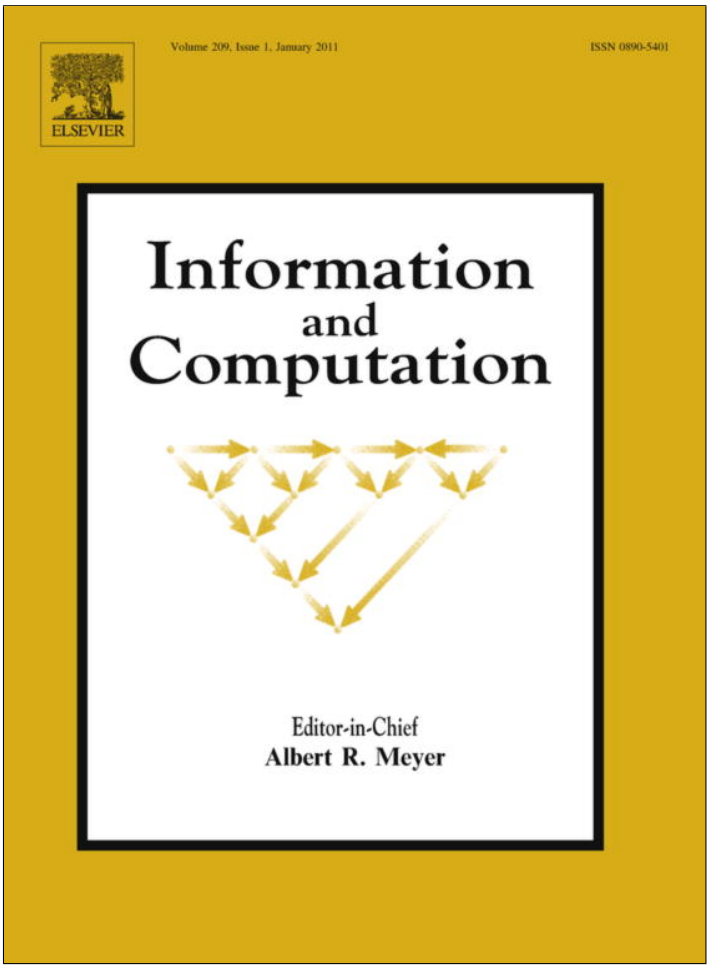

(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

Information and Computation 212 (2012) 64–91

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

An Algebra of Behavioural Types

António Ravara [a,*,1], Pedro Resende [b], Vasco T. Vasconcelos [c]

[a] Software Systems area at Center for Informatics and Information Technologies and Dep. of Informatics, FCT, Univ. Nova de Lisboa, Portugal
[b] Center for Mathematical Analysis, Geometry, and Dynamical Systems and Dep. of Mathematics, IST, Univ. Técnica de Lisboa, Portugal
[c] Group of Software Systems at Large-Scale Informatics Systems Laboratory and Dep. of Informatics, FC, Univ. de Lisboa, Portugal

ARTICLE INFO

Article history:
Received 6 June 2002
Revised 6 November 2011
Available online 25 January 2012

ABSTRACT

We propose a process algebra, the Algebra of Behavioural Types, as a language for typing concurrent objects. A type is a higher-order labelled transition system that characterises all possible life cycles of a concurrent object. States represent interfaces of objects; state transitions model the dynamic change of object interfaces. Moreover, a type provides an internal view of the objects that inhabits it: a synchronous one, since transitions correspond to message reception. To capture this internal view of objects we define a notion of bisimulation, strong on labels and weak on silent actions. We study several algebraic laws that characterise this equivalence, and obtain completeness results for image-finite types.

1. The role of types

Type systems in programming languages are used to discipline the computational mechanism of the language, ruling out program behaviours judge as erroneous. Examples of such errors are the application of a function with the wrong number of arguments, or the invocation of a non-existing method in an object. A type system is a collection of axiom schemas and inference rules, and acts as a proof system, guaranteeing the absence of erroneous program behaviours. Therefore, types are abstract representations of the correct behaviour of the various entities of a program, constituting partial behavioural specifications.[2]

To ensure the absence of a particular form of bad program behaviour, *i.e.* a specific safety property, a good notion of type is an important ingredient. Our aim is a language of types capable of expressing behavioural aspects of computing entities like objects. This language should be expressible enough to be used in (decidable) proof systems for ensuring statically not only safety properties of such entities, but also some (limited, although interesting) liveness properties.

A programming language is *type safe* if it is equipped with a (static) type system that guarantees the absence of run-time errors in well-typed programs. This important safety property may be obtained combining two properties:

1. the *absence of run-time errors* in well-typed programs; and
2. Curry's *Subject-Reduction*, which ensures that if a program is typable, then the computation mechanism preserves the typability of all the programs resulting from the intermediate steps.

In sequential and functional languages, types are assigned to the terms of the language. The information that a type describes can be very simple (a set of values, booleans or integers), more elaborate (a function, from integers to booleans,

* Corresponding author.
E-mail addresses: aravara@fct.unl.pt (A. Ravara), pmr@math.ist.utl.pt (P. Resende), vv@di.fc.ul.pt (V. T. Vasconcelos).

[1] Research reported herein done while the author was at the Dep. of Mathematics, IST, Univ. Técnica de Lisboa, Portugal.

[2] Readers interested in a general introduction to this issue should consult Pierce's thorough work [48].

0890-5401/$ – see front matter
doi:10.1016/j.ic.2011.12.005

for example), or can even be a complex structure (a graph or a term of a process algebra), depending on the purpose of the type system. Type systems can ensure a wide range of properties, from basic, like all operations are invoked with the adequate arguments, to more elaborate, like guaranteeing termination or deadlock-freedom.

In systems of objects, types usually record the methods of objects and the types of its parameters, constituting interfaces for these objects. In a programming language with objects, a type system should prevent the usually known as '*method-not-understood*' error, a run-time error due to the erroneous call of a method at the target object (non-existence or wrong number of arguments passed).

1.1. Types in a concurrent scenario

To ensure a safety result for a given program, one needs mathematical tools to deal with, and reason about, the behaviour of such program. Ideas, concepts, and techniques from the typed λ-calculus and from (name-passing) process calculi have been successfully applied to the study of behavioural properties and of type systems for concurrent object-oriented languages. A calculus of mobile—or name-passing—processes is one where the communication topology changes dynamically. Processes communicate via channels—called names—and may also exchange names during the interaction, acquiring new acquaintances that they can use for further communications. The precursor and paradigmatic case is the π-calculus of Milner, Parrow and Walker [39].

As a process algebra, one may use a mobile calculus not only to specify (concurrent) systems, but also to verify properties of those systems using the rich algebraic theory that such a calculus possesses. On one hand, its features, like referencing—or naming—and scoping, make process calculi approaches suitable for describing and studying object-oriented programming. Thus, not surprisingly, there are many works on the semantics of (concurrent) objects as (mobile) processes (a brief synopsis may be found in [45]). On the other hand, process calculi provide: (1) structural operational semantics—an essential element for describing the operational behaviour of programs; (2) various static type systems—ensuring the absence of run-time errors in well-typed programs; and (3) several notions of behavioural equivalences together with proof techniques, algebraic laws, and logical characterisations—providing tools to reason about properties of programs.

In mobile calculi, types are usually assigned to names, constituting a discipline for communication: they determine the arity of a name (and, in some systems, its directionality—input or output), and recursively, the arity of the names carried by that name [38,49,67]. The roles of a type system in a mobile calculus are two-fold: (1) it avoids communication errors, due to arity-mismatch; and (2) it allows refinements on the algebraic theory, leading to specialised behavioural equivalences.

Nonetheless, the referred systems provide little information about process behaviour, and to ensure more than the usual safety properties one needs richer notions of types, able of capturing the information flow within the processes. A natural approach is to consider processes as types, as for instance done in [7,13,23,28–30,53,71].

1.2. Typing non-uniform objects

Objects exhibiting methods that may be enabled or disabled according to their internal state—*non-uniform objects*—are very common in object-oriented programming. Simple examples are a stack (from which one cannot pop elements if it is empty); a finite buffer (where one cannot write if it is full); a cash machine (from which one can only get a balance if the connection with the bank is enabled).

A static notion of typing, as interfaces-as-types, is not powerful enough to capture this kind of dynamic properties of the behaviour of objects. A rigid interface, exhibiting all methods of an object, gives misleading information about the functionality of the object. In the beginning of the 90s, Nierstrasz proposes the use of a regular language to type active objects, *i.e.*, objects that may dynamically change behaviour [47]. The purpose is to characterise all the traces of the menus offered by objects and to define a notion of behavioural subtyping. Although the idea applies to a sequential setting, the work on this topic has mainly been developed for concurrent objects. Notice however that the type theory developed herein can be used in sequential object-oriented languages (almost) straightforwardly.

Concurrent setting. In a name-passing calculus of objects such as TyCO [68] or π_a^V [59], processes denote the behaviour of a community of interacting objects, where each object has a location identified by a name. As in the π-calculus, processes determine an assignment of types to names that reflects a discipline for communication.

Statically detecting '*method-not-understood*' errors is a more delicate problem in systems of (possibly distributed) concurrent objects, since the enabling conditions of methods are harder to verify in this scenario. The usual records-as-types paradigm gives each name a static type that contains information about all the methods of the object, regardless of whether they are enabled or not. Is this an adequate notion of type of an object, in the presence of concurrency? Nierstrasz argued that typing concurrent objects posed particular problems, due to the '*non-uniform service availability of concurrent objects*'. By synchronisation constraints, the availability of a service depends upon the internal state of the object (which reflects the state of the system). Despite having developed a calculus of objects, Nierstrasz did not apply these ideas in the form of a type system for it, neither did he show how to model non-uniform objects. This task has then been taken by several authors [7,20–22,42,44,50,52,51,55,56]. Most of these works propose a specific calculus, and develop a particular language of types aiming at guaranteeing some envisaged property. Herein we study the semantics of a language of types designed to cope with behavioural aspects of non-uniform objects, using in this study process algebraic tools and results. At the end of this paper we compare the referred works with our own.

1.3. Rationale of our approach

The purpose of this work is to study the *semantic foundations of types for concurrent objects*, using the tools and the body of knowledge of the theory of process algebras. We propose a *process algebra of types*, discuss a notion of equivalence, and study its algebraic properties.

To capture the behaviour of non-uniform objects, we advocate the use of non-uniform types, types that are themselves modelled as processes. Then, several questions arise: What is the appropriate syntax and operational semantics? What is a good notion of behavioural equality? Our purpose is to address theses questions: we develop herein the *Algebra of Behavioural Types*, ABT, where a type characterises all possible life cycles of an object. A type (a term of ABT) is basically a collection of enabled methods (an interface); such a type is dynamic in the sense that the execution of a method can change it—thus, a type may express temporal properties as ordering sequences of events, and reflects a dependency of the interface of an object upon its internal state. Hence, the type of an object is a partial representation of its behaviour, modelled as a labelled transition system: states represent interfaces of objects; state transitions model the dynamic change of object interfaces.

Therefore, apart from sequencing, other operators of process algebra naturally express behaviour common to objects: sum represents an object interface; parallel composition represents communities of objects. Types are thus built with the following operators:

1. non-deterministic labelled sum, denoting the collection of methods that an object offers at a given state;
2. blocking υ, expressing that a collection of methods is not currently enabled;
3. parallel composition, merging types[3];
4. recursion via a least fixed-point.

The operational semantics describes how the "execution" of a method yields a new type; the equivalence notion equates behaviourally similar objects.

1.4. A paradigmatic example

Consider a one-place buffer, where one may write an integer value if the buffer is empty, and from where one may read a value, providing a return address, if the buffer is full. An interface type for that buffer would look like [*read*:*nam*, *write*:*int*]. This type provides no information on the right order of calls the object may attend. In TyCO, the buffer may be simply written as follows.[4]

$$\textbf{def}\ \mathsf{Empty}(b) = b?\{write(u) = \mathsf{Full}\langle b, u\rangle\},$$
$$\textbf{and}\ \mathsf{Full}(b, u) = b?\{read(r) = (r!val\langle u\rangle \mid \mathsf{Empty}\langle b\rangle)\}.$$

The buffer alternates between an Empty state, where it only allows **write** operations, and a Full state, where it only allows **read** operations. In each state the object waits (at name b) for requests of the method offered (**write** or **read**), requests passing as argument the value to store (in the case of **write**) or the return address (r) where the client waits for the value that was stored. Notice that b is the identity of the object, following the '?' sign one finds a set of methods, each with a name, a list of parameters and a body (after the '=' sign). A message (or method call) like $r!val\langle u\rangle$ indicates the identity of the called object (r), the name of the method (**val**), and the argument of the call (u). The vertical bar ('|') denotes parallel composition. It is worth noticing that the calculus is asynchronous: the method call is non-blocking and, in the example, is in parallel with the object in state Empty (after the consumption of the value in the buffer). A behavioural type clearly expresses this ordering: $\mu t.write(int).read(nam).t$.

Elsewhere we use ABT as syntactic types for non-uniform concurrent objects in TyCO. We formalise a notion of process with a communication error that copes with non-uniform service availability, and define a type system that assigns terms of the type algebra to names occurring in processes. This system enjoys the subject-reduction property, and guarantee that typable processes do not deadlock or run into errors [56].

1.5. A novel behavioural equivalence

We assume that objects communicate via asynchronous message-passing; nevertheless, types, as defined here, essentially correspond to a notion of object behaviour as it would be perceived by an internal observer located within an object (the object's private "gnome"). This observer can see methods being invoked and can detect whether the object is blocked, even though its methods may be enabled for self calls. Therefore, this notion of behaviour is synchronous, as the gnome can

[3] Parallel composition is a merge operator since the operands—types—do not communicate among themselves.

[4] Available for http://www.dcc.fc.up.pt/~tyco/. This version is not typable by the current type system of TyCO, which uses an interface-like notion of types. The problem is the change in the interface. A workaround is a busy-waiting implementation, less readable and less natural.

detect refusals of methods when they are not enabled for outside calls. The action of unblocking a method, denoted by υ, corresponds to an invocation of a method in another object. Thus, this action is similar to CCS's τ in that it is hidden, but it is external rather than internal [36].

To illustrate what we mean by 'blocked method', consider a buffer that copies the stored value to a stack. Thus, after accepting a **write** operation, the buffer performs an internal operation—inserts the value in a stack s—before allowing **read** operations. The method **read** is blocked until receiving a message acknowledging the insertion of the value.

$$\begin{aligned}(\nu s)\big(\ &\textbf{def}\ \mathsf{Empty}(b) = b?\big\{write(u) = (\nu c)\big(s!push\langle u, c\rangle \mid c?\big\{done = \mathsf{Full}\langle b, u\rangle\big\}\big)\big\}\\ &\textbf{and}\ \mathsf{Full}(b, u) = b?\big\{read(r) = r!val\langle u\rangle \mid \mathsf{Empty}\langle b\rangle\big\}\\ &\mid \mathsf{Stack}(s)\ \big).\end{aligned}$$

A behavioural type describing this buffer should now take into account that a **read** operation is not immediately available, depending on some computation taking place in other object (pushing the value; sending the acknowledgement message). Thus, a possible type is $\mu t.write(int).\upsilon.read(nam).t$.

The unblocking action is also useful to express internal non-determinism. Consider the following object.

$$a?\big\{l = (\nu\, c)\ \big((c!m_1 \mid c!m_2) \mid c?\big\{m_1 = a?\big\{l_1\big\}, m_2 = a?\{l_2\}\big\}\big)\big\}.$$

The type of a expresses the internal choice: $l.(\upsilon.l_1 + \upsilon.l_2)$.

Naturally, the resulting notion of equivalence has an intuition different from that of bisimulation in CCS, since the observer is internal, rather than external:

1. distinguishes a blocked from an unblocked type

$$read(nam) \not\approx \upsilon.read(nam), \quad \text{hence}$$
$$\mu t.write(int).read(nam).t \not\approx \mu t.write(int).\upsilon.read(nam).t,$$

2. does not count blockings

$$\mu t.write(int).\upsilon.read(nam).t \approx \mu t.write(int).\upsilon.\upsilon.read(nam).t.$$

The nature of the silent action—unblocking—induces an original behavioural equivalence notion. To capture the referred internal view of objects, we define a notion of *bisimulation, strong on labels and weak on silent actions*. Naturally, we reach a new set of algebraic laws, different from those we are aware of, in particular different from Milner's τ-laws.

An object o with methods l and m simultaneously available, as $o?\{l = P, m = Q\}$, would be described by the ABT type $l + m$, assuming that o does not occur free in P and Q. Similarly, two objects sharing the same reference o, one with a single method l and the other with a single method m, as $o?\{l = P\} \mid o?\{m = Q\}$, would be described by the type $l \parallel m$. This type also characterises a parallel composition of messages targeting the same object as $o!l \mid o!m$. As usual, the behaviour of this object system should not be distinguishable from that of $o?\{l = P \mid o?\{m = Q\}, m = Q \mid o?\{l = P\}\}$, described by the type $l.m + m.l$. Hence, an expansion law holds.

