An Algebra of Behavioural Types

António Ravara [*]   Pedro Resende [*]   Vasco T. Vasconcelos [†]

Abstract

We propose a process algebra, the Algebra of Behavioural Types, as a language for typing concurrent objects in process calculi. 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 inhabit 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.

Contents

[*]Center for Logic and Computation, Department of Mathematics, Instituto Superior Técnico, Lisbon, Portugal. Email: {amar,pmr}@math.ist.utl.pt

[†]Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal.
Email: vv@di.fc.ul.pt

1 Introduction

Types in a programming language are used to discipline the computational mechanism of the language, disallowing interactions that may lead to erroneous operations. Examples of such errors are the assignment of some value of an invalid type to a variable, or the invocation of a non-existing method in an object. Therefore, the types are abstract representations of the correct behaviour of the various entities of a program, constituting partial behavioural specifications. A programming language is *type safe* if it is equipped with a (static) type system that guarantees a safety property known as Curry's subject-reduction: if a program is typable, then the computation mechanism preserves the typability of all the programs resulting from the intermediate steps. Thus, it results as a corollary that subject reduction guarantees the absence of run-time errors in well-typed programs. In sequential and functional languages, types are assigned to the terms of the language. The information that a type encode can be very simple (e.g. a set of values, like booleans or integers), more elaborate (e.g. a function, like from integers to booleans), or can even be some complex structure (e.g. a graph or a term of a process algebra), depending on the purpose of the type system. They can ensure a wide range of properties, from basic ones, like all operations are invoked with the adequate arguments, to more elaborate ones, like guaranteeing termination or deadlock freedom. In systems of objects, the types record the methods of the 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 (non-existence, wrong number the arguments, or arguments of incorrect types).

To be able of ensuring a safety result like subject reduction for example, for a program, one needs mathematical tools to deal with, and reason about, these notions of types. Ideas, concepts, and techniques from the typed lambda-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 [MPW92]. 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 very suitable for studying and representing object-oriented programming. Thus, not surprisingly, there are many works on the semantics of (concurrent) objects as (mobile) processes [NR99]. On the other hand, process calculi provide: (1) structural operational semantics—an essential element for reasoning about the correctness 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 [Mil93, PS96, VH93]. The role of a type system in a mobile calculus is two-fold: (1) it avoids communication errors, due to arity mismatch; and (2) it allows refinements to the algebraic theory, leading to specialised behavioural equivalences. Nonetheless, the refereed systems provide little information about process behaviour, and to ensure more than the usual safety properties one needs richer notions of types. Note that then the types should capture causality of actions, as properties like deadlock or even livelock are global properties (and not compositional—as it is usually the case of liveness properties). Thus, the types should capture the flow of the processes, being themselves processes [Bou98, GH99, HVK98, IK01, Kob00, Yos96]. Note that some of these systems assign types to processes, not to names.

In a name-passing calculus of objects such as TyCO [VT93] or π_a^V [San98], processes denote the behaviour of a community of interacting objects, where each object has a location identified by a name. Statically detecting "method-not-understood" errors is a more delicate problem in systems of (possibly distributed) concurrent objects. 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? In the beginning of the 90's, 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) [Nie95]. Objects exhibiting methods that may be enabled or disabled, according to their internal state, are very common in object-oriented programming, e.g. 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. Therefore, a static notion of typing, like interfaces-as-types, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. A rigid interface, exhibiting all the methods of an object, gives misleading information about the functionality of the object. Accordingly, Nierstrasz proposes the use of a regular language to type active objects, i.e., objects that may dynamically change behaviour. The purpose is to characterise all the traces of menus offered by those objects and define a notion of behavioural subtyping. He proposes as notion of subtyping a relation called '*request substitutability*', which is based on a generalisation of the '*principle of substitutability*' by Wegner and Zdonick [WZ88], which states that "services may be refined as long as the original promises are still upheld". Despite having developed a calculus of objects, Nierstrasz did not apply these ideas in a type system for it, neither did he show how to model non-uniform objects. This task has be taken by several authors [Bou98, Col97, CPS97, CPDS99, NN97, NNS99b, Pun96, Pun97, Pun99, RL99, RV00]. Some of these works propose a specific calculus and develop a particular language of types for that calculus to guarantee some envisaged property.

The aim 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. Since non-uniform types should capture the behaviour of objects, the types themselves can be modelled as processes. Then, several questions arise: what is the appropriate syntax and operational semantics? what is a good notion of behavioural equality? This paper addresses theses questions, proposing a (partial) solution: we develop herein the *Algebra of Behavioural Types*, ABT, where a type characterises all possible life cycles of an object: it 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, and reflect a dependency of the interface of an object upon its internal state, conveying information about dynamic properties of objects. Hence, the type of an object is a partial representation of its behaviour, modelled as a labelled transition system; the operational semantics describes how the execution of a method yields a new type. Therefore, a type may express temporal properties like ordering sequences of events. 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 internally enabled. Hence, this notion of behaviour is synchronous, as the gnome can detect refusals of methods. The action of unblocking an object, 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 [Mil89a]. Naturally, the resulting notion of equivalence has an intuition different from the one for bisimulations in CCS, since the observer is internal, rather than external: it distinguishes a blocked from an unblocked type, but does not distinguishes between types blocked by a different number of υs—does not count unblockings. Accordingly, we define a notion of bisimulation, strong on labels and weak on silent actions, and reach a set of algebraic laws different from Milner's τ-laws. The proof system based on these laws is complete with respect to the notion of bisimulation, at least for image-finite types.

ABT is similar to the Basic Parallel Processes, BPP [Chr93], 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, 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 an 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 the 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.

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 stands for an interface of an object—the collection of methods it offers. As we allow the same label to appear more than once (possibly with different continuations under the prefix), 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 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 type due to some action in other 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 realm of process algebra any equivalence notion build with 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* l, m, possibly subscripted.

Definition 2.1 (Non-deterministic sequential finite types).
The following grammar defines the set $\mathcal{T}_{sf}$ 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 possessing parameters of type $\widetilde{\alpha}$; the type α under the prefix prescribes the behaviour of the object after the execution of the method l with parameters of type $\widetilde{\alpha}$. A term of the form $\upsilon.l(\widetilde{\alpha}).\alpha$ is a *blocked method type*, the type of an unavailable method type $l(\widetilde{\alpha}).\alpha$. Thus, 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 released, 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.* Consider also the following abbreviations:

1. $\mathbf{0}$ 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. $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$.

Henceforth we call *blocked types* the blocked sums. A non-empty interface presents the active methods of an object (having that type).

DEFINITION 2.2 (INTERFACE OF A TYPE).
In a labelled sum $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ the set $\{l_i(\widetilde{\alpha}_i) \mid i \in I\}$ is the interface *of the object type, denoted by $\mathsf{int}(\alpha)$. The interface of a blocked type is empty.*

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

The action υ denotes a silent transition that releases a blocked object; a action like $l(\widetilde{\alpha})$ 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 $\mathcal{T}_{sf}$.*

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

ACT is in fact an axiom-schema that captures two cases:

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

Terminology. Some terminology regarding the transition relation will make the proofs clearer.

