q\,] \mid b^{\mathsf{h}+1}[\,R \mid \blacksquare \mid S\,] \\
\text{(GETD)} & \mathbf{put}^{\downarrow}.P \mid \blacksquare \mid a^{\mathsf{k}}[\,\mathbf{get}^{\uparrow}.Q \mid R\,] & \searrow & P \mid a^{\mathsf{k}+1}[\,\blacksquare \mid Q \mid R\,] \\
\text{(SPAWN)} & \mathsf{k} \rhd P \mid \blacksquare^{\mathsf{k}} & \searrow & P \\
\text{(EXCHANGE)} & (x:\chi)P \mid \langle M \rangle Q & \searrow & P\{x := M\} \mid Q \\
\text{(STRUCT)} & P \equiv P' \quad P' \searrow Q' \quad Q' \equiv Q & \Longrightarrow & P \searrow Q \\
\text{(CONTEXT)} & P \searrow Q & \Longrightarrow & E\{P\} \searrow E\{Q\}
\end{array}$$

Fig. 1. Structural Congruence and Reduction

The reduction relation $\searrow$ is defined according to the intuitions discussed in the introduction; we denote with $\searrow_{*}$ the reflexive and transitive closure of $\searrow$. Structural congruence is essentially standard. The assumption of well-formedness is central to both relations. In particular, the congruence $!P \equiv P \mid !P$ only holds with P a proper term of weight 0. Thus, to duplicate arbitrary processes we need to first "freeze" them under $\mathsf{k}\rhd$, i.e. we decompose arbitrary duplication into "template replication" and "process activation." We define $!^{\mathsf{k}} \triangleq\, !\mathsf{k}\rhd$, which gives us $!^{\mathsf{k}}P \mid \blacksquare^{\mathsf{k}} \searrow\, !^{\mathsf{k}}P \mid P$.

A few remarks are in order on the form of the transfer capabilities. The **put** capability (among siblings) does not name the target ambient, as is the case for the dual capability **get**. We select this particular combination because it is the most liberal one for which our results hold. Of course, more stringent notions are possible, as e.g. when both partners in a synchronisation use each other's names. Adopting any of these would not change the nature of the calculus and preserve, mutatis mutandis, the validity of our results. In particular, the current choice makes it easy and natural to express interesting programming examples (cf. the memory management in §1.3), and protocols: e.g., it enables us to provide simple encoding of named (and private) resources allocated for spawning (cf. §3). Secondly, a new protocol is easily derived for transferring resources "upwards" from children to parents using the following pair of dual put and get.

$$\mathbf{get}^{\downarrow}a.P \triangleq (\boldsymbol{\nu} m)(\mathbf{opn}\ m.P \mid m[\,\mathbf{get}\ a.\overline{\mathbf{opn}}\,]), \quad \text{and} \quad \mathbf{put}^{\uparrow} \triangleq \mathbf{put}$$

Transfers affect the amount of resources allocated at different nesting levels in a system. We delegate to the type system of §2 to control that no nesting level suffers from resource over- or under-flows. The reduction semantics itself guarantees that the global amount of resources is preserved, as it can be proved by an inspection of the reduction rules.

Proposition 1 (Resource Preservation). *If $w(P) \neq \bot$, and $P \searrow_{*} Q$, then $w(Q) = w(P)$.*

Two remarks about the above proposition are in order. First, resource preservation is a distinctive property of *closed* systems; in open systems, instead, a process may acquire new resources from the environment, or transfer resources to the environment, by exercising the **put** and **get** capabilities. Secondly, the fact that the global weight of a process is invariant through reduction does not imply that the amount of resources available for computation also is invariant. Indeed, our notion of slot is an economical way to convey the three different concepts of a resource being *free*, *allocated*, or *wasted* , according to the context in which $\blacksquare$ occurs during the computation. Unguarded slots, as in $a[\,\blacksquare \mid P\,]$, represent resources available for spawning or mobility at a given nesting level; guarded slots, like $M.\blacksquare$, represent allocated resources, which may potentially be released and become free; and unreachable slots, like $(\boldsymbol{\nu} a)\mathbf{in}\ a.\blacksquare^{\mathsf{k}}$ or $(\boldsymbol{\nu} a)a^{\mathsf{k}}[\,\blacksquare^{\mathsf{k}}\,]$, represent wasted resources that will never be released.

Computation changes the state of resources in the expected ways: allocated resources may be freed, as in $\mathbf{opn}\ a.\blacksquare \mid a[\,P\,] \searrow \blacksquare \mid P$; free resources may be allocated, as in $\blacksquare \mid 1 \triangleright M.\blacksquare \searrow M.\blacksquare$, or wasted as in $\mathbf{put}^{\downarrow} \mid \blacksquare \mid (\boldsymbol{\nu} a)a[\,\mathbf{get}^{\uparrow}\,] \searrow (\boldsymbol{\nu} a)a[\,\blacksquare\,]$. No further transition for wasted resource is possible: in particular, it may never become free, and re-allocated. Accordingly, while the global amount of resources is invariant through reduction, as stated in Proposition 1, the computation of a process does in general consume resources and leaves a non-increasing amount of free and allocated resources. We leave to our future work the development of a precise analysis of resource usage based on the characterization we just outlined, and focus on the behavioural semantics of the calculus instead.

1.2 Behavioural Semantics

The semantic theory of BoCa is based on *barbed congruence* [13], a standard equality relation based on reduction and a notion of observability. As usual in ambient calculi, our observation predicate, $P\downarrow_a$, indicates the possibility for process P to interact with the environment via an ambient named a. In Mobile Ambients (MA) this is defined as follows:

$$(1) \qquad P\downarrow_a \quad \triangleq \quad P \equiv (\nu \tilde{m})(a[\,P'\,] \mid Q) \qquad a \notin \tilde{m}$$

Since no authorisation is required to cross a boundary, the presence of an ambient a at top level denotes a potential interaction between the process and the environment via a. In the presence of co-capabilities [12], however, the process $(\boldsymbol{\nu}\tilde{m})(a[\, P\,] \mid Q)$ only represents a potential interaction if P can exercise an appropriate co-capability. The same observation applies to BoCa, as many aspects of its dynamics rely on co-capabilities: notably, mobility, opening, and transfer across ambients. Correspondingly, we have the following reasonable choices of observation (with $a \notin \{\tilde{m}\}$):

$$P \downarrow_a^{opn} \quad \triangleq \quad P \equiv (\boldsymbol{\nu}\tilde{m})(a[\, \overline{\mathbf{opn}}.P' \mid Q\,] \mid R) \tag{2}$$

$$P \downarrow_a^{slt} \quad \triangleq \quad P \equiv (\boldsymbol{\nu}\tilde{m})(a[\, \blacksquare \mid Q\,] \mid R) \tag{3}$$

$$P \downarrow_a^{put} \quad \triangleq \quad P \equiv (\boldsymbol{\nu}\tilde{m})(a[\, \mathbf{put}.P' \mid \blacksquare \mid Q\,] \mid R) \tag{4}$$

As it turns out, definitions (1)–(4) yield the same barbed congruence relation. Indeed, the presence of 0-weighted ambients makes it possible to rely on the same notion of observation as in MA, that is (1), without consequences on barbed congruences. We discuss this in further detail below.