However, one must define carefully this expansion. Since terms of the algebra are intended to be types of objects, it is useful to rule out meaningless terms: a type as $\upsilon + l$ is not an object interface, *i.e.*, the collection of its enabled method names. To interpret sums as interfaces, mixed sums should not be allowed. Therefore, we do not want to express the type $l \parallel \upsilon.m$ as $l.\upsilon.m + \upsilon.(l \parallel m)$, since no object interface would be described by such a type. Moreover, the former type gives a more precise information: one may request method l; in the environment there is another instance of the object providing a method m, which is however unavailable. The technical work developed herein is simplified by this assumption.

Assuming object interfaces as collections of enabled method names, the rationale for this interpretation of types of objects is:

> *labelled sums denote interfaces*; *blocked sums denote currently unavailable objects that after unblocking offer an interface* (*a menu of options*).

In conclusion, the expansion law should not generate mixed sums. This choice leads to a novel setting, not studied before in process algebra, since parallel composition usually enjoys expansion.

The problem of axiomatising our equivalence notion without eliminating the parallel composition operator is also interesting from a mathematical point of view: we are not aware of any axiomatic system for a process algebra where the parallel is not reduced to sum. We develop a proof system, based on these laws, which is complete with respect to the notion of bisimulation, at least for image-finite types. Notice that the absence of mixed sums in ABT leads to a simpler axiomatic system than that of CCS.

Applications. The type theory developed herein allows to reason about objects using an abstract representation of their behaviour. Since an ABT term describes all the possible sequences of method calls for a particular type, and the equivalence notion semantically characterises its behaviour, one may not only prove properties using equational reasoning, but also perform program optimisations by transforming the type (*e.g.*, the normal form of the type indicates its simplest syntactic form).

1.6. Summary

A process algebra is a natural choice when the aim is to use behavioural types to statically enforce some behavioural properties of systems. We show herein that a simple process algebra—ABT—may be used to cope with non-uniform concurrent objects. Since ABT is used as a language of types for concurrent objects, we tailored it *not* to be Turing powerful, so we may envisage decidable simulation and bisimulation relations—such results would be useful to develop type checking and inference algorithms.

ABT is similar to the Basic Parallel Processes, BPP [14], a fragment of CCS proposed by Christensen where communication is not present—parallel composition is simply a merge of processes. The differences are basically three. In ABT: (1) all sums are prefixed; (2) mixed sums (with labels and silent actions) are not allowed; and (3) the silent action represents activity external, rather than internal, to the process. Since these items correspond to our main criteria for the envisaged notion of type, to avoid confusions instead of using BPP, we decided to design a new process algebra.

We construct ABT gradually. The next section presents finite types built with a non-deterministic labelled sum and a blocking operator; then defines the operational semantics and a novel equivalence notion; and finally provides a complete axiomatisation for that equivalence. In Section 3 we add a parallel composition operator, this operator being a merge of processes (*i.e.* without communication), and extend the previous axiomatic system with two expansion laws—consequence of the absence of mixed sums—and a saturation law. Since normal forms include parallel compositions, the proof of completeness of the axiomatic system is not standard, and the result depends on a new inference rule that we add to the proof system. The rule does not seem to be derivable. Finally, Section 4 presents the full algebra, with dynamic types obtained by a recursive constructor. The axiomatic system contains three more laws to deal with recursion. In Section 5 we prove the completeness of the axiomatic system for image-finite terms. The paper closes with comparisons with related work and with some directions for future research, namely on a modal logic and on a notion of subtyping.

Contributions. In short, the novelties introduced are the following.

1. A very simple process language, yet expressive enough to capture the behaviour of (non-uniform) concurrent objects: a term represents sequences of method offers.
2. An original notion of behavioural equivalence for (non-uniform) concurrent objects, to our knowledge the only one proposed so far. This novel notion was neither proposed nor studied before.
3. An axiomatisation of the equivalence notion, where parallel composition is not reduced to choice, along the lines of the works on spatial, and on separation, logic, treating the parallel operator as separating resources (different instances of an object).
4. The axiomatisation is sound and complete (for image-finite terms). Soundness is proved for full language. Completeness does not restrict the language to sequential terms, as in most process algebras (including CCS), demanding only image-finiteness.

2. Non-deterministic finite types

We start by presenting an algebra of *non-deterministic sequential finite types*. The basic term is an *object type*, a labelled sum that denotes an object *interface*—the collection of the names of the methods offered by an object. As we allow the same label to appear more than once (possibly with different continuations), the sum is non-deterministic. This fact makes possible the definition of an expansion law later on when we introduce a parallel composition operator. When an object is in a state where its methods are disabled, its type reflects the situation: unavailable, or *blocked*, object types are types prefixed by a blocking operator, denoted by υ. A sum of blocked object types—a blocked sum—represents the possible types of an object, after becoming enabled. Hence, the silent transition is labelled with υ and corresponds to the release—or unblocking—of the blocked sum due to some action in another object: it is an *inter-object choice* that makes available one of the types in the sum. Thus, it should be interpreted as an action that is external to the object.

The intended meaning of a labelled sum and of a blocked sum clarify that it does not make sense to allow mixed sums: we want to distinguish an object that is enabled and offers a certain collection of methods from one it is blocked. Furthermore, we did not find in the literature of process algebra any equivalence notion build on this intuition. Therefore, we develop a notion of type equivalence accordingly to the requirements explained above. This fact leads to axiomatic systems that are not standard and require new proof techniques.

2.1. Syntax

Assume a countable set of *method names*, denoted by l, m, possibly subscripted.

Definition 2.1 *(Non-deterministic sequential finite types).* The grammar below defines the set $\mathcal{T}_{\mathbf{sf}}$ of *sequential finite types*.

$$\alpha ::= \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \mid \sum_{i\in I} \upsilon.\alpha_i$$

where I is a finite, possibly empty, indexing set, and each $\widetilde{\alpha}_i$ is a finite sequence of types.

A term of the form $l(\widetilde{\alpha}).\alpha$ is a *method type*. The label l in the prefix stands for the name of a method expecting arguments of types $\widetilde{\alpha}$; the type α under the prefix prescribes the behaviour of the object after the execution of the method l. A term of the form $\upsilon.l(\widetilde{\alpha}).\alpha$ is a *blocked method type*, the type of an unavailable method type $l(\widetilde{\alpha}).\alpha$. The term $\upsilon.\alpha$ denotes thus a blocked type.

The only type composition operator of the algebra is the sum, '$\sum$', which has two uses:

1. gathers together several method types to form the type of an object that offers the corresponding collection of methods: the *labelled sum* $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$;
2. associates several blocked types in the *blocked sum* $\sum_{i\in I} \upsilon.\alpha_i$; after being unblocked, the object behaves according to one of the types α_i.

Notation. We write $\alpha \equiv \beta$ when the types α and β are *syntactically identical*. We consider also the following abbreviations:

1. the term $\mathbf{0}$ denotes the empty type (sum with empty indexing set); we omit the sum symbol if the indexing set is singular, and we use the plus ('+') to denote binary sums of types, which we assume associative;
2. the term $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$.

This simple language has sufficient ingredients to support the specification of non-deterministic finite behaviours. To illustrate the use of the language, we progressively develop a running example throughout the paper, the specification of an Automatic Teller Machine (ATM, for short). In this section we present a finite ATM, *i.e.*, a machine allowing only a finite number of interactions.

Example 2.2. In its initial state, the ATM offers a *welcome* method that waits for two values—card number and pin. The data provided by the user is validated by the bank while the user waits—activity denoted by the first υ—and depending on the bank's reply—again not visible to the user—either the interaction is refused (method *sorry*) or the user selects one of the operations offered: *balance*, *deposit*, or *withdraw*.

$$\mathsf{fATM} \stackrel{\text{def}}{=} \mathit{welcome}(\mathit{int},\mathit{int}).\upsilon.(\upsilon.\mathit{sorry} + \upsilon.\mathsf{Menu}), \quad \text{where}$$
$$\mathsf{Menu} \stackrel{\text{def}}{=} \mathit{balance} + \mathit{deposit}(\mathit{int}) + \mathit{withdraw}(\mathit{int}).$$

2.2. Operational semantics

A labelled transition relation on types defines the structural operational semantics for non-deterministic sequential finite types.

Definition 2.3 *(Actions).* The following grammar defines the set of *actions*:

$$\pi ::= \upsilon \mid l(\widetilde{\alpha}).$$

Action υ denotes a silent transition that releases a blocked object; an action $l(\widetilde{\alpha})$ denotes a transition corresponding to the invocation of method l with actual parameters of types $\widetilde{\alpha}$. When occurring in sums, we refer to actions as prefixes. We write $\sum_{i\in I} \pi_i.\alpha_i$ to refer to an arbitrary sum, either with prefixes $l_i(\widetilde{\alpha}_i)$—*labelled*—or with prefix υ—*blocked*.

Definition 2.4 *(Labelled transition relation).* The following rule inductively defines a *labelled transition relation* on $\mathcal{T}_{\mathbf{sf}}$.

$$\textsc{Act} \quad \sum_{i\in I} \pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I).$$

Act is in fact an axiom schema that captures two cases:

1. the labelled transition $l(\widetilde{\alpha})$—*execution of a method*—corresponds to the invocation of a method with name l with arguments of types $\widetilde{\alpha}$, yielding the type α of the object in the method body;
2. the silent transition υ—*unblocking*—releases a blocked sum.

Notation. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^{*}$ (the reflexive and transitive closure of $\xrightarrow{\upsilon}$), and let $\overset{\upsilon}{\Longrightarrow}$ denote $\xrightarrow{\upsilon}{}^{+}$ (the transitive closure of $\xrightarrow{\upsilon}$).

Terminology. The following terminology regarding the transition relation simplifies the presentation of some proofs.

1. If $\alpha \xrightarrow{\pi} \alpha'$ then the type α' is an *immediate derivative* of the type α.
   When $\pi = l(\widetilde{\alpha})$ we say the transition is *labelled* and α' is an l-derivative of α.
   When $\pi = \upsilon$ we say the transition is *silent* and α' is an υ-derivative of α.
2. If $\alpha \overset{\widetilde{\pi}}{\Longrightarrow} \alpha'$ then the type α' is a *derivative* of the type α.
3. A type is
   (a) *blocked*, if it only has immediate υ-derivatives, and is *strictly blocked* if all its derivatives are *blocked*;
   (b) *unblocked*, if it has at least an immediate l-derivative, and is *strictly unblocked* if all its derivatives are *unblocked*;
   (c) *inert*, if it has no transitions.[5]

2.3. Notion of equivalence

We want two types to be equivalent if they offer the same methods—have the same interface—and if, after each transition, they continue to be equivalent, in a bisimulation style. Furthermore, from the point of view of each type, transitions of other types can be regarded as hidden transitions, which would suggest weak bisimulation as the right notion of equivalence for our types, with υ playing the role of Milner's τ, but representing external interaction rather than internal. However, we want types to distinguish an object that immediately makes available a method, from another that makes it available only after being unblocked (by some object). Therefore, υ should be externally unobservable—unobservable from the outside of an object, but internally observable—an internal observer should detect that the object is blocked. Hence, we would expect $\upsilon.l$ to be different from l, since all the internal observer can see is that the object is blocked, and after being released it can eventually execute the method l. This discards weak bisimulation as a candidate for type equivalence. Furthermore, we want $\upsilon.l$ and $\upsilon.\upsilon.l$ to be equivalent, because the number of unblockings cannot be counted from within the object as they correspond to transitions on other objects, thus discarding strong bisimulation [36] and progressing bisimulation [41]. We also want to distinguish $l.\upsilon.m$ from $l.m$ on the grounds that, for the latter, a blocking after l cannot be observed, and thus observational congruence [36] and rooted bisimulation [2] are unsuitable. Also, notice that all the above mentioned equivalences, with the exception of weak bisimulation, are finer than what we need, because they are congruences with respect to binary sums, as in CCS [36], whereas in this work we stick to prefixed sums.

These considerations lead to the choice of a notion of equivalence that we call *label-strong bisimulation*, or **lsb**. It is a higher-order *strong* bisimulation *on labels* and a *weak* bisimulation *on unblockings*.

Hence, we require that if α and β are bisimilar then:

1. if α offers a particular method, then also β offers that method, and the parameters and the bodies of the methods are pairwise bisimilar;
2. if α offers a hidden transition, then β can offer zero or more hidden transitions.

Conversely, the intuition is that two types are *not bisimilar* if they have different *interfaces* (set of the outermost labels of a labelled sum), possibly after some matching transitions.

In Section 6.2 (p. 85) we further discuss the choices leading to this particular notion.

Definition 2.5 *(Bisimilarity on types).*

1. A symmetric binary relation $R \subseteq \mathcal{T}_{\mathbf{sf}} \times \mathcal{T}_{\mathbf{sf}}$ is a *label-strong bisimulation*, or simply a bisimulation, if, whenever $\alpha \mathrel{R} \beta$:
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \widetilde{\alpha} \mathrel{R} \beta' \widetilde{\beta})$[6];
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists \beta'$ $(\beta \Longrightarrow \beta'$ and $\alpha' \mathrel{R} \beta')$.
2. Two types α and β are *label-strong bisimilar*, or simply bisimilar, and we write $\alpha \approx \beta$, if there is a label-strong bisimulation R such that $\alpha \mathrel{R} \beta$.

The usual properties of a bisimilarity hold: it is an *equivalence relation*, the *largest bisimulation*, and a *greatest fixed-point*. The proofs of these results are standard (*cf.* [36, Propositions 4.2 and 4.16]). The following characterisation of **lsb** is useful.

[5] If a type α is inert, then $\alpha \equiv \mathbf{0}$.

[6] Let $(\alpha_1 \cdots \alpha_n) \mathrel{R} (\beta_1 \cdots \beta_n)$ denote $\alpha_1 \mathrel{R} \beta_1 \wedge \cdots \wedge \alpha_n \mathrel{R} \beta_n$.

Proposition 2.6 *(Label-strong bisimilarity). Types α and β are bisimilar, if, and only if,*

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}\ (\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha' \widetilde{\alpha} \approx \beta' \widetilde{\beta})$;
2. $\alpha \Longrightarrow \alpha'$ *implies* $\exists \beta'\ (\beta \Longrightarrow \beta'$ *and* $\alpha' \approx \beta')$.

Proof. Notice that $\alpha \Longrightarrow \alpha'$ means $\alpha \xrightarrow{\upsilon}{}^{n} \alpha'$, for some natural number n. The proof is by induction on n.[7] □

The sum of method types and the sum of blocked sums preserve bisimilarity. This result is simple to verify, since both sums are guarded.

Proposition 2.7 *(Congruence). Let $\widetilde{\alpha_i} \approx \widetilde{\beta_i}$ and $\alpha_i \approx \beta_i$ for all i in an indexing set I.*

1. $\sum_{i\in I} l_i(\widetilde{\alpha_i}).\alpha_i \approx \sum_{i\in I} l_i(\widetilde{\beta_i}).\beta_i$, *and*
2. $\sum_{i\in I} \upsilon.\alpha_i \approx \sum_{i\in I} \upsilon.\beta_i$.

Briefly, **lsb** is a congruence relation with respect to prefixing and summation.

Proof. Proving that labelled sums and blocked sums preserve **lsb** is a direct application of Definition 2.5, and of the fixed-point property of **lsb**. □

2.4. Algebraic characterisation

We present an axiomatisation of the equivalence notion and show that it is sound and complete. The proof scheme for completeness is standard: a definition of a normal form for the types; a lemma ensuring that for all types there exists an equivalent term in normal form; and finally the completeness theorem says that for all pairs of equivalent normal forms there exists a derivation of their equality using the rules of the axiomatic system. However, the proofs differ from those in the literature, since the particular syntactic conditions and restrictions of ABT make these proofs an elaborate combinatoric problem.

Prop/Definition 2.8 *(Axiomatisation). The following equivalences, sound with respect to* **lsb***, inductively define the* axiomatic system $\mathcal{A}_{\mathbf{sf}}$.

Commutativity: *for any permutation*[8] $\sigma : I \to I$ *we have* $\sum_{i\in I} \pi_i.\alpha_i = \sum_{i\in I} \pi_{\sigma(i)}.\alpha_{\sigma(i)}$;
Idempotence: $\pi.\alpha + \pi.\alpha + \beta = \pi.\alpha + \beta$;
U1: $\upsilon.\sum_{i\in I} \upsilon.\alpha_i = \sum_{i\in I} \upsilon.\alpha_i$.

Proof. It is straightforward to build the respective bisimulations. □

Example 2.9. Using the υ-law **U1**, one may simplify the specification of the finite ATM (*cf.* Example 2.2).

$$\mathit{welcome}(\mathit{int},\mathit{int}).\upsilon.(\upsilon.\mathit{sorry} + \upsilon.\mathsf{Menu}) = \mathit{welcome}(\mathit{int},\mathit{int}).(\upsilon.\mathit{sorry} + \upsilon.\mathsf{Menu}).$$

Notation. We write $\vdash \alpha = \beta$ when we can prove $\alpha \approx \beta$ using the laws above and the usual rules of equational logic.

