Interaction, computer science and formal topology

Peter Hancock* and Pierre Hyvernat†

November 2002

Abstract

A pattern of interaction that arises again and again in programming, is a "handshake", in which two agents exchange data. The exchange is thought of as provision of a service. The interaction is initiated or offered by one agent – the client or angel, and concluded or accepted by a second agent – the server or demon.

It turns out that formal spaces of a certain simple kind can serve as contractual descriptions of services provided over a handshaken interace; moreover that a component that provides or exports one such service, relying on or importing others can be modelled by a continuous map between these formal spaces.

We present a category in which the objects —called interaction structures in the paper— model interfaces, and the morphisms —called refinements— model components. The morphisms are relations between the underlying sets of the interaction structures, and the identity morphisms are simply the equality relations on these sets: for methodological reasons we prefer to isolate points at which a general equality relation is needed – so we approach this category via a "pre-category", in which there is only an associative composition relation. Refinements are relations that are post-fixed points for a certain operator. The proof that a relation is a refinement can serve (at least, in theory) as an executable program, whose specification is that it provides the service described by its domain, on the basis of something which provides the service described by its codomain.

This "pre-category" is then shown to be, uncannily, exactly the inductive part of basic topology in the terminology of Sambin, or topology given by a closure operator which is not required to distribute over finite unions. This provides topologists with a natural source of examples for non-distributive formal topology. It raises a number of questions of interest both for formal topology and component based programming.

What and why...

One task for computer science is to devise effective and appropriate mathematical models that capture central programming concepts. These models should provide intellectual leverage that we can bring to bear on problems arising in practice that defy unguided ingenuity. We try to analyze these problems using concepts embodied in a clear and powerful theory. A central concept of programming, prominent in recent trends toward "component-based systems", is the notion of an interface: a boundary across which components interact. The notions of interface and interaction are fundamentally linked.

The problem of how to model interaction has had a lot of attention, starting with the pioneering work in process algebra by Milner and Hoare in the 1970's developing respectively CCS [12] and CSP [10]. The focus in process algebra is on events in which independent components or agents engage or participate simultaneously. The events are so to speak shared synchronization points in the curricula vitae of those agents. The models of CCS are provided by labeled transition

*e.mail: `hancock@spamcop.net`

†current address: Chalmers University of Technology, Department of computing science, S-412 96 Göteborg, Sweden; e.mail: `hyvernat@iml.univ-mrs.fr`

systems, in which the transitions represent instances of atomic actions, private or shared. The simplest models of CSP are provided by relations between finite sequences of events and "refusals", which are sets of events in which a process can refuse to engage.

The notion of an atomic, synchronising event involving or requiring the participation of two or more agents is manifestly of great importance in the science of interaction. But turning to more practical matters, most of the work of a programmer is to write code (a module in a larger system) which implements or provides one or more procedural (call and return) interfaces, given and relying on other such interfaces, or a hardware interface. There is a binary interaction, in which one of the agents plays the role of a client, which is to initiate the interaction, perhaps passing in some data while the other has the role of concluding the interaction, perhaps returning some data or results that were originally solicited by the client. The notion of a contract or specification to be fulfilled by an implementation of a procedural, client-server interface is of pivotal practical importance. How should we model procedural interfaces in the framework of process algebra?

- At a coarse level of granularity, we might represent the entire execution of a procedure as an atomic action, in which the calling and the called program both participate, exchanging output results for input data. This loses us the simple idea that a particular party goes first.
- At a finer level, we might represent entry to a procedure as one action, and return from it as another.

It is difficult to argue that process algebra provides a model that captures the notion of a procedural interface particularly well. To view the execution of a procedure as an indivisible join action makes it difficult to model phenomena such as deadlock, in which a call takes place but no return is possible without breach of contract. To view (at a finer granularity) the execution of a procedure as a sequence of actions bracketed by a call and a return action requires extra structure on the labels to tie the elements of the pair together. This structure is an extraneous complication on top of straightforward labeled transition systems.

We propose a different model of procedural interfaces (and command-response interaction) in which there is an intrinsic "bracketing" between actions, which exploits the kind of cumulative dependency between types built-in to dependently typed functional programming languages deriving from constructive type-theory. Much of the inspiration for this model comes formal topology, particularly as studied by Sambin, Coquand, and others.

The model takes the form of an order enriched "pre-category", in which the objects (analogous to formal spaces) represent interfaces, and the morphisms (analogous to continuous maps) represent program modules or components. There is an associative form of composition, which is moreover monotone on both sides.

Modeling interaction There is a certain ambiguity in the word "interface" between what one can perhaps call syntactical aspects of an interface, and what one might call semantical or contractual aspects. To take a crude analogy, it is part of the syntax of British power plugs and sockets that the pins and holes have a rectangular shape, with two holes the same size and one a bit bigger, arranged in an isoscele triangle of a certain size and shape with the biggest pin at the apex of the triangle. It is a semantical feature that the socket supplies 60 Hz AC at 240 Volts, that the big pin is the earth and that other properties hold which are not merely a matter of convention, but of "urgent practical importance". We shall speak of the specification of the service that is provided or made available through an interface, versus the form of the interface. The notion we seek to capture is that of a procedural interface, in the sense of a specification that can be realized in various forms.

What do we do in practice to specify a service to a flesh-and-blood user? We posit a service state, for example the instantaneous state of the global banking system, which is sufficiently rich to explain the effect of successive service calls. We try to pick a state space which allows a description of the observable effect of a procedure call which is as short and simple to understand

as we can make it. The claim to be able to provide a component that implements this interface means that one can simulate the existence of an updateable variable or observable that successively attains or contains states in this state-space, jumping from one state to another in discrete steps. The simulation has to respect only the observations that can be made of the system; here an observation consists of issuing a procedure call with particular input arguments, and dispatching on the result values when the calls return.

To describe the semantics of a procedure, we have first to say which calls are legitimate (for the caller of the procedure) in a given state, and then say which returns are legitimate (for the called procedure) for a given call. Here a call can be thought of as a pair of an input state together with a vector of input arguments for some procedure chosen by the caller. A return can be thought of as a pair of an output state together with a result chosen by the procedure. What is crucial here are that there are two levels of choice, with the scope of choice available to the second agent dependent on the choice previously made by the first. The definition is given in section 1.3 on page 6.

1 Interaction structures

1.1 Type theory, notations, remarks

We work in a form of predicative type theory (see [11, 13]) without any general identity relation: intensional (like in [13]) or extensional (like in [11]). The reasons for this are partly ideological (a proof of identity has no computational meaning), partly psychiatric (traumatisation of one of the authors) and partly experimental (we wanted to see how far we could get without it).

In some parts we make use of co-inductively defined types. The treatment and justification of co-induction in a theory of types without a general equality relation is problematic, and beyond the scope of this paper[1].

Some notation:

- if S is a "set" (small type) and x is an element of S, we write $x \in S$;
- the collection Set of all sets is a (proper, or large) type;
- if $\mathcal{A}$ is a proper type, and A is an element of $\mathcal{A}$, we write $A : \mathcal{A}$;
- Π is the dependent function type. We write $(s \in S) \to T$ as a synonym for $(\Pi\, s \in S)\, T$.

We use transparently the isomorphism $A \to B \to C \equiv A \times B \to C$, which means that we write $f(a, b)$ for $\big(f(a)\big)(b)$ (f is of type $A \to B \to C$.)

Following [13] (part II) and [18], we formalize subsets of a set S as propositional functions over S.
$U \subseteq S$ is a shorthand for $U : S \to \mathsf{Set}$. The (proper) type of all subsets of a set S is written $\mathbb{P}(S)$. Saying that $s \in S$ is "in" U means that $U(s)$ is true (inhabited). We use ε as an infix operator for "backward" application $s\, \varepsilon\, U$ of propositional functions U to argument states s.

If $U \subseteq S$ and $V \subseteq S$, to say that U is "included" in V means that for all elements s of S, if $s\, \varepsilon\, U$ then $s\, \varepsilon\, V$. We write $U \subseteq V$ for this:

$$U \subseteq V \equiv (s \in S) \to U(s) \to V(s)$$

where U and V range over subsets of S.
"$\subseteq$" is thus used for two different things: to say that something is a subset of a set, and to say that a subset is included inside another subset. Any ambiguity should be resolved from the context.

[1] However, see remark on page 11

Finally, there is a new symbol $\between$ (read "meets") which is an existential dual of $\subseteq$. Its definition is:

$$U \between V \equiv (\Sigma\, s \in S)\,(s\ \varepsilon\ U \text{ and } s\ \varepsilon\ V)$$

where U and V range over subsets of S.
It provides a positive way to assert that a subset is not empty: $U \between U$.

For further details, see [18].

Type theory and programming Usually, the Curry-Howard isomorphism is interpreted as "proofs are programs", *i.e.* any (intuitionistic) proof can be viewed as an algorithm. Here, the emphasis is in the other direction, namely "programs are proofs". Some of the proofs (see sections 1.7 and 1.8 for examples) are really programs.

Even the more "mathematical" proofs of section 2 were developed as real programs in type theory. We present here the mathematical intuition behind them to make them more readable, but the actual code for the proofs exists (see reference on page 13).