1. If $\alpha \xrightarrow{l(\tilde{\alpha})} \alpha'$ or $\alpha \xrightarrow{\upsilon} \alpha'$ then α' is a *derivative* of α. In the first case it is an l-derivative and the second an υ-derivative.

2. Types of the form $\sum_{i \in I} l_i(\tilde{\alpha}_i).\alpha_i$ are *unblocked*, and types of the form $\sum_{i \in I} \upsilon.\alpha_i$ are *blocked*. If $I \neq \emptyset$, then $\sum_{i \in I} l_i(\tilde{\alpha}_i).\alpha_i$ is *strictly unblocked*, and $\sum_{i \in I} \upsilon.\alpha_i$ is *strictly blocked*.

Notation. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}^{*}$ and $\overset{\upsilon}{\Longrightarrow}$ denote $\xrightarrow{\upsilon}^{+}$.

2.3 Equivalence notion

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). This is because, although υ is supposed to be unobservable, we assume that an internal observer—the object's gnome—can 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 [Mil89a] and progressing bisimulation [MS92]. 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 [Mil89a] and rooted bisimulation [BBK87] are unsuitable. Also, notice that all the above mentioned equivalences, with the exception of weak bisimulation, are finer than necessary, because they are congruences with respect to binary sums, as in CCS [Mil89a], 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.

Thus, the intuition is: two types are *not bisimilar* if they have different interfaces or, after some matching transitions, their derivatives have different interfaces.

DEFINITION 2.5 (BISIMILARITY ON TYPES).

1. *A symmetric binary relation* $R \subseteq \mathcal{T}_{sf} \times \mathcal{T}_{sf}$ *is a* label-strong bisimulation*, or simply a bisimulation, if, whenever* $\alpha \, R \, \beta$*:*
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha'\widetilde{\alpha} \, R \, \beta'\widetilde{\beta})^1$*;*
   (b) $\alpha \xrightarrow{v} \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \, R \, \beta')$*.*
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* $\alpha \, R \, \beta$*.*

The usual properties of a bisimilarity hold.

PROPOSITION 2.6.
Label-strong bisimilarity is an equivalence relation and the largest bisimulation.

PROOF. The proof is standard (cf. [Mil89a], Proposition 4.2). □

Label strong bisimilarity is a greatest fixed point.

PROPOSITION 2.7 (FIXED POINT).
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. $\alpha \xrightarrow{v} \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \approx \beta')$*.*

PROOF. The proof is standard (cf. [Mil89a], Proposition 4.16). □

The following characterisation of *lsb* is useful for proofs.

PROPOSITION 2.8 (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 $\alpha \xrightarrow{v}^n \alpha'$, for some natural number n. The proof is by induction on n. □

^{1}Let $(\alpha_1 \cdots \alpha_n) \, R \, (\beta_1 \cdots \beta_n)$ denote $\alpha_1 \, R \, \beta_1, \ldots,$ and $\alpha_n \, R \, \beta_n$; the order is irrelevant.

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

Proposition 2.9 (Congruence).
Let $\widetilde{\alpha}_i \approx \widetilde{\beta}_i$ and $\alpha_i \approx \beta_i$ for all i in some 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 (higher-order) 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* (Proposition 2.7). $\square$

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 2.10 (Axiomatisation).
The following equivalences inductively define the axiomatic system $\mathcal{A}_{sf}$.

Commutativity: *for any permutation* $\sigma : I \to I$ *we have* $\sum_{i\in I} \pi_i.\alpha_i \approx \sum_{i\in I} \pi_{\sigma(i)}.\alpha_{\sigma(i)}$;

Idempotence: $\pi.\alpha + \pi.\alpha + \beta \approx \pi.\alpha + \beta$;

υ-law: $\upsilon.\sum_{i\in I} \upsilon.\alpha_i \approx \sum_{i\in I} \upsilon.\alpha_i$.

Proof. It is straightforward to build the respective bisimulations. $\square$

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

Proof. Follows from Proposition 2.9 and of Prop/Definition 2.10. $\square$

Remark. The novelty of this system is the υ-law: 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 the υ-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 $\vdash \sum_{i\in I} l_i(\widetilde{\alpha_i}).\alpha_i = \sum_{j\in J} m_j(\widetilde{\beta_j}).\beta_j$.

   The soundness of these rules is a consequence of Proposition 2.9, 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 τ-law $\alpha.\tau.P = \alpha.P$ when α is τ.

   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.
   (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 one does not hold, as the following counter-example 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 path is known: via normal forms.

Definition 2.12 (Normal forms of sequential finite types).

1. *A type α is* saturated *if $\alpha \overset{\upsilon}{\Longrightarrow} \alpha'$ 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) *$\alpha \equiv \mathbf{0}$;*
   (b) *$\alpha \equiv \sum_{i \in I} \upsilon.\alpha_i$ and each α_i is a normal form;*
   (c) *$\alpha \equiv \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i$, where each component of each sequence $\alpha_i\widetilde{\alpha}_i$ is a normal form.*

The proof technique to establish the next result uses the notion of depth of a type.

Definition 2.13 (Depth of a type).
The following rules inductively define the depth of a type.

$$\begin{aligned}
\mathit{depth}(\mathbf{0}) &= 0, \\
\mathit{depth}(\textstyle\sum_{i \in I} \upsilon.\alpha_i) &= 1 + \max\{\mathit{depth}(\alpha_i) \mid i \in I\}, \text{ and} \\
\mathit{depth}(\textstyle\sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i) &= 1 + \max\{\mathit{depth}(\widetilde{\alpha}_i) + \mathit{depth}(\alpha_i) \mid i \in I\},
\end{aligned}$$

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

We must prove that all types have equivalent normal forms, a crucial lemma to achieve completeness.

Lemma 2.14 (Normal form lemma).
For all types α there exists a normal form α' such that $\vdash \alpha = \alpha'$, with $\mathit{depth}(\alpha') \leq \mathit{depth}(\alpha)$.

Proof. By induction on the depth of α. The base case is when $\mathsf{depth}(\alpha) = 0$; so, $\alpha \equiv \mathbf{0}$, and it is a normal form by definition. Otherwise, if $\alpha \equiv \sum_{i \in I} \pi_i.\alpha_i$ then, by induction hypothesis, for each α_i there exists a normal form α'_i such that $\vdash \alpha_i = \alpha'_i$, with $\mathsf{depth}(\alpha'_i) \leq \mathsf{depth}(\alpha_i)$. 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, $\mathsf{depth}(\sum_{i \in I} l_i(\widetilde{\alpha}'_i).\alpha'_i) \leq \mathsf{depth}(\alpha)$.
2. Case $\pi_i \equiv \upsilon$, for all i.