Theorem 2.10 *(Soundness of $\mathcal{A}_{\mathbf{sf}}$). If $\vdash \alpha = \beta$ then $\alpha \approx \beta$.*

Proof. Follows from Proposition 2.7 and of Prop/Definition 2.8. □

Remark. The novelty of this system is the υ-law **U1**: one can observe if a method is enabled or not, but in the latter case, one cannot count how many unblockings must occur to enable the method. Notice some more facts about this law:

1. The proof of completeness uses two derived rules.
   (a) If $\forall_{i\in I}\exists_{j\in J}\ \vdash \alpha_i = \beta_j$ and $\forall_{j\in J}\exists_{i\in I}\ \vdash \alpha_i = \beta_j$, then $\vdash \sum_{i\in I} \upsilon.\alpha_i = \sum_{j\in J} \upsilon.\beta_j$.
   (b) If $\forall_{i\in I}\exists_{j\in J}\ (l_i \equiv m_j,\ \vdash \widetilde{\alpha_i} = \widetilde{\beta_j}$, and $\vdash \alpha_i = \beta_j)$
   and $\forall_{j\in J}\exists_{i\in I}\ (l_i \equiv m_j,\ \vdash \widetilde{\alpha_i} = \widetilde{\beta_j}$, and $\vdash \alpha_i = \beta_j)$,
   then $\vdash \sum_{i\in I} l_i(\widetilde{\alpha_i}).\alpha_i = \sum_{j\in J} m_j(\widetilde{\beta_j}).\beta_j$.

[7] This proof technique is known as "transition induction"—it is an induction on the maximum length of the derivation of the transition.
[8] A permutation is a bijection of a set into itself.

The soundness of these rules is a consequence of Proposition 2.7, but it results also from the fact that they are derived rules (the proof is simple). Therefore, the proof system without them is still complete.

2. An interesting instance of the υ-law is that $\alpha \approx \mathbf{0}$, if α is a blocked sum with all its derivatives being also blocked sums (*i.e.* α is a tree with all branches labelled by υ).
3. The υ-law corresponds to an instance of the CCS's first τ-law, $\alpha.\tau.P = \alpha.P$, when α is τ (υ, in our case).
   One can easily recognise particular instances of the remaining τ-laws of CCS that hold in this setting. For example, the following derivable laws are instances of the second and third τ-laws ($P + \tau.P = \tau.P$ and $\alpha.(P + \tau.Q) = \alpha.(P + \tau.Q) + \alpha.Q$ respectively).
   (a) $\upsilon.P + \upsilon.\upsilon.P = \upsilon.\upsilon.P$.
   (b) $\upsilon.(\upsilon.P + \upsilon.Q) = \upsilon.(\upsilon.P + \upsilon.Q) + \upsilon.Q$.
   The first clause is easy to prove, since $\upsilon.\upsilon.P = \upsilon.P$; to prove the second clause use first the υ-law, then idempotence, and then again the υ-law.
   However, the τ-laws do not hold in general in this setting; for instance, the third τ-law does not hold, as the following counterexample shows:

$$l.(\upsilon.m + \upsilon.n) \not\approx l.(\upsilon.m + \upsilon.n) + l.n.$$

   After an l-transition, the right-hand side offers an n-transition, which is not available in the left-hand side.

From these remarks it is easy to conclude that both weak bisimulation, which is a congruence for prefixed sums, and observation congruence are coarser than **lsb**, when considered in this setting.

We proceed now towards the completeness result for the axiomatic system, with respect to **lsb**. The proof technique to establish the result uses the notion of depth of a type.

Definition 2.11 *(Depth of a type).* The following rules inductively define the *depth of a type*.

$$\mathsf{depth}(\mathbf{0}) = 0,$$
$$\mathsf{depth}\Big(\sum_{i\in I} \upsilon.\alpha_i\Big) = 1 + \max\big\{\mathsf{depth}(\alpha_i) \mid i \in I\big\}, \quad \text{and}$$
$$\mathsf{depth}\Big(\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i\Big) = 1 + \max\big\{\mathsf{depth}(\widetilde{\alpha}_i) + \mathsf{depth}(\alpha_i) \mid i \in I\big\},$$

where $\mathsf{depth}(\widetilde{\alpha}) = \max\{\mathsf{depth}(\beta) \mid \beta \in \widetilde{\alpha}\}$.

Theorem 2.12 *(Completeness of $\mathcal{A}_{\mathbf{sf}}$). If $\alpha \approx \beta$ then $\vdash \alpha = \beta$.*

Proof. By induction on the sum of the depths of the types α and β.

Base case: $\mathsf{depth}(\alpha) = \mathsf{depth}(\beta) = 0$; by definition of depth, $\alpha \equiv \mathbf{0} \equiv \beta$.
It follows from the reflexivity law of the proof system that $\vdash \alpha = \beta$.
Induction step: there are two cases to consider.

1. Case α is a labelled sum, and hence also β is a labelled sum, as $\alpha \approx \beta$.
   Thus, if $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ then $\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha'\widetilde{\alpha} \approx \beta'\widetilde{\beta}$.
   Since $\mathsf{depth}(\alpha'\widetilde{\alpha}) + \mathsf{depth}(\beta'\widetilde{\beta}) \leqslant \mathsf{depth}(\alpha) + \mathsf{depth}(\beta)$, it follows by the induction hypothesis that $\vdash \alpha'\widetilde{\alpha} = \beta'\widetilde{\beta}$.
2. Case α is a blocked sum, and hence also β is a blocked sum, as $\alpha \approx \beta$.
   Thus, if $\alpha \xrightarrow{\upsilon} \alpha'$ then $\beta \Longrightarrow \beta'$ and $\alpha' \approx \beta'$.
   Again, since $\mathsf{depth}(\alpha') + \mathsf{depth}(\beta') \leqslant \mathsf{depth}(\alpha) + \mathsf{depth}(\beta)$, we conclude by the induction hypothesis that $\vdash \alpha' = \beta'$.

The result follows using the derived inference rules: rule 1(b) in the first case and rule 1(a) in the second. □

3. Concurrent finite types

We extend the algebra of non-deterministic sequential finite types to concurrent types, adding a parallel composition operator. Since the algebra does not have communication, this operator is simply a merge of types (*cf.* the parallel composition operator of BPP). A type constructed with the parallel composition operator denotes the behaviour of a parallel composition of objects with the same name. In the parallel composition of types each component is the type of an element of the parallel composition of objects.

The axiomatic system includes two new expansion laws: the parallel composition of labelled sums is equivalent to a labelled sum; the parallel composition of blocked sums is equivalent to a blocked sum. However, the absence of mixed sums in the grammar of types prohibits a general expansion law. The main consequence of this fact is that normal forms

include parallel compositions, and the standard proof technique to establish the completeness of the axiomatic system must be refined. The mathematical study of the implications of the absence of mixed sums is interesting by itself.

To prove the axiomatic system complete, we were forced to add a new inference rule to the proof system of equational logic, which we prove sound. The rule does not seem to be derivable.

We studied an alternative proof of completeness (without this new inference rule) which uses only induction. However, the proof requires that the converse of the congruence for the parallel holds for normal forms. We conjecture that the result holds, and leave it as an open problem, since its proof turned out to be quite difficult, due to the highly combinatorial nature of the problem. If proved, the conjecture can be used as a lemma in a simpler proof of the completeness of the axiomatic system for the notion of equivalence, system that would not require the new inference rule mentioned before.

3.1. Syntax

Take the set of method names assumed in the previous section.

Definition 3.1 *(Concurrent finite types).* The grammar below defines the set $\mathcal{T}_{\mathbf{f}}$ of *concurrent finite types*.

$$\alpha, \beta ::= \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \;\mid\; \sum_{i\in I} \upsilon.\alpha_i \;\mid\; (\alpha \parallel \beta)$$

where I is a finite, possibly empty, indexing set, and each $\widetilde{\alpha}_i$ is a finite sequence of types.

The parallel composition operator, denoted by '$\parallel$', represents the existence of several objects located at (sharing) the same name, and executing in parallel (interpreted as different copies of the same object, possibly in different states). The prefixes of a sum bind tighter than the parallel constructor '$\parallel$', *i.e.*, $l.m \parallel n$ is $(l.m) \parallel n$.

Example 3.2. We refine the specification of the ATM, adding a method after the welcome that shows some messages to the user while the communication with the bank is taking place.

$$\mathsf{fpATM} \stackrel{\text{def}}{=} \mathit{welcome}(\mathit{int}, \mathit{int}).\big(\mathsf{show} \parallel \upsilon.(\upsilon.\mathit{sorry} + \upsilon.\mathsf{Menu})\big), \quad \text{where}$$

$$\mathsf{Menu} \stackrel{\text{def}}{=} \mathit{balance} + \mathit{deposit}(\mathit{int}) + \mathit{withdraw}(\mathit{int}).$$

Notice that a mixed sum here would be confusing because it would look like a user's choice.

3.2. Operational semantics

Take the set of labels specified by Definition 2.3.

Definition 3.3 *(Labelled transition relation).* The axiom schema of Definition 2.4 together with the two rules below inductively define the *labelled transition relation* of the algebra of concurrent finite types.

$$\textsc{Rpar}\ \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \textsc{Lpar}\ \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'}.$$

To prove that **lsb** is still a congruence in this extended language it suffices to show that the parallel composition operator preserves **lsb**.

Proposition 3.4 *(Preservation of* **lsb** *by* $\parallel$*). The parallel composition operator preserves* **lsb**.

Proof. It is easy to see that the relation $R \stackrel{\text{def}}{=} \{(\alpha \parallel \beta, \alpha' \parallel \beta) \mid \alpha \approx \alpha'\}$ is a bisimulation, and thus that $\parallel$ preserves $\approx$. We proceed by transition induction.

Take $(\alpha \parallel \beta, \alpha' \parallel \beta) \in R$ and let $\alpha \parallel \beta \xrightarrow{\pi} \delta$. We have two cases to consider:

Case $\alpha \xrightarrow{\pi} \alpha_1$ and $\delta \equiv \alpha_1 \parallel \beta$.
  By hypothesis, $\alpha \approx \alpha'$, thus there is an α'_1 such that $\alpha' \xrightarrow{\pi} \alpha'_1$ and $\alpha_1 \approx \alpha'_1$;
  hence, by induction hypothesis, $(\alpha_1 \parallel \beta, \alpha'_1 \parallel \beta) \in R$.
Case $\beta \xrightarrow{\pi} \beta_1$ and $\delta \equiv \alpha \parallel \beta_1$.
  Then, by rule Lpar, $\alpha \parallel \beta \xrightarrow{\pi} \alpha \parallel \beta_1$ and obviously, also $\alpha' \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta_1$;
  hence, by induction hypothesis, $(\alpha \parallel \beta_1, \alpha' \parallel \beta_1) \in R$.

By symmetry we conclude that R is an **lsb**. □

Corollary 3.5 *(Congruence).* **lsb** *is a congruence relation on* $\mathcal{T}_{\mathbf{f}}$.

As in the previous section, a result like Proposition 2.6 is convenient for some proofs.

Proposition 3.6 *(Label-strong bisimilarity). Types* α *and* β *are bisimilar, if, and only if,*

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}\ (\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha' \widetilde{\alpha} \approx \beta' \widetilde{\beta})$;
2. $\alpha \Longrightarrow \alpha'$ *implies* $\exists \beta'\ (\beta \Longrightarrow \beta'$ *and* $\alpha' \approx \beta')$.

It is useful to characterise *active types*, *i.e.* types that are not bisimilar to **0**. Syntactically, one can do it with a two-level grammar. Semantically, one uses the following lemma.

Lemma 3.7 *(Active types).* $\alpha \not\approx \mathbf{0}$ *if and only if* $\exists_{\alpha', l(\widetilde{\alpha})}\ \alpha \Longrightarrow \alpha' \xrightarrow{l(\widetilde{\alpha})}$.

Proof. Straightforward. □

The contrapositive of the lemma above is also interesting, as it characterises strictly blocked types: they are equivalent to **0**.

The following results make use of the notion of depth of a type, which we refine.

Definition 3.8 *(Depth of a type).* The rules of Definition 2.11, together with the rule below inductively define the *depth of a type*.

$$\mathsf{depth}(\alpha \parallel \beta) = \mathsf{depth}(\alpha) + \mathsf{depth}(\beta).$$

The subsequent proofs use the notion of *normal form of a type*.

Definition 3.9 *(Normal forms of concurrent finite types).*

1. A type α is *saturated* if $\alpha \stackrel{\upsilon}{\Longrightarrow} \alpha'$ implies $\alpha \xrightarrow{\upsilon} \alpha'$.
2. A type α is a *normal form* if it is saturated, and furthermore, one of the following conditions holds:
   (a) $\alpha \equiv \mathbf{0}$; or
   (b) $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ and each α_i is a normal form; or
   (c) $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ and each component of each $\alpha_i\widetilde{\alpha}_i$ is a normal form; or
   (d) $\alpha \equiv \alpha_1 \parallel \alpha_2$, where α_1 and α_2 are respectively as in (b) and (c) above, with $\alpha_1 \not\approx \mathbf{0}$.

We show now that all types have equivalent normal forms. We need an auxiliary result, a second υ-law.

Lemma 3.10 *(Law* **U2***).* $\upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) = \upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \upsilon.(\alpha_k \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j)$, *with* $k \in I$.

Proof. It is straightforward to build the respective bisimulation. □

Lemma 3.11 *(Normal form lemma). For all* α *there exists a normal form* α' *such that* $\vdash \alpha = \alpha'$, *with* $\mathsf{depth}(\alpha') \leqslant \mathsf{depth}(\alpha)$.

Proof. By induction on the depth of α.

Base case: $\mathsf{depth}(\alpha) = 0$; by definition of depth, $\alpha \equiv \mathbf{0}$, a normal form by definition.

Induction step: α is now either a sum or a parallel composition. Let us consider first the former case.

Case $\alpha \equiv \sum_{i\in I} \pi_i.\alpha_i$; then, by induction hypothesis, for each α_i there exists a normal form α_i' such that $\vdash \alpha_i = \alpha_i'$, with $\mathsf{depth}(\alpha_i') \leqslant \mathsf{depth}(\alpha_i)$. The prefix can be of two forms.

1. Case $\pi_i \equiv l_i(\widetilde{\alpha}_i)$, for all i.
   Since the types $\widetilde{\alpha}_i$ also have normal forms $\widetilde{\alpha}_i'$, we conclude that $\vdash \alpha = \sum_{i\in I} l_i(\widetilde{\alpha}_i').\alpha_i'$, as clearly, $\mathsf{depth}(\sum_{i\in I} l_i(\widetilde{\alpha}_i').\alpha_i') \leqslant \mathsf{depth}(\alpha)$.
2. Case $\pi_i \equiv \upsilon$, for all i.
   We only have to guarantee saturation. Thus, for each i such that $\alpha_i' \xrightarrow{\upsilon}$, either:
   (1) there is a $J_i \neq \emptyset$ such that $\alpha_i' \equiv \sum_{j\in J_i} \upsilon.\alpha_j$; for each $j \in J_i$ we then have $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i' \xrightarrow{\upsilon.\upsilon} \alpha_j$;
   by idempotency and law **U1**, we conclude $\vdash \sum_{i\in I} \upsilon.\alpha_i' = \sum_{i\in I} \upsilon.\alpha_i' + \upsilon.\alpha_j$; therefore, using these laws for all such α_j we thus obtain a normal form of α, since the resulting type is saturated, whose depth clearly equals that of $\sum_{i\in I} \upsilon.\alpha_i'$;

(2) or $\alpha_i' \equiv (\sum_{j\in J_i} \upsilon.\alpha_j) \parallel \beta$, with $\beta \equiv \sum_{k\in K_i} l_k(\widetilde{\beta}_k).\beta_k$ and $J_i, K_i \neq \emptyset$; for each $j \in J_i$ we have $\sum_{i\in I} \upsilon.\alpha_i' \xrightarrow{\upsilon.\upsilon} \alpha_j \parallel \beta$; by **U2**, we conclude

$$\vdash \sum_{i\in I} \upsilon.\alpha_i' = \sum_{i\in I} \upsilon.\alpha_i' + \upsilon.(\alpha_j \parallel \beta).$$

Notice that $(\alpha_j \parallel \beta)$ may not be a normal form (*e.g.*, if $\alpha_j \equiv \mathbf{0}$), but since clearly $\mathsf{depth}(\alpha_j \parallel \beta) < \mathsf{depth}(\alpha)$, thus by induction hypothesis, there is a normal form γ such that $\vdash \alpha_j \parallel \beta = \gamma$; hence

$$\vdash \sum_{i\in I} \upsilon.\alpha_i' = \sum_{i\in I} \upsilon.\alpha_i' + \upsilon.\gamma.$$

We still have to saturate $\upsilon.\gamma$. If it is a blocked sum, using **U1** we obtain a saturated form γ'; if it is a parallel composition we use **U2** to obtain γ'. So,

$$\vdash \sum_{i\in I} \upsilon.\alpha_i' = \sum_{i\in I} \upsilon.\alpha_i' + \gamma'.$$

Repeatedly applying **U2** in the same way to all α_i' of this form, and applying **U1** as in the proof of Theorem 2.12, we attain a normal form of α, whose depth obviously does not exceed that of $\sum_{i\in I} \upsilon.\alpha_i'$.