1.2 Two notions of powerset

Not using any equality type amounts to a distinction between two forms of (predicative) power set, that we distinguish with the words "family" and "predicate"[2]. If S is a set, a family of S's is an indexed family of S's, and a predicate of S's is a propositional (set-valued) function with domain S.

$$\begin{array}{lcl} \mathbb{F}\,(S) & \equiv & (\Sigma\ I : \mathsf{Set})I \to S \\ \mathbb{P}\,(S) & \equiv & S \to \mathsf{Set} \end{array}$$

Note that (in a predicative type theory) both these operators cross a size-boundary: applied to sets, they return proper types. The canonical forms of a family f and a predicate P will be written as follows

$$\begin{array}{ll} \{\, f[i] \mid i \in f_{\mathrm{D}} \,\} & \text{where } f_{\mathrm{D}} : \mathsf{Set} \text{ and } f[\_] : f_{\mathrm{D}} \to S \\ \{\, s \in S \mid P(s) \,\} & \text{where } P : S \to \mathsf{Set} \end{array}$$

In the presence of an equality relation $=_S$ on an arbitrary set S, there is little difference between a family and a predicate. For any family

$$\{\, f[i] \mid i \in f_{\mathrm{D}} \,\} : \mathbb{F}\,(S)$$

there is the predicate (in essence the union of an indexed family of singleton predicates)

$$\{\, s \in S \mid (\exists i \in f_{\mathrm{D}})\, f[i] =_S s \,\} : \mathbb{P}\,(S)$$

and for any set

$$\{\, s \in S \mid P(s) \,\} : \mathbb{P}\,(S)$$

there is the family (which uses left projection as its indexing function)

$$\{\, s \mid \langle s, \_ \rangle \in (\Sigma\ s \in S)\ P(s) \,\} : \mathbb{F}\,(S)$$

One slight difference is that the passage from a predicate to a family (with small index set) fails when S is not a set but a type. A more conspicuous difference is that if we consider $\mathbb{P}\,(\_)$ and $\mathbb{F}\,(\_)$ as functors from the category of types (and functions between them) to the category of types,

[2] One may complain that a predicate is something syntactical, belonging to grammar, and the terminology is therefore inappropriate. That may be, but the terminology in which "predicate" is synonymous with boolean-valued (or propositional) function has been used by computer scientists for about 4 decades, starting with Dijkstra. For the better or not, language has changed.

then $\mathbb{P}(\_)$ is contravariant, and $\mathbb{F}(\_)$ is co-variant. The actions of the two functors on morphisms $f : A \to B$ are as follows.

$$\begin{array}{lll} \mathbb{F}(f) : \mathbb{F}(A) \to \mathbb{F}(B), & a = \{\, a[i] \mid i \in a_{\mathrm{D}} \,\} & \mapsto \{\, f(a[i]) \mid i \in a_{\mathrm{D}} \,\} \\ \mathbb{P}(f) : \mathbb{P}(B) \to \mathbb{P}(A), & P = \{\, b \in B \mid P(b) \,\} & \mapsto \{\, a \in A \mid P(f(a)) \,\} \end{array}$$

The notations f^{-1} and $(\cdot f)$ are also used for $\mathbb{P}(f)$.

In a sense, a family is something computational —a function which can be used to compute a element or datum given an index in the family's domain— whereas a predicate is something propositional, that belongs to the realm of specification.

1.2.1 Varieties of relation

If we distinguish predicates from families, then there are two subtly different notions of relation:

$$\begin{array}{lll} A & \to & \mathbb{P}(B) \\ A & \to & \mathbb{F}(B) \end{array}$$

We shall call a relation of the first kind an "honest-to-God" relation from A to B, and a relation of the second, more computational kind a "transition structure" from A to B. A transition structure $f : S \to \mathbb{F}(S)$ models a program which waits in state $s \in S$ for an input $i \in f(s)_{\mathrm{D}}$, then moves deterministically to the state $f(s)[i]$. On the other hand an honest-to-God relation $R : S \to \mathbb{P}(S)$ models a specification that such a deterministic program might meet.

Both kinds of relation are closed under indexed unions, and in general the structure of families and predicates lifts "pointwise" to relations. Both have in addition a monoid structure of composition (which we denote with a semi-colon), and a notion of reflexive-transitive closure. However, in the absence of equality, transition relations have some advantages. In particular, the unit for composition of honest-to-God relations is the problematic relation of equality.

Apart from this subtle defect, honest-to-God relations enjoy more structure. An honest-to-God relation is invertible: given $R : A \to \mathbb{P}(B)$ we have $R^{-1} : B \to \mathbb{P}(A)$ defined by $R^{-1}(a, b) = R(b, a)$. In contrast, transition structures do not in general possess inverses. Moreover, honest-to-God relations are closed under complementation, whereas transition structures are not. Finally, they are closed under intersection, whereas transition structures are not, in the absence of equality.

1.2.2 Varieties of predicate transformer

If we distinguish predicates from families then also there are several subtly different notions of predicate transformer. By a predicate transformer is meant a function of type

$$\mathbb{P}(B) \to \mathbb{P}(A)$$

transforming predicates over B to predicates over A. In fact it is more convenient to consider the inverse of a predicate transformer, that is a function of type.

$$A \to \mathbb{P}(\mathbb{P}(B))$$

Two interesting isotopes of honest-to-God predicate transformers are then

$$\begin{array}{l} A \to \mathbb{F}(\mathbb{F}(B)) \\ A \to \mathbb{F}(\mathbb{P}(B)) \end{array}$$

We focus on the first of these in this paper. The second (which is closer to the formulation of a covering relation in formal topology, see [3]) appears to be of equal interest; it may even be more realistic (true to life, or appropriate in real situations) as a model of interface specifications.

1.3 Interaction structures

Before setting up a model for interaction, one needs to describe the "universe" in which it takes place. Depending on one's perspective, that may be for example:

- a computational system, if our perspective is programming.
- an "arena[3]" if we want to use a game-theoretic or gladiatorial metaphor.
- the physical universe if we are modeling experiments in physics.
- a deductive system, if our concern is with arguments and inference.

Definition 1. An interaction structure is a quadruple $\langle S, A, D, n\rangle$ where:

- $S : \mathsf{Set}$;
- $A : S \to \mathsf{Set}$;
- $D : (s \in S) \to A(s) \to \mathsf{Set}$;
- $n \in (s \in S) \to \big(a \in A(s)\big) \to D(s,a) \to S$.

The intuition is as follows:

- S is a set of *states*. For example $s \in S$ may represent the state of the board in a game after an even number of moves, or the content of the memory in a computer between execution of instructions.
- $A(s)$ represents the possible actions (choices) the first agent (actor) can take in state s (the possible moves or the allowed assembly instructions). We'll refer to $a \in A(s)$ as an *action*.
  A is for "Angel", the first player in the game, or the user who requests a service. (Our terminology is biased toward the side of the angels.)
- $D(s,a)$ represents all the reactions permitted to the second actor in response to action a in state s. We'll refer to $d \in D(s,a)$ as a *reaction*.
  D is for "Demon", the opponent in the game, or the system which responds to a solicited request.
- $n(s,a,d) \in S$ is the state reached after the ordered pair of actions a then d.

If an interaction structure is seen a generalized Post-system (or deductive system without discharging of assumptions), S represents its formulas, or sequents; $A(s)$ is the set of possible rules having s as conclusion; and while d varies over $D(s,a)$, $n(s,a,d)$ exhausts the premises required for the rule $a \in A(s)$.

note: by using a weak form of impredicative quantification, one can define a universal interaction structure: $\mathbf{S} = (\Sigma\ w : IS)\ w.S$, $\mathbf{A}(\langle w, s\rangle) = w.A(s)$, $\mathbf{D}(\langle w, s\rangle, a) = w.D(s,a)$ and $\mathbf{n}(\langle w, s\rangle, a, d) = w.n(s,a,d)$. With predicative quantification, the universal structure is large, in that it has a proper type rather than a set of states.

Observation 1.1. *An interaction structure is equivalent to a pair* $\langle S : \mathsf{Set}, f : S \to \mathbb{F}^2 S\rangle$

[3] a game semantics term. An arena gives the rules for playing a 2 player game

note: There are other structures of a slightly more general form which seem to be useful in practice. Instead of considering only one state space, one can have 2 state spaces: one for the Angel, one for the Demon. For example, a *double sided interaction structure* on a pair of sets S and S' is a pair of functions with the following types.

$$f : S \to \mathbb{F}(S') \quad g : S' \to \mathbb{F}(S)$$

By putting $S' = (\Sigma\, s \in S)\ A(s)$, one recovers the usual "single sided" form of interaction structure. Another isotope which seems particularly useful from the perspective of specifying procedural interfaces uses both notions of powerset together, changing the types to the following.

$$f : S \to \mathbb{F}(S') \quad g : S' \to \mathbb{P}(S)$$

These structures will not be studied here.

In practice, most interaction structures present themselves with a given *initial state*. We call such an object, comprising an interaction structure and an initial state, an *interaction system*:

Definition 2. an interaction *system* is a triple $\langle S : \mathsf{Set}, f : S \to \mathbb{F}^2 S, s_0 \in S \rangle$

Transition systems $\langle S : \mathsf{Set}, f : S \to \mathbb{F}(S), s_0 \in S \rangle$ (mentioned in section 1.2.1) are a special case of interaction structures. One can recover them in different (but isomorphic) ways. Each has its own intuition:

- put $A'(s) = f(s)_{\mathrm{D}}$, $D'(s,a) = \{*\}$, and $n'(s,a,*) = f(s)[a]$ for all $s \in S$, $a \in A(s)$. In such an interaction system, the angel has all the choice. The demon's reactions are mere acknowledgment.[4]
- put $A'(s) = \{*\}$, $D'(s,*) = f(s)_{\mathrm{D}}$, and $n'(s,*,i) = f(s)[i]$ for all $s \in S$ and $i \in D(s,*)$. In such an interaction system, the demon has all the choice. The angel's actions merely solicit the demon's choices.[5]

note: In the rest of this section, we work in the context of a interaction structure $\langle S, A, D, n \rangle$.