Our notion of barbed congruence is standard, except that we require closure by well-formed contexts. Say that a relation $\mathcal{R}$ is *reduction closed* if $P\mathcal{R}Q$ and $P \searrow P'$ imply the existence of some Q' such that $Q \searrow_* Q'$ and $P'\mathcal{R}Q'$; it is *barb preserving* if $P\mathcal{R}Q$ and $P\downarrow_a$ imply $Q\Downarrow_a$, i.e. $Q \searrow_* \downarrow_a$.

Definition 2 (Barbed Congruence). Barbed bisimulation, noted $\simeq$, is the largest symmetric relation on closed processes that is reduction closed and barb preserving. Two processes P and Q are *barbed congruent*, written $P \cong Q$, if for all contexts $\mathcal{C}[\cdot]$, preterm $\mathcal{C}[P]$ is a term iff so is $\mathcal{C}[Q]$, and then $\mathcal{C}[P] \simeq \mathcal{C}[Q]$.

Let then $\cong_i$ be the barbed congruence relation resulting from Definition 2 and from choosing the notion of observation as in (i) above (with $i \in [1..4]$).

Proposition 2 (Independence from Barbs). $\cong_i = \cong_j$ *for all* $i, j \in [1..4]$.

Since the relations differ only on the choice of barb, Proposition 2 is proved by just showing that all barbs imply each other. This can be accomplished, as usual, by exhibiting a corresponding context. For instance, to see that $\cong_3$ implies $\cong_2$ use the context $\mathcal{C}[\cdot] = [\cdot] \mid \mathbf{opn}\ a.b^1[\, \blacksquare\,]$, and note that for all P such that b is fresh in P one has $P \Downarrow_a^{opn}$ if and only if $\mathcal{C}[P] \Downarrow_b^{slt}$.

The import of the processes' weight in the relation of behavioural equivalence is captured directly by the well-formedness requirement in Definition 2. In particular, processes of different weight are distinguished, irrespective of the their "purely" behavioral properties. To see that, note that any two processes P and Q of weight, say, k and h with $\mathsf{h} \neq \mathsf{k}$, are immediately distinguished by the context $\mathcal{C}[\cdot] = a^{\mathsf{k}}[\, [\cdot]\,]$, as $\mathcal{C}[P]$ is well-formed while $\mathcal{C}[Q]$ is not.

1.3 Examples

We complete the presentation of the calculus with some encodings of systems and examples in which space usage and control is modelled.

Recovering Mobile Ambients. The Ambient Calculus [2] is straightforwardly embedded in (an untyped version of) BoCa: it suffices to insert a process $\mathbf{!\overline{opn}}$ in all ambients. The relevant clauses of the embedding are as follows:

$$[\![a[\,P\,]]\!] \triangleq a^0[\,\mathbf{!\overline{opn}} \mid [\![P]\!]\,], \quad [\![(\boldsymbol{\nu} a)P]\!] \triangleq (\boldsymbol{\nu} a)[\![P]\!]$$

and the remaining ones are derived similarly; clearly all resulting processes weigh 0.

Encoding Father-Son Swap. In BoCa, like in any situation where ambient weighs, this swap is possible only in case the father and child nodes have the same weight. We present it for example in the case of weight 1. Notice the use of the primitives for child to father slot exchange that we have defined in §1.

$$b^1[\,\mathbf{get}^{\downarrow}a\,.\,\mathbf{put}\,.\,\mathbf{in}\;a\,.\,\mathbf{get}^{\uparrow} \mid a^1[\,\mathbf{put}^{\uparrow}\,.\,\mathbf{out}\;b\,.\,\mathbf{get}\;b\,.\,\mathbf{put}^{\downarrow} \mid \blacksquare\,]\,] \searrow_* a^1[\,b^1[\,\blacksquare\,]\,]$$

Encoding Ambient Renaming. We can represent in BoCa a form of ambient self-renaming capability. First, define $\mathbf{spw}_a b^{\mathsf{k}}[\,P\,] \triangleq exp^0[\,\mathbf{out}\;a\,.\,\overline{\mathbf{opn}}\,.\,\mathsf{k} \triangleright b^{\mathsf{k}}[\,P\,]\,]$ and then use it to define

$$a\;\mathbf{be}^{\mathsf{k}} b\,.\,P \triangleq \mathbf{spw}_a b^{\mathsf{k}}[\,\blacksquare^{\mathsf{k}} \mid \mathbf{opn}\;a\,] \mid \mathbf{in}\;b\,.\,\overline{\mathbf{opn}}\,.\,P$$

Since $\mathbf{opn}\;exp \mid \blacksquare^{\mathsf{k}} \mid a^{\mathsf{h}}[\,\mathbf{spw}_a b^{\mathsf{k}}[\,P\,] \mid Q\,] \searrow_* b^{\mathsf{k}}[\,P\,] \mid a^{\mathsf{h}}[\,Q\,]$ where k, h are the weights of P and Q, respectively, we get

$$a^{\mathsf{k}}[\,a\;\mathbf{be}^{\mathsf{k}} b\,.\,P \mid R\,] \mid \blacksquare^{\mathsf{k}} \mid \mathbf{opn}\;exp \searrow_* b^{\mathsf{k}}[\,P \mid R\,] \mid \blacksquare^{\mathsf{k}}$$

So, an ambient needs to *borrow* space from its parent in order to rename itself. We conjecture that renaming cannot be obtained otherwise.

A Memory Module. A user can take slots from a memory module MEM_MOD using MALLOC and release them back to MEM_MOD after their use.

$$\begin{aligned}
\text{MEM\_MOD} &\triangleq mem[\,\blacksquare^{256MB} \mid \overbrace{\mathbf{opn}\;m \mid \ldots \mid \mathbf{opn}\;m}^{256MB}\,] \\
\text{MALLOC} &\triangleq m[\,\mathbf{out}\;u\,.\,\mathbf{in}\;mem\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{put}\,.\,\mathbf{get}\;u\,.\,\mathbf{opn}\;m\,] \\
\text{USER} &\triangleq u[\,\ldots.\text{MALLOC} \mid \ldots.\mathbf{get}\;mem\ldots\mathbf{put} \mid \ldots\,]
\end{aligned}$$

2 Bounding Resources, by Typing

In this section we discuss a type system that provides static guarantees for a simple behavioural property, namely the absence of space under- and over-flows arising as a result of transfers during the computation. To deal with this satisfactorily, we need to take into account that transfer (co)capabilities can be acquired by way of exchanges. The type of a capability will hence have to express how it affects the space of the ambient in which it can be performed.