   We have to guarantee saturation. For each i such that $\alpha'_i \xrightarrow{\upsilon}$, there is a $J_i \neq \emptyset$ such that $\alpha'_i \equiv \sum_{j \in J_i} \upsilon.\alpha_j$ and, for each $j \in J_i$ we have $\sum_{i \in I} \upsilon.\alpha'_i \xrightarrow{\upsilon.\upsilon} \alpha_j$. By idempotence and the υ-law, we have $\vdash \sum_{i \in I} \upsilon.\alpha'_i = \sum_{i \in I} \upsilon.\alpha'_i + \upsilon.\alpha_j$. We can add $\upsilon.\alpha_j$ to $\sum_{i \in I} \upsilon.\alpha'_i$ for all such α_j and thus obtain a normal form of α whose depth clearly equals that of $\sum_{i \in I} \upsilon.\alpha'_i$.

Since there are no remaining cases, the proof is complete. □

We are now in a position to prove the main result of this chapter: the completeness of the axiomatic system $\mathcal{A}_{sf}$ with respect to the notion of equivalence.

THEOREM 2.15 (COMPLETENESS OF $\mathcal{A}_{sf}$).
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, following Lemma 2.14).

Base case: $\mathsf{depth}(\alpha) = \mathsf{depth}(\beta) = 0$;

Then, $\alpha \approx \mathbf{0} \approx \beta$, and it follows from the reflexivity law of the proof system that $\vdash \alpha = \beta$.

Induction step: there are two cases under consideration.

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 $\alpha'\widetilde{\alpha} \approx \beta'\widetilde{\beta}$; 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.
   Thus, if $\alpha \xrightarrow{v} \alpha'$ then $\beta \xrightarrow{v} \beta'$, and again we conclude that $\vdash \alpha' = \beta'$.

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

3 CONCURRENT FINITE TYPES

We extend the algebra of non-deterministic sequential finite types with 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 input processes with the same location. In the parallel composition of types each component is the type of an element of the parallel composition of input processes. Moreover, it also allows to distinguish enabled from blocked types, a crucial feature of this calculus.

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. In particular, we are 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, but we believe that it is possible that the axiomatic system without it is still complete.

An alternative proof of completeness uses only induction, but 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 prove turned out to be quite difficult, due to the highly combinatoric nature of the problem. Once 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 chapter.

Definition 3.1 (Concurrent finite types).
The following grammar defines the set $\mathcal{T}_f$ 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 $(l.m) \parallel n$.

Terminology. Unblocked types are now not only labelled sums, but also parallel compositions involving (at least) a labelled sum. Types that are not of the form $\sum_{i\in I}\upsilon.\alpha_i$ are *unblocked*. Furthermore, each α_i is *strictly blocked* if $I \neq \emptyset$, and is *strictly unblocked* if it has an l-derivative.[2]

Therefore, if α is an unblocked type, its interface is the union of the interfaces of the labelled sums that are involved in the parallel composition.

Definition 3.2 (Interface of a type).
The following rules inductively define the interface of a type.

$$\begin{aligned} \mathit{int}(\textstyle\sum_{i\in I} \upsilon.\alpha_i) &= \emptyset \\ \mathit{int}(\textstyle\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i) &= \{l_i(\widetilde{\alpha}_i) \mid i \in I\} \\ \mathit{int}(\alpha_1 \parallel \alpha_2) &= \mathit{int}(\alpha_1) \cup \mathit{int}(\alpha_2) \end{aligned}$$

[2]Syntactic characterisations, e.g. via grammars, are easily definable.

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.*

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

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 [3]*.*

Proof. It is easy to see that the relation $R \overset{\text{def}}{=} \{(\alpha \parallel \beta, \alpha' \parallel \beta) \mid \alpha \approx \alpha'\}$ is a bisimulation, and thus that $\parallel$ preserves $\approx$. 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 $\delta \equiv \alpha_1 \parallel \beta$;

1. if $\pi \equiv l(\widetilde{\alpha})$, since by hypothesis $\alpha \approx \alpha'$ there are $\widetilde{\alpha}', \alpha'_1$ such that $\widetilde{\alpha} \approx \widetilde{\alpha}'$ and $\alpha_1 \approx \alpha'_1$, hence $(\alpha_1 \parallel \beta, \alpha'_1 \parallel \beta) \in R$;
2. if $\pi \equiv \upsilon$, the reasoning is similar to that of the previous case.

Case $\beta \xrightarrow{\pi} \beta_1$ and $\delta \equiv \alpha \parallel \beta_1$;

then $\alpha \parallel \beta \xrightarrow{\pi} \alpha \parallel \beta_1$ and obviously, also $\alpha' \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta_1$, hence $(\alpha \parallel \beta_1, \alpha' \parallel \beta_1) \in R$.

By symmetry we conclude that R is a *lsb*. $\square$

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

Lemma 3.5 (Active types).
$\alpha \not\approx \mathbf{0}$*, if and only if,* $\exists_{\alpha', l(\widetilde{\alpha})}\, \alpha \Longrightarrow \alpha' \xrightarrow{l(\widetilde{\alpha})}$.

Proof. The 'if' direction follows easily by absurd, considering $\alpha' \approx \mathbf{0}$. The 'only-if' direction is by induction on the structure of α. $\square$

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

[3] cf. Definition 2.5.

Definition 3.6 (Depth of a type).
The rules of Definition 2.13, together with the rule below inductively define the depth of a type.

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

The following result is a consequence of the equivalence relation: a non-active type is the neutral element of the parallel composition.

Proposition 3.7 (Neutral element of parallel composition).
Let α be a labelled sum and γ be a blocked type, such that $\alpha \approx \beta \parallel \gamma$, for β a sum (either labelled or blocked). Then $\gamma \approx \mathbf{0}$.

Proof. If β is blocked then clearly $\alpha \approx \mathbf{0}$, hence $\gamma \approx \mathbf{0}$. For the remainder of this proof, let us assume that β is unblocked. The proof follows by induction on $\mathsf{depth}(\alpha)$.

Base case: if $\mathsf{depth}(\alpha) = 1$, then $\alpha \approx \beta$ and it follows easily that $\gamma \approx \mathbf{0}$.

For the induction step we assume $\gamma \not\approx \mathbf{0}$ and we derive a contradiction as follows. By Lemma 3.5, $\gamma \Longrightarrow \gamma' \xrightarrow{l(\widetilde{\gamma})} \gamma''$ for some γ', l, $\widetilde{\gamma}$, and γ''. Since α is unblocked and $\alpha \approx \beta \parallel \gamma$, then $\alpha \approx \beta \parallel \gamma'$ and $\alpha' \approx \beta \parallel \gamma''$, for some l-derivative α' of α; we are assuming that β is strictly unblocked and thus so is α'. Furthermore, since $\alpha \approx \beta \parallel \gamma$ and γ is blocked, there is some l-derivative β' of β such that $\alpha' \approx \beta' \parallel \gamma$ where β' is strictly unblocked because α' is and γ is blocked. Finally, $\mathsf{depth}(\alpha') < \mathsf{depth}(\alpha)$, and thus by induction we conclude $\gamma \approx \mathbf{0}$, a contradiction.