Here is a summary of the different interpretations of an interaction structure: (the topological interpretation will be developed in section 3)

	$s \in S$	$a \in A(s)$	$d \in D(s,a)$	$n(s,a,d)$
programming	state	call to a server	return value	next state
games	state	move	countermove	next state
physics	state	observation	reading	next state
Post system	sequent	inference rule	identifier...	...for a premise
topology	basic open	basic covering	identifier...	...for a basic open

1.4 Examples of interaction structures

Simple interaction structures There are many examples of "atomic" interaction structures:

$$\begin{array}{lrcl} \text{ABORT} : & S & = & \{*\} \\ & A(*) & = & \emptyset \\ & D(*,\_) & = & \_ \\ & n(*,\_,\_) & = & \_ \end{array}$$

In such an interaction system, the first agent (Angel) can do nothing. Not only does such an interaction fail to terminate, it fails even to start!

[4] In terms of automata, those are automata which "generate" a language. The angel produces tokens.
[5] In terms of automata, those "recognize" a language. The angel reads tokens from the demon.

$$\begin{array}{rlcl} \text{STOP} : & S & = & \{*\} \\ & A(*) & = & \{*\} \\ & D(*,*) & = & \emptyset \\ & n(*,*,\_) & = & \_ \end{array}$$

This is opposite to the preceding example. The first agent has a canonical choice $(*)$, but the second agent (Demon) cannot respond, or is deadlocked. The interaction provides a way for the first agent to cease further interaction.

This interaction system seems to have great logical meaning. It is denoted by ✠ in [9] and called "daimon"[6].

$$\begin{array}{rlcl} \text{NOOP} : & S & = & \{*\} \\ & A(*) & = & \{*\} \\ & D(*,*) & = & \{*\} \\ & n(*,*,*) & = & * \end{array}$$

This is a "dummy" interaction system. After one 'tick' and one 'tock', we terminate back in the starting position. Despite its supreme uselessness, NOOP plays a central rôle in the general theory of interaction structures.

$$\begin{array}{rlcl} \text{CHAOS}_X : & S & = & X \\ & A(x) & = & \{*\} \\ & D(x,*) & = & X \\ & n(x,*,x') & = & x' \end{array}$$

Chaos is a very familiar phenomenon in computer programming. In this interaction system, the second agent can choose the new state. Although interaction terminates, the only properties it brings about are those which hold universally, and so hold already.

Other constructions

Definition 3. To each interaction structure w on S, we associate a predicate transformer $w : \mathbb{P}(S) \to \mathbb{P}(S)$, defined as follows.

$$w(U,s) \qquad \equiv \qquad \big(\Sigma\ a \in A(s)\big)\ \big(d \in D(s,a)\big) \to U\big(n(s,a,d)\big)$$

It should be clear from the context whether w stands for an interaction structure or the associated predicate transformer.

There is a canonical notion of preorder on predicate transformers (extensional inclusion), and this gives a notion of preorder on interaction structures (on the same set of states), namely pointwise extensional inclusion. Note the definition involves universal quantification over a powerset, and so is a 'large' relation.

Definition 4. Let $w_1, w_2 : S \to \mathbb{F}^2(S)$.

- We write $w_1 \sqsubseteq w_2$ if for all $U \subseteq S$, $w_1(U) \subseteq w_2(U)$;
- we write $w_1 \approx w_2$ if $w_1 \sqsubseteq w_2$ and $w_2 \sqsubseteq w_1$.

The following is trivial:

[6]Not to be confused with our demon. The daimon is a way for the angel to "cheat" by giving no way for the demon to continue.

Lemma 1.1. *$\sqsubseteq$ is a preorder; $\approx$ is an equivalence relation.*

With this notion of order, interaction structures are closed under (set indexed) sups and infs: suppose $w_i \in S \to \mathbb{F}^2(S)$ for all $i \in I$. We can define:

$$\begin{array}{lllll} \sqcup_{i\in I} w_i & : & S & = & S \\ & & A(s) & = & (\Sigma\ i \in I)\ A_i(s) \\ & & D(s, \langle i, a_i\rangle) & = & D_i(s, a_i) \\ & & n(s, \langle i, a_i\rangle, d_i) & = & n_i(s, a_i, d_i) \\ \text{and} & & & & \\ \sqcap_{i\in I} w_i & : & S & = & S \\ & & A(s) & = & (i \in I) \to A_i(s) \\ & & D(s, f) & = & (\Sigma\ i \in I)\ D_i\big(s, f(i)\big) \\ & & n(s, f, \langle i, d_i\rangle) & = & n_i\big(s, f(i), d_i\big) \end{array}$$

It is straightforwardly verified that $\sqcup$ and $\sqcap$ really mean sups and infs with respect to extensional inclusion of predicate transformers.

Interaction structures are also closed under sequential composition. We write this operation with an infix ";":
for $w_1, w_2 : S \to \mathbb{F}^2(S)$

$$\begin{array}{lllll} (w_1; w_2) & : & S & = & S \\ & & A(s_1) & = & w_1(A_2, s_1) \\ & & D(s_1, \langle a_1, k\rangle) & = & \big(\Sigma\ d_1 \in D_1(s_1, a_1)\big)\ D_2\big(n_1(s_1, a_1, d_1), k(d_1)\big) \\ & & n(s_1, \langle a_1, k\rangle, \langle d_1, d_2\rangle) & = & n_2\big(n_1(s_1, a_1, d_1), k(d_1), d_2\big) \end{array}$$

We have the following:

Lemma 1.2. *Let w_1 and w_2 be interaction structures on S:*

- *for all $U \subseteq S$, $(w_1; w_2)(U) = w_1\big(w_2(U)\big)$;*
- *composition is monotonic in its 2 arguments.*
- *composition is commutes with arbitrarily indexed unions in its first argument, and with arbitrarily indexed intersections in its second.*

Interaction structures are also closed under an operation w^* of reflexive transitive closure. We defer discussion of this operation to section 2.3.1 on page 15.

note: $\bigcup_i w_i$ and $\bigcap_i w_i$ are defined whenever $w_i : S_1 \to \mathbb{F}^2 S_2$ (and then both $\sqcup_i w_i$ and $\sqcap_i w_i$ are in $S_1 \to \mathbb{F}^2(s_2)$), and $w_1; w_2$ is defined for any $w_1 : S_1 \to \mathbb{F}^2(S')$, $w_2 : S' \to \mathbb{F}^2(S_2)$ (and then $w_1; w_2 : S_1 \to \mathbb{F}^2(S_2)$).

note: Interaction structures encode predicate transformers. They support most of the algebraic structure of Back and von Wright's refinement calculus ([2]).

1.5 Experiments

Actions and reactions are the basic building blocks of interaction. We now want to consider "strategies" to conduct *complex interactions*, from the point of view of the active or reactive agents. The first kind of strategy we shall consider is one to conduct an *experiment*.
An experiment is a way for the first agent to choose appropriate actions depending on the state and the preceding sequence of actions and reactions. Moreover, the agent may at any time choose to stop and return a result.

Definition 5. For $U \subseteq S$ and $s \in S$, $\mathcal{A}(U,s)$ is the set inductively generated by:

$$\frac{u \in U(s)}{i(u) \in \mathcal{A}(U,s)}\ (1) \text{ and } \frac{a \in A(s) \qquad k \in \Big(d \in D(s,a) \to \mathcal{A}\big(U, n(s,a,d)\big)\Big)}{j(a,k) \in \mathcal{A}(U,s)}\ (2)$$

$\mathcal{A}(U,s)$ is the set of experiments with results in U starting in state s.

Thus, $\mathcal{A}(U)$ is the least solution $X : \mathbb{P}(S)$ of:

$$X(U,s) = U(s) + \Big(\big(\Sigma\ a \in A(s)\big)\ \big(\Pi\ d \in D(s,a)\big)\ \big(X\big(U, n(s,a,d)\big)\big)\Big)$$

or more concisely, $X(U) = U \cup w\big(X(U)\big)$

Rule (1) is used to return a value and rule (2) is used to compose actions.

note: The type for experiments introduced above already appeared in the literature under the name of "tree-sets" ([14]) , in the form in which the predicate U is empty. It is a generalization of the type of well founded trees of Martin-Löf's type theory. Whereas a W-type is defined by a single generalized inductive definition, the tree sets are an S-indexed family of sets defined by simultaneous generalized inductive definition.

If interaction systems are regarded as Post systems, then an element of $\mathcal{A}(U,s)$ is a proof of the formula or sequent s from hypotheses among U.

Note that by abstraction of the state variable, we have that $\mathcal{A}(U) \in (S \to \mathsf{Set})$. $\mathcal{A}$ is thus a predicate transformer, or operator on $\mathbb{P}(S)$.

Finally, a useful notation which greatly improves readability:

Definition 6. We use the notation $s \triangleleft U$ to mean $s\ \varepsilon\ \mathcal{A}(U)$.
$U \triangleleft V$ (for $U, V \subseteq S$) means $U \subseteq \mathcal{A}(V)$ (*i.e.* $(\forall s\ \varepsilon\ U)\ s \triangleleft V$.)