2.1 The Types

We use $\mathcal{Z}$ to denote the set of integers, and note $\mathcal{Z}^+$ and $\mathcal{Z}^-$ the sets of non-negative and non-positive integers respectively. We define the following domains:

$$\begin{array}{rl} \textit{Intervals} & \iota \in \mathfrak{I} \triangleq \{[\mathsf{n},\mathsf{N}] \mid \mathsf{n},\mathsf{N} \in \mathcal{Z}^+,\ \mathsf{n} \leq \mathsf{N}\} \\ \textit{Effects} & \varepsilon \in \mathcal{E} \triangleq \{(\mathsf{d},\mathsf{i}) \mid \mathsf{d} \in \mathcal{Z}^-,\ \mathsf{i} \in \mathcal{Z}^+\} \\ \textit{Thread Effects} & \phi \in \Phi \triangleq \mathcal{E} \to \mathcal{E} \end{array}$$

Intervals and effects are ordered in the usual way, namely: $[\mathsf{n},\mathsf{N}] \leq [\mathsf{n}',\mathsf{N}']$ when $\mathsf{n}' \leq \mathsf{n}$ and $\mathsf{N} \leq \mathsf{N}'$ and $(\mathsf{d},\mathsf{i}) \leq (\mathsf{d}',\mathsf{i}')$ when $\mathsf{d}' \leq \mathsf{d}$ and $\mathsf{i} \leq \mathsf{i}'$. It is also convenient to define the component-wise sum operator for effects: $(\mathsf{d},\mathsf{i}) + (\mathsf{d}',\mathsf{i}') = (\mathsf{d}+\mathsf{d}',\mathsf{i}+\mathsf{i}')$, and lift it to Φ pointwise: $\phi_1 + \phi_2 = \lambda\varepsilon.\phi_1(\varepsilon) + \phi_2(\varepsilon)$.

The syntax of types is defined by the following productions:

$$\begin{array}{rl} \textit{Message Types} & W ::= \mathit{Amb}\langle\iota,\varepsilon,\chi\rangle \mid \mathit{Cap}\langle\phi,\chi\rangle \\ \textit{Exchange Types} & \chi ::= \mathit{Shh} \mid W \\ \textit{Process Types} & \pi ::= \mathit{Proc}\langle\varepsilon,\chi\rangle \end{array}$$

Type $\mathit{Proc}\langle\varepsilon,\chi\rangle$ is the type of processes with ε effects and χ exchanges. Specifically, for a process P of type $\mathit{Proc}\langle(\mathsf{d},\mathsf{i}),\chi\rangle$, the effect (d,i) bounds the number of slots delivered (d) and acquired (i) by P as the cumulative result of exercising P's transfer capabilities.

Type $\mathit{Amb}\langle\iota,\varepsilon,\chi\rangle$ is the type of ambients with weight ranging in ι, and enclosing processes with ε effects and χ exchanges. As in companion type systems, values that can be exchanged include ambients and (paths of) capabilities, while the type Shh indicates no exchange. As for capability types, $\mathit{Cap}\langle\phi,\chi\rangle$ is the types of capabilities which, when exercised, unleash processes with χ exchanges, and compose the effect of the unleashed process with the thread effect ϕ. The functional domain of thread effects helps compute the composition of effects. In brief, thread effects accumulate the results from **get**s and **put**s, and compose these with the effects unleashed by occurrences of **opn**.

We introduce the following combinators (functions in Φ) to define the thread effects of the **put**, **get** and **open** capabilities.

$$\begin{aligned} \mathsf{Put} &= \lambda(\mathsf{d},\mathsf{i}).(\mathsf{d}-1,\max(0,\mathsf{i}-1)) \\ \mathsf{Get} &= \lambda(\mathsf{d},\mathsf{i}).(\min(0,\mathsf{d}+1),\mathsf{i}+1) \\ \mathsf{Open}(\varepsilon) &= \lambda(\mathsf{d},\mathsf{i}).(\varepsilon+(\mathsf{d},\mathsf{i})) \end{aligned}$$

The intuition is as follows. A **put** that prefixes a process P with cumulative effect (d,i), contributes to a "shift" in that effect of one unit. The effect of a **get** capability is dual. To illustrate, take $P =$ **put. put. get** a. The thread effect associated with P is computed as follows, where we use function composition in standard order (i.e. $f \circ g(x) = f(g(x))$):

$$\varepsilon \ = \ (\mathsf{Put} \circ \mathsf{Put} \circ \mathsf{Get})((0,0)) \ = \ (-2,0).$$

The intuition about an open capability is similar, but subtler, as the effect of opening an ambient is, essentially, the effect of the process unleashed by the open: in **opn** $n.P$, the process unleashed by **opn** n runs in parallel with P. As a consequence, open has an additive import in the computation of the effect. To motivate, assume that $n : Amb\langle\iota,\varepsilon,\chi\rangle$. Opening n unleashes the enclosed process in parallel to the process P. To compute the resulting effect. we may rely on the effect ε declared by n to bound the effect of the unleashed process: that effect is then is added to the effect of the continuation P. Specifically, if P has effect ε', the composite effect of **opn** $n.P$ is computed as $\mathsf{Open}(\varepsilon)(\varepsilon') = \varepsilon + \varepsilon'$.

2.2 The Typing Rules

The typing rules are collected in Figures 2 and 3, where we denote with id_Φ the identity element in the domain Φ.

The rules in Figure 2 derive judgements $\Gamma \vdash M : W$ for well-typed messages. The rules draw on the intuitions we gave earlier. Notice, in particular, that the capabilities **in**, **out** and the cocapability $\overline{\mathbf{opn}}$ have no effect, as reflected by the use id_Φ in their type. The same is true also of the the co-capability $\mathbf{put}^\downarrow$. In fact, by means of the superscript k in $a^{\mathsf{k}}[\,P\,]$ we can record the actual weight of the ambient (cf. reduction rule (GETD)). This implies that the weight of an ambient in which $\mathbf{put}^\downarrow$ is executed does not change: the ambient loses a slot, but the weight of one of its sub-ambients increases.

$$\frac{}{\Gamma, M : W \vdash M : W}\ (axiom) \qquad \frac{}{\Gamma \vdash nil : Cap\langle \mathsf{id}_\Phi, \chi\rangle}\ (nil)$$

$$\frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{get}\ M : Cap\langle \mathsf{Get}, \chi\rangle}\ (\mathbf{get}\ M) \qquad \frac{}{\Gamma \vdash \mathbf{put} : Cap\langle \mathsf{Put}, \chi\rangle}\ (\mathbf{put})$$

$$\frac{}{\Gamma \vdash \mathbf{get}^\uparrow : Cap\langle \mathsf{Get}, \chi\rangle}\ (\mathbf{get}^\uparrow) \qquad \frac{}{\Gamma \vdash \mathbf{put}^\downarrow : Cap\langle \mathsf{id}_\Phi, \chi\rangle}\ (\mathbf{put}^\downarrow)$$

$$\frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{in}\ M : Cap\langle \mathsf{id}_\Phi, \chi\rangle}\ (\mathbf{in}\ M) \qquad \frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{out}\ M : Cap\langle \mathsf{id}_\Phi, \chi\rangle}\ (\mathbf{out}\ M)$$

$$\frac{\Gamma \vdash M : Amb\langle -,\varepsilon,\chi\rangle}{\Gamma \vdash \mathbf{opn}\ M : Cap\langle \mathsf{Open}(\varepsilon), \chi\rangle}\ (\mathbf{opn}\ M) \qquad \frac{}{\Gamma \vdash \overline{\mathbf{opn}} : Cap\langle \mathsf{id}_\Phi, \chi\rangle}\ (\overline{\mathbf{opn}})$$

$$\frac{\Gamma \vdash M : Cap\langle \phi, \chi\rangle \quad \Gamma \vdash M' : Cap\langle \phi', \chi\rangle}{\Gamma \vdash M.M' : Cap\langle \phi \circ \phi', \chi\rangle}\ (path)$$