The proof is complete. □

3.3 Algebraic characterisation

We extend the axiomatic system $\mathcal{A}_{sf}$ 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 3.8 (Axiomatisation).
The laws of Prop/Definition 2.10, together with the following laws inductively define the axiomatic system $\mathcal{A}_f$.[4]

CM $\langle \mathcal{T}_f/\!\approx, \parallel, \mathbf{0}\rangle$ *is a commutative monoid;*

EXP1 $\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j \approx$
$\sum_{i\in I} \upsilon.(\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j) + \sum_{j\in J} \upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \beta_j)$;

[4]Notice that the υ-law of system $\mathcal{A}_{sf}$ is now law U1.

EXP2 $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j \approx$

$\quad\sum_{i\in I} l_i(\widetilde{\alpha}_i).(\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \sum_{j\in J} m_j(\widetilde{\beta}_j).(\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \beta_j)$;

U1 $\upsilon.\sum_{i\in I} \upsilon.\alpha_i \;\approx\; \sum_{i\in I} \upsilon.\alpha_i$;

U2 $\upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) \approx$

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

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

We ensure first that the rules above are sound.

Theorem 3.9 (Soundness of $\mathcal{A}_f$).
If $\vdash \alpha = \beta$ *then* $\alpha \approx \beta$.

Proof. A consequence of Proposition 2.9 and of Prop/Definition 3.8. □

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

1. Allow arbitrary labelled sums, i.e. mixing labels and υ in sums, and having a single expansion law; the proof of completeness is standard;
2. Add a new inference rule to the equational logic and proceed as usual.

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

3. Keep the proof system and use induction directly.

The first alternative is somewhat unnatural, since labelled sums represent interfaces of objects, and an arbitrary sum does not represent a valid object interface. Hence, we do not follow this path. 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 type; thus the proofs of the normal form lemma and of the completeness theorem are different from those for CCS. The third alternative turns out to be quite difficult, thus we proceed now according to the second alternative. First we have to show that the new inference rule is sound.

Lemma 3.10 (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 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 $\widetilde{\alpha}_i \approx \widetilde{\beta}_k$, as by hypothesis $\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k$ and the axiomatic system is sound, then also $\beta \xrightarrow{m_k(\widetilde{\beta}_k)} \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.
   Moreover, $\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j \approx \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, again because by hypothesis we have $\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, and the system is sound.

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 $\alpha \approx \beta$. □

Definition 3.11 (Normal forms of concurrent finite types).
A type α is a normal form *if it is saturated (cf. Definition 2.12), and furthermore, one of the following conditions holds:*

1. $\alpha \equiv \mathbf{0}$;
2. $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ *and each α_i is a normal form;*
3. $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ *and each component of each $\alpha_i\widetilde{\alpha}_i$ is a normal form;*
4. $\alpha \equiv \alpha_1 \parallel \alpha_2$, *where α_1 and α_2 are respectively as in 2 and 3 above, with $\alpha_1, \alpha_2 \not\approx \mathbf{0}$ and $I, J \neq \emptyset$.*

Again, we show that all types have equivalent normal forms.

Lemma 3.12 (Normal form lemma).
For all α there exists a normal form α' such that $\vdash \alpha = \alpha'$, with $\mathsf{depth}(\alpha') \leq \mathsf{depth}(\alpha)$.

Proof. By induction on the depth of α. The proof is similar to that of Lemma 2.14, so we omit the common cases. Hence, we only have two cases to consider.

Case $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$.

By induction hypothesis, for each α_i there exists a normal form α'_i such that $\vdash \alpha_i = \alpha'_i$, with $\mathsf{depth}(\alpha'_i) \leq \mathsf{depth}(\alpha_i)$. The interesting case now is when

$$\alpha'_i \equiv (\sum_{j\in J_i} \upsilon.\alpha_j) \parallel \beta, \text{ with } \beta \equiv \sum_{k\in K_i} l_k(\widetilde{\beta}_k).\beta_k \text{ and } J_i, K_i \neq \emptyset\,.$$

Then, we have $\sum_{i\in I} \upsilon.\alpha'_i \xrightarrow{\upsilon.\upsilon} \alpha_j \parallel \beta$, and, by U2, $\vdash \sum_{i\in I} \upsilon.\alpha'_i = \sum_{i\in I} \upsilon.\alpha'_i + \upsilon.(\alpha_j \parallel \beta)$. 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 2.15, 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, $\alpha_1 \parallel \alpha_2$ is a normal form. Moreover, notice that $\mathsf{depth}(\alpha_1) \leq \mathsf{depth}(\sum_{i\in I} \upsilon.\alpha_i)$ and $\mathsf{depth}(\alpha_2) \leq \mathsf{depth}(\sum_{j\in J} l_j(\widetilde{\beta}_j).\beta_j)$.

Therefore, $\mathsf{depth}(\alpha_1 \parallel \alpha_2) \leq \mathsf{depth}(\alpha)$.

Since we inspected all cases, the proof is complete. □

Proposition 3.13.
If $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ is a normal form, then no α_i is a parallel composition.

Proof. Assume that $\alpha_1 \equiv \sum_{j\in J_i} \upsilon.\alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, which is a normal form. Then $\alpha \xrightarrow{\upsilon\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$. Since α is saturated it follows that $\alpha \xrightarrow{\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, i.e. 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, $\alpha_j \not\approx \mathbf{0}$. Since the types are finite, applying repeatedly this procedure lead us to $\alpha_i \equiv \sum_{m\in M_i} l_m(\widetilde{\alpha}_m).\alpha_m \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, for some $i \in I$, which is not a normal form, and we attain an absurd. □

We are now in a position to prove the completeness of $\mathcal{A}_{sf}$, the main result of this section.

Theorem 3.14 (Completeness of $\mathcal{A}_{sf}$).
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 3.12).

Taking into account the proof of Theorem 2.15, 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. If $J \neq \emptyset$, then, by Lemma 3.7, also $L \neq \emptyset$.
3. Thus, consider $I, J, K, L \neq \emptyset$, and consider the blocked sums not equivalent to $\mathbf{0}$. The proof depends on the transition done by 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 $\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.

   (b) case $\alpha \xrightarrow{\upsilon} \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j$.
   If $\beta \overset{\upsilon}{\Longrightarrow} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$ then, as β is saturated, $\beta \xrightarrow{\upsilon} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$, and 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 like $\mu t.\alpha$ denotes a solution of the equation $t = \alpha$. These recursive types allow us to characterise the behaviour of persistent objects, as well as that of objects living in methods of persistent objects.