1.6 Reactive strategies

The predicate transformer $\mathcal{A}$ was introduced to describe strategies for the first agent to conduct experiments comprising a sequence of observations that (should a response be forthcoming in each case) eventually yield a result. The second agent also has strategies to guide its successive reactions. For reasons of symmetry, we make the following definition.

Definition 7. For $V \subseteq S$, $\mathcal{J}(V)$ is defined to be the greatest solution $X : \mathbb{P}(S)$ of:

$$X(V,s) = V(s) \times \Big(\big(\Pi\ a \in A(s)\big)\ \big(\Sigma\ d \in D(s,a)\big)\ \big(X\big(V, n(s,a,d)\big)\big)\Big)$$

For $V \subseteq S$ and $s \in S$, $\mathcal{J}(V,s)$ is the set of reactive strategies to maintain V starting in state s, yet evade deadlock.

At this point, there is a definite departure from the game-theoretical intuition, at least in so far as the agents in game-theory are antagonistic. While the first player has well-founded ("finite") strategies to achieve certain outcomes U, the second player has non well-founded ("infinite") strategies, to remain in play, in a region of the statespace satisfying V.

Whereas an experiment always *ends* in a state satisfying U (since it ends by returning an element of $U(s)$), a reactive strategy always *stays* in states satisfying V. As it were, at each step not only must it be prepared with a response to any request, but it can also be asked to return a proof that V holds in the current state.

In terms of specifications, $\mathcal{A}(U,s)$, which we can also write $s \triangleleft U$ means that the angel has a strategy to bring about U starting from state s, provided that the demon evades deadlock – situations in which there is no contractually legitimate reaction to an action of the angel. This is an example of a (conditional) liveness property, requiring that certain things must happen. On the other hand, $\mathcal{J}(V,s)$ means that the demon has a strategy to maintain V starting in state s, while evading deadlock. This is an example of a safety property, requiring that only certain things may happen.

If interactive systems are regarded as Post systems, then a reactive strategy K in $\mathcal{J}(U,s)$ is a procedure to *test* or probe into a purported proof of s. The angel (who has the proof) says what the last inference is, and the demon responds by identifying a premise of this inference. In this way a branch is traced out in the proof in which the formulas that appear all satisfy U, and the inference steps that appear all have more than one premise.

note: The 'definition' of $\mathcal{J}$ above is difficult to formulate or justify in the framework of Martin-Löf's type theory. To our knowledge, there is as yet no fully satisfactory way to deal with co-induction inside Martin-Löf type theory, without extensional equality. Our feeling is that the principles needed are safe, and have a solid computational meaning; yet there remain serious foundational issues to clarify and resolve.

1.7 Properties of $\mathcal{A}$ and $\mathcal{J}$

$\mathcal{A}$ and J enjoy dual properties.

Proposition 1. *$\mathcal{A}$ is a closure operator on $\mathbb{P}(S)$; similarly, $\mathcal{J}$ is an interior operator on $\mathbb{P}(S)$.*

Proof. The first part ($\mathcal{A}$ is a closure operator) has been type-checked by the system Agda (see references on page 13). We present here the "program" version of the proof as it is more relevant than the mathematical version, which is trivial.

$\longrightarrow \mathcal{A}$ is a closure operator:

(1): $$\frac{f \in (s \in S) \to U(s) \to V(s)}{\mathsf{map}\,(f) \in (s \in S) \to \mathcal{A}(U,s) \to \mathcal{A}(V,s)}\ \text{monotonicity} \qquad \left(\frac{U \subseteq V}{\mathcal{A}(U) \subseteq \mathcal{A}(V)} \right)$$

$$\begin{aligned} \mathsf{map}\,(f)\big(s, i(u)\big) &= i\big(f(s,u)\big) \\ \mathsf{map}\,(f)\big(s, j\langle a,k\rangle\big) &= j\big\langle\ a\ ,\ \big(\lambda d \in D(s,a)\big)\ .\ \mathsf{map}\,(f)\big(n(s,a,d),k(d)\big)\ \big\rangle \end{aligned}$$

(2): $$\frac{u \in U(s)}{\eta(u) \in \mathcal{A}(U,s)} \qquad \Big(U \subseteq \mathcal{A}(U)\Big)$$

$$\eta(u) = i(u)$$

(3): $$\frac{f \in (s \in S) \to U(s) \to \mathcal{A}(V,s)}{f^\sharp \in (s \in S) \to \mathcal{A}(U,s) \to \mathcal{A}(V,s)} \qquad \left(\frac{U \subseteq \mathcal{A}(V)}{\mathcal{A}(U) \subseteq \mathcal{A}(V)} \right)$$

$$\begin{aligned} f^\sharp\big(s, i(u)\big) &= f(s,u) \\ f^\sharp\big(s, j\langle a,k\rangle\big) &= j\big\langle\ a\ ,\ \big(\lambda d \in D(s,a)\big)\ .\ f^\sharp\big(n(s,a,d),k(d)\big)\big\rangle \end{aligned}$$

This last rule can be rewritten as $\dfrac{s \triangleleft U \qquad U \triangleleft V}{s \triangleleft V}$ which we call "transitivity."

$\longrightarrow\!\triangleright\ \mathcal{J}$ is an interior operator:

(1): $\dfrac{f \in (s \in S) \to U(s) \to V(s)}{\mathsf{pam}\,(f) \in (s \in S) \to \mathcal{J}(U,s) \to \mathcal{J}(V,s)}$ monotonicity $\left(\dfrac{U \subseteq V}{\mathcal{J}(U) \subseteq \mathcal{J}(V)} \right)$

$\mathsf{pam}\,(f)\big(s, \langle u, k\rangle\big) \quad = \quad \Big\langle\ f(u) \quad , \quad \big(\lambda a \in A(s)\big)\ .\ \big\langle\ \pi_1\big(k(a)\big)\ ,\ \mathsf{pam}\,(f)\big(\pi_2\big(k(a)\big)\big)\ \big\rangle\ \Big\rangle$

(2): $\dfrac{K \in \mathcal{J}(U,s)}{\mu(K) \in U(s)}$ $\Big(\mathcal{J}(U) \subseteq U\Big)$

$\mu\big(\langle u, k\rangle\big) \quad = \quad u$

(3): $\dfrac{f \in (s \in S) \to \mathcal{J}(U,s) \to V(s)}{f^\flat \in (s \in S) \to \mathcal{J}(U,s) \to \mathcal{J}(V,s)}$ $\left(\dfrac{\mathcal{J}(U) \subseteq V}{\mathcal{J}(U) \subseteq \mathcal{J}(V)} \right)$

$f^\flat\big(s, \langle u, k\rangle\big) = \Big\langle\ f\big(s, \langle u, k\rangle\big) \quad , \quad \big(\lambda a \in A(s)\big)\ .\ f^\flat\Big(n\big(s, a, \pi_1 k(a)\big), \pi_2 k(a)\Big)\ \Big\rangle$

This is what one would like to write as it has a straightforward computational meaning (if one uses head reduction or lazy evaluation). The treatment of co-induction should allow us to write something similar. □

note: there is a stronger statement which is more algebraic:

Proposition 2. *On the category $S^d \to Set$, $\mathcal{A}$ is a monad and $\mathcal{J}$ a comonad*[7]*.*

This is slightly stronger because is also requires some equalities to be satisfied. In particular, the above "$\_^\sharp$" and "$\_^\flat$" should respect the structure of their arguments. For this to be provable however, we need a notion of equality.

1.8 Interaction between experiments and reactive strategies

Interaction occurs when an experiment (first agent) and a reactive strategy (second agent) are put together. The experiment gives a first action (or returns a result), the reactive strategy gives a reaction, and the process goes on... We call this process *execution*. It always gives an element of $U(\text{final state})$.

Here is the way to compute $\mathsf{exec}\,\big(E \in \mathcal{A}(U,s), K \in \mathcal{J}(V,s)\big)$:

$$
\begin{array}{lll}
\mathsf{exec}\,\big(i(u), \langle v, f\rangle\big) = u & & \\
\mathsf{exec}\,\big(j(a,k), \langle v, f\rangle\big) = & \textbf{let} & \text{reaction} = \pi_1(f(a)) \\
 & & \text{exp}' = k(\text{reaction}) \\
 & & \text{reac}' = \pi_2(f(\text{reaction})) \\
 & \textbf{in} & \mathsf{exec}\,(\text{exp}', \text{reac}')
\end{array}
$$

This is a bare-bones presentation of the execute function. In an actual implementation of type theory, we have to add the (implicit) arguments U, V and s.

Notice also that when we reach the result, we also have at our disposal a reactive strategy in the current state. Here is the less skeletal and more fleshy definition of exec:

$$
\begin{array}{lll}
\mathsf{exec}\,\big(U \subseteq S, V \subseteq S, s \in S, E \in \mathcal{A}(U,s), K \in \mathcal{J}(V,s)\big) \quad \in \quad \big(\Sigma\, s' \in S\big)\ \big(U(s') \times \mathcal{J}(V,s')\big) & & \\
\mathsf{exec}\,\big(U, V, s, i(u), K\big) = \langle s, \langle u, K\rangle\rangle & & \\
\mathsf{exec}\,\big(U, V, s, j(a,k), \langle v, f\rangle\big) = & \textbf{let} & d = \pi_1(f(a)) \\
 & & E' = k(d) \\
 & & K' = \pi_2(f(a)) \\
 & \textbf{in} & \mathsf{exec}\,\big(U, V, n(s,a,d), E', K'\big)
\end{array}
$$

[7] S^d is the discrete category on S.

note: It is not used here that a reactive strategy provides evidence that V continues to hold.