Fig. 2. Good Messages

The rules in Figure 3 derive judgements $\Gamma \vdash P : Proc\langle\varepsilon,\chi\rangle$ for well-typed processes. An inspection of the typing rules shows that any well-typed process is also well-formed (in the sense of Definition 1. We let $0_{\mathcal{E}}$ denote the null effect $(0,0)$: thus, rules (**0**) and (▬) simply state that the inhert process and the slot form have no effects. Rule (*prefix*) computes the effects of prefixes, by applying the thread effect of the capability to the

$$\frac{}{\Gamma \vdash \blacksquare : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\blacksquare) \qquad \frac{}{\Gamma \vdash \mathbf{0} : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\mathbf{0})$$

$$\frac{\Gamma \vdash M : Cap\langle \phi, \chi\rangle \quad \Gamma \vdash P : Proc\langle \varepsilon, \chi\rangle}{\Gamma \vdash M.P : Proc\langle \phi(\varepsilon), \chi\rangle}\ (\mathit{prefix})$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon, \chi\rangle \quad \Gamma \vdash Q : Proc\langle \varepsilon', \chi\rangle}{\Gamma \vdash P \mid Q : Proc\langle \varepsilon + \varepsilon', \chi\rangle}\ (\mathit{par}) \qquad \frac{\Gamma, x : W \vdash P : Proc\langle \varepsilon, W\rangle}{\Gamma \vdash (x : W)P : Proc\langle \varepsilon, W\rangle}\ (\mathit{input})$$

$$\frac{\Gamma \vdash M : W \quad \Gamma \vdash P : Proc\langle \varepsilon, W\rangle}{\Gamma \vdash \langle M\rangle P : Proc\langle \varepsilon, W\rangle}\ (\mathit{output}) \qquad \frac{\Gamma, a : Amb\langle \iota, \varepsilon, \chi\rangle \vdash P : Proc\langle \varepsilon', \chi'\rangle}{\Gamma \vdash (\boldsymbol{\nu} a : Amb\langle \iota, \varepsilon, \chi\rangle)P : Proc\langle \varepsilon', \chi'\rangle}\ (\mathit{new})$$

$$\frac{\begin{array}{lll} \Gamma \vdash M : Amb\langle [\mathsf{n}, \mathsf{N}], \varepsilon, \chi'\rangle & & [\max(\mathsf{k}+\mathsf{d}, 0), \mathsf{k}+\mathsf{i}] \leq [\mathsf{n}, \mathsf{N}] \\ \Gamma \vdash P : Proc\langle (\mathsf{d}, \mathsf{i}), \chi'\rangle & w(P) = \mathsf{k} & (\mathsf{d}-\mathsf{i}, \min(\mathsf{N}-\mathsf{n}, \mathsf{i}-\mathsf{d})) \leq \varepsilon \end{array}}{\Gamma \vdash M^{\mathsf{k}}[P] : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\mathit{amb})$$

$$\frac{\Gamma \vdash P : Proc\langle 0_{\mathcal{E}}, \chi\rangle \quad w(P) = \mathsf{k}}{\Gamma \vdash \mathsf{k} \triangleright P : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\mathit{spawn}) \qquad \frac{\Gamma \vdash P : Proc\langle 0_{\mathcal{E}}, \chi\rangle \quad w(P) = 0}{\Gamma \vdash !P : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\mathit{bang})$$

Fig. 3. Good Processes

effect of the process. Rule (*par*) adds up the effects of two parallel threads, while the constructs for input, output and restriction do not have any effect.

Rule (*amb*) governs the formation of ambient processes. The declared weight k of the ambient must reflect the weight of the enclosed process. Two further conditions ensure (*i*) that k modified by the effect (d, i) of the enclosed process lies within the interval $[\mathsf{n}, \mathsf{N}]$ declared by the ambient type, and (*ii*) that effect ε declared by the ambient type is a sound approximation for the effects released by opening the ambient itself. Condition (*i*) is simply $[\max(\mathsf{k}+\mathsf{d}, 0), \mathsf{k}+\mathsf{i}] \leq [\mathsf{n}, \mathsf{N}]$, where the use of $\max(\mathsf{k}+\mathsf{d}, 0)$ is justified by observing that the weight of an ambient may never grow negative as a result the enclosed process exercising **put** capabilities. To motivate condition (*ii*), first observe that opening an ambient which encloses a process with effect (d, i) may only release effects $\varepsilon \leq (\mathsf{d}-\mathsf{i}, \mathsf{i}-\mathsf{d})$. The lower bound arises in a situation in which the ambient is opened right after the enclosed process has completed its i **get** 's and is thus left with $|\mathsf{d}-\mathsf{i}|$ **put**'s unleashed in the opening context. Dually, the upper bound arises when the ambient is opened right after the enclosed process has completed its d **put**'s, and is left with $\mathsf{i}-\mathsf{d}$ **get** 's. On the other hand, we also know that the maximum increasing effect released by opening ambients with weight ranging in $[\mathsf{n}, \mathsf{N}]$ is $\mathsf{N}-\mathsf{n}$. Collectively, these two observations justify the condition $(\mathsf{d}-\mathsf{i}, \min(\mathsf{N}-\mathsf{n}, \mathsf{i}-\mathsf{d})) \leq \varepsilon$ in rule (*amb*).

In rule (*spawn*), the effect of $\mathsf{k} \triangleright P$ is the same as that of the reduct P. Finally, to prevent the effects of duplicated processes to add up beyond control, with unpredictable consequences, rule (*bang*) enforces duplicated process to have null effects.

A first property of the given type system is that all typable preterms are terms.

The following result complements Proposition 1 and shows that capacity bounds on ambients are preserved during computations, while the processes' ability to shrink or expand reduces.

Theorem 1 (Subject Reduction). *Assume* $\Gamma \vdash P : Proc\langle \varepsilon, \chi \rangle$ *and* $P \searrow_* Q$*. Then* $\Gamma \vdash Q : Proc\langle \varepsilon', \chi \rangle$ *for some* $\varepsilon' \leq \varepsilon$*.*

It follows as a direct corollary that no ambient may be subject to under/over-flows during the computation of a process.