Case $\alpha \equiv \gamma_1 \parallel \gamma_2$; we use the commutative monoid laws and the expansion laws to rewrite α either as a blocked sum, a labelled-prefixed sum, or a parallel composition of the previous two with $I, J \neq \emptyset$, and without increasing the depth of the types. The first two cases are treated as above, and in the parallel composition case we apply the same reasoning to each sum separately, obtaining $\vdash \alpha = \alpha_1 \parallel \alpha_2$, where α_1 and α_2 are respectively a blocked sum and a labelled-prefixed sum, both normal forms. If $\vdash \alpha_1 = \mathbf{0}$ then $\vdash \alpha_1 \parallel \alpha_2 = \alpha_2$; otherwise, by definition, $\alpha_1 \parallel \alpha_2$ is a normal form. Moreover, notice that $\mathsf{depth}(\alpha_1) \leqslant \mathsf{depth}(\sum_{i\in I} \upsilon.\alpha_i)$ and $\mathsf{depth}(\alpha_2) \leqslant \mathsf{depth}(\sum_{j\in J} l_j(\widetilde{\beta}_j).\beta_j)$. Therefore, $\mathsf{depth}(\alpha_1 \parallel \alpha_2) \leqslant \mathsf{depth}(\alpha)$. □

An auxiliary useful result for some proofs ahead is the following.

Proposition 3.12. *If $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ is a normal form, then no α_i is a parallel composition.*

Proof. Let, for some $l \in I$, $\alpha_l \equiv \sum_{j\in J_i} \upsilon.\alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, which is a normal form.

For some $j \in J_i$, one may infer that $\alpha \xrightarrow{\upsilon\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, and since α is saturated, it is also the case that $\alpha \xrightarrow{\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$.

Therefore, for some $i \in I$, $\alpha_i \equiv \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$. But then α_j is blocked since α is a normal form, and furthermore, $\alpha_j \not\approx \mathbf{0}$; hence, by Lemma 3.7, α_j has a derivative which is a labelled sum.

Since the types are finite, applying repeatedly this procedure leads to, for some $i \in I$, $\alpha_i \equiv \sum_{m\in M_i} l_m(\widetilde{\alpha}_m).\alpha_m \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, which is not a normal form, and we attain a contradiction. □

The following result is also useful, and builds on the previous one.

Proposition 3.13. *Let α, β, and γ be types such that α and β are labelled sums, γ is a blocked sum, and $\alpha \approx \beta \parallel \gamma$. Then $\gamma \approx \mathbf{0}$.*

Proof. By induction on the depth of the types (*i.e.*, on $\mathsf{depth}(\alpha) + \mathsf{depth}(\beta)$), considering the three types in normal form (otherwise, use Lemma 3.11).

Base case: $\mathsf{depth}(\alpha) + \mathsf{depth}(\beta) = 0$. Then $\alpha \approx \beta \approx \mathbf{0}$, hence $\mathbf{0} \approx \mathbf{0} \parallel \gamma$, and since $\mathbf{0} \parallel \gamma \approx \gamma$, by transitivity we conclude that $\gamma \approx \mathbf{0}$.

Induction step. If $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ then, since by hypothesis $\alpha \approx \beta \parallel \gamma$ and γ is a blocked sum, we have $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \approx \beta' \parallel \gamma)$. We perform now a case analysis on the form of α'.

Case α' is again a labelled sum, so is β' since $\alpha' \approx \beta' \parallel \gamma$ and γ is a blocked sum; as clearly we have $\mathsf{depth}(\alpha)' + \mathsf{depth}(\beta)' < \mathsf{depth}(\alpha) + \mathsf{depth}(\beta)$, we conclude using the induction hypothesis that $\gamma \approx \mathbf{0}$.

Case α' is a blocked sum, so is β' since $\alpha' \approx \beta' \parallel \gamma$. If $\alpha' \approx \mathbf{0}$ then also $\beta' \parallel \gamma \approx \mathbf{0}$, and we conclude $\gamma \approx \mathbf{0}$. So, consider now $\alpha' \not\approx \mathbf{0}$; by Lemma 3.7, there is an υ-derivative α'' of α' which, by Proposition 3.12, is also a labelled sum; since $\alpha' \approx \beta' \parallel \gamma$, there is an υ-derivative δ of $\beta' \parallel \gamma$ which is a parallel composition of a labelled sum and a blocked sum. We have two cases to consider: either $\delta \equiv \beta'' \parallel \gamma$ with β'' being a labelled sum, and we conclude by induction that $\gamma \approx \mathbf{0}$, or $\delta \equiv \beta' \parallel \gamma'$ with γ' being a labelled sum. The latter situation is however impossible: let $\beta \parallel \gamma \xrightarrow{\upsilon} \beta \parallel \gamma' \xrightarrow{l(\widetilde{\beta})} \delta$; then $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha_1$ such that $\alpha_1 \approx \delta$ (hence α_1 is a labelled sum), and again as $\alpha \approx \beta \parallel \gamma$ by hypothesis, also $\beta \parallel \gamma \xrightarrow{l(\widetilde{\alpha})} \beta_1 \parallel \gamma$ with β_1 being a labelled sum; but now we conclude by induction that $\gamma \approx \mathbf{0}$, attaining a contradiction. □

3.3. Algebraic characterisation

We extend the axiomatic system $\mathcal{A}_{\mathbf{sf}}$ with laws regarding the parallel composition operator, and show that the resulting axiomatic system is sound and complete. Notice that we need two expansion laws, since the syntax of finite types does not allow mixing labels and υ in sums, *e.g.*, as in $l.\alpha + \upsilon.\beta$. Moreover, we further need an extra υ-law which allows us to saturate blocked parallel types that do not expand.

Prop/Definition 3.14 *(Axiomatisation). The laws of Prop/Definition* 2.8*, together with the following laws, sound with respect to* **lsb***, inductively define the* axiomatic system $\mathcal{A}_{\mathbf{f}}$.

CM $\langle \mathcal{T}_{\mathbf{f}}/=, \|, \mathbf{0}\rangle$ *is a commutative monoid*;
EXP1 $\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j = \sum_{i\in I} \upsilon.(\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j) + \sum_{j\in J} \upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \beta_j)$;
EXP2 $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j = \sum_{i\in I} l_i(\widetilde{\alpha}_i).(\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \sum_{j\in J} m_j(\widetilde{\beta}_j).(\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \beta_j)$;
U1 $\upsilon.\sum_{i\in I} \upsilon.\alpha_i = \sum_{i\in I} \upsilon.\alpha_i$;
U2 $\upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) = \upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \upsilon.(\alpha_k \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j)$*, with* $k \in I$.

Proof. It is straightforward to build the respective bisimulations. □

Theorem 3.15 *(Soundness of* $\mathcal{A}_{\mathbf{f}}$*). If* $\vdash \alpha = \beta$ *then* $\alpha \approx \beta$.

Proof. A consequence of Propositions 2.7, 3.4, and of Prop/Definition 3.14. □

To obtain a system that is provably complete, there are three alternatives.

1. Allow arbitrary prefixed sums, *i.e.* mixing labels and υ in sums, and having a single (CCS like) expansion law; the proof of completeness is standard. Notice that the other laws still hold.
2. Add a new inference rule to the equational logic and proceed as usual.

$$
\begin{aligned}
&\text{If} \quad \forall_{i\in I}\exists_{k\in K}\left(\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k,\ l_i \equiv m_k,\ \text{and} \vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l\right)\\
&\quad \text{and} \quad \forall_{k\in K}\exists_{i\in I}\left(\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k,\ l_i \equiv m_k,\ \text{and} \vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l\right)\\
&\quad \text{and} \quad \forall_{j\in J}\exists_{l\in L}\left(\vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l\right)\\
&\quad \text{and} \quad \forall_{l\in L}\exists_{j\in J}\left(\vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l\right),\\
&\quad \text{then} \quad \vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l. \qquad (1)
\end{aligned}
$$

3. Keep the proof system and use induction directly.

The first alternative is somewhat unnatural, since labelled sums represent interfaces of objects (*i.e.*, collections of enabled method names), and thus an arbitrary sum does not represent a valid object interface.[9] Moreover, the problem of axiomatising an equivalence notion without eliminating the parallel composition operator is also interesting from a mathematical point of view: we are not aware of any axiomatic system for a process algebra where the parallel is not reduced to sum. Hence, we rule out mixed sums.

For the last two alternatives, normal forms include a parallel composition, as there is no expansion law for the parallel composition of a labelled sum and a blocked sum; thus the proofs of the normal form lemma and of the completeness theorem are different from those for CCS.

The third alternative (using only induction) turns out to be quite difficult, due to the highly combinatorial nature of the problem. The proof requires that the converse of the congruence result for the parallel operator holds for normal forms: if $\alpha_1 \parallel \alpha_2 \approx \alpha_1' \parallel \alpha_2'$ then $\alpha_1 \approx \alpha_1'$ and $\alpha_2 \approx \alpha_2'$. We did not prove (nor disprove) this result, but succeeded for a particular case—the new inference rule.

[9] *Cf.* Example 3.2.

Therefore, we proceed now according to the second alternative listed above. First we have to show that the new inference rule is sound.

Lemma 3.16 *(Soundness of the new inference rule). The inference rule* (1) *is sound.*

Proof. Let $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$ and $\beta \equiv \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$. We proceed by induction on the depth of the types (*i.e.*, on $\mathsf{depth}(\alpha) + \mathsf{depth}(\beta)$), and conduct a case analysis of the possible immediate transitions of α and β.

1. Case $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$.
   Since by hypothesis there is a $k \in K$ such that $l_i \equiv m_k$, and since by induction hypothesis $\widetilde{\alpha}_i \approx \widetilde{\beta}_k$, as by hypothesis $\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k$, then we also have
   $$\beta \xrightarrow{m_k(\widetilde{\beta}_k)} \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l.$$
   Moreover, $\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j \approx \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, again by induction hypothesis and because by hypothesis we have
   $$\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l.$$
2. Case $\alpha \xrightarrow{\upsilon} \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j$.
   Let $\beta \xrightarrow{\upsilon} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$ for some $l \in L$.
   Using again the hypotheses and the soundness the axiomatic system, it follows that
   $$\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j \approx \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l.$$

By symmetry we conclude that $\alpha \approx \beta$. □

Again, we show a completeness result for the axiomatic system, with respect to **lsb**.

Theorem 3.17 *(Completeness of $\mathcal{A}_{\mathbf{sf}}$). If $\alpha \approx \beta$ then $\vdash \alpha = \beta$.*

Proof. By induction on the sum of the depths of the types α and β (assumed to be normal forms, by Lemma 3.11).
Taking into account the proof of Theorem 2.12, we only have one case to consider.
Case $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$ and $\beta \equiv \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.
We examine the indexing sets:

1. If $I \neq \emptyset$, then obviously also $K \neq \emptyset$, as $\alpha \approx \beta$.
2. Note that $J \neq \emptyset$ and also $L \neq \emptyset$, since clause 3 of the definition of normal forms demands $\sum_{j\in J} \upsilon.\alpha_j \not\approx \mathbf{0}$ and thus also $\sum_{l\in L} \upsilon.\beta_l \not\approx \mathbf{0}$.
3. Thus, consider $I, J, K, L \neq \emptyset$. The proof analyses the possible transitions of α. There are two cases to consider:
   (a) Case $\alpha \xrightarrow{l_i(\widetilde{\alpha}_i)} \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$.
   Since, by hypothesis, $\alpha \approx \beta$, then there exists a $k \in K$ such that $l_i \equiv m_k$, $\widetilde{\alpha}_i \approx \widetilde{\beta}_k$, and $\beta \xrightarrow{m_k(\widetilde{\beta}_k)} \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, with $\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j \approx \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.
   By induction hypothesis it follows that $\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k$ and thus,
   $$\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l.$$
   (b) Case $\alpha \xrightarrow{\upsilon} \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j$.
   If $\beta \xRightarrow{\upsilon} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$ then, as β is saturated, thus
   $$\beta \xrightarrow{\upsilon} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l;$$
   the proof proceeds similarly to the previous case.
   Otherwise, $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j \approx \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$ with the sum of their depths being lesser than the sum of the original depths, and we can again use the induction hypothesis.
   The result for the case we are examining follows by the inference rule 1.

As there are no more cases, the proof is complete. □

4. Behavioural types

Finally, we present the *Algebra of Behavioural Types*, ABT for short. We obtain it by extending the algebra of concurrent finite types with a recursive operator μ to denote infinite types. A type $\mu t.\alpha$ denotes a solution to the equation $t = \alpha$. Recursive types allow us to characterise the behaviour of persistent objects, as well as that of (possibly non-persistent) objects, created when executing methods of persistent objects.

In this section, we present the syntax and operational semantics of ABT, add to the axiomatic system of the previous section three new laws regarding the behaviour of recursive types, laws that we prove correct. Note that these axioms are different from those of CCS. The next section shows that the axiomatic system—simpler than that of CCS because of absence of mixed sums in ABT—is complete for image-finite types. The result holds in a setting more general than that of CCS, as we do not require processes to be sequential.

4.1. Syntax

Assume a countable set of variables, denoted by t, possibly subscripted, disjoint from the set of method names considered in the previous sections.

Definition 4.1 *(Behavioural types).* The grammar below defines the set $\mathcal{T}$ of *behavioural types*.

$$\alpha, \beta ::= \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \;\mid\; \sum_{i\in I} \upsilon.\alpha_i \;\mid\; (\alpha \parallel \beta) \;\mid\; t \;\mid\; \mu t.\alpha$$

where I is a finite, possibly empty, indexing set, each $\widetilde{\alpha}_i$ is a finite sequence of types.

The recursive operator μ allows the definition of possibly infinite types, as in the next example.

Example 4.2. We refine again the specification of the ATM, allowing recurrent behaviour.

$$\mathsf{ATM} \stackrel{\text{def}}{=} \big(\mathsf{show} \parallel \mu t.\mathit{welcome}(\mathit{int},\mathit{int}).\upsilon.(\upsilon.\mathit{sorry}.t + \upsilon.\mathsf{Menu})\big), \quad \text{where}$$

$$\mathsf{Menu} \stackrel{\text{def}}{=} \mathit{balance}.t + \mathit{deposit}(\mathit{int}).t + \mathit{withdraw}(\mathit{int}).t.$$

This version of the ATM has now a "kind" behaviour, allowing a user to perform several operations.

Since types may have type variables, we define when a type variable occurs free in a type, and when it occurs bound.

Definition 4.3 *(Free and bound variables).* An occurrence of the variable t in the type α is *bound* if it occurs in a part $\mu t.\alpha$ of α; otherwise the occurrence of t in α is *free*.

Alpha-conversion in a type $\mu t.\alpha$ is defined as usual. Henceforth we consider only *contractive types*, *i.e.*, terms where, in any subexpression of the form $(\mu t.(\mu t_1 \ldots (\mu t_n.\alpha)))$ (with $n \geqslant 0$), the body α is not t.

Notation. For simplicity, we use the following conventions.

1. Assume a variable convention like in Barendregt [3], and assume types equal up-to alpha-conversion. Moreover, $\mathrm{fv}(\alpha)$ denotes the set of variables that occur free in type α and $\mathrm{var}(\alpha)$ denotes the set of all variables (free or bound) in α.
2. The type $\alpha\{\beta/t\}$ denotes the result of the substitution in α of β for the free occurrences of t. Furthermore, the type $\alpha\{\widetilde{\beta}/\tilde{t}\}$ denotes the simultaneous substitution[10] of $\widetilde{\beta}$ for the free occurrences of $\tilde{t}$ in α.
3. Let $\{\tilde{t}\}$ denote the set of the elements, and $|\tilde{t}|$ the length, of the sequence $\tilde{t}$.
4. For simplicity, we sometimes write $\alpha(\beta)$ instead of $\alpha\{\beta/t\}$.

Terminology. The following concepts will be useful to prove the subsequent results.

Definition 4.4 *(Guarded variables and guarded types).*

1. A type without free variables is said *closed*; otherwise it is *open*.
2. A free variable t *is guarded in* α if all its occurrences are within some label-prefixed part of α.
3. A type α *is guarded* if all its free variables are guarded. Otherwise, we say α *is unguarded*.

Example 4.5. The variable t is guarded in $l(t).\alpha$, in $l(\widetilde{\alpha}).t$, and in $l(\upsilon.t).\mathbf{0}$, but not in $\upsilon.t$.

[10] Standard notion (see, for instance, Barendregt [3]).

Table 1
The labelled transition relation of the Algebra of Behavioural Types.

ACT	$\sum_{i\in I}\pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I)$			
RPAR	$\dfrac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta}$	LPAR	$\dfrac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'}$	REC $\dfrac{\alpha\{\mu t.\alpha/t\} \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$

4.2. Operational semantics

Assume the set of labels specified by Definition 2.4. We define the operational semantics of ABT by adding a new rule to the labelled transition relation defined in Definition 3.3. Table 1 presents all the rules together.