2 An appropriate "pre-category"

2.1 Starting remarks

All the categories described in the following will have interaction structures as objects, and some special relations between sets of states as morphisms. We could do exactly the same using interaction systems (*i.e.* initialized interaction structures).

As already mentioned, we use no identity type whatsoever. This makes it impossible to define an identity morphism on object. Our "categories" are thus somewhat ill-defined.

Definition 8. A pre-category $\mathcal{C}$ is given by a collection (usually a large type) of objects together with, for any pair A, B of objects, a collection $\mathcal{C}[A, B]$ (again, usually large) of morphisms.

There is moreover an equivalence relation on each $\mathcal{C}[A, B]$, and an operation of composition ";" (if $f : \mathcal{C}[A, B]$ and $g : \mathcal{C}[B, C]$ then $f; g : \mathcal{C}[A, C]$) which is associative.

Any category is trivially an example of pre-category.

Notation: interaction structures are written w, w_1... and their component S, A, D, n or S_1, A_1, D_1, n_1...

To get more legible proofs, we'll omit the domain of variables as often as possible. For example, a_1^* will implicitly be an element of $A_1^*(s_1)$.

Finally, all the proofs in this section (except lemma 2.3) were computer checked using the system Agda[8] / Alfa[9]. The relevant files with a typeset description of them can be found at `http://iml.univ-mrs.fr/~hyvernat/academics.html`

2.2 Interaction structures with simulations

2.2.1 Simulations

The morphisms of the first pre-category are "simulations" between states. A simulation is a relation between states satisfying some extra property:

Definition 9. Let w_1 and w_2 be 2 interaction structures. A simulation R from w_1 to w_2 (written $R : w_1 \to w_2$) is a (honest to God) relation between S_1 and S_2 such that the following holds:

$$\begin{array}{c} s_1 \; R \; s_2 \\ \Rightarrow \\ \big(\forall a_1 \in A_1(s_1)\big) \quad \big(\exists a_2 \in A_2(s_2)\big) \quad \big(\forall d_2 \in D_2(s_2, a_2)\big) \quad \big(\exists d_1 \in D_1(s_1, a_1)\big) \\ \text{s.t. } n_1(s_1, a_1, d_1) \; R \; n_2(s_2, a_2, d_2) \end{array}$$

If one replaces interaction structures by interaction systems, we add the condition that the initial states are related through R.

note: by using the predicate transformer associated to w_2, and defining $s \lhd_2 U$ to mean $w_2(U, s)$, one can rewrite the condition as:

$$\big(\forall a_1 \in A_1(s_1)\big) \quad R(s_1) \lhd_2 \Big(\bigcup_{d_1 \in D_1(s_1, a_1)} R\big(n_1(s_1, a_1, d_1)\big)\Big)$$

[8] `http://www.cs.chalmers.se/~catarina/agda/`
[9] `http://www.cs.chalmers.se/~hallgren/Alfa/`

In a constructive framework, a proof of $s_1\ R\ s_2$ gives a function to translate actions $a : A_1(s_1)$ into actions $a' : A_2(s_2)$, and a function to translate reactions $d : D_2(s_2, a')$ to the latter into reactions $d' : D_1(s_1, a)$ to the former, in such a way that we can prove that $n_1(s_1, a, d')\ R\ n_2(s_2, a', d)$ for all $a : A_1(s_1)$ and $d : D_2(s_2, a')$.

This is easily seen to be a direct generalization of the notion of simulation on transition structures.

In type theory, a simulation from w_1 to w_2 is a pair $\langle \mathrm{REL}, \mathrm{SIM} \rangle$ where

- $\mathrm{REL} : S_1 \to S_2 \to \mathsf{Set}$ (usual coding of relations);
- $$\begin{array}{lll} \mathrm{SIM} \in & (s_1 \in S_1) \to (s_2 \in S_2) \to \mathrm{REL}(s_1, s_2) \to & (\text{whenever } s_1\ R\ s_2) \\ & \big(a_1 \in A_1(s_1)\big) \to & (\forall a_1) \\ & \big(\Sigma\ a_2 \in A_2(s_2)\big) & (\exists a_2) \\ & \big(d_2 \in D_2(s_2, a_2)\big) \to & (\forall d_2) \\ & \big(\Sigma\ d_1 \in D_1(s_1, a_1)\big) & (\exists d_1) \\ & \mathrm{REL}\big(n_1(s_1, a_1, d_1), n_2(s_2, a_2, d_2)\big) & (\text{s.t. } n_1(s_1, a_1, d_1)\ R\ n_2(s_2, a_2, d_2)) \end{array}$$

2.2.2 Composition

Composition of simulations R_1 and R_2 is easy. Just compose the relations as usual. The fact that this is still a simulation is obtained by composing the proofs of simulations of R_1 and R_2.

Lemma 2.1. *if $R_1 : w_1 \to w_2$ and $R_2 : w_2 \to w_3$, then $R_1; R_2 : w_1 \to w_3$.*

Proof. `computer checked...`
Easy. □

2.2.3 Equality on simulations

By using extensional equality on relations, one obtains trivially that composition is associative. Extensional equality on subsets is interesting because it doesn't require any other notion of equality.

In type theory, it is defined as

$$\begin{array}{lll} (R = T) \equiv & (s_1 \in S_1) \to (s_2 \in S_2) \to R(s_1, s_2) \to T(s_1, s_2) & (R \subseteq T) \\ & \times & (\text{and}) \\ & (s_1 \in S_1) \to (s_2 \in S_2) \to T(s_1, s_2) \to R(s_1, s_2) & (T \subseteq R) \end{array}$$

i.e. whenever $s_1\ R\ s_2$ then $s_1\ T\ s_2$, and vice and versa.

Definition 10. The pre-category **IntSim** has interaction structures as object, and simulations as morphisms.

If one adds the identity type, then **IntSim** is in fact a true category.

2.3 Interaction structures with refinements

IntSim is a very simple "interaction relevant" category, but it is not entirely satisfying, as each simulated action must be matched by exactly one simulating action. In common situations, one wants to allow simulation by 'composite' actions, or processes.

We now introduce "processes", a variant of experiments and construct a new pre-category: morphisms will translate processes into processes rather than actions into actions.

2.3.1 Reflexive transitive closure, processes

Intuitively, a process is a (dependent) list of actions, or a strategy for the angel to initiate actions, depending on the reaction to previous actions, which eventually terminates.

Definition 11. Let w be an interaction structure. The *reflexive transitive closure* of w (written w^*) is defined by:

- $S^* = S$
  (same set of states)
- $A^*(s) = \mathtt{NOP} \quad | \quad \mathtt{Cons}\Big(a \in A(s)\Big)\Big(t \in \big(d \in D(s,a) \to A^*(n(s,a,d))\big)\Big)$
  (tree of actions, with branching determined by reactions)
- $D^*\big(s, \mathtt{NOP}\big) = \{*\}$
  $D^*\big(s, \mathtt{Cons}(a,t)\big) = \big(\Sigma\ d \in D(s,a)\big)\Big(d^* \in D^*\big(n(s,a,d), t(d)\big)\Big)$
  (branch in such a tree)
- $n^*(s, \mathtt{NOP}, *) = s$
  $n^*\big(s, \mathtt{Cons}(a,t), \langle d, d^*\rangle\big) = n^*\big(n(s,a,d), t(d), d^*\big)$
  (state of the leaf reached by the branch)

$a^* \in A^*(s)$ is called a process in state s.

A^*, D^* and n^* are all inductively defined[10], and are thus directly definable in Martin-Löf's type theory.

Definition 12. a process $a^* : A(s)$ that starts in state s is said to terminate in $U \subseteq S$ if $\big(\forall d^* \in D^*(s,a^*)\big) \quad n^*(s,a^*,d^*)\ \varepsilon\ U$
In type theory, the set of processes that start in state s and terminate in U can be defined by $\mathcal{B}(U,s) \equiv \big(\Sigma\ a^* \in A^*(s)\big)\ \Big(\big(d^* \in D^*(s,a^*)\big) \to U\big(n^*(s,a^*,d^*)\big)\Big)$.

Note that $\mathcal{B}$ is actually the same object as the predicate transformer associated with the interactive structure w^*.

Lemma 2.2. *As operators on $\mathbb{P}(S)$, $\mathcal{A}$ and $\mathcal{B}$ are extensionally equal.*

In type theory, it means that there are 2 functions: $f \in (s \in S) \to \mathcal{A}(U,s) \to \mathcal{B}(U,s)$ and $g \in (s \in S) \to \mathcal{B}(V,s) \to \mathcal{A}(U,s)$.

Proof. `computer checked...`
Easy. □

f and g are canonical in the sense that, with equality, one can show that $f = g^{-1}$.

Corollary. *$\mathcal{B}$ is a closure operator on $\mathbb{P}(S)$*

note: we will now use $s \triangleleft U$ for either $s\ \varepsilon\ \mathcal{A}(U)$ or $s\ \varepsilon\ \mathcal{B}(U)$.

[10] A^* is the least solution to $A^* = N_1 + w(A^*)$.

2.3.2 Refinements

Definition 13. If w_1 and w_2 are interaction structures, then R is a refinement from w_1 to w_2 if R is a simulation from w_1 to w_2^*. $(R : w_1 \to w_2^*)$

Refinements are closer to what we have in mind: w_2 refines w_1 if actions of w_1 can be replaced by processes of w_1.