Theorem 2 (Absence of Under/Over-Flow). *Assume* $\Gamma \vdash P : Proc\langle \varepsilon, \chi \rangle$ *and let* $P \searrow_* Q$*. If* $a : Amb\langle [\mathsf{n}, \mathsf{N}], -, - \rangle \in \Gamma$*, then, for any subterm of* Q *of the form* $a^{\mathsf{k}}[\, R\,]$*, not in the scope of a binder for a, we have* $\mathsf{n} \leq \mathsf{k} \leq \mathsf{N}$*.*

2.3 Typed Examples

A Typed Memory Module. As a first illustration of the typing system at work we give a typed version of the memory module of Section 1.3. All other examples in that section are typeable too, and this can be easily verified. We start with the *malloc* ambient

$$\textsc{malloc} \triangleq m[\, \mathbf{out}\ u\,.\,\mathbf{in}\ mem\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{put}\,.\,\mathbf{get}\ u\,.\,\mathbf{opn}\ m\,]$$

Since there are no exchanges, we give the typing annotation and derivation disregarding the exchange component from the types. Let P_{malloc} denote thread enclosed within the ambient m. If we let $m : Amb\langle [0,0], (-1,0) \rangle \in \Gamma$, an inspection of the typing rules for capabilities and paths shows that the following typing is derivable for any ambient type assigned to *mem*:

$$\Gamma \vdash \mathbf{out}\ u\,.\,\mathbf{in}\ mem\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{put}\,.\,\mathbf{get}\ u\,.\,\mathbf{opn}\ m : Cap\langle (\mathsf{Put} \circ \mathsf{Get} \circ \lambda\varepsilon.((-1,0) + \varepsilon))(0_{\mathcal{E}}) \rangle$$

Composing the thread effects, one has: $\mathsf{Put} \circ \mathsf{Get}\,(-1,0) = (-1,0)$. From this one derives $\Gamma \vdash P_{malloc} : Proc\langle (-1,0) \rangle$, which gives $\Gamma \vdash \textsc{malloc} : Proc\langle 0_{\mathcal{E}} \rangle$. As to the memory module itself, it is a routine check to verify that the process

$$\textsc{mem\_mod} \triangleq mem[\, \blacksquare^{256MB} \mid \mathbf{opn}\ m \mid \ldots \mid \mathbf{opn}\ m\,]$$

typechecks with $m : Amb\langle [0,0], (-1,0) \rangle$, $mem : Amb\langle [0, 256MB], (-256MB, 256MB) \rangle$.

A Cab Trip. As a further example, we give a new version of the the cab trip protocol from [17], formulated in our calculus. A customer sends a request for a cab, which then arrives and takes the customer to his destination. The use of slots here enables us to model very naturally the constraint that only one passenger (or actually any fixed number of them) may occupy a cab. The typing environment contains $call : W_1$, $cab : W_1$, $trip : W_0$, $loading : W_0$, $unloading : W_0$, $bye : W_0$ where $W_1 = Amb\langle [1,1], 0_{\mathcal{E}} \rangle$, $W_0 = Amb\langle [0,0], 0_{\mathcal{E}} \rangle$.

$$\begin{array}{ll}
\mathit{CALL}(\mathit{from},\mathit{client}) & \triangleq \\
\quad \mathit{call}^1[\,\mathbf{out}\,\mathit{client}\,.\,\mathbf{out}\,\mathit{from}\,.\,\mathbf{in}\,\mathit{cab}\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{in}\,\mathit{from}\,.(\mathit{loading}^0[\,\mathbf{out}\,\mathit{cab}\,.\,\mathbf{in}\,\mathit{client}\,.\,\overline{\mathbf{opn}}\,] \mid \rule{0.8em}{0.4em})\,] & \\
\mathit{TRIP}(\mathit{from},\mathit{to},\mathit{client}) & \triangleq \mathit{trip}^0[\,\mathbf{out}\,\mathit{client}\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{out}\,\mathit{from}\,.\,\mathbf{in}\,\mathit{to}\,.\,\mathit{unloading}^0[\,\mathbf{in}\,\mathit{client}\,.\,\overline{\mathbf{opn}}\,]\,] \\
\mathit{CLIENT}(\mathit{from},\mathit{to}) & \triangleq (\nu c : W_1)c^1[\mathit{CALL}(\mathit{from},c) \mid \mathbf{opn}\,\mathit{loading}\,.\,\mathbf{in}\,\mathit{cab}\,.\,\mathit{TRIP}(\mathit{from},\mathit{to},c) \\
 & \qquad \mid \mathbf{opn}\,\mathit{unloading}\,.\,\mathbf{out}\,\mathit{cab}\,.\,\mathit{bye}^0[\,\mathbf{out}\,c\,.\,\mathbf{in}\,\mathit{cab}\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{out}\,\mathit{to}\,]] \\
\mathit{CAB} & \triangleq \mathit{cab}^1[\,\rule{0.8em}{0.4em} \mid !(\mathbf{opn}\,\mathit{call}\,.\,\mathbf{opn}\,\mathit{trip}\,.\,\mathbf{opn}\,\mathit{bye})\,] \\
\mathit{SITE}(i) & \triangleq \mathit{site}_i[\,\mathit{CLIENT}(\mathit{site}_i,\mathit{site}_j) \mid \mathit{CLIENT}(\mathit{site}_i,\mathit{site}_l) \mid \cdots \mid \rule{0.8em}{0.4em} \mid \rule{0.8em}{0.4em} \mid \cdots\,] \\
\mathit{CITY} & \triangleq \mathit{city}[\,\mathit{CAB} \mid \mathit{CAB} \mid \cdots \mid \cdots \mid \mathit{SITE}(1) \mid \cdots \mid \mathit{SITE}(n) \mid \rule{0.8em}{0.4em} \mid \rule{0.8em}{0.4em} \mid \cdots\,]
\end{array}$$

The fact that only one slot is available in *cab* together with the weight 1 of both *call* and *client* prevents the cab to receive more than one call and/or more than one client. Moreover this encoding limits also the space in each site and in the whole city.

Comparing with [17], we notice that we can deal with the cab's space satisfactorily with no need for 3-way synchronisations. Unfortunately, as already observed in [17], this encoding may lead to unwanted behaviours, since there is no way of preventing a client to enter a cab different from that called and/or the ambient bye to enter a cab different from that the client has left. We will give a safe encoding of this example in Subsection 3.2 using named slots.