In this section, we present the syntax and operational semantics of ABT, add to the axiomatic system of the previous chapter three new laws regarding the behaviour of recursive types, laws that we prove correct. The next section shows that the axiomatic system is complete for image-finite types.

4.1 Syntax

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

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

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

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

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

Definition 4.2 (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*.*

A type without free variables is said *closed*; otherwise it is *open. Alpha-conversion* in a type $\mu t.\alpha$ is defined as usual.

Notation. To simplify our work, we use the following conventions.

1. Assume a variable convention like in Barendregt [Bar84], and that types are equal up-to alpha-conversion. Moreover, $\mathsf{fv}(\alpha)$ denotes the set of variables that occur free in α and $\mathsf{var}(\alpha)$ denotes the set of all variables of the type α.
2. The type $\alpha[\beta/t]$ denotes the substitution in α of β for the free occurrences of t. Furthermore, the type $\alpha\{\widetilde{\beta}/\tilde{t}\}$ denotes the simultaneous substitution[5] 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 $\alpha[\beta/t]$.

[5]Standard notion (see, for instance, Barendregt [Bar84]).

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

$$\text{Rpar} \quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \text{Lpar} \quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'} \qquad \text{Rec} \quad \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

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

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

Definition 4.3 (Guarded variables and guarded types).

1. *A free variable t* is guarded in α *if all its occurrences are within some label-prefixed part of α.*
2. *A type α* is guarded *if all its free variables are guarded. Otherwise, we say α* is unguarded.

Example 4.4. *The variable t is guarded in $l(t).\alpha$, in $l(\widetilde{\alpha}).t$, and in $l(\upsilon.t).\mathbf{0}$, but not in $\upsilon.t$.*

4.2 Operational semantics

Assume the set of labels defined in 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 4.5 (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 page 9), only applies to closed terms. Following the usual approach (see, e.g. [Ren]), we extend it to open terms by requiring them to be bisimilar if all their closed instantiations are bisimilar.

Definition 4.6 (Bisimilarity on open types). *Let $\mathsf{fv}(\alpha) \cup \mathsf{fv}(\beta) \subseteq \{\tilde{t}\}$. Then, $\alpha \approx \beta$ if, for all indexed sets of closed types $\widetilde{\gamma}$, we have $\alpha[\widetilde{\gamma}/\tilde{t}] \approx \beta[\widetilde{\gamma}/\tilde{t}]$.*

It is straightforward to verify that Propositions 2.6 and 2.7 still hold, i.e., that this new definition of *lsb* is an equivalence relation and a fix point.

An important property is *substitutivity*: according to Rensink [Ren], a relation is substitutive if it is preserved by insertion— replacing 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. To prove this result, as well as others in this section, we use the technique of "transition induction"—see [Mil89a]—an induction on the maximum length of the derivation of the transition.

PROPOSITION 4.7 (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 instead of 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 4.8 (UNFOLDING).
$\mu t.\alpha \approx \alpha[\mu t.\alpha/t]$.

PROOF. Immediate, since by the rule REC both have the same transitions. □

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

PROPOSITION 4.9 (PRESERVATION OF *lsb* BY μ).
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 $\mu t.\alpha \xrightarrow{v} \alpha'$. 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. $\square$

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. one to 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 4.10 (The axiomatic system $\mathcal{A}$).
The laws of Prop/Definition 3.8, together with the following recursion laws, inductively define the axiomatic system $\mathcal{A}$.

R1 $\mu t.\alpha \approx \alpha[\mu t.\alpha/t]$;

R2 *if* $\beta \approx \alpha[\beta/t]$ *then* $\beta \approx \mu t.\alpha$, *provided that* α *is guarded;*

R3 $\mu t.(v.t + \sum_{i\in I} v.\alpha_i) \approx \mu t.\sum_{i\in I} v.\alpha_i$.

Remark. Laws R3 and R5 of CCS 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 law R4. The soundness of the axioms above is not a trivial result. To prove it, one has first to ensure that *the equations have unique solutions*, i.e.,

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

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. Law R2 of Prop/Definition 4.10 is a corollary of the uniqueness of the solutions of equations and of law R1 (which is the unfolding lemma—just proved). To prove law R3, one simply has to build the respective bisimulation. $\square$

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 slightly simpler proof [Yin99].

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

Definition 4.11 (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 4.12 (Lsb up to $\approx$ is a bisimulation).
Let R be an lsb up to $\approx$. Then,

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

Proof. Similar to the proof for weak bisimulation up to weak bisimilarity in [Mil89a]. $\square$

Remark. The definition of *lsb* up to $\approx$ is slightly different from that of *lsb*: with $\xrightarrow{v}$ instead of $\Longrightarrow$ in the antecedent of the second condition of Definition 4.11, the previous proposition would not hold, as the example $R = \{(v.v.a.\mathbf{0}, v.\mathbf{0})\}$ shows[6].

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

Lemma 4.13.
Let α be guarded with 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* $\alpha(\widetilde{\beta}) \xrightarrow{v} \gamma$ *then there exists* α' *with free variables in* $\tilde{t}$ *such that* $\gamma \equiv \alpha'(\widetilde{\beta})$ *and, for all* $\widetilde{\beta}'$, *we have* $\alpha(\widetilde{\beta}') \xrightarrow{v} \alpha'(\widetilde{\beta}')$. *Furthermore,* α' *is guarded.*

Proof. By transition induction.

[6] notice the similarities with the Exercise 5.14 in [Mil89a].

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 trivial. For the induction step, we have three different cases to consider, according to the possible forms of α.
   (a) Case $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$.
   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 $\widetilde{\alpha} \equiv \widetilde{\alpha}_i$.
   (b) 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 $\alpha' \equiv \alpha_1' \| \alpha_2$ and $\widetilde{\alpha} \equiv \widetilde{\alpha}_1$.
   (c) 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 4.14.
Let α be guarded with 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 main result for infinite types: the theorem of the uniqueness of the solutions of equations.

THEOREM 4.15 (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 $\{(\gamma(\widetilde{\alpha}_1), \gamma(\widetilde{\alpha}_2)) \mid \mathsf{var}(\gamma) \subseteq \tilde{t}\}$. 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 $\alpha_1 \approx R \approx \alpha_2$.

1. So let us prove the first item: since $\approx$ is a congruence,
$$\gamma(\widetilde{\alpha}_1) \approx \gamma(\widetilde{\beta}(\widetilde{\alpha}_1))\ R\ \gamma(\widetilde{\beta}(\widetilde{\alpha}_2)) \approx \gamma(\widetilde{\alpha}_2)\,.$$
Hence, 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 4.13, 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 4.14, instead of Lemma 4.13.

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 $i = 1, \ldots, |\tilde{t}|$. $\square$

Note that from this result it follows as a corollary that the recursive constructor preserves the equivalence notion (at least for guarded types): 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 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, but the recursive operator, together with the parallel composition operator, allows us to define image-infinite types like $\mu t.v.(l\|t)$, the type of an ephemeral object with a method l that lives inside a method of a persistent object.