Our aim is thus to define composition for refinements. This will show that as well as having a pre-category **IntSim** in which the morphisms are simulations, we have a pre-category **IntRef** in which the morphisms are refinements.

There are 2 other interesting characterizations of refinement relations:

Proposition 3. *let w_1 and w_2 be 2 interaction structures, and R a relation on $S_1 \times S_2$. The following are equivalent:*

1. *R is a refinement relation;*
2. *for all $s_1 \in S_1$ and $a_1 \in A_1(s)$, $R(s_1) \lhd \bigcup_{d_1 \in D_1(s_1,a_1)} R\big(n_1(s_1, a_1, d_1)\big)$;*
3. *for all $s_1 \in S_1$ and $a_1^* \in A_1^*(s)$, $R(s_1) \lhd \bigcup_{d_1^* \in D_1^*(s_1,a_1^*)} R\big(n_1^*(s_1, a_1^*, d_1^*)\big)$.*

Proof. `computer checked...`
Here is a "friendly" version:

$\multimap$ *1 $\Rightarrow$ 2*
suppose that R is a refinement; let $s_1 \in S_1$, $a_1 \in A_1(s_1)$ and $s_2 \in S_2$ such that $s_1\ R\ s_2$.
We need to show that $s_2 \lhd \bigcup_{d_1 \in D_1(s_1,a_1)} R\big(n_1(s_1, a_1, d_1)\big)$.

Since $s_1\ R\ s_2$ and R is a refinement, we know that there is $a_2^* \in A_2^*(s_2)$ s.t. for all $d_2^* \in D_2^*(s_2, a_2^*)$ there is $d_1 \in D_1(s_1, a_1)$ s.t. $n_1(s_1.a_1, d_1)\ R\ n_2^*(s_2, a_2^*, d_2^*)$.

In other words, $\exists a_2^*\ \forall d_2^* \quad n_2^*(s_2, a_2^*, d_2^*)\ \varepsilon \bigcup_{d_1} R\big(n_1(s_1, a_1, d_1)\big)$, *i.e.* $s_2\ \varepsilon\ \mathcal{B}\Big(\bigcup_{d_1} R\big(n_1(s_1, a_1, d_1)\big)\Big)$

$\multimap$ *2 $\Rightarrow$ 1*
Similar...

$\multimap$ *2 $\Rightarrow$ 3*
Suppose:

- $\forall s_1, a_1\ R(s_1) \lhd \bigcup_{d_1} R\big(n_1(s_1, a_1, d_1)\big)$;
- $s_1 \in S_1$, $s_2 \in S_2$, $s_1\ R\ s_2$ (*i.e.* $s_2 \in R(s_1)$);
- $a_1^* \in A_1^*(s_1)$.

We will show that $s_2 \lhd \bigcup_{d_1^*} \Big(R\big(n_1^*(s_1, a_1^*, d_1^*)\big)\Big)$ by induction on a_1^*.

base case: $a_1^* = \texttt{NOP}$. We need to show $s_2 \lhd R(s_1)$.[11]
This holds by hypothesis: $s_2\ \varepsilon\ R(s_1) \subseteq \mathcal{B}\big(R(s_1)\big)$.

induction: $a_1^* = \texttt{Cons}(a_1, tl_1)$
By hypothesis, we have that $s_2 \lhd \bigcup_{d_1 \in D_1(s_1,a_1)} R\big(n_1(s_1, a_1, d_1)\big)$ (see above);

- by induction hypothesis, we have that for all $d_1 \in D_1(s_1, a_1)$,

$$R\big(n_1(s_1, a_1, d_1)\big) \lhd \bigcup_{d_1^* \in D_1^*\left(n_1(s_1,a_1,d_1), tl_1(d_1)\right)} R\Big(n_1^*\big(n_1(s_1, a_1, d_1), tl_1(d_1), d_1^*\big)\Big);$$

[11] Recall that in this case $D_1^*(s_1, \texttt{NOP}) = \{*\}$ and that $n_1^*(s_1, \texttt{NOP}, *) = s_1$.

- the right-hand-side subset is equal to $\bigcup_{d_1^* \in D_1^*(s_1, a_1^*)} R(n_1^*(s_1, a_1^*, d_1^*))$;
- so,

$$R\big(n_1(s_1, a_1, d_1)\big) \lhd \bigcup_{d_1^* \in D_1^*(s_1, a_1)} R\big(n_1^*(s_1, a_1^*, d_1^*)\big);$$

This shows that

$$\bigcup_{d_1 \in D_1(s_1, a_1)} R\big(n_1(s_1, a_1, d_1)\big) \lhd \bigcup_{d_1^* \in D_1^*(s_1, a_1)} R\big(n_1^*(s_1, a_1^*, d_1^*)\big)$$

We get the result by transitivity.

$\longrightarrow$ *3* $\Rightarrow$ *2* We leave the (easy) proof for the reader. □

We also have a "sub-commutativity" of refinement:

Proposition 4. *If R is a refinement relation from w_1 to w_2, then: $R\big(\mathcal{B}_1(U)\big) \subseteq \mathcal{B}_2\big(R(U)\big)$.*

One can rewrite the above to: if $s \lhd_1 U$ then $R(s) \lhd_2 R(U)$.

Proof. `computer checked...`

Suppose that R is a refinement relation, that $s_1\ R\ s_2$, and that $\langle a_1^*, p_1\rangle \in \mathcal{B}_1(U, s_1)$. We are going to construct an element of $\mathcal{B}_2\big(R(U), s_2\big)$.

Since R is a refinement relation, a_1^* has a translation a_2^*. We show that $n_2^*(s_2, a_2^*, d_2^*)\ \varepsilon\ R(U)$ for all d_2^*. Given d_2^*, let d_1^* be its translation according to R's properties as a simulation. We have $n_1^*(s_1, a_1^*, d_1^*)\ R\ n_2^*(s_2, a_2^*, d_2^*)$ (since R is a refinement) and $n_1^*(s_1, a_1^*, d_1^*)\ \varepsilon\ U$ (using p_1), so that $n_2^*(s_2, a_2^*, d_2^*)\ \varepsilon\ R(U)$. □

2.3.3 Composition of refinements

Morally, the idea is to show that $\_^*$ is a monad on the category **IntSim**, so that one can construct the Kleisli category.

For $\_^*$ to be a monad means that we have a morphism : $\eta : w \to w^*$ and an operator

$$\frac{R : w_1 \to w_2^*}{R^\sharp : w_1^* \to w_2^*}$$ such that the following hold: $\eta; f^\sharp = f$, $\eta^\sharp = \mathbf{Id}$ and $(f; g^\sharp)^\sharp = f^\sharp; g^\sharp$.

Since **IntSim** has no identity, we can't use this definition. Instead, we use the following:

Definition 14. Let $\mathcal{C}$ be a pre-category. A "pre-monad" on $\mathcal{C}$ is a function $\mathcal{M}$ from the objects of $\mathcal{C}$ to the objects of $\mathcal{C}$ with 2 operators $\_^\sharp$ and $\_\sharp$:

$$\frac{f : \mathcal{C}[A, \mathcal{M}\,B]}{f^\sharp : \mathcal{C}[\mathcal{M}\,A, \mathcal{M}\,B]} \quad \text{and} \quad \frac{f : \mathcal{C}[\mathcal{M}\,A, \mathcal{M}\,B]}{f_\sharp : \mathcal{C}[A, \mathcal{M}\,B]}$$

which satisfy: $(f^\sharp)_\sharp = f$, $f_\sharp; g^\sharp = (f; g^\sharp)_\sharp$ and $f^\sharp; g^\sharp = (f; g^\sharp)^\sharp$

note: This is exactly equivalent to dropping the condition "$\eta^\sharp = \mathbf{Id}$" (which cannot be expressed in a pre-category) from the usual definition of a monad. (Put $\eta = \mathbf{Id}_\sharp$ or $f_\sharp = f; \eta$.) It thus looks like the appropriate notion of monad in pre-categories.

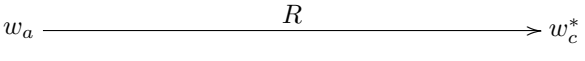
For any pre-monad, we can construct a "Kleisli pre-category": composition of f and g is defined as $f; g^\sharp$. Associativity follows from the law $(f; g^\sharp)^\sharp = f^\sharp; g^\sharp$ and associativity on the ground category. (The other two conditions are not necessary.)

Proposition 5. *"$\_^*$" is a pre-monad on* **IntSim**.

Proof. `computer checked...`
(only for the definition of $\_^\sharp$ and $\_^\sharp$. That the equalities hold is trivial because the underlying relations don't change.)
Easily derived from proposition 3. □

Definition 15. Let's give the name **IntRef** to the "Kleisli pre-category" on **IntSim** with respect to the pre-monad $\_^*$. This is the category of interaction structures and refinements.