3 Controlling Races for Resources

The calculus of the previous sections provides a simple, yet effective, framework for reasoning on resource usage and consumption. On the other hand, it is less effective in expressing *policies* to govern the allocation and distribution of space to distinct, possibly, competitive components. Indeed, with the current semantics it is not entirely obvious that a given resource unit can be selectively allocated to a specific agent, and protected against unintended use. To illustrate, consider the following term (and assume it well-formed):

$$a^1[\,\mathbf{in}\,b.P\,] \mid b[\,1 \triangleright Q \mid \rule{0.8em}{0.4em} \mid d[\,c^1[\,\mathbf{out}\,d.R\,]\,]\,]$$

Three agents are competing for the resource unit in ambient b: ambients a and c, which would use it for their move, and the local spawner inside ambient b. While the race between a and c may be acceptable – the resource unit may be allocated by b to any migrating agent – it would also be desirable for b to reserve resources for internal use, i.e. for spawning new processes. In fact, reserving private space for spawning is possible with the current primitives, by encoding a notion of "named resource". This can be accomplished by defining:

$$\rule{0.8em}{0.4em}_a^{\mathsf{k}} \triangleq a[\,\mathbf{put}^{\mathsf{k}} \mid \rule{0.8em}{0.4em}^{\mathsf{k}}\,], \quad \text{and} \quad \mathsf{k} \triangleright (a,P) \triangleq (\nu n)(n[\,(\mathbf{get}\,a)^{\mathsf{k}}.\mathsf{k} \triangleright \overline{\mathbf{opn}}.P\,] \mid \mathbf{opn}\,n)$$

Then, assuming $w(P) = \mathsf{k}$, one has $(\nu a)(\rule{0.8em}{0.4em}_a^{\mathsf{k}} \mid \mathsf{k} \triangleright (a,P)) \cong P$, as desired. It is also possible to encode a form of "resource renaming", by defining:

$$\{x/y\}.P \triangleq (\nu n)(n[\,\mathbf{get}\,y.\mathbf{put}.\overline{\mathbf{opn}}\,] \mid x[\,\mathbf{get}\,n.\mathbf{put}\,] \mid \mathbf{opn}\,n.P)$$

Then, a y-resource can be turned in to an x-resource: $\{x/y\}.P \mid \rule{0.8em}{0.4em}_y \searrow_* P \mid \rule{0.8em}{0.4em}_x$.

Encoding a similar form of named, and reserved, resources for mobility is subtler. On the one hand, it is not difficult to encode a construct for reserving a x-slot for ambients named x. For example, ambients a and b may agree on the following protocol to reserve a private slot for the move of a into b. If we want to use the space in ambient b for moving a we can write the process:

$$(\boldsymbol{\nu} p,q)(p[\,\mathbf{in}\ b.\mathbf{get}\ q.1 \triangleright \overline{\mathbf{opn}}.a^1[\,\text{▬}\,]\,] \mid b[\,P \mid q[\,\text{▬} \mid \mathbf{put}\,] \mid \mathbf{opn}\ p\,])$$

On the other hand, defining a mechanism to release a named resource to the context from which it has been received is more complex, as it amounts to releasing a resource with the *same* name it was allocated to. This can be simulated loosely with the current primitives, by providing a mechanism whereby a migrating ambient releases an anonymous slot, which is then renamed by the context that is in control of it. The problem is that such a mechanism of releasing and renaming lacks the *atomicity* required to guard against unexpected races for the released resource. Indeed, we conjecture that such atomic mechanisms for named resources can not be defined in the current calculus.

3.1 The Calculus, Refined

We counter the problem by refining the calculus with named resources as primitive notions, and by tailoring the constructs for mobility, transfer and spawning accordingly. Resource units come now always with a tag, as in ▬_η, where $\eta \in \mathcal{N} + \{*\}$ is the unit name. To make the new calculus a conservative extension of the one presented in §1, we make provision for a special tag '*', to be associated with *anonymous* units: any process can be spawned on an anonymous slot, as well any ambient can be moved on it. In addition, we extend the structure of the transfer capabilities, as well as the construct for spawning and ambient as shown in the productions below, which replace the corresponding ones in §1.

$$\begin{array}{lll} \textit{Processes} & P \ ::= \text{▬}_\eta \mid \mathsf{k} \triangleright_\eta P \mid M[\,P\,]_\eta \mid \ldots & \text{as in Section 1} \\ \textit{Capabilities} & C \ ::= \mathbf{get}_\eta\ M \mid \mathbf{get}^{\uparrow}{}_\eta \mid \ldots & \text{as in Section 1} \\ \textit{Messages} & M ::= \ldots & \text{as in Section 1} \end{array}$$

Again, a (well-formed) *term* is a preterm such that in any subterm of the form $a^{\mathsf{k}}[P]$ or $\mathsf{k} \triangleright_\eta P$, P has weight k. The weight of a process can be computed by rules similar to those of Section 1. The anonymous slots ▬_* will be often denoted simply as ▬, and in general η will be omitted when equal to $*$; subscripts on ambients are omitted when irrelevant.

The dynamics of the refined calculus is again defined by means of structural congruence and reduction. Structural congruence is exactly as in Figure 1, the top-level reductions are defined in Figure 4.

The reductions for the transfer capabilities are the natural extensions of the original reductions of §1. Here, in addition to naming the target ambient, the **get** capabilities also indicate the name of the unit they request. The choice of the primitives enables natural forms of scope extrusion for the names of resources, even though resource tags may not be variables. Consider the following system:

$$S \triangleq n[\,(\boldsymbol{\nu} a)(\mathbf{put}.P \mid \text{▬}_a \mid p[\,\mathbf{out}\ n.\mathbf{in}\ m.\overline{\mathbf{opn}}.\mathbf{get}_a\ n\,])\,] \mid m[\,\mathbf{opn}\ p.Q\,]$$

The reductions for ambient opening and exchanges are as in Figure 1, and the rules (ENTER) and (EXIT) have $\eta \in \{a, \star\}$ as side condition. The omitted subscripts ρ on ambients are meant to remain unchanged by the reductions.

$$\begin{array}{llcl}
\text{(ENTER)} & a^{\mathsf{k}}[\,\mathbf{in}\,b.P \mid Q\,]_\rho \mid b[\,\blacksquare^{\mathsf{k}}_{\eta} \mid R\,] & \searrow & \blacksquare^{\mathsf{k}}_{\rho} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,]_\eta \mid R\,] \\
\text{(EXIT)} & \blacksquare^{\mathsf{k}}_{\eta} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\,b.Q \mid R\,]_\rho\,] & \searrow & a^{\mathsf{k}}[\,Q \mid R\,]_\eta \mid b[\,P \mid \blacksquare^{\mathsf{k}}_{\rho}\,] \\
\text{(GETS)} & b^{\mathsf{h}+1}[\,\mathbf{put}.P \mid \blacksquare_\eta \mid Q\,] \mid a^{\mathsf{k}}[\,\mathbf{get}_\eta\ b.R \mid S\,] & \searrow & b^{\mathsf{h}}[\,P \mid Q\,] \mid a^{\mathsf{k}+1}[\,R \mid \blacksquare_\eta \mid S\,] \\
\text{(GETU)} & \mathbf{put}^{\downarrow}.P \mid \blacksquare_\eta \mid a^{\mathsf{k}+1}[\,\mathbf{get}^{\uparrow}{}_\eta.Q \mid R\,] & \searrow & P \mid a^{\mathsf{k}}[\,\blacksquare_\eta \mid Q \mid R\,] \\
\text{(SPAWN)} & \mathsf{k} \rhd_\eta .P \mid \blacksquare^{\mathsf{k}}_{\eta} & \searrow & P
\end{array}$$

Fig. 4. Top-level reductions with named units

Here, the private resource enclosed within ambient n is communicated to ambient m, as $S \searrow_* (\boldsymbol{\nu} a)(n[\,P\,] \mid m[\,Q \mid \blacksquare_a\,])$.