It is well known that a process algebra where it is possible to define such terms does not have complete axiomatisations of the equivalence notions, since the process algebra has full computational power (is *turing complete*).

Therefore, in this section we use an image-finite ABT, i.e., the dynamic types without the parallel composition operator, and show that the axiom system of the previous section is complete for image-finite types.

The completeness proof of the axiomatisation of the equivalence notion for image-finite dynamic types is not trivial, and so we dedicate to it this section. For a rigorous definition of image-finiteness see, e.g., van Glabbeek [vG93a], who proves that a language with action prefixes, choice, and recursion is image-finite. The completeness proof follows the "standard" structure of that for image-finite CCS [Mil89b], 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, i.e., all types satisfy a particular kind of set of equations.

Terminology. Consider a set of variables $\{\tilde{t}\}$, and a set of types $\{\widetilde{\alpha}\}$ where $\mathsf{fv}(\widetilde{\alpha}) \subseteq \tilde{t}$. Let S: $\tilde{t} = \widetilde{\alpha}$ denote a system of (possibly mutually recursive) formal equations, where $\mathsf{var}(S)$ is its set of variables. Then,

1. We write $t_i \xrightarrow{\pi} t$, if $\pi.t$ is a summand of α_i.
2. System S is *guarded*, if $\forall_i\ t_i \not\overset{\upsilon}{\Longrightarrow} t_i$ and it is *saturated* if $t_i \overset{\upsilon}{\Longrightarrow} t'$ implies $t_i \xrightarrow{\upsilon} t'$.
3. System S is *standard*, if $\alpha_i \equiv \sum_{j\in J} \upsilon.t_{f(i,j)}$ or $\alpha_i \equiv \sum_{j\in J} l_{ij}(\tilde{t}'_{(i,j)}).t_{f(i,j)}$, where $\tilde{t}'_{(i,j)} \subseteq \tilde{t}$.
4. Let α_1 be a closed type. We write $\alpha_1 \Vdash S$ if there is $\widetilde{\alpha}{=}\alpha_1 \cdots \alpha_n$ such that $\vdash \widetilde{\alpha} = \widetilde{\beta}(\widetilde{\alpha})$.

The following lemma is crucial to prove the theorem that ensures that semantically equivalent types satisfying two different sets of equations also satisfy a common set of equations.

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.

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 2.14, obtaining α'_1.

Now in S substitute α'_1 for α_1 and repeat this process for the remaining equations. $\square$

We prove now the referred theorem.

Theorem 5.2 (Common set of equations).
Consider two systems of equations S: $\tilde{t} = \widetilde{\gamma}$ and T: $\tilde{u} = \widetilde{\delta}$, 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 $\alpha \approx \beta$, then:

1. $t_1 \xrightarrow{l(\tilde{t}')} t_i$ implies $\exists_{u_j, \tilde{u}'}(u_1 \xrightarrow{l(\tilde{u}')} u_j$ and $\alpha_i \widetilde{\alpha}' \approx \beta_j \widetilde{\beta}')$[7];
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. $t_1 \xrightarrow{v} t_i$ implies $\exists_{u_j}(u_1 \xrightarrow{v} u_j$ and $\alpha_i \approx \beta_j)$;
4. $u_1 \xrightarrow{v} u_j$ implies $\exists_{t_i}(t_1 \xrightarrow{v} t_i$ 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) $t \xrightarrow{v} t'$ implies $(t' \, R \, u$ or $\exists_{u'}(u \xrightarrow{v} u'$ and $t' \, R \, u'))$;
   (d) $u \xrightarrow{v} u'$ implies $(t \, R \, u'$ or $\exists_{t'}(t \xrightarrow{v} t'$ and $t' \, R \, u'))$.
2. $t_1 \, R \, u_1$.

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

1. $l(\tilde{v}').v_{kl}$, if $t_i \xrightarrow{l(\tilde{t}')} t_k$ and $u_j \xrightarrow{l(\tilde{u}')} u_l$ and $t_k \tilde{t}' \, R \, u_l \tilde{u}'$;[8]
2. $v.v_{kl}$, if $t_i \xrightarrow{v} t_k$ and $u_j \xrightarrow{v} u_l$ 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 $\vdash \widetilde{\varphi} = \tilde{\varepsilon}[\widetilde{\varphi}/\tilde{v}]$.

[7]Consider subfamilies of types $\widetilde{\alpha}'$ and $\widetilde{\beta}'$, corresponding respectively to $\tilde{t}'$ and $\tilde{u}'$.
[8]Consider v'_n the variable associated with $t'_n \times u'_n$, e.g., $v'_n = v_{25}$ if $t'_n = t_2$ and $u'_n = u_5$.

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 $l(\tilde{v}').v_{kl}$, and α_i has a summand $l(\alpha', \ldots).\alpha_k$; thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{v}]$ has a summand $l(\alpha', \ldots).\alpha_k$, and we conclude the equality.
2. Case $t_i \xrightarrow{v} t_k$ and $t_k \, R \, u_l$.
   Then ε_{ij} is a blocked sum with a summand $v.v_{kl}$, and α_i has a summand $v.\alpha_k$; thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{v}]$ has a summand $v.\alpha$, with $v.t_k \approx v.u_l$, 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 guarded type α with free variables $\tilde{t}$ there is S standard and guarded such that $\alpha \Vdash S$, and $\mathsf{fv}(S) \subseteq \tilde{t}$. 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, similarly to the construction in the proof of Theorem 4.1 in [Mil89b]. □

5.2 Completeness for image-finite types

From the results in the previous subsection we establish the main result of this section: the completeness of the axiom system $\mathcal{A}$ 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 cardinal of S.

The base case is immediate: consider the system S: $t = \delta$ with t guarded in δ; making $\alpha \overset{\text{def}}{=} \mu t.\delta$, 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 [Mil89b]. □

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. First, one should perform a case analysis on the form of α, but obviously, 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 [Mil89b]):

For every type α 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 three required conditions.
2. Finally, 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 the behaviour of a concurrent object with several co-existing clones (the model emerging naturally from the calculus in its full generality) via a type, a process algebra is a natural idea. Since a type (partially) specifies an object, there are three 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 or by the silent action, i.e. there are no free 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.

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 CCS, for the calculus is computationally complete. Since the full computational power comes from the substitution mechanisms (communication in this case), in calculi without communication it still makes sense to look for completeness. The study of image-infinite (or infinite-state) systems is a lively area of concurrency theory, with several important results established [BE97, CH93, Mol96]. We focus our attention in two process algebras: BPA and BPP. BPA is the class of Basic Process Algebra of Bergstra and Klop [BK85], 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 [Chr93], which is the parallel counterpart of BPA but with arbitrary derivations. Strong bisimilarity is decidable for BPA [CHS95] and for BPP [CHM93b, CHM93a]. 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 [Hir97], and a possible decision procedure for full BPP is NP-hard [Stř98].