Definition 4.6 *(Labelled transition relation).* The axiom schema of Definition 2.4 together with the two rules of Definition 3.3 and with the axiom and the rule below inductively define the *labelled transition relation* of the Algebra of Behavioural Types.

$$\text{REC}\quad \frac{\alpha\{\mu t.\alpha/t\} \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}.$$

The definition of **lsb** that we have been using (Definition 2.5 in p. 70), only applies to closed terms. Following the usual approach (see, *e.g.* [58]), we extend it to open terms by requiring them to be bisimilar if all their closed instantiations are bisimilar.

Definition 4.7 *(Bisimilarity on open types).* Let $\mathsf{fv}(\alpha) \cup \mathsf{fv}(\beta) \subseteq \{\tilde{t}\}$. Then, $\alpha \approx \beta$ if, for all sequences of closed types $\widetilde{\gamma}$, we have $\alpha\{\widetilde{\gamma}/\tilde{t}\} \approx \beta\{\widetilde{\gamma}/\tilde{t}\}$.

It is straightforward to verify that this new definition of **lsb** is an equivalence relation and a fix point.

An important property is *substitutivity*: according to Rensink [58], a relation is substitutive if it is preserved by *insertion*—substituting equivalent types for variables do not change the behaviour of a type—and by *instantiation*—replacing a closed type for a variable in equivalent types result in equivalent types. Since preservation by instantiation is built into the new definition of **lsb**, it suffices to prove that **lsb** is preserved by insertion.

Proposition 4.8 *(Substitutive). If $\alpha_1 \approx \alpha_2$ then $\alpha\{\alpha_1/t\} \approx \alpha\{\alpha_2/t\}$.*

Proof. It is easy to show that the relation $\{(\alpha\{\alpha_1/t\}, \alpha\{\alpha_2/t\}) \mid \alpha_1 \approx \alpha_2\}$ is an **lsb**. □

It is necessary to verify that **lsb** is still a congruence—an expected result, but since we conduct the proof on recursive terms rather than on equations, it turns out to be simpler (in particular, a bisimulation suffices, whereas for equations one needs to establish a bisimulation up to).

An important auxiliary result is a consequence of the rule REC: folding or unfolding a recursive term does not change its behaviour, since $\mu t.\alpha$ and $\alpha\{\mu t.\alpha/t\}$ have the same transitions, and thus one expects them to be bisimilar.

Lemma 4.9 *(Unfolding). $\mu t.\alpha \approx \alpha\{\mu t.\alpha/t\}$.*

Proof. Immediate. □

We prove now that the operator μt preserves **lsb** and hence, that our notion of equivalence is still a congruence.

Proposition 4.10 *(Preservation of* **lsb** *by recursion). The recursive operator preserves label-strong bisimulation.*

Proof. We show that the relation $\{(\mu t.\alpha, \mu t.\beta) \mid \alpha \approx \beta\}$ is an **lsb**, and thus, that μt preserves $\approx$. The proof is by transition induction. The base case is trivial, as the definition of **lsb** implies that if $\alpha \approx \beta$ and $\alpha \equiv t$ then $\beta \equiv t$. There are two cases to consider in the induction step.

1. Let $\mu t.\alpha \xrightarrow{l(\widetilde{\alpha}(\mu t.\alpha))} \gamma$. By a shorter derivation (see rule REC) also $\alpha(\mu t.\alpha) \xrightarrow{l(\widetilde{\alpha}(\mu t.\alpha))} \gamma$. Since by hypothesis $\alpha \approx \beta$, we have

$$\alpha(\mu t.\alpha) \approx \beta(\mu t.\alpha) \xrightarrow{l(\widetilde{\beta}(\mu t.\alpha))} \beta'(\mu t.\alpha) \approx \alpha'(\mu t.\alpha) \equiv \gamma,$$

as there are $\widetilde{\beta}$ such that $\widetilde{\alpha}(\mu t.\alpha) \approx \widetilde{\beta}(\mu t.\alpha)$.

Therefore, $\mu t.\beta$ has an l-transition, hence also $\beta(\mu t.\beta) \xrightarrow{l(\widetilde{\beta}(\mu t.\beta))} \beta'(\mu t.\beta)$, and the result follows as by induction hypothesis $\alpha'(\mu t.\alpha) \approx \beta'(\mu t.\beta)$ and $\widetilde{\alpha}'(\mu t.\alpha) \approx \widetilde{\beta}(\mu t.\beta)$.

2. Let $\mu t.\alpha \xrightarrow{\upsilon} \alpha'$. The proof is as in the first case, except that we do not need to worry about the parameters in the prefixes.

The proof is complete. □

Therefore, **lsb** in ABT is a congruence.

4.3. Axiomatic system

We present an axiomatisation of the equivalence notion, adding three recursion rules to the previous axiomatic system, and show its soundness. Completeness will be the topic of the next section. The axiomatisation of **lsb** requires three more laws:

1. unfolding recursive types preserves **lsb**;
2. equalities involving recursive types have unique solutions up to **lsb**;
3. allow the saturation of recursive types.

Notice that the absence of mixed sums in ABT leads to a simpler axiomatic system than that of CCS.

Prop/Definition 4.11 *(Axiomatic system). The laws of Prop/Definition* 3.14*, together with the following recursion laws, inductively define an* axiomatic system*.*

R1 $\mu t.\alpha = \alpha\{\mu t.\alpha/t\}$;
R2 *if* $\beta = \alpha\{\beta/t\}$ *then* $\beta = \mu t.\alpha$*, provided that* α *is guarded*;
R3 $\mu t.(\upsilon.t + \sum_{i\in I} \upsilon.\alpha_i) = \mu t.\sum_{i\in I} \upsilon.\alpha_i$.

Remark. Laws **R3** and **R5** of CCS[11] have no correspondence in this setting, as ABT does not have binary sums. Thus, our law **R3** corresponds to (is an instance of) the CCS's **R4** law. Soundness of the above axioms is not a trivial result. To prove it, one has first to ensure that *the equations have unique solutions*, *i.e.*,

if β is guarded, $\alpha_1 \approx \beta\{\alpha_1/t\}$, and $\alpha_2 \approx \beta\{\alpha_2/t\}$ then $\alpha_1 \approx \alpha_2$.

The next subsection is dedicated to the proof of that result. Once we establish it, the proof of the soundness of the axioms follows.

Proof of Prop/Definition 4.11. The Unfolding Lemma 4.9 states that law **R1** is sound. Law **R2** of Prop/Definition 4.11 is a corollary of the uniqueness of the solutions of equations (Theorem 4.16 in p. 81) and of law **R1**. To prove law **R3**, one simply has to build the appropriate bisimulation. □

4.4. Unique solutions

The proof follows a method analogous to that used for CCS, but we attain a more general result, since we do not require the type to be sequential. The method was set up by Milner; Ying has a simpler proof that we follow here [70].

We need the auxiliary notion of **lsb** up to $\approx$.

Definition 4.12 *(Lsb up to* $\approx$*).* An **lsb** *up to* $\approx$ is a symmetric binary relation R on types such that, whenever $\alpha\ R\ \beta$ then

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists_{\widetilde{\beta},\beta'}\ \beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha'\widetilde{\alpha} \approx R \approx \beta'\widetilde{\beta}$, and
2. $\alpha \Longrightarrow \alpha'$ implies $\exists_{\beta'}\ \beta \Longrightarrow \beta'$ and $\alpha' \approx R \approx \beta'$.

We show that **lsb** up to $\approx$ is still a label-strong bisimulation.

Proposition 4.13 *(Lsb up to* $\approx$ *is a bisimulation). Let* R *be an* **lsb** *up to* $\approx$*. Then*:

1. $\approx R \approx$ *is an* **lsb**.
2. $R \subseteq \approx$.

[11] CCS' recursion laws: $\mu t.(t + \alpha) = \mu t.\alpha$ (**R3**), $\mu t.(\tau.t + \alpha) = \mu t.\tau.\alpha$ (**R4**), and $\mu t.(\tau.(t + \alpha) + \beta) = \mu t.(\tau.t + \alpha + \beta)$ (**R5**).

Proof. Similar to the proof for weak bisimulation up to weak bisimilarity in [36]. □

Remark. The definition of **lsb** up to $\approx$ is slightly different from that of **lsb**: with $\xrightarrow{\upsilon}$ instead of $\Longrightarrow$ in the antecedent of the second condition of Definition 4.12, the previous proposition would not hold, as the example $R = \{(\upsilon.\upsilon.a.\mathbf{0}, \upsilon.\mathbf{0})\}$ shows.[12]

Two technical lemmas are necessary to prove the result. We present and prove them below, and proceed to the main result: equations have unique solutions (up to **lsb**).

Lemma 4.14. *Let α be guarded, and consider all its free variables in $\{\tilde{t}\}$.*

1. *If $\alpha(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$ then there exist α' and $\widetilde{\alpha}$ with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}(\widetilde{\beta})$, and, for all $\widetilde{\beta}'$, $\alpha(\widetilde{\beta}') \xrightarrow{l(\widetilde{\alpha}(\widetilde{\beta}'))} \alpha'(\widetilde{\beta}')$.*
2. *If $\alpha(\widetilde{\beta}) \xrightarrow{\upsilon} \gamma$ then there exists α' with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and, for all $\widetilde{\beta}'$, we have $\alpha(\widetilde{\beta}') \xrightarrow{\upsilon} \alpha'(\widetilde{\beta}')$. Furthermore, α' is guarded.*

Proof. By transition induction.[13] There are two cases to consider, depending on the transition performed.

1. Let $\alpha(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$. The base case is simple: let $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$. Then $\alpha(\widetilde{\beta}) \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i(\widetilde{\beta})).\alpha_i(\widetilde{\beta})$, $l(\widetilde{\gamma}) \equiv l_i(\widetilde{\alpha}_i(\widetilde{\beta}))$, and $\gamma \equiv \alpha_i(\widetilde{\beta})$ for some $i \in I$. The result follows from taking $\alpha' \equiv \alpha_i$ and $\widetilde{\alpha} \equiv \widetilde{\alpha}_i$.
   For the induction step, we have two different cases to consider, according to the possible forms of α.
   (a) Case $\alpha \equiv \alpha_1 \parallel \alpha_2$.
   Then $\alpha(\widetilde{\beta}) \equiv \alpha_1(\widetilde{\beta}) \parallel \alpha_2(\widetilde{\beta})$. There are two sub-cases to consider:
   i. either $\gamma \equiv \gamma_1 \parallel \alpha_2(\widetilde{\beta})$, with $\alpha_1(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma_1$,
   ii. or $\gamma \equiv \alpha_1(\widetilde{\beta}) \parallel \gamma_2$, with $\alpha_2(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma_2$, by a shorter derivation.
   Without loss of generality, assume the first case. Since α is guarded, so is α_1, and thus, by induction hypothesis, it follows that $\gamma_1 \equiv \alpha_1'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}_1(\widetilde{\beta})$. The result follows from taking $\alpha' \equiv \alpha_1' \parallel \alpha_2$ and $\widetilde{\alpha} \equiv \widetilde{\alpha}_1$.
   (b) Case $\alpha \equiv \mu t.\beta$.
   Then $\alpha(\widetilde{\beta}) \equiv \mu t.\beta(\widetilde{\beta})$, where the free variables of β are taken from t and $\tilde{t}$. The variables $\tilde{t}$ must be guarded in β, otherwise they would not be guarded in α. If $\mu t.\beta(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$ then $\beta\{\mu t.\beta(\widetilde{\beta})/t\} \xrightarrow{l(\widetilde{\gamma})} \gamma$, by a shorter derivation. But the variables of $\tilde{t}$ are guarded in $\beta\{\mu t.\beta/t\}$, and thus, by induction hypothesis, we conclude that $\gamma \equiv \alpha'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}(\widetilde{\beta})$.
2. Let $\alpha(\widetilde{\beta}) \xrightarrow{\upsilon} \gamma$. The proof is as in the first case, except that we need not worry about parameters in prefixes. The main difference is that we must also prove that α' is guarded. The base case is trivial. Case $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$, all the α_i must be guarded because α is, and thus α' is guarded. In the remaining cases, the conclusion is a consequence of the induction hypothesis.

The proof is complete. □

Lemma 4.15. *Let α be guarded, and consider all its free variables in $\tilde{t}$. If $\alpha(\widetilde{\beta}) \Longrightarrow \gamma$ then there exists α' with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and, for all $\widetilde{\beta}'$, we have $\alpha(\widetilde{\beta}') \Longrightarrow \alpha'(\widetilde{\beta}')$.*

Proof. Let $\alpha(\widetilde{\beta}) \Longrightarrow \gamma$ and let n be the actual number of υ's in the transition. The proof follows easily from the previous lemma, by induction on n. □

We are finally in a position to prove the uniqueness of the solutions of equations, the result that leads to the correctness of the axiom system.

Theorem 4.16 *(Unique solutions of equations). Let $\widetilde{\beta}$ be guarded, $\widetilde{\alpha}_1 \approx \widetilde{\beta}(\widetilde{\alpha}_1)$ and $\widetilde{\alpha}_2 \approx \widetilde{\beta}(\widetilde{\alpha}_2)$. Then $\widetilde{\alpha}_1 \approx \widetilde{\alpha}_2$.*

Proof. Let R be the relation $\{(\gamma(\widetilde{\alpha}_1), \gamma(\widetilde{\alpha}_2)) \mid \mathrm{var}(\gamma) \subseteq \tilde{t}\}$. We will show that:

1. $\gamma(\widetilde{\alpha}_1) \xrightarrow{l(\widetilde{\delta}_1)} \alpha_1$ implies $\exists \alpha_2, \widetilde{\delta}_2\ \gamma(\widetilde{\alpha}_2) \xrightarrow{l(\widetilde{\delta}_2)} \alpha_2$ and $\alpha_1(\widetilde{\delta}_1) \approx R \approx \alpha_2(\widetilde{\delta}_2)$;
2. $\gamma(\widetilde{\alpha}_1) \Longrightarrow \alpha_1$ implies $\exists \alpha_2\ \gamma(\widetilde{\alpha}_2) \Longrightarrow \alpha_2$ and $\alpha_1 \approx R \approx \alpha_2$.

[12] Notice the similarities with Exercise 5.14 in [36].
[13] Induction on the length of the derivation of the transition.

1. So let us prove the first item: since $\approx$ is a congruence,
$$\gamma(\widetilde{\alpha}_1) \approx \gamma\big(\widetilde{\beta}(\widetilde{\alpha}_1)\big)\ R\ \gamma\big(\widetilde{\beta}(\widetilde{\alpha}_2)\big) \approx \gamma(\widetilde{\alpha}_2).$$
Therefore, by hypothesis $\widetilde{\alpha}_1 \approx \widetilde{\beta}(\widetilde{\alpha}_1)$ and $\widetilde{\beta}(\widetilde{\alpha}_2) \approx \widetilde{\alpha}_2$. Let $\gamma(\widetilde{\alpha}_1) \xrightarrow{l(\widetilde{\delta}_1)} \alpha_1$. Then, $\gamma(\widetilde{\beta}(\widetilde{\alpha}_1)) \xrightarrow{l(\widetilde{\delta}'_1)} \alpha'_1$ with $\widetilde{\delta}_1 \approx \widetilde{\delta}'_1$ and $\alpha_1 \approx \alpha'_1$. By Lemma 4.14, there are $\widetilde{\gamma}$ and γ' such that $\widetilde{\delta}'_1 \equiv \widetilde{\gamma}(\widetilde{\alpha}_1)$, $\alpha'_1 \equiv \gamma'(\alpha_1)$ and $\gamma(\widetilde{\beta}(\widetilde{\alpha}_2)) \xrightarrow{l(\widetilde{\gamma}(\widetilde{\alpha}_2))} \alpha'_2 \equiv \gamma'(\widetilde{\alpha}_2)$, which implies $\widetilde{\gamma}(\widetilde{\alpha}_2) \xrightarrow{l(\widetilde{\delta}_2)} \alpha_2$, with $\widetilde{\gamma}(\widetilde{\alpha}_2) \approx \widetilde{\delta}_2$ and $\alpha'_2 \approx \alpha_2$.
We conclude that $\widetilde{\delta}_1 \approx R \approx \widetilde{\delta}_2$ and $\alpha_1 \approx R \approx \alpha_2$.
2. We prove 2 by similar reasoning, but using Lemma 4.15, instead of Lemma 4.14.

By the results we have seen before, and by symmetry, this establishes that R is a label-strong bisimulation up to label-strong bisimilarity, and also that $\gamma(\widetilde{\alpha}_1) \approx \gamma(\widetilde{\alpha}_2)$ for all γ, which includes the cases $\alpha_{1i} \approx \alpha_{2i}$ ($\gamma \equiv t_i$), for all $i = 1, \ldots, |\tilde{t}|$. □

Note that from this result follows as a corollary that the recursive constructor preserves the equivalence notion (at least for guarded types), as stated in Prop/Definition 4.11. We finally present the proof of that result.

Proof. If $\alpha \approx \beta$ and both are guarded, then since $\mu t.\alpha \approx \alpha(\mu t.\alpha)$, also $\mu t.\alpha \approx \beta(\mu t.\alpha)$, thus $\mu t.\alpha \approx \mu t.\beta$. □

5. Completeness of the axiom system for image-finite types

The presence of the recursive operator in the algebra allows us to define infinite types like $\mu t.(l.t)$, a type that represents an infinite sequence of l-actions. It represents the behaviour of a persistent object that repeatedly offers a method l. This is actually an image-finite type,[14] but the recursive operator, together with the parallel composition operator, allows us to define image-infinite types like $\mu t.\upsilon.(l \parallel t)$. This would be the type of an ephemeral object (usable only once) with a method l that is created by a method of a persistent object.