The dynamics of mobility solves the problem we discussed above. To complete a move an ambient a must be granted an anonymous resource or an a-resource. The migrating ambient releases a resource under the name that it was assigned upon the move (as recorded in the tag associated with the ambient construct). Finally, the new semantics of spawning acts as expected, by associating the process to be spawned with a specific set of resources.

These definitions suggest a natural form of resource renaming (or rebinding), noted $\{^{\eta}/_{\rho}\}_{\mathsf{k}}$ with the following operational semantics.

$$\{^{\eta}/_{\rho}\}_{\mathsf{k}}.P \mid \blacksquare^{\mathsf{k}}_{\rho} \searrow P \mid \blacksquare^{\mathsf{k}}_{\eta}$$

Notice that this is a dangerous capability, since it allows processes to give particular names to anonymous slots, and for instance put in place possible malicious behaviours to make all public resources their own: $!\{^{y}/_{*}\}$. This suggests that in many situations one ought to restrict $\mathsf{k}\rhd_\eta$ to $\eta \in \mathcal{N}$. The inverse behaviour, that is a "communist for y spaces," is also well-formed and it is often useful (even though not commendable by everyone). Notice however that it can be harmful too: $!\{^{*}/_{y}\}$. We have not defined the name rebinding capability as a primitive of our calculus since it can be encoded using the new form of spawning as follows, for a fresh.

$$\{^{\eta}/_{\rho}\}_{\mathsf{k}}.P \triangleq (\boldsymbol{\nu} a)(\mathsf{k} \rhd_\rho (\blacksquare^{k}_{\eta} \mid a^0[\,\overline{\mathbf{opn}}\,]) \mid \mathbf{opn}\ a.P)$$

Observe that the simpler encoding $\mathsf{k} \rhd_\rho (\blacksquare^{k}_{\eta} \mid P)$ is allowed only for processes P of weight 0.

It is easy to check that the type system of Section 2 can be used without modifications also for the calculus with named slots. For this calculus the same properties proved in Section 2 hold.

Theorem 3 (Subject Reduction and Under/Over-Flow Absence). *For the processes and reduction relation of this section, we have:*

(i) $\Gamma \vdash P : Proc\langle \varepsilon, \chi \rangle$ *and* $P \searrow_* Q$ *imply* $\Gamma \vdash Q : Proc\langle \varepsilon', \chi \rangle$ *with* $\varepsilon' \le \varepsilon$.
(ii) *If* $\Gamma, a : Amb\langle [\mathsf{n}, \mathsf{N}], \chi \rangle \vdash P : Proc\langle \varepsilon, \chi \rangle$, $P \searrow_* C[a^{\mathsf{k}}[\,R\,]]$, *and the showed occurrence of a is not in the scope of a binder for a, then* $\mathsf{n} \le \mathsf{k} \le \mathsf{N}$.

3.2 More Examples

The Cab Trip Revisited. Named slots allow us to avoid unwanted behaviours when encoding the cab trip example. The *cab* initially contains one slot named *call*, but after reaching the client's site it will contain one slot with the client private name, and lastly when the *client* goes out of the cab it leaves one slot named *bye*. The *call* exiting the *client* leaves one slot tagged *forbye*, which is reserved for spawning *bye*. The (resource) renaming in the *site*s and in the *city* allow the public reuse of resources.
Let $W_1 = Amb\langle [1,1], 0_{\mathcal{E}} \rangle$, $W_0 = Amb\langle [0,0], 0_{\mathcal{E}} \rangle$. As in Subsection 2.3 the typing environment contains $call : W_1$, $cab : W_1$, $trip : W_0$, $unloading : W_0$, but $bye : W_1$ and we do not need the ambient *loading*.

$$
\begin{array}{ll}
CALL(from, client) & \triangleq call^1[\,\mathbf{out}\ client\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{in}\ from\,.\,\blacksquare_{client}\,]_{forbye} \\
TRIP(from, to, client) & \triangleq trip^0[\,\mathbf{out}\ client\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ to\,.\,unloading^0[\,\mathbf{in}\ client\,.\,\overline{\mathbf{opn}}\,]\,] \\
CLIENT(from, to) & \triangleq (\boldsymbol{\nu} c : W_1)c^1[CALL(from, c) \mid \mathbf{in}\ cab\,.\,TRIP(from, to, c) \\
 & \quad \mid \mathbf{opn}\ unloading\,.\,\mathbf{out}\ cab\,.\,1 \triangleright_{forbye} bye^1[\,\mathbf{out}\ c\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{out}\ to\,.\,\blacksquare_{call}\,]_*]_{bye} \\
CAB & \triangleq cab^1[\,\blacksquare_{call} \mid !(\mathbf{opn}\ call\,.\,\mathbf{opn}\ trip\,.\,\mathbf{opn}\ bye)\,]_* \\
SITE(i) & \triangleq \\
\quad site_i[\,CLIENT(site_i, site_j) \mid CLIENT(site_i, site_l) \mid \cdots \mid \blacksquare \mid \blacksquare \mid \cdots \mid !\{^*/_{forbye}\}_1 \mid !\{^*/_{bye}\}_1\,] & \\
CITY & \triangleq \\
 & city[\,CAB \mid CAB \mid \cdots \mid SITE(1) \mid \cdots \mid SITE(n) \mid \blacksquare \mid \blacksquare \mid \cdots \mid !\{^*/_{forbye}\}_1\,]
\end{array}
$$

A Travel Agency. We conclude the presentation with an example that shows the expressiveness of the naming mechanisms for resources in the refined calculus. We wish to model clients buying tickets from a travel agency, paying them one slot (the $\blacksquare_{fortkt}$ inside the client), and then use them to travel by plane. At most two clients may enter the travel agency, and they are served one by one. The three components of the systems are defined below.