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) $\alpha \xrightarrow{v} \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \, R \, \beta')$*;*

2. *Two types* α *and* β *are* label-semi-strong bisimilar*, and we write* $\alpha {\approx_s} \beta$*, if there is a label-semi-strong bisimulation* R *such that* $\alpha \, R \, \beta$*.*

Again, $\approx_s$ is an equivalence relation and $\alpha \approx_s \beta$ holds if and only if conditions 1(a) and 1(b) of the previous definition hold with R replaced by $\approx_s$. 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)). For deterministic finite types the two notions coincide, as we have previously shown [RRV98]. 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 Figure 1. In $l.v.l$, 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 $l.v.l = l.v.l + l.l$ holds for some equivalence notion then the 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).

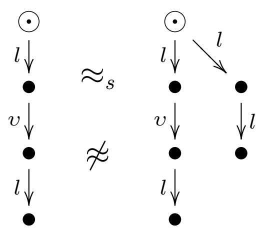

Figure 1: Lsb vs. lssb: comparing $l.v.l$ to $l.v.l + l.l$

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 system in Figure 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 [vG93b]. 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*) [MS92]. Notice that *lsc* is incomparable to *pb*, as, e.g., $l.v.m \not=_{pb} l.v.v.m$ but $l.v.m \approx_{lsc} l.v.v.m$, 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 v-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., $l.v.m$ from $l.v.v.m$ (hence, the law U1—valid for *lsb* and for weak bisimulation—is not valid for *pb*).

6.3 Comparison with related work on types.

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

Types and effects. The *type and effect discipline* is a framework for principal typing reconstruction in implicitly typed polymorphic functional languages [NN96, TJ94]. Types describe what expressions compute (sets of values) and effects describe how expressions compute (behaviour). In a concurrent functional scenario this discipline is particularly useful to prove 'subject reduction' and related results. 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, and makes behaviours look like a process algebra, which is an abstraction of the semantics of the concurrent functional language. In the context of name-passing calculi, types are assigned to names, not to processes, and are the semantics of the usage of names. Hence, we merge types and behaviours in a single entity.

Behavioural typing and subtyping. Behavioural typing and behavioural subtyping are notions of, respectively, typing and subtyping for concurrent object-oriented programming, which take into account dynamic aspects of the behaviour of objects.

Several researchers are working on this track, developing behavioural notions of typing and subtyping. We give here 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 [Nie95]. He proposes a notion of subtyping, *request substitutability*, which is based on a generalisation of the *principle of substitutability* by Wegner and Zdonick [WZ88], according to the extension relation of Brinksma *et al.* [BSS87]. It 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 [Col97, CPS97, CPDS99]. The authors define a type system which aims at the detection of "orphan messages", i.e. messages that may never be accepted by some actor, either because the requested service is not available, or because, due to dynamic changes in a actor's interface,the requested service is no longer available. Types are interface-like, with multiplicities (how often may 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 [NN97, NNS99a, NNS99b]. 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: use a process algebra as types for objects.

1. Boudol proposes a dynamic type system for the Blue Calculus, a variant of the π-calculus directly incorporating the λ-calculus [Bou97]. The types are functional and assign 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. Bowman *et al.* study the problem of defining behavioural subtyping using the process algebra LOTOS to describe the types [BBSDS97]. Since none of the standard preorders fits the requirements, the authors re-define the reduction relation to use it as an instantiation of behavioural subtyping.
3. Puntigam defines a calculus of active objects and process types [Pun97, Pun99]. A static type system ensures that all sequences of messages sent to an object are received and answered, 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, 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.
4. Kobayashi *et al.* having been studying deadlock and livelock detection in mobile calculi for a while [Kob00, KSS00]. 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. Recently, Igarashi and Kobayashi proposed a generic framework to develop type systems to ensure various properties, where types are processes themselves and are abstractions of mobile processes [IK01].

6.4 Further work

The first priority is to find out if ABT is completely axiomatisable. 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 [HM85, Mil89a]. 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.*

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

The relation below defines when does a type satisfy 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 formulas. □

We would like to extend this modal logic with recursion (in the lines of the modal μ-calculus [Koz83]), 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* states that "a type β is a subtype of a type α, if β can safely be used in place of α". "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 it is expected; and (2) a parameter type for one with more methods (contra-variant in the arguments), as it is safe to assume that that the argument has less behaviour than it really has.

Instead of defining the subtyping relation via typing rules, as for instance, in [PS96] 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.

Notice that our equivalence notion *lsb* is a symmetric *simulation*.

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* $\alpha \leq \beta$*, 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 page 9). 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 $\alpha \leq \alpha'$.

Example 6.8. *Some examples of subtyping.*

1. $(n \parallel l(m)) \leq l(m)$ *and* $(n + l(m)) \leq l(m)$;
2. $l(m) \leq l(m+n)$;
3. $l(m) \leq \upsilon.l(m)$.

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

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

1. $\alpha \leq \alpha$;
2. *if* $\alpha \leq \beta$ *and* $\beta \leq \gamma$ *then* $\alpha \leq \gamma$;

Proof. Straightforward, simply using the definition □

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 $\leq$. $\square$

This pre-order relation induces an equivalence relation (if $\alpha \leq \beta$ and $\beta \leq \alpha$ 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.

7 CONCLUDING REMARKS

The approach of behavioural types in other works is done by using an existing process algebra as a language of types for a calculus (of objects). We took the inverse approach: we first define adequate types for our setting—non-uniform objects—and then show that the types are a process algebra with convenient properties.

The Algebra of Behavioural Types, ABT is a process algebra in the style of CCS. It is 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 [RRV98], 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 [Rav00].

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 [RV00]: 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 [RV00, Rav00].

We believe that ABT can be use not only to type other concurrent calculi with extensions for objects, but also to type calculi with localities. 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.

Acknowledgements

Special thanks to Gérard Boudol, Ilaria Castellani, Silvano Dal Zilio, and Massimo Merro, for very fruitful discussions and careful reading of parts of this document. Several anonymous referees made very useful comments. This work was partially supported by the Portuguese Foundation for Science and Technology (FCT), as well as by Project POSI/CHS/39789/2001 - MIMO: Models and Infra-structures for Mobile cOmputing, and by EU proactive initiative FET-Global Computing via project Profundis: Proofs of Functionality for Mobile Distributed Systems (contract IST-2001-33100).

References

[Bar84] Henk Barendregt. *The Lambda Calculus - Its Syntax and Semantics.* North-Holland, 1981 (1st ed.) revised 84.

[BBK87] Jos C. M. Baeten, Jan A. Bergstra, and Jan W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1–2):129–176, 1987.

[BBSDS97] Howard Bowman, Charles Briscoe-Smith, John Derrick, and Ben Strulo. On behavioural subtyping in LOTOS. In Howard Bowman and John Derrick, editors, *Proceedings of FMOODS'97.* Chapman & Hall, 1997.