Due to decidability issues in process algebra, the study of complete axiomatisations of equivalence notions is restricted to languages generating only image-finite terms. Therefore, in this section we use a sublanguage of ABT, obtained by removing the parallel composition operator from the grammar in Definition 4.1. The resulting language is image-finite (van Glabbeek [65] proves that a language with action prefixes, choice, and recursion is image-finite).

The main result herein is the completeness of the axiom system of the previous section for this language of image-finite types. The proof is not trivial, and so we dedicate to it this section. It follows the "standard" structure of that for image-finite CCS [37], being significantly simpler, as ABT does not have communication.

5.1. Equational characterisation

The purpose of this first step of the completeness proof is to show an equational characterisation theorem showing that all types satisfy some set of equations.

Terminology. Consider a set of variables $T = \{t_1, \ldots, t_n\}$ and a set of types $A = \{\alpha_1, \ldots, \alpha_n\}$ where $\mathsf{fv}(A) \subseteq T$ and $n \geqslant 0$. Let S: $t_1 = \alpha_1, \ldots, t_n = \alpha_n$ denote a system of (possibly mutually recursive) equations, and let $\mathsf{var}(S) = T$. We use the following abbreviations: $\tilde{t} = t_1, \ldots, t_n$, $\widetilde{\alpha} = \alpha_1, \ldots, \alpha_n$ and S: $\tilde{t} = \widetilde{\alpha}$ denotes a system of equations. Furthermore, $\vdash \widetilde{\alpha} = \widetilde{\beta}(\widetilde{\alpha})$ stands for $\vdash \alpha_1 = \beta_1(\alpha_1)$ and … and $\vdash \alpha_n = \beta_n(\alpha_n)$, for some $n \geqslant 0$.

Then, $\forall i.1 \leqslant i \leqslant n$,

1. we write $t_i \xrightarrow{\pi} t$, if $\pi.t$ is a summand of α_i;
2. let α_1 be a closed type; we write $\alpha_1 \Vdash S$ if, for any $\widetilde{\beta}$ it is the case that $\vdash \widetilde{\alpha} = \widetilde{\beta}(\widetilde{\alpha})$;
3. we say the system S is: *guarded*, if $\forall_i\ t_i \overset{\upsilon}{\not\Longrightarrow} t_i$; *saturated* if $t_i \overset{\upsilon}{\Longrightarrow} t'$ implies $t_i \xrightarrow{\upsilon} t'$; and *standard*, if $\alpha_i \equiv \sum_{j\in J} \upsilon.t_{f(i,j)}$ or $\alpha_i \equiv \sum_{j\in J} l_{ij}(\tilde{t}'_{f(i,j)}).t_{f(i,j)}$, where $\tilde{t}'_{f(i,j)} \subseteq \tilde{t}$.

The following lemma is crucial to prove that semantically equivalent types satisfying two different sets of equations also satisfy a common set of equations (Theorem 5.2).

Lemma 5.1 *(Saturation). Let $\alpha \Vdash S$, with S standard and guarded. There is an S' standard, guarded, and saturated such that $\alpha \Vdash S'$.*

Proof. From S obtain S' saturated, by saturating each equation. Consider S: $\tilde{t} = \widetilde{\alpha}$ and take $t_1 = \alpha_1$. It is now necessary to perform a case analysis on the structure of α_1. Since S is standard, there are only two cases to be considered.

[14] A type α is *image-finite*, if the collection $\{\beta \mid \alpha \xrightarrow{\pi} \beta\}$ is finite for each action π.

1. Case α_1 is a labelled sum, it is already (trivially) saturated.
2. Case α_1 is a blocked sum, as it is guarded by hypothesis, if $t_1 \xrightarrow{\upsilon.\upsilon} t_j$ then $j \neq 1$. To saturate α_1 proceed like in the second case of the proof of Lemma 3.11, obtaining α_1'.

Now in S substitute α_1' for α_1 and repeat this process for the remaining equations. □

Theorem 5.2 *(Common set of equations). Consider two systems of equations $S\colon \tilde{t} = \widetilde{\gamma}$ and $T\colon \tilde{u} = \widetilde{\delta}$, standard and guarded, where* $\mathsf{var}(S)$ *is disjoint from* $\mathsf{var}(T)$*. Let $\alpha \Vdash S$, and $\beta \Vdash T$, and let $\alpha \approx \beta$. Then, there is a system U standard and guarded such that $\alpha \Vdash U$ and $\beta \Vdash U$.*

Proof. By Lemma 5.1, assume S and T saturated. We construct the common set U as follows.
There are $\widetilde{\alpha}$ and $\widetilde{\beta}$, with $\alpha_1 \equiv \alpha$ and $\beta_1 \equiv \beta$ such that $\vdash \widetilde{\alpha} = \widetilde{\gamma}[\widetilde{\alpha}/\tilde{t}]$ and $\vdash \widetilde{\beta} = \widetilde{\delta}[\widetilde{\beta}/\tilde{u}]$.
Since by hypothesis $\alpha \approx \beta$ and S and T are saturated:

1. $t_1 \xrightarrow{l(\tilde{t}')} t_i$ implies $\exists_{u_j, \tilde{u}'}\ (u_1 \xrightarrow{l(\tilde{u}')} u_j$ and $\alpha_i \widetilde{\alpha}' \approx \beta_j \widetilde{\beta}')$[15];
2. $u_1 \xrightarrow{l(\tilde{u}')} u_j$ implies $\exists_{t_i, \tilde{t}'}\ (t_1 \xrightarrow{l(\tilde{t}')} t_i$ and $\alpha_i \widetilde{\alpha}' \approx \beta_j \widetilde{\beta}')$;
3. $t_1 \xrightarrow{\upsilon} t_i$ implies $\exists_{u_j}\ (u_1 \xrightarrow{\upsilon} u_j$ and $\alpha_i \approx \beta_j)$;
4. $u_1 \xrightarrow{\upsilon} u_j$ implies $\exists_{t_i}\ (t_1 \xrightarrow{\upsilon} t_i$ and $\alpha_i \approx \beta_j)$.

Consider the following bisimulation relation R:

1. $R \subseteq \tilde{t} \times \tilde{u}$ such that
   (a) $t \xrightarrow{l(\tilde{t}')} t'$ implies $\exists_{u', \tilde{u}'}\ (u \xrightarrow{l(\tilde{u}')} u'$ and $t' \tilde{t}'\ R\ u' \tilde{u}')$;
   (b) $u \xrightarrow{l(\tilde{u}')} u'$ implies $\exists_{t', \tilde{t}'}\ (t \xrightarrow{l(\tilde{t}')} t'$ and $t' \tilde{t}'\ R\ u' \tilde{u}')$;
   (c) $t \xrightarrow{\upsilon} t'$ implies $(t'\ R\ u$ or $\exists_{u'}\ (u \xrightarrow{\upsilon} u'$ and $t'\ R\ u'))$;
   (d) $u \xrightarrow{\upsilon} u'$ implies $(t\ R\ u'$ or $\exists_{t'}\ (t \xrightarrow{\upsilon} t'$ and $t'\ R\ u'))$;
2. $t_1\ R\ u_1$.

We aim at $U\colon \tilde{\nu} = \tilde{\varepsilon}$, where $\tilde{\nu} = \{v_{ij} \mid t_i\ R\ u_j\}$, and $\tilde{\varepsilon} = \{\varepsilon_{ij} \mid t_i\ R\ u_j\}$, with ε_{ij} being a sum with summands:

1. $l(\tilde{\nu}').v_{kl}$, if $t_i \xrightarrow{l(\tilde{t}')} t_k$ and $u_j \xrightarrow{l(\tilde{u}')} u_l$ and $t_k \tilde{t}'\ R\ u_l \tilde{u}'$[16];
2. $\upsilon.v_{kl}$, if $t_i \xrightarrow{\upsilon} t_k$ and $u_j \xrightarrow{\upsilon} u_l$ and $t_k\ R\ u_l$.

To finally prove '$\alpha \Vdash U$', with v_{11} being the leading variable, we must find $\widetilde{\varphi}$ such that $\varphi_1 \equiv \alpha$ and $\vdash \widetilde{\varphi} = \tilde{\varepsilon}[\widetilde{\varphi}/\tilde{\nu}]$.
Let $\varphi_{ij} \equiv \alpha_i$. Since $t_i \approx u_j$ we have two cases to consider:

1. Case $t_i \xrightarrow{l(\tilde{t}')} t_k$, where $t_k\ R\ u_l$ and $\tilde{t}'\ R\ \tilde{u}$.
   Then ε_{ij} is a labelled sum with a summand $l(\tilde{\nu}').v_{kl}$, and α_i has a summand $l(\alpha', \ldots).\alpha_k$;
   thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{\nu}]$ has a summand $l(\alpha', \ldots).\alpha_k$, and we conclude the equality.
2. Case $t_i \xrightarrow{\upsilon} t_k$ and $t_k\ R\ u_l$.
   Then ε_{ij} is a blocked sum with a summand $\upsilon.v_{kl}$, and α_i has a summand $\upsilon.\alpha_k$;
   thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{\nu}]$ has a summand $\upsilon.\alpha$, with $\upsilon.t_k \approx \upsilon.u_l$, and we conclude the equality.

The proof is complete. □

We prove now that all types satisfy a standard and guarded set of equations.

Proposition 5.3 *(Equational characterisation). For every image-finite type α with free variables $\tilde{t}$ there is S standard and guarded such that $\alpha \Vdash S$, and* $\mathsf{var}(S) \subseteq \tilde{t}$*. Moreover, if t is guarded in α, then t is guarded in S.*

Proof. By induction on the structure of α. Construct the set S, standard and guarded, as in the proof of Theorem 4.1 in [37]. □

[15] Consider subfamilies of types $\widetilde{\alpha}'$ and $\widetilde{\beta}'$, corresponding respectively to $\tilde{t}'$ and $\tilde{u}'$.
[16] Consider v_n' the variable associated with $t_n' \times u_n'$, *e.g.*, $v_n' = v_{25}$ if $t_n' = t_2$ and $u_n' = u_5$.

5.2. Completeness for image-finite types

From the results in the previous subsection we establish the main result of this section: the axiom system $\mathcal{A}$ is complete with respect to the equivalence notion for image-finite types, that is, types without the parallel composition operator.

We do this in two steps, as usual: first prove the completeness of the axiom system for image-finite guarded types, and then show that every type has a provably equivalent guarded one, hence the axiomatisation is complete for all image-finite types. The former step is the critical one.

So let us prove first that the types that satisfy a set of equations are unique up to bisimulation.

Theorem 5.4 *(Unique solution of equations). If S is guarded with free variables $\tilde{t}$, then there is a type α such that $\alpha \Vdash S$. Moreover, if for some β with free variables $\tilde{t}$, $\beta \Vdash S$, then $\vdash \alpha = \beta$.*

Proof. By induction on the cardinality of S.

The base case is immediate: consider the system S: $t = \delta$ with t guarded in δ; making $\alpha \stackrel{\text{def}}{=} \mu t.\delta$, rule **R1** ensures $\mu t.\delta \Vdash S$; moreover, if there is a β with free variables t such that $\beta \Vdash S$, *i.e.* $\vdash \beta = \delta(\beta)$, then by rule **R2**, $\vdash \beta = \mu t.\delta$, as required. For the induction step proceed similarly to the proof of Theorem 4.2 in [37]. □

Theorem 5.5 *(Completeness for image-finite guarded types). If α and β are image-finite guarded types, and $\alpha \approx \beta$ then $\vdash \alpha = \beta$.*

Proof. Proposition 5.3 ensures that there is an S standard and guarded such that $\alpha \Vdash S$ and an S' standard and guarded such that $\beta \Vdash S'$. But then Theorem 5.2 guarantees that there is a single set of equations that they both satisfy, and hence the result follows using Theorem 5.4. □

Proposition 5.6 *(Reduction to guarded types). For every type α there is a guarded type β such that $\vdash \alpha = \beta$.*

Proof. One should perform a case analysis on the form of α, but since we consider only contractive types (*i.e.*, $\alpha \neq \mu t.t$), it is enough to consider types of the form $\mu t.\alpha$. The difficulty is that t may occur arbitrarily deep in α, possibly within other recursions. Therefore, it is useful to prove a stronger result (according to the proof for CCS by Milner [37]):

For every type α such that if $t \in \mathsf{fv}(\alpha)$ then $t \neq \alpha$ there is a guarded type β for which:

1. *t is guarded in β*;
2. *no free unguarded occurrence of any variable in β lies within a recursion in β*;
3. *$\vdash \mu t.\alpha = \mu t.\beta$.*

We prove this by induction on the depth of the nesting of recursions in α.

1. The first step is to remove from α free unguarded occurrences of variables occurring within recursions. By induction hypothesis, for every $\mu t'.\gamma$ in α such that the recursion depth of γ is smaller than that of α, there is a γ' for which the result above holds. Thus, no free unguarded occurrence of any variable in $\gamma'[\mu t'.\gamma/t']$ lies within a recursion. Now substitute in α every top-level $\mu t'.\gamma$ by $\gamma'[\mu t'.\gamma/t']$, obtaining a type α' that fulfils the four required conditions.
2. Then, we only need to remove the remaining free unguarded occurrence of t in α', which do not lie within recursions. A case analysis on the structure of α' leads to the conclusion $\alpha' \equiv \mu t.(\upsilon.t + \sum_{i\in I} \upsilon.\alpha_i)$; applying rule **R3** yields $\vdash \alpha' = \mu t.\sum_{i\in I} \upsilon.\alpha_i$. Repeatedly applying this procedure yields the envisage type $\mu t.\beta$, and the result follows by transitivity.

Since we conclude the proof of the stronger result, we're done. □

Corollary 5.7 *(Completeness for image-finite types). If α and β are image-finite types, and $\alpha \approx \beta$ then $\vdash \alpha = \beta$.*

Proof. Straightforward, using the previous proposition and Theorem 5.5. □

6. Final discussion

To represent with a type the behaviour of a (non-uniform, possibly distributed) concurrent object, a process algebra is a natural idea. Since a type (partially) specifies an object, three are the basic requirements:

1. method calls are the basic actions; and thus,
2. the silent action is external rather than internal, as it corresponds to an action in another object (not directly observable);

3. sums (records-as-objects) are either prefixed by method calls, representing objects' interfaces, or by the silent action, representing disabled—or blocked—objects; in short, there are neither free nor mixed sums.

Hence, actions are either method calls or the silent action, the basic process is the (possibly empty) action-prefixed sum, and the composition operators are parallel composition and recursion.

No existing process algebra has all these characteristics together. BPP is similar to, but not exactly, what we need. For the sake of simplicity and to avoid confusion we define ABT, a new process algebra. Furthermore, since the notion of observation differs from the usual one in process algebra, it leads to a new, simple, and natural notion of equivalence, **lsb**, which has a complete axiom system, at least for image-finite types. Moreover, the absence of mixed sums in ABT leads to a simpler axiomatic system than that of CCS.

To conclude, we discuss three last questions:

1. Can the proof system be complete, when considering image-infinite types?
2. Why is **lsb** our notion of type equivalence?
3. What else remains to be done?

6.1. Completeness for image-infinite types

Completeness for infinite-state types is a considerably more difficult problem. One cannot hope for completeness of axiomatisations of equivalence notions in process algebras like CCS, since the problem of checking weak notions of equivalences like bisimulation is undecidable [35,60].

The study of image-infinite (or infinite-state) systems is a lively area of concurrency theory, with several important results established [10,15,40]. Srba wrote a comprehensive survey on (un)decidability results of equivalence notions and decision problems on infinite-state systems, which he keeps up-to-date [60].

We focus our attention in two process algebras: BPA and BPP. BPA is the class of Basic Process Algebra of Bergstra and Klop [4], corresponding to the transition systems associated with Greibach Normal Form (GNF) context-free grammars, in which only left-most derivations are allowed. BPP is the class of Basic Parallel Processes of Christensen [14], which is the parallel counterpart of BPA but with arbitrary derivations. Strong bisimilarity is decidable for BPA [18] and for BPP [17,16]. However there is still no such result for weak bisimilarity on full BPA and BPP, although the result is already established for the totally normed subclasses [27], and a possible decision procedure for full BPP is NP-hard [62]. Recent results are reported by Křetínský et al. [33]. It is thus an open problem if an equivalence notion like the one we propose herein is decidable.

Nevertheless, even if ultimately decidability is an important result to ensure the applicability of our equivalence notion, we are looking for completeness, since decidability is stronger than what we need: the existence of a proof for each equation suffices.

6.2. The notion of bisimulation

Why is **lsb** our equivalence notion? Could it be different? Could we have used an existent notion? We now approach these questions.

An alternative notion of bisimulation. Consider the following definition of a bisimulation relation.