The programmer's predicament The idea of refinement is quite natural from a programmer's point of view. The programmer's task, most of the time, is to implement an "abstract" or "high-level" interface (such as a filesystem, or an abstract machine) by using a "concrete" or "low-level" interface (such as that provided by a hard disk, or a specific processor). We can represent the locus operandi of the programmer, by placing him in a certain 'box', or hom-set as follows:



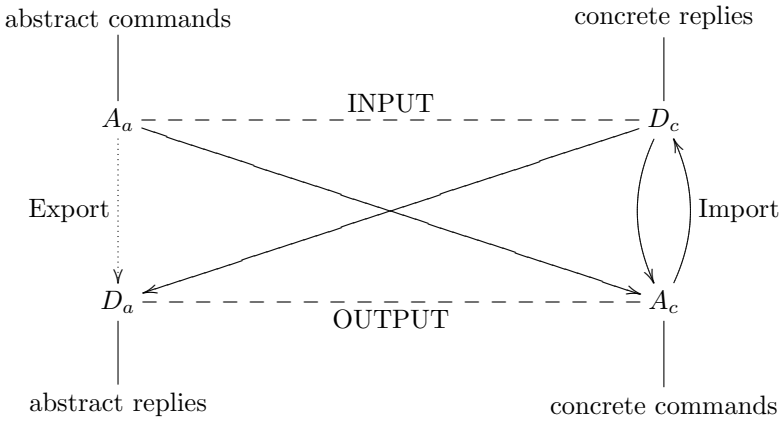


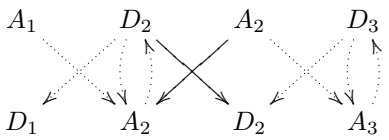
Figure: the programmer's predicament

Things start when an abstract, high-level command arrives, top-left. Such a command has to be mapped to something which in most cases is a composite program, or an articulated strategy for issuing low-level commands and dispatching[12] on the results. Running this program on the low level interfaces means that a sequence of commands is issued across the low-level interface (bottom-right), in zip-wise alternation with a sequence of results returned across it (top-right). The results are as it were fed-back to control future use of the low-level interface. In the picture, this is represented by the cycle between the A and D 'poles' of the imported interface.

Given such a mapping, what remains is to provide a way to translate complete sequences of low-level responses (top-right) to abstract replies, and return them (bottom-left). In general, this is translation from low-level execution traces to final high-level results is built up as the programs run.

In this way, we export the abstract high-level interface by importing (making use of) the concrete one – zero, one or more times, but only finitely often.

To compose several of those boxes, one just needs to plug them together using a "cross-link":



[12]Dispatching means transferring control to the next point in a program, as it were 'steered' by the result of a previously issued command.

Alas, in reality, things are not always so straightforward.

2.4 Saturation and preorder

The (pre) category **IntRef** is closer to a real "interaction aware" category[13], but the equality relation between morphisms (extensional equality between refinement relations) is too strong.

The "purpose" of a refinement $R : w_1 \to w_2^*$ is to give, whenever $s_1\ R\ s_2$, a way to translate actions in $A_1(s_1)$ into processes in $A_2^*(s_2)$ (or equivalently, processes into processes. See proposition 3).

There may however be refinements that are merely 'implicit' in a relation R which satisfies the refinement condition. The intuition is perhaps easier to grasp in the simpler case of transition structures:

2.4.1 The case of transition structures

Definition 16. A transition structure is an interaction structure in which $D(s,a) = \{*\}$ for all $s \in S$, $a \in A(s)$. Equivalently, a transition structure is a pair $\langle S, f : S \to \mathbb{F}(S)\rangle$.
We write $s \xrightarrow{a} s'$ for $n(s,a) = s'$.

Note that $\mathcal{B}(U)$ is the set of states from which some chain of transitions leads into U.

On such a structure, a refinement is a relation between states such that whenever $s_1\ R\ s_2$ and $s_1 \xrightarrow{a_1} s_1'$ there is a chain of transitions $s_2 \xrightarrow{a_{2,1}} \dots \xrightarrow{a_{2,n}} s_2'$ such that $s_1'\ R\ s_2'$

Suppose now that $s_2 \xrightarrow{a_{2,1}} \dots \xrightarrow{a_{2,n}} s_2'$ and that $s_1\ R\ s_2'$. Even though s_1 is not (necessarily) related to s_2, s_2 refines s_1:

$$\text{if } s_1 \xrightarrow{a_1} s_1' \text{ then } s_2 \underbrace{\xrightarrow{a_{2,1}} \dots \xrightarrow{a_{2,n}}}_{\text{hypothesis}} s_2' \underbrace{\xrightarrow{a'_{2,1}} \dots \xrightarrow{a'_{2,m}}}_{s_1 R s_2'} s_2'' \text{ with } s_1'\ R\ s_2''$$

What we have just shown is that if there is a path (process) in $\mathcal{B}\big(R(s_1), s_2\big)$ then s_2 refines s_1.

The same holds for general interaction structures:

2.4.2 The case of interaction structures: saturation

Following the intuition of the previous paragraph, we define:

Definition 17. let R be a refinement relation from w_1 to w_2. The saturation of R (written $\overline{R}$) is defined by:

$$s_1\ \overline{R}\ s_2 \quad \equiv \quad s_2\ \varepsilon\ \mathcal{B}\big(R(s_1)\big)$$

We have:

Proposition 6. *If R is a refinement from w_1 to w_2, then $\overline{R}$ is also a refinement from w_1 to w_2.*

In some sense, $\overline{R}$ is the weakest relation having the same "refinement power" as R.

Proof. `computer checked...`
We will show that $\overline{R}\, s_1 \lhd \bigcup_{d_1} \overline{R}\big(n_1(s_1,a_1,d_1)\big)$.
Since we have $R(s_1) \lhd \bigcup_{d_1} R\big(n_1(s_1,a_1,d_1)\big)$ and because $\mathcal{B}$ is a closure operator, we have

$$\mathcal{B}\big(R(s_1)\big) \equiv \overline{R}(s_1) \quad \lhd \quad \bigcup_{d_1} R\big(n_1(s_1,a_1,d_1)\big).$$

[13] proposition 3 shows that refinements are just translation of processes.

For any d_1, $R\big(n_1(s_1,a_1,d_1)\big) \lhd R\big(n_1(s_1,a_1,d_1)\big)$ which implies (still because $\mathcal{B}$ is a closure operator) $R\big(n_1(s_1,a_1,d_1)\big) \quad \lhd \quad \mathcal{B}\Big(R\big(n_1(s_1,a_1,d_1)\big)\Big) \equiv \overline{R}\,(n_1(s_1,a_1,d_1))$.

This in turn implies that

$$\bigcup_{d_1} R\big(n_1(s_1,a_1,d_1)\big) \lhd \bigcup_{d_1} \overline{R}\,\big(n_1(s_1,a_1,d_1)\big).$$

We get the result by transitivity. □

note: since transition structures are special interaction structures, the same holds for those. (See preceding paragraph.)

However, in this case, the result degenerates: suppose $s_1 \ \overline{R}\ s_2$. We can "refine" s_1 by s_2 by translating any $s_1 \xrightarrow{a_1} s_1'$ to $s_2 \xrightarrow{\texttt{NOP}} s_2$.

We have $s_1' \ \overline{R}\ s_2$ because $s_2 \underbrace{\longrightarrow \cdots}_{s_1 \overline{R}\, s_2}\ s_2' \ \underbrace{\cdots \longrightarrow}_{s_1 R s_2'} s_2''$ with $s_1' \ R\ s_2''$.

The above trick works only works in transition structures because we don't need to translate the reactions.

2.4.3 A poset enriched pre-category

Recall the definition of a poset enriched category ([1]):

Definition 18. A poset enriched category $(\mathcal{C}, \sqsubseteq)$ is a category $\mathcal{C}$ with a partial order $\sqsubseteq$ on homsets such that composition is monotonic: if $f_1 \sqsubseteq f_2$ and $g_1 \sqsubseteq g_2$ then $f_1; g_1 \sqsubseteq f_2; g_2$.

This definition is directly extended to pre-categories.

The notion of a poset enriched category seem to be particularly illuminating in connection with two categories that are pervasive in imperative programming, namely the categories of sets and relations, and that of sets and monotonic predicate transformers. The connection is described in [5].

We argued above that saturation provides us with an appropriate "normalization" operation when comparing refinements. So we put:

Definition 19. let R_1, R_2 be 2 refinements from w_1 to w_2; we say that

- R_2 is *stronger* than R_1 (written $R_1 \sqsubseteq R_2$) if $\overline{R_1} \subseteq \overline{R_2}$;
- R_1 is *equivalent* to R_2 ($R_1 \approx R_2$) if $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_1$.

The following is trivial:

Lemma 2.3. *we have:*

- $\sqsubseteq$ *is a preorder on refinements from* w_1 *to* w_2*;*
- $\approx$ *is an equivalence relation.*

We can now defined our final pre-category:

Definition 20. The pre-category **IntSat** is defined by:

- the objects are interaction structures;
- morphisms from w_1 to w_2 are refinement;

- equality on morphisms is $\approx$.