- ▷ <u>THE AGENCY</u>: $ag^5[\,\blacksquare^2_{cl} \mid \blacksquare_{req} \mid$
  $\quad \mathrm{desk}^1[\,\blacksquare_{req} \mid !(\mathbf{opn}\ req\,.\,1 \triangleright_{fortkt}\,.\,tkt^1[\,\mathbf{out}\ desk\,.\,\mathbf{in}\ cl\,.\,CONT\,]_{req})]\,]$
  where $CONT = (\overline{\mathbf{opn}}\,.\,\mathbf{out}\ ag\,.\,\mathbf{in}\ plane\,.\,rdy^0[\,\mathbf{out}\ cl\,] \mid \blacksquare_{getoff} \mid \mathbf{opn}\ getoff)$
- ▷ <u>THE CLIENT</u>: $\mathrm{cl}^1[\,\mathbf{in}\ ag\,.\,req^1[\,\mathbf{out}\ cl\,.\,\mathbf{in}\ desk\,.\,\overline{\mathbf{opn}}\,.\,\blacksquare_{fortkt}\,]_{tkt} \mid \mathbf{opn}\ tkt\,]_{cl}$
- ▷ <u>THE AIRCRAFT</u>: $\mathrm{plane}[\,\blacksquare^2_{cl} \mid \mathbf{opn}\ rdy^0\,.\,\mathbf{opn}\ rdy^0\,.\,ROUTE\,.(GETOFF \mid GETOFF)\,]$
  where $GETOFF = \mathrm{getoff}^1[\,\mathbf{in}\ \mathrm{cl}\,.\,\overline{\mathbf{opn}}\,.\,\mathbf{out}\ plane \mid \blacksquare\,]$ and *ROUTE* is the unspecified path modelling the route of the aircraft.

We assume that there exists only one sort of ticket, but it is easy to extend the example with as many kinds of ticket as possible plane routes. What makes the example interesting is the possibility of letting two clients into the agency, but serving them non-deterministically in sequence. Notice that the use of the named slots is essential for a correct implementation of the protocol. When the request goes to the desk, a slot named *tkt* is left in the client. This slot allows the ticket to enter the client. In this way we guarantee that no ticket can enter a client before its request has reached the desk.

We assume the aircraft to leave only when full. This constraint is implemented by means of the *rdy* ambient. The ambient *getoff* enables the passengers to get off once at destination; assigning weight 1 to the *getoff* ambients prevents them to get both into the same client.

4 Conclusion and Future Work

We have presented an ambient-like calculus centred around an explicit primitive representing a resource unit: the space "slot" ▬. The calculus, dubbed BoCa, features capabilities for resource control, namely pairs **get/put** to transfer spaces between sibling ambients and from parent to child, as well as the capabilities **in** a and **out** a for ambient migration, which represent an abstract mechanism of resource negotiation between travelling agent and its source and destination environments. A fundamental ingredient of the calculus is $\triangleright(\_)$, a primitive which consumes space to activate processes. The combination of such elements makes of BoCa a suitable formalism, if initial, to study the role of resource consumption, and the corresponding safety guarantees, in the dynamics of mobile systems. We have experimented with the all important notion of private resource, which has guided our formulation of a refined version of the calculus featuring named resources.

The presence of the space construct ▬ induces a notion of weight on processes, and by exercising their transfer capabilities, processes may exchange resources with their surrounding context, so making it possible to have under- and over-filled ambients. We have introduced a type system which prevents such unwanted effects and guarantees that the contents of each ambient remain within its declared capacity.

As we mentioned in the Introduction, our approach is related to the work on *Controlled Mobile Ambients* (CMA) [17] and on *Finite Control Mobile Ambients* [3]. There are, however, important difference with respect to both approaches.

In CMA the notions of process weight and capacity are entirely characterized at the typing level, and so are the mechanisms for resource control (additional control on ambient behavior is achieved by means of a three-way synchronization for mobility, but that is essentially orthogonal to the mechanisms targeted at resource control). In BoCa, instead, we characterize the notions of space and resources directly in the calculus, by means of an explicit process constructor, and associated capabilities. In particular, the primitives for transferring space, and more generally for the explicit manipulation of space and resources by means of spawning and replication appear to be original to BoCa, and suitable for the development of formal analyses of the fundamental mechanism of the usage and and consumption of resources which do not seem to be possible for CMA.

As to [3], their main goal is to isolate an expressive fragment of Mobile Ambients for which the model checking problem against the ambient logic can be made decidable. Decidability requires guarantees of finiteness which in turn raise boundedness concerns that are related to those we have investigated here. However, a more thorough comparison between the two approaches deserves to be made and we leave it to our future work.

Plans for future include further work in several directions. A finer typing discipline could be put in place to regulate the behavior of processes in the presence of primitive notions of named slots. Also, the calculus certainly needs behavioral theories and proof techniques adequate for reasoning about resource usage and consumption. Such theories and techniques could be assisted by enhanced typing systems providing static guarantees of a controlled, and bounded, use of resources, along the lines of the work by Hofmann and Jost in [10].

A further direction for future development is to consider a version of weighed ambients whose "external" weight is independent of their "internal" weight, that is the weight of their contents. This approach sees an ambient as a packaging abstraction whose weight may have a different interpretation from that of contents'. For instance, modelling a wallet the weight of its contents could represent the value of the money inside, whereas its external weight could measure the physical space it occupies. A directory's internal weight could be the cumulative size of its files, while the external weight their number.

Last, but not least, we would like to identify logics for BoCa to formulate (quantitative) resource properties and analyses; and to model general resource bounds negotiation and enforcement in the Global Computing scenario.

Acknowledgements

We gratefully acknowledge the anonymous referees for careful reading and useful suggestions.

References

1. M. Bugliesi and G. Castagna. Secure safe ambients. In *POPL'01*, pages 222–235, New York, 2001. ACM Press.
2. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, D. Le Métayer Editor.
3. W. Charatonik, A. D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In D. Le Métayer, editor, *ESOP'02*, volume 2305 of *LNCS*, pages 295–313, Berlin, 2002. Springer-Verlag.
4. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*, pages 184–198, New York, 2000. ACM Press.
5. J. C. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 272–287, Berlin, 2002. Springer-Verlag.
6. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed system (extended abstract). In A. D. Gordon, editor, *FOSSACS'03*, volume 2620 of *LNCS*, pages 282–299, Berlin, 2003. Springer-Verlag.

7. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.
8. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
9. M. Hofmann. The strength of non size-increasing computation. In *POPL'02*, pages 260–269, New York, 2002. ACM Press.
10. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, pages 185–197, New York, 2003. ACM Press.
11. A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL'02*, pages 331–342, New York, 2002. ACM Press.
12. F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL'00*, pages 352–364. ACM Press, New York, 2000.
13. R. Milner and D. Sangiorgi. Barbed bisimulation. In W.Kuich, editor, *ICALP'92*, volume 623 of *LNCS*, pages 685–695, Berlin, 1992. Springer-Verlag.
14. R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.
15. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
16. D. Sangiorgi and A. Valente. A distributed abstract machine for safe ambients. In F. Orejas, P. Spirakis, and J. Leeuwen, editors, *ICALP'01*, volume 2076 of *LNCS*, pages 408–420, Berlin, 2001. Springer-Verlag.
17. D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 288–303, Berlin, 2002. Springer-Verlag.
18. N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order mobile processes (extended abstract). In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99*, volume 1664 of *LNCS*, pages 557–573, Berlin, 1999. Springer-Verlag.