Definition 6.1 *(Label-semi-strong bisimilarity).*

1. A symmetric binary relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a *label-semi-strong bisimulation*, (**lssb**), if whenever $\alpha\ R\ \beta$ then
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}, \gamma\ (\beta \xrightarrow{l(\widetilde{\beta})} \gamma \Longrightarrow \beta'$ and $\alpha' \widetilde{\alpha}\ R\ \beta' \widetilde{\beta})$;
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists \beta'\ (\beta \Longrightarrow \beta'$ and $\alpha'\ R\ \beta')$.
2. Two types α and β are *label-semi-strong bisimilar*, and we write $\alpha \approx_{\mathbf{s}} \beta$, if there is a label-semi-strong bisimulation R such that $\alpha\ R\ \beta$.

Again, $\approx_{\mathbf{s}}$ is an equivalence relation and $\alpha \approx_{\mathbf{s}} \beta$ holds if and only if conditions 1(a) and 1(b) of the previous definition hold with R replaced by $\approx_{\mathbf{s}}$. Furthermore, **lssb** is a congruence relation (the proofs of these results are very similar to those done previously for **lsb**).

This notion differs from **lsb** by allowing unblockings after method calls (condition 1(a)). In the context of deterministic finite types the two equivalences coincide, as we have previously shown [57]. However, as we discuss in that paper, the notions do not coincide in more general transition systems, namely in non-deterministic ones.

Take the systems in Fig. 1. In $l.\upsilon.l$, the second l is only observable after the occurrence of the unblocking, which corresponds to the execution of some action in another object. There is a causal dependency between the first l, the action corresponding to the unblocking, and the second l. If the law $l.\upsilon.l = l.\upsilon.l + l.l$ holds for some equivalence notion then the

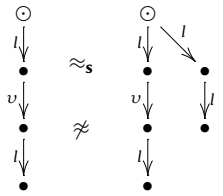

Fig. 1. lsb *vs.* **lssb**: comparing $l.\upsilon.l$ to $l.\upsilon.l + l.l$.

notion does not capture causality between action execution in different objects, and thus it is a *local* notion, whereas a notion that distinguishes the types in the law is *global* (with respect to the community of objects).

Theorem 6.2 *(Comparing* **lsb** *and* **lssb***). Label-strong bisimulation is finer than label-semi-strong bisimulation.*

Proof. Clearly, a label-strong bisimulation is also a label-semi-strong bisimulation. The converse does not hold, as, *e.g.*, the systems in Fig. 1 show. □

We have adopted **lsb** as the "right" notion of bisimulation, for it is global, and it is technically simpler. Furthermore, it is finer than **lssb**.

Relation to other notions. What is the position of **lsb** in the lattice of bisimulation equivalences? Since it is a bisimulation it is above a large spectrum of equivalence notions [66]. Obviously, its relative position varies according to the characteristics of the transition system in consideration. We focus now on CCS and ABT.

As with weak bisimulation (*wb*), **lsb** is not a congruence in CCS. However, one defines from **lsb** a congruence (let us call it *lsc*) just by demanding that a silent action should be matched by at least one silent action (*cf.* the observational congruence, *oc*). Hence, *lsc* is finer than *oc* (as **lsb** is finer than *wb*), since the laws of *lsc* are particular cases of the laws of *oc*. In CCS, the coarsest bisimulation which is still a congruence is the progressing bisimulation (*pb*) [41]. Notice that *lsc* is incomparable to *pb*, as, *e.g.*, $l.\upsilon.m \neq_{pb} l.\upsilon.\upsilon.m$ but $l.\upsilon.m \approx_{lsc} l.\upsilon.\upsilon.m$, and $l + \tau.l =_{pb} \tau.l$ but $l + \tau.l \not\approx_{lsc} \tau.l$.

In ABT, *wb* is a congruence, as the sums are prefixed. Since this setting has no mixed sums, the υ-laws are particular cases of the laws holding for *wb*. Thus, *wb* is still coarser than **lsb**, but notice that *pb* is, in this setting, finer than the previous two, since it distinguishes, *e.g.*, $l.\upsilon.m$ from $l.\upsilon.\upsilon.m$ (hence, the law **U1**—valid for **lsb** and for weak bisimulation—is not valid for *pb*).

6.3. Further work

The first priority is to find out if **lsb** is completely axiomatisable in the context of ABT. From there, apart from the decision procedure for **lsb**, two topics are interesting: a modal characterisation, to specify properties, and a notion of subtyping, to allow program refinement.

Modal characterisation. To specify/verify properties of types it is useful to have a logical characterisation of the equivalence notion. In the process algebra realm this is done with a modal action logic like the Hennessy–Milner logic [25,36]. In the same way we define a modal logic for ABT.

Definition 6.3 *(Syntax).* The grammar below defines the set $\mathcal{F}$ of *formulae* of the logic.

$$\varphi, \psi ::= \top \mid \neg\varphi \mid (\varphi \wedge \psi) \mid \langle\upsilon\rangle\varphi \mid \left\langle l(\widetilde{\alpha})\right\rangle\varphi.$$

The relation below defines when a type satisfies a formula.

Definition 6.4 *(Semantics).* The following rules inductively define the *satisfaction relation* $\models \subseteq \mathcal{T} \times \mathcal{F}$:

1. $\alpha \models \top$, for any α;
2. $\alpha \models \neg\varphi$, if not $\alpha \models \varphi$;
3. $\alpha \models \varphi \wedge \psi$, if $\alpha \models \varphi$ and $\alpha \models \psi$;
4. $\alpha \models \langle\upsilon\rangle\varphi$, if $\exists_{\alpha'}\ (\alpha \Longrightarrow \alpha'$ and $\alpha' \models \varphi)$;
5. $\alpha \models \langle l(\widetilde{\alpha})\rangle\varphi$, if $\exists_{\alpha'}\ (\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ and $\alpha' \models \varphi)$.

An equivalence relation rises naturally from the satisfaction relation.

Definition 6.5 *(Logical equivalence).* Types α and β are *logically equivalent*, $\alpha =_{lg} \beta$, if, for all φ, we have $\alpha \models \varphi$, if, and only if, $\beta \models \varphi$.

Logical equivalence is sound with respect to **lsb**. The converse direction is a conjecture. Usually, it requires assuming image-finite systems, but $\Longrightarrow$ is not image-finite.

Theorem 6.6 *(Soundness). If $\alpha =_{lg} \beta$ then $\alpha \approx \beta$.*

Proof. By induction on the structure of the formulae. □

We would like to extend this modal logic with recursion (in the lines of the modal μ-calculus [32]), study our types as logical formulae, and see how to specify and verify certain properties of systems of objects.

Subtyping. Since types are partial specifications of the behaviour of objects, the subtyping relation gives us the possibility of specifying that behaviour in more detail. In fact, the *principle of substitutability*[17] states that "a type β is a subtype of a type α, if β can safely be used in place of α" [34]. "Safely" means that the program is still typable and thus no run-time error arises. Therefore, subtyping allows the substitution of: (1) a type for one with less methods (co-variant in width), as it is safe to provide more than what is expected; and (2) a parameter type for one with more methods (contra-variant in the arguments), as it is safe to assume that the argument has less behaviour than it really has.

Instead of defining the subtyping relation via typing rules, as for instance, in [49] we propose a semantic definition. It would be interesting to define those rules and study the relationship among both notions; we leave that for future work.

Definition 6.7 *(Similarity on types).*

1. A binary relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a *label-strong simulation*, or simply a simulation, if whenever $\alpha\ R\ \beta$ we have:
   (a) $\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ implies $\exists \alpha', \widetilde{\alpha}$ $(\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ and $\alpha' \widetilde{\beta}\ R\ \beta' \widetilde{\alpha})$;
   (b) $\beta \xrightarrow{\upsilon} \beta'$ implies $\exists \beta'$ $(\alpha \Longrightarrow \alpha'$ and $\alpha'\ R\ \beta')$.
2. Type β is *label-strong similar* to type α, or α *simulates* β, and we write $\alpha \leqslant \beta$, if there is a label-strong simulation R such that $\alpha\ R\ \beta$.

A symmetric simulation is a label-strong bisimulation (Definition 2.5 in p. 70). The simulation is a *subtyping* relation, since it is a pre-order (reflexive and transitive). Thus, if α *simulates* α', we say that α is a *subtype* of α', and write $\alpha \leqslant \alpha'$.

Example 6.8. Subtyping provides flexibility, allowing to change/update (parts of) a system without compromising the overall behavioural and correctness. The following examples show, on the left-hand side, types specifying systems that can safely replace those specified by the type on the right-hand side.

1. $(n \parallel l(m)) \leqslant l(m)$ and $(n + l(m)) \leqslant l(m)$.
2. $l(m) \leqslant l(m + n)$.
3. $l(m) \leqslant \upsilon.l(m)$.
4. Recall Example 2.2 (p. 69) where

   $$\mathsf{Menu} \stackrel{\text{def}}{=} \mathit{balance} + \mathit{deposit}(\mathit{int}) + \mathit{withdraw}(\mathit{int}).$$

   It is possible to add a new functionality like money transfer without compromising the correctness of the system. Let

   $$\mathsf{Menu}' \stackrel{\text{def}}{=} \mathit{balance} + \mathit{deposit}(\mathit{int}) + \mathit{withdraw}(\mathit{int}) + \mathit{transfer}(\mathit{int}, \mathit{int}).$$

   One easily checks that $\mathsf{Menu}' \leqslant \mathsf{Menu}$.

The following result ensures that subtyping is a pre-order.

Proposition 6.9 *($(\mathcal{T}, \leqslant)$ is a pre-ordered set).*

1. $\alpha \leqslant \alpha$.
2. *If $\alpha \leqslant \beta$ and $\beta \leqslant \gamma$ then $\alpha \leqslant \gamma$.*

Proof. Straightforward, simply using the definition □

[17] AKA the Liskov substitution principle.

The operators of ABT, as well as **lsb**, preserve the simulation relation.

Proposition 6.10 *(Congruence).*

1. *Similarity is a congruence relation.*
2. **lsb** *preserves similarity.*

Proof. The proof of the first clause is standard. The proof of the second clause is trivial, since $\approx$ implies $\leqslant$. □

This pre-order relation induces an equivalence relation (if $\alpha \leqslant \beta$ and $\beta \leqslant \alpha$ then $\alpha = \beta$) that is coarser than **lsb**, since usually there are types that can simulate each other without being bisimilar. A simple example is the pair of types $\upsilon.(a+b)$ and $\upsilon.(\upsilon.a + \upsilon.(a+b))$.

We would like to define a syntactic notion of subtyping and develop a proof system via subtyping rules, sound and possibly complete with respect to the semantic notion based on simulation that we just presented.

6.4. Related work

In sequential computational settings, since they were proposed, types have been interpreted as predicates, *i.e.*, abstract behavioural specifications of a program, and have thus formal semantic interpretations [26,61]. In the context of object-oriented programming, types are used to statically guarantee semantic interoperability, capturing behavioural aspects of the specified systems. Barbara Liskov's substitution principle allows to safely replace objects of type T in a program with objects of type S, if S is a subtype of T [34]. Oscar Nierstrasz noticed that objects may exhibit non-uniform method availability (one cannot pop from an empty buffer—push should be called first), thus requiring types to represent possible sequences of method calls [47].

Concurrency theory inspires dynamic notions of typing and subtyping, often called *behavioural*. These notions have (at least) three different forms: *types and effects*, *regular types*, and *processes as types*. In the following paragraphs we briefly present each approach and compare it to ours.

Types and effects. The *type and effect discipline* is a framework for principal typing reconstruction in implicitly typed polymorphic functional languages [46,64]. An effect system extends a type system to statically describe the dynamic behaviour of a computation (its effect). Types describe what expressions compute (sets of values) and effects describe how expressions compute (behaviour). In the context of polymorphic functional languages, these systems are used to control resource usage, like memory manipulation. When such languages are concurrent, effects resemble processes, and the effect system is akin to a labelled transition system [24]. Types and effects may decrease with computation. As effects (also called *behaviours*) model communication, their decrease corresponds to consuming prefixes, which suggests an operational semantics. Thus, behaviours look like process algebra terms, an abstraction of the semantics of the language. In the context of name-passing process calculi, types and behaviours may be merged to become abstract specifications of systems behaviour. We give a detailed account of this approach ahead, when presenting process types.

Behavioural typing. Several researchers are working on this track, developing behavioural notions of typing for concurrent object calculi. We give herein a brief account of their work. Consider two main approaches:

Regular types: use a regular language as types for objects.

1. Nierstrasz characterises the traces of menus offered by (active) objects [47]. He proposes a notion of subtyping, *request substitutability*, based on a generalisation of the Liskov substitution principle by Wegner and Zdonik [69], which states that "services may be refined as long as the original promises are still upheld". According to the extension relation of Brinksma et al. [9], request substitutability is a transition relation, close to the failures model.
2. Colaço et al. propose a calculus of actors based on an extended TyCO, supporting objects that dynamically change behaviour [20–22]. The authors define a type system which aims at the detection of "orphan messages", *i.e.* messages that may never be accepted by any actor, either because the requested service is not available, or because, due to dynamic changes in an actor's interface, the requested service is no longer available. Types are interface-like, with multiplicities (how often can a method be invoked), thus without dynamic information, and the type system requires complex operations on a lattice of types. Nonetheless, they define a type inference algorithm based on set-constraints resolution, a well-known technique widely used in functional languages.
3. Najm and Nimour propose a calculus of objects that features dynamically changing interfaces [42–44]. The authors develop a typing system handling dynamic method offers in interfaces, and guaranteeing a liveness property: all pending requests are treated. Types are sets of deterministic guarded parametric equations, equipped with a transition relation, and representing infinite-state systems. A type inference algorithm is built on an equivalence relation, a compatibility relation, and a subtyping relation on types, based on the simulation and on the bisimulation relations (strong versions, thus decidable).

Process types. To capture with types behavioural aspects of a system, a natural idea, inspired by the effect analysis techniques, is to consider processes as types. Approaches in the context of concurrency, namely in process calculi, where mainly syntactical, but recently, through the combination of both type and model checking, semantic approaches have emerged, leading to behavioural type systems: types are sound abstractions of the behaviour of processes, and the analysis performed is akin to model checking. As the properties are checked on types, not on processes, they become decidable, and thus this approach benefits from the advantages of both type and model checking. Some significant works are the following.

1. Boudol proposes a dynamic type system for the Blue Calculus, a variant of the π-calculus directly incorporating the λ-calculus [6]. Types are functional and assigned to terms, in the style of Curry simple types, and incorporate Hennessy–Milner logic with recursion—modalities interpreted as resources of names. So, processes inhabit the types, and this approach captures some causality in the usage of names in a process, ensuring that messages to a name will meet a corresponding offer. Well-typed processes behave correctly, a property preserved under reduction.
2. Puntigam defines a calculus of active objects with process types that specify constraints on the ordering of messages [50, 52,51]. A static type inference system (with polynomial time complexity) ensures that all sequences of messages sent to an object are acceptable, even if the set of acceptable messages changes dynamically. Objects are syntactically constrained to a unique identity and messages are received in the order they were sent and not suppressed by deadlocks, as every object is associated with a FIFO queue. The expressiveness of types is that of a non-regular language, which is equipped with a subtyping relation.
3. Kobayashi et al. have studied deadlock and livelock detection in mobile calculi [30,31]. Channel types have information not only about their arity, but also about their usage (sequences of possible inputs and outputs), about when they should be used, and if they must be used.
4. To avoid having a dedicated proof system, tailored to the specific target property, often with its own language of types, Igarashi and Kobayashi proposed a generic framework to develop type systems to ensure various properties, the Generic Type System [29]. The language of types is the restriction-free fragment of CCS, hence types are abstract representations of a process' behaviour. Particular type systems for concrete properties emerge as instances of the generic one: a given property is captured by instantiating a general subtyping relation and by defining a consistency condition on types. One needs to prove that reduction on types preserves consistency and that consistency on types implies the desired condition on processes. This process works with safety properties (like simple arity-mismatch, race-freedom and even deadlock-freedom), but not with liveness properties, which require model checking.
   This line of work has been pursued by several authors [1,11,13,53].

Protocols types specify the sequence and form of messages passing over communication channels between a number of parties, in distributed systems. Correctness of such systems implies that protocols are obeyed. Types are terms of a simple process algebra that allows to describe one side of a communication process, like ABT. There are two main trends.

1. *Session types* allow the specification of a protocol to be expressed as a type [28,63]; when a communication channel is created, a session type is associated with it; the two parties at each end of the channel have dual types. Such a type specifies not only the data types of individual messages, but also the state transitions of the protocol and hence the allowable sequences of messages. Static type checking makes possible to verify, at compile-time, that an agent using the channel does so in accordance with the protocol.
   Unlike ABT, session types distinguish incoming from outgoing actions, but do not have a parallel composition operator. The purpose of session types is however rather different from that of ABT: to discipline communication protocols running on private channels, instead of representing the behaviour of a distributed object.
2. *Conversation types* capture the interactive behaviour of a service-based system, describing multiparty interactions [12]. The aim is to discipline the conversations between an unanticipated number of participants, ensuring at the same time progress in the presence of several interleaved conversations.
   ABT can be seen as a sublanguage of conversation types, which also uses both spatial and behavioural operators.