[BE97] Olaf Burkart and Javier Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 62:138–159, 1997.

[BK85] Jan A. Bergstra and Jan W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[Bou97] Gérard Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and Kazunori Ueda, editors, *Proceedings of ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 1997.

[Bou98] Gérard Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11:177–208, 1998. An extended abstract appeared in *Proceedings of POPL'97*, pages 228–241.

[BSS87] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementations and their tests. In *Protocol Specification, Testing and Verification VI (IFIP)*, pages 349–360. North-Holland, 1987.

[CH93] Søren Christensen and Hans Hüttel. Decidability issues for infinite-state processes—a survey. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 51:156–166, 1993.

[CHM93a] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Bisimulation equivalence is decidable for basic parallel processes. In Eike Best, editor, *Proceedings of CONCUR'93*, volume 1243 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, 1993.

[CHM93b] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Decomposability, decidability and axiomatisability for bisimulation equivalence on basic parallel processes. In *Proceedings of LICS'93*, pages 386–396. IEEE, Computer Society Press, 1993.

[Chr93] Søren Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, U. K., 1993.

[CHS95] Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. *Journal of Information and Computation*, 121(2):143–148, 1995.

[Col97] Jean-Louis Colaço. *Analyses Statiques d'un calcul d'acteurs par typage.* Thèse d'État, Institut National Polytechnique de Toulouse, France, 1997.

[CPDS99] Jean-Louis Colaço, Mark Pantel, Fabien Dagnat, and Patrick Sallé. Safety analysis for non-uniform service availability in actors. In Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri, editors, *Proceedings of FMOODS'99.* Kluwer Academic Publishers, 1999.

[CPS97] Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. A set constraint-based analyses of actors. In Howard Bowman and John Derrick, editors, *Proceedings of FMOODS'97.* Chapman & Hall, 1997.

[GH99] Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems: Proceedings of the 8th European Symposium on Programming (ESOP'99), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'99), Amsterdam, The Netherlands*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 1999.

[Hir97] Yoram Hirshfeld. Bisimulation trees and the decidability of weak bisimulations. In *Proceedings of the International Workshop on the Verification of Infinite-State Systems*, volume 5 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1997.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP'98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), Lisbon, Portugal*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *28th Annual Symposium on Principles of Programming Languages (POPL) (London, UK)*, pages 128–141. ACM, 2001.

[Kob00] 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*, volume 1872 of *Lecture Notes in Computer Science*, pages 365–389. IFIP, Springer-Verlag, 2000.

[Koz83] Dexter Kozen. Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[KSS00] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, 2000.

[Mil89a] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mil89b] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Journal of Information and Computation*, 81(2):227–247, 1989.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of the International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, U. K., 1991.

[Mol96] Faron Moller. Infinite results. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer-Verlag, 1996.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992. Available as Technical Reports ECS-LFCS-89-85 and ECS-LFCS-89-86, University of Edinburgh, U. K., 1989.

[MS92] Ugo Montanari and Vladimiro Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16(2):171–199, 1992.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[NN96] Flemming Nielson and Hanne R. Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996. An extended abstract appeared in *Proceedings of CONCUR'93*, volume LNCS 1243, 493–508.

[NN97] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In Howard Bowman and John Derrick, editors, *Proceedings of FMOODS'97*. Chapman & Hall, 1997.

[NNS99a] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioral typing. In *Proceedings of FORTE/PSTV'99*. Kluwer Academic Publishers, 1999.

[NNS99b] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Infinite types for distributed objects interfaces. In Paolo Ciancarini, Alesandro Fantechi, and Roberto Gorrieri, editors, *Proceedings of FMOODS'99*. Kluwer Academic Publishers, 1999.

[NR99] Uwe Nestmann and António Ravara. Semantics of objects as processes (SOAP). In Ana Moreira and Serge Demeyer, editors, *ECOOP '99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, pages 314–325.

Springer-Verlag, 1999. An introduction to, and summary of, the 2nd International SOAP-Workshop.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract appeared in *Proceedings of LICS'93*: 376–385.

[Pun96] Franz Puntigam. Types for active objects based on trace semantics. In *IFIP Conference on Formal Methods for Open Object-based Distributed Systems.* Chapman & Hall, 1996.

[Pun97] Franz Puntigam. Coordination requirements expressed in types for active objects. In Memet Aksit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388. Springer-Verlag, 1997.

[Pun99] Franz Puntigam. Non-regular process types. In *European Conference on Parallel Processing (Euro-Par)*, volume 1685 of *Lecture Notes in Computer Science*, pages 1334–1343. Springer-Verlag, 1999.

[Rav00] António Ravara. *Typing Non-Uniform Concurrent Objects.* PhD thesis, Instituto Superior Técnico, Technical University of Lisbon, Portugal, 2000.

[Ren] Arend Rensink. Bisimilarity of open terms. *Journal of Information and Computation.*

[RL99] António Ravara and Luís Lopes. Programming and implementation issues in non-unifom 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).* URL: http://www.tec.informatik.uni-rostock.de/IuK/congr/oosds99/program.htm.

[RRV98] António Ravara, Pedro Resende, and Vasco T. Vasconcelos. Towards an algebra of dynamic object types. In *Semantics of Objects as Processes (SOAP)*, volume NS-98-5 of *BRICS Notes Series*, pages 25–30. Danish Institute of Basic Research on Computer Science (BRICS), 1998.

[RV00] António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2000. Extended version available as DM-IST Research Report 12/2000, Portugal.

[San98] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. 143(1):34–73, 1998. Earlier version published as Rapport de Recherche RR-3000, INRIA, August 1996.

[Stř98] Jitka Stříbrná. Hardness results for weak bisimilarity of simple process algebras. In *Proceedings of MFCS'98 Workshop on Concurrency*, volume 18 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1998.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Journal of Information and Computation*, 111(2):245–296, 1994. An extended abstract appeared in *Proceedings of LICS'92*: 162–173.

[vG93a] 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), Gdansk, Poland*, volume 711 of *Lecture Notes in Computer Science*, pages 473–484. Springer-Verlag, 1993.

[vG93b] Rob J. van Glabbeek. The linear time—branching time spectrum II (the semantics of sequential systems with silent moves). In Eike Best, editor, *Proceedings of CONCUR'93*, volume 1243 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 1993.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In Eike Best, editor, *Proceedings of CONCUR'93*, volume 1243 of *Lecture Notes in Computer Science*, pages 524–538. Springer-Verlag, 1993.

[VT93] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, 1993.

[WZ88] Peter Wegner and 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*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer-Verlag, 1988.

[Yin99] Mingsheng Ying. A shorter proof to uniqueness of solutions of equations (note). *tcs*, 216:395–397, 1999.

[Yos96] 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)*, volume 1180 of *Lecture Notes in Computer

Science, pages 371–386. Springer-Verlag, 1996. Extended version as Technical Report ECS-LFCS-96-350, University of Edinburgh.