Concluding remarks. None of the works referred above study semantically the language of types or propose equivalence notions, so in that respect our work is original. ABT serves well as a language for partial specification of object behaviour: a term captures all the possible behaviour of a concurrent (possibly distributed) object. It may even be consider as a denotation of a labelled transition system representing such behaviour. Some of the notions of process types presented before are also suited for such representation. In particular, ABT is very similar to session types or to conversation types, thus the work presented here may be applied to those languages.

7. Conclusions

The approach to behavioural types for a concurrent calculus in other works is done either by using an existing process algebra as a language of types, or by designing a new language appropriate to a particular calculus. We instead propose a simple and natural notion of types for a general setting—systems of (non-uniform) concurrent objects—and then show that these types are adequate to the purpose, and are a process algebra with convenient properties.

The Algebra of Behavioural Types, ABT is a process algebra in the style of CCS. It is syntactically very similar to its proper subclass BPP; in particular, communication is not present. The actions have terms of the process algebra as parameters. The nature of the silent action induces an original equivalence notion, different from all other equivalences known for process algebras. Naturally the set of axioms that characterises the equivalence notion is also original. However, the proof techniques are basically the same, but some crucial proofs are simpler. The interesting aspect is that normal forms include a parallel composition, as there is no expansion law for the parallel composition of a labelled sum and a blocked type. Thus the proof of the normal form lemma and of the completeness theorem are different from those for CCS.

Some of the ideas presented in this paper, namely regarding external silent actions and label-(semi-)strong bisimulation, appeared first in [57], where however the algebra was much less tractable (*e.g.*, with non-associative sums) and no completeness results were obtained. The developments presented here are part of the first author's PhD thesis [54].

We use ABT to type non-uniform concurrent objects in TyCO, where we formalise a notion of process with a communication error that copes with non-uniform service availability [56]: we advocate that the right notion of communication error in systems of concurrent objects is that no message should be forever not understood. Using ABT as the language of types, we have developed for TyCO a static type system that assigns terms of ABT to TyCO processes, and enjoys the subject-reduction property, ensuring that typable processes are not locally deadlocked, and do not run into errors [56,54].

We believe that ABT can be used not only to type other concurrent calculi with extensions for objects, but also to type distribute calculi. To fully use its expressiveness one can define in its favourite calculus functionalities like a method update that changes the type of the method, object extension adding methods, distributed objects without uniqueness of objects' identifiers, and non-uniform objects.

Acknowledgments

Special thanks to Gérard Boudol, Ilaria Castellani, Silvano Dal Zilio, and Massimo Merro, for fruitful discussions and careful reading of parts of this document. Several anonymous referees made useful comments.

References

[1] Lucia Acciai, Michele Boreale, Spatial and behavioural types in the pi-calculus, J. Inf. Comput. 208 (2010) 1118–1153.
[2] Jos C.M. Baeten, Jan A. Bergstra, Jan W. Klop, On the consistency of Koomen's fair abstraction rule, Theoret. Comput. Sci. 51 (1–2) (1987) 129–176.
[3] Henk Barendregt, The Lambda Calculus – Its Syntax and Semantics, first ed., revised 84, North-Holland, 1981.
[4] Jan A. Bergstra, Jan W. Klop, Algebra of communicating processes with abstraction, Theoret. Comput. Sci. 37 (1) (1985) 77–121.
[5] Eike Best (Ed.), 4th International Conference on Concurrency Theory (CONCUR), Hildesheim, Germany, Lecture Notes in Comput. Sci., vol. 1243, Springer-Verlag, 1993.
[6] Gérard Boudol, Typing the use of resources in a concurrent calculus, in: R.K. Shyamasundar, Kazunori Ueda (Eds.), Proceedings of ASIAN'97, in: Lecture Notes in Comput. Sci., vol. 1345, Springer-Verlag, 1997, pp. 239–253.
[7] Gérard Boudol, The π-calculus in direct style, Higher-Order and Symbolic Computation 11 (1998) 177–208.
[8] Howard Bowman, John Derrick (Eds.), 2nd IFIP Conference on Formal Methods for Open Object-Based Distributed Systems, Canterbury, UK Chapman & Hall, 1997.
[9] Ed Brinksma, Giuseppe Scollo, Chris Steenbergen, LOTOS specifications, their implementations and their tests, in: Protocol Specification, Testing and Verification VI (IFIP), North-Holland, 1987, pp. 349–360.
[10] Olaf Burkart, Javier Esparza, More infinite results, Bull. Eur. Assoc. Theor. Comput. Sci. EATCS 62 (1997) 138–159.
[11] Luís Caires, Logical semantics of types for concurrency, in: Proceedings of the 2nd Conference on Algebra and Coalgebra in Computer Science (CALCO'07), in: Lecture Notes in Comput. Sci., vol. 4624, Springer-Verlag, 2007, pp. 16–35.
[12] Luís Caires, Hugo T. Vieira, Conversation types, Theoret. Comput. Sci. 411 (51–52) (2010) 4399–4440.
[13] Sagar Chaki, Sriram K. Rajamani, Jakob Rehof, Types as models: model checking message-passing programs, in: Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL), ACM Press, 2002, pp. 45–57.
[14] Søren Christensen, Decidability and decomposition in process algebras, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, UK, 1993.
[15] Søren Christensen, Hans Hüttel, Decidability issues for infinite-state processes—a survey, Bull. Eur. Assoc. Theor. Comput. Sci. EATCS 51 (1993) 156–166.
[16] Søren Christensen, Yoram Hirshfeld, Faron Moller, Bisimulation equivalence is decidable for basic parallel processes, in: Best [5], pp. 143–157.
[17] Søren Christensen, Yoram Hirshfeld, Faron Moller, Decomposability, decidability and axiomatisability for bisimulation equivalence on basic parallel processes, in: Proceedings of LICS'93, IEEE, Computer Society Press, 1993, pp. 386–396.
[18] Søren Christensen, Hans Hüttel, Colin Stirling, Bisimulation equivalence is decidable for all context-free processes, J. Inf. Comput. 121 (2) (1995) 143–148.
[19] Paolo Ciancarini, Alesandro Fantechi, Roberto Gorrieri (Eds.), 3rd IFIP Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, Kluwer Academic Publishers, 1999.
[20] Jean-Louis Colaço, Analyses Statiques d'un calcul d'acteurs par typage, Thèse d'État, Institut National Polytechnique de Toulouse, France, 1997.
[21] Jean-Louis Colaço, Mark Pantel, Patrick Sallé, A set constraint-based analyses of actors, in: Bowman and Derrick [8].
[22] Jean-Louis Colaço, Mark Pantel, Fabien Dagnat, Patrick Sallé, Safety analysis for non-uniform service availability in actors, in: Ciancarini et al. [19].
[23] Simon J. Gay, Malcolm J. Hole, Types and subtypes for client-server interactions, in: Proceedings of the 8th European Symposium on Programming (ESOP'99), in: Lecture Notes in Comput. Sci., vol. 1576, Springer-Verlag, 1999, pp. 74–90, full version available as Types and subtypes for correct communication in client-server systems, Technical Report TR-2003-131, Department of Computing Science, University of Glasgow, February 2003.
[24] Chris Hankin, Flemming Nielson, Hanne Riis Nielson, Principles of Program Analysis, Springer, 1999.
[25] Matthew Hennessy, Robin Milner, Algebraic laws for nondeterminism and concurrency, J. ACM 32 (1) (1985) 137–161.
[26] Roger Hindley, Basic Simple Type Theory, Cambridge University Press, 1997.
[27] Yoram Hirshfeld, Bisimulation trees and the decidability of weak bisimulations, in: Proceedings of the International Workshop on the Verification of Infinite-State Systems, in: Electron. Notes Theor. Comput. Sci. (ENTCS), vol. 5, Elsevier Science Publishers, 1997.

[28] Kohei Honda, Vasco T. Vasconcelos, Makoto Kubo, Language primitives and type discipline for structured communication-based programming, in: Chris Hankin (Ed.), Proceedings of the 7th European Symposium on Programming (ESOP'98), in: Lecture Notes in Comput. Sci., vol. 1381, Springer-Verlag, 1998, pp. 122–138.
[29] Atsushi Igarashi, Naoki Kobayashi, Generic type system for the pi-calculus, Theoret. Comput. Sci. 311 (1–3) (2004) 121–163.
[30] Naoki Kobayashi, Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness, in: Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, in: Lecture Notes in Comput. Sci., vol. 1872, IFIP, Springer-Verlag, 2000, pp. 365–389.
[31] Naoki Kobayashi, Shin Saito, Eijiro Sumii, An implicitly-typed deadlock-free process calculus, in: Catuscia Palamidessi (Ed.), CONCUR 2000: Concurrency Theory, 11th International Conference, University Park, PA, USA, in: Lecture Notes in Comput. Sci., vol. 1877, Springer-Verlag, 2000, pp. 489–503.
[32] Dexter Kozen, Results on the propositional mu-calculus, Theoret. Comput. Sci. 27 (3) (1983) 333–354.
[33] M. Křetínský, V. Řehák, J. Strejček, Refining the undecidability border of weak bisimilarity, in: Proceedings of the 7th International Workshop on Verification of Infinite-State Systems (INFINITY'05), in: Electron. Notes Theor. Comput. Sci., vol. 149, Elsevier Science Publishers, 2006, pp. 17–36.
[34] Barbara H. Liskov, Jeannette M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. (TOPLAS) 16 (6) (1994) 1811–1841.
[35] Richard Mayr, Process rewrite systems, J. Inf. Comput. 156 (1) (2000) 264–286.
[36] Robin Milner, Communication and Concurrency, Int. Ser. Comput. Sci., Prentice Hall, 1989.
[37] Robin Milner, A complete axiomatisation for observational congruence of finite-state behaviours, J. Inf. Comput. 81 (2) (1989) 227–247.
[38] Robin Milner, The polyadic π-calculus: A tutorial, in: Friedrich L. Bauer, Wilfried Brauer, Helmut Schwichtenberg (Eds.), Logic and Algebra of Specification, Proceedings of the International NATO Summer School, Marktoberdorf, Germany, 1991, in: Ser. F, NATO ASI, vol. 94, Springer-Verlag, 1993, available as Technical Report ECS-LFCS-91-180, University of Edinburgh, UK, 1991.
[39] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, part I/II, J. Inf. Comput. 100 (1992) 1–77, available as Technical Reports ECS-LFCS-89-85 and ECS-LFCS-89-86, University of Edinburgh, UK, 1989.
[40] Faron Moller, Infinite results, in: Ugo Montanari, Vladimiro Sassone (Eds.), Proceedings of CONCUR'96, in: Lecture Notes in Comput. Sci., vol. 1119, Springer-Verlag, 1996, pp. 195–216.
[41] Ugo Montanari, Vladimiro Sassone, Dynamic congruence vs. progressing bisimulation for CCS, Fund. Inform. 16 (2) (1992) 171–199.
[42] Elie Najm, Abdelkrim Nimour, A calculus of object bindings, in: Bowman and Derrick [8].
[43] Elie Najm, Abdelkrim Nimour, Jean-Bernard Stefani, Guaranteeing liveness in an object calculus through behavioral typing, in: Proceedings of FORTE/PSTV'99, Kluwer Academic Publishers, 1999.
[44] Elie Najm, Abdelkrim Nimour, Jean-Bernard Stefani, Infinite types for distributed objects interfaces, in: Ciancarini et al. [19].
[45] Uwe Nestmann, António Ravara, Semantics of objects as processes (SOAP), in: Ana Moreira, Serge Demeyer (Eds.), ECOOP'99 Workshop Reader, in: Lecture Notes in Comput. Sci., vol. 1743, Springer-Verlag, 1999, pp. 314–325, an introduction to, and summary of, the 2nd International SOAP-Workshop.
[46] Flemming Nielson, Hanne Riis Nielson, Type and effect systems, in: Correct System Design, 1999, pp. 114–136.
[47] Oscar Nierstrasz, Regular types for active objects, in: Object-Oriented Software Composition, Prentice Hall, 1995, pp. 99–121.
[48] Benjamin Pierce, Types and Programming Languages, The MIT Press, 2002.
[49] Benjamin C. Pierce, Davide Sangiorgi, Typing and subtyping for mobile processes, Math. Structures Comput. Sci. 6 (5) (1996) 409–454, an extended abstract appeared in: Proceedings of LICS'93, pp. 376–385.
[50] Franz Puntigam, Strong types for coordinating active objects, Concurrency Comput. Pract. Exp. 13 (2001) 293–326.
[51] Franz Puntigam, State inference for dynamically changing interfaces, Comput. Lang. 27 (2002) 163–202.
[52] Franz Puntigam, Christof Peter, Types for active objects with static deadlock prevention, Fund. Inform. 49 (2001) 1–27.
[53] Sriram K. Rajamani, Jakob Rehof, A behavioral module system for the pi-calculus, in: Patrick Cousot (Ed.), Static Analysis: 8th International Symposium, SAS 2001, in: Lecture Notes in Comput. Sci., vol. 2126, Springer-Verlag, 2001, pp. 375–394.
[54] António Ravara, Typing non-uniform concurrent objects, PhD thesis, Instituto Superior Técnico, Technical University of Lisbon, Portugal, 2000.
[55] António Ravara, Luís Lopes, Programming and implementation issues in non-uniform TyCO, Technical Report, Department of Computer Science, Faculty of Sciences, University of Porto, 4150 Porto, Portugal, 1999, presented at the Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours (OOSDS'99), Satellite event of the 1st Conference on Principles, Logics and Implementations of High-Level Programming Languages (PLI'99), http://www.tec.informatik.uni-rostock.de/IuK/congr/oosds99/program.htm.
[56] António Ravara, Vasco T. Vasconcelos, Typing non-uniform concurrent objects, in: Catuscia Palamidessi (Ed.), CONCUR 2000: Concurrency Theory, 11th International Conference, University Park, PA, USA, in: Lecture Notes in Comput. Sci., vol. 1877, Springer-Verlag, 2000, pp. 474–488, extended version available as DM-IST Research Report 12/2000, Portugal.
[57] António Ravara, Pedro Resende, Vasco T. Vasconcelos, Towards an algebra of dynamic object types, in: Semantics of Objects as Processes (SOAP), in: BRICS Notes Ser., vol. NS-98-5, Danish Institute of Basic Research on Computer Science (BRICS), 1998, pp. 25–30.
[58] Arend Rensink, Bisimilarity of open terms, J. Inf. Comput. 156 (2000).
[59] Davide Sangiorgi, An interpretation of typed objects into typed π-calculus, J. Inf. Comput. 143 (1) (1998) 34–73, earlier version published as Rapport de Recherche RR-3000, INRIA, August 1996.
[60] Jiri Sbrna, Roadmap of infinite results, 2008.
[61] Jonathan Seldin, The logic of church and curry, in: The Handbook of the History of Logic, vol. 5, Elsevier, 2008.
[62] Jitka Stříbrná, Hardness results for weak bisimilarity of simple process algebras, in: Proceedings of MFCS'98 Workshop on Concurrency, in: Electron. Notes Theor. Comput. Sci. (ENTCS), vol. 18, Elsevier Science Publishers, 1998.
[63] Kaku Takeuchi, Kohei Honda, Makoto Kubo, An interaction-based language and its typing system, in: Parallel Architectures and Languages Europe, in: Lecture Notes in Comput. Sci., vol. 817, Springer-Verlag, 1994.
[64] Jean-Pierre Talpin, Pierre Jouvelot, The type and effect discipline, J. Inf. Comput. 111 (2) (1994) 245–296, an extended abstract appeared in: Proceedings of LICS'92, pp. 162–173.
[65] Rob J. van Glabbeek, A complete axiomatization for branching bisimulation congruence of finite-state behaviours, in: Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS'93), in: Lecture Notes in Comput. Sci., vol. 711, Springer-Verlag, 1993, pp. 473–484.
[66] Rob J. van Glabbeek, The linear time—branching time spectrum II (the semantics of sequential systems with silent moves), in: Best [5], pp. 66–80.
[67] Vasco T. Vasconcelos, Kohei Honda, Principal typing schemes in a polyadic π-calculus, in: Best [5], pp. 524–538.
[68] Vasco T. Vasconcelos, Mario Tokoro, A typing system for a calculus of objects, in: Proceedings of the 1st International Symposium on Object Technologies for Advanced Software, in: Lecture Notes in Comput. Sci., vol. 742, Springer-Verlag, 1993, pp. 460–474.
[69] Peter Wegner, Stanley B. Zdonik, Inheritance as an incremental modification mechanism or what like is and isn't like, in: Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway, in: Lecture Notes in Comput. Sci., vol. 322, Springer-Verlag, 1988, pp. 55–77.
[70] Mingsheng Ying, A shorter proof to uniqueness of solutions of equations (note), Theoret. Comput. Sci. 216 (1999) 395–397.
[71] Nobuko Yoshida, Graph types for monadic mobile processes, in: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FST/TCS), in: Lecture Notes in Comput. Sci., vol. 1180, Springer-Verlag, 1996, pp. 371–386, extended version as Technical Report ECS-LFCS-96-350, University of Edinburgh.