For this to be well-defined, one needs to check that equality is compatible with composition. This is trivially implied by the following proposition:

Proposition 7. $(\mathbf{IntSat}, \sqsubseteq)$ *is a poset enriched pre-category.*

Proof. `computer checked...`
We only need to check that composition is monotonic.

let $F_1, F_2 : w_1 \to w_2^*$ and $G_1, G_2 : w_2 \to w_3^*$, suppose $F_1 \sqsubseteq F_2$ and $G_1 \sqsubseteq G_2$; suppose furthermore that $s_1 \in S_1$. We need to show that $\overline{F_1; G_1}(s_1) \subseteq \overline{F_2; G_2}(s_1)$:

- $\overline{F_1}(s_1) \subseteq \overline{F_2}(s_1)$ since $F_1 \sqsubseteq F_2$;
- $G_1\big(\overline{F_1}(s_1)\big) \lhd G_2\big(\overline{F_2}(s_1)\big)$:
  let $s_3 \varepsilon G_1\big(\overline{F_1}(s_1)\big)$, *i.e.* $s_2\ G_1\ s_3$ for some $s_2 \varepsilon \overline{F_1} s_1$. We want $s_3 \lhd G_2\big(\overline{F_2}(s_1)\big)$:
  - $G_2(s_2) \subseteq G_2\big(\overline{F_2}(s_1)\big)$ since $s_2 \varepsilon \overline{F_1}(s_1) \subseteq \overline{F_2}(s_1)$;
  - $s_3 \varepsilon \overline{G_2}(s_2)$ because $G_1 \sqsubseteq G_2$ and $s_3 \varepsilon \overline{G_1}(s_2)$ (since $s_3 \varepsilon G_1(s_2)$);
  - so that by monotonicity, $s_3 \varepsilon \mathcal{B}\Big(G_2\big(\overline{F_2}(s_1)\big)\Big)$
- from the above, we get $\mathcal{B}\Big(G_1\big(\overline{F_1}(s_1)\big)\Big) \subseteq \mathcal{B}\Big(G_2\big(\overline{F_2}(s_1)\big)\Big)$ ($\mathcal{B}$ is a closure operator);
- we also have $\mathcal{B}\Big(R\big(\mathcal{B}(U)\big)\Big) = \mathcal{B}\big(R(U)\big)$ (for any refinement R)
  - $\subseteq$: because $R\big(\mathcal{B}(U)\big) \subseteq \mathcal{B}\big(R(U)\big)$ and $\mathcal{B}$ is a closure operator;
  - $\supseteq$: $U \subseteq \mathcal{B}(U) \Rightarrow R(U) \subseteq R\big(\mathcal{B}(U)\big) \Rightarrow \mathcal{B}\big(R(U)\big) \subseteq \mathcal{B}\Big(R\big(\mathcal{B}(U)\big)\Big)$
- so that we have $\mathcal{B}\Big(G_1\big(F_1(s_1)\big)\Big) \equiv \overline{F_1; G_1}(s_1) \subseteq \mathcal{B}\Big(G_2\big(F_2(s_1)\big)\Big) \equiv \overline{F_2; G_2}(s_1)$.

□

3 What about formal topology

3.1 Formal and basic topology

An introduction to the subject of formal topology, and further details can be found in [6, 17, 8]. Let's recall some important points:

Basic topologies see [7] for details

Basic topologies represent the formal side of basic pairs (see [6, 15]). A basic topology is just a set S together with 2 operators $\mathcal{A}$ and $\mathcal{J}$ on $\mathbb{P}(S)$ such that:

- $\mathcal{A}$ is a closure operator;
- $\mathcal{J}$ is an interior operator;
- they satisfy the *compatibility rule*: $\dfrac{\mathcal{A}(U) \between \mathcal{J}(V)}{U \between \mathcal{J}(V)}$

$s \in S$ is called a formal *basic open*. Subsets of S of the form $\mathcal{A}(U)$ are called *open*, subsets of the form $\mathcal{J}(V)$ are *closed*[14].

[14] No mistakes: a formal open is closed in the sense of $\mathcal{A}$. See [15] for justifications.

Formal continuity see [8] for details

For reasons of symmetry, in formal topology continuity is defined for relations rather than functions. If $(S_1, \mathcal{A}_1, \mathcal{J}_1)$ and $(S_2, \mathcal{A}_2, \mathcal{J}_2)$ are basic topologies, and R is a relation between S_1 and S_2; R is *continuous* if:

$$\frac{s_1\ R\ s_2 \qquad s_2\ \varepsilon\ \mathcal{A}_2(V)}{s_1\ \varepsilon\ \mathcal{A}_1\big(R^-(V)\big)}\ (1) \text{ and } \frac{s_1\ R\ s_2 \qquad s_1\ \varepsilon\ \mathcal{J}_1(U)}{s_2\ \varepsilon\ \mathcal{J}_2\big(R(U)\big)}\ (2)$$

or more concisely, $R^-\big(\mathcal{A}_2(V)\big) \subseteq \mathcal{A}_1\big(R^-(V)\big)$ and $R\big(\mathcal{J}_1(U)\big) \subseteq \mathcal{J}_2\big(R(U)\big)$. (There are several other equivalent ways to characterize continuity. See [8].)

Classically, the 2 conditions are equivalent.

Two continuous relations R and T from S_1 to S_2 are equal if $\mathcal{A}(R^- s_2) = \mathcal{A}(T^- s_2)$ for all $s_2 \in S_2$.

The category of formal topologies and continuous relation (modulo the equality just defined) is called **BFTop**.

Balanced basic topologies see [16] for details

The above structure is enough to start doing topology, but it still lacks many properties found in "real" topologies. In particular, we did not mention intersection once. This last rule is fundamental to get a "real" formal topology: (it is sometimes called "summability of approximations".)

$$\frac{s\ \varepsilon\ \mathcal{A}(U) \qquad s\ \varepsilon\ \mathcal{A}(V)}{s\ \varepsilon\ \mathcal{A}(U \downarrow V)}\ \text{balance}$$

where $U \downarrow V = \big\{s \mid (\exists s'\ \varepsilon\ U)\ s\ \varepsilon\ \mathcal{A}(\{s'\}) \text{ and } (\exists s''\ \varepsilon\ V)\ s\ \varepsilon\ \mathcal{A}(\{s''\})\big\}$

This property is equivalent to the infinite distributivity rule found in the definition of locales and frames. This is why basic topologies are sometimes called "non-distributive" topologies.

note: the above definition of "$\downarrow$" uses singletons sets. This requires an equality to be defined on $U(s)$ and $V(s)$

Inductive generation see [3] for details

As shown in [3], we can define a formal topology $(S, \mathcal{A}, \mathcal{J})$ which satisfies the balance rule as soon as we are given a localised axiom set on S:

Definition 21. an *axiom set* on S is a pair $\langle I, C\rangle$ where $I : S \to \mathsf{Set}$ and $C : (s \in S) \to i \in I(s) \to \mathbb{P}(S)$ is a (I indexed) family of subsets of S. Equivalently, an axiom set is a pair $\langle S, f : S \to \mathbb{F}(\mathbb{P}(S))\rangle$.
That a binary relation $\sqsubseteq$ is a *simulation* of the axiom set means that:

$$s \sqsubseteq s' \quad \Rightarrow \quad \big(\forall\ i' \in I(s')\big)\ \big(\exists\ i \in I(s)\big)\ \big(\forall\ t\ \varepsilon\ C(s,i)\big)\ \big(\exists\ t'\ \varepsilon\ C(s',i')\big) \quad t \sqsubseteq t'$$

If $\sqsubseteq$ is a preorder on S, we say that the axiom set is *localized* if $\sqsubseteq$ is a simulation on S.

The main difference between a formal topology in the sense of [3] and a balanced basic topology is that the latter replaces the unary positivity predicate $\mathrm{P}(\_)$ by an "infinitary" one $\_\ \varepsilon\ \mathcal{J}(\_)$. $P(s)$ is then defined to be $s\ \varepsilon\ \mathcal{J}(S)$. Moreover, while P was given, $\mathcal{J}$ needs to be co-inductively defined.

If one doesn't ask for the axiom set to be localized, the resulting structure is a (not necessarily balanced) basic topology.

3.2 The structure $(S, \mathcal{A}, \mathcal{J})$

Let's recall some properties: $\mathcal{A}$ is a closure operator on the subsets of S, $\mathcal{J}$ is an interior operator, and we have the execution property:

$$\frac{E \in \mathcal{A}(U,s) \qquad K \in \mathcal{J}(V,s)}{\mathtt{exec}(E,K) \in U(s') \times \mathcal{J}(V,s')} \text{ where } s' \text{ is the final state.}$$

We can rewrite that using the set theoretic notations (see section 1.1):

$$\frac{s \,\varepsilon\, \mathcal{A}(U) \qquad s \,\varepsilon\, \mathcal{J}(V)}{s' \,\varepsilon\, U \cap \mathcal{J}(V)} \text{ which shortens to } \frac{\mathcal{A}(U) \between \mathcal{J}(V)}{U \between \mathcal{J}(V)}$$

Therefore, the execution property is a computational expression of the compatibility rule!
$(S, \mathcal{A}, \mathcal{J})$ is a "computation aware" basic topology!

One can place cardinality restrictions on the angel's and the demon's choices in an interaction structure, for example to require say that there are at most finitely many choices, or at most one, or exactly one. There are at least two cases of special interest.

Definition 22. An interaction structure $\langle S, A, D, n\rangle$ is *finitary* if $D(s,a)$ is finite for any $s \in S$, $a \in A(s)$. It is *unitary* if $D(s,a) = \{*\}$.

If the balance law holds, these special cases are called Stone spaces, and Scott spaces respectively. (See [3].) In terms of interaction, the significance of the cardinality restrictions is as follows:

- in the finitary case, we have simple, automata-like processes, in which the inputs and outputs are taken from finite alphabets. Many interactive devices in real life fall in this case. (Cash machines, ticket machines, *etc.*)
- in the unitary case the transfer of information is in one direction, from the angel to the demon. A unitary interaction structure is essentially a transition structure which has been lifted by giving all choice to the angel.

To inductively generate the corresponding topology, [3] uses the two following rules:

$$\frac{s \,\varepsilon\, U}{s \,\varepsilon\, \mathcal{A}(U)} \text{ reflexivity and } \frac{a \in A(s) \qquad \big(\forall d \in D(s,a)\big)\,\big(n(s,a,d) \,\varepsilon\, \mathcal{A}(U)\big)}{s \,\varepsilon\, \mathcal{A}(U)} \text{ infinity}$$

The only difference between axiom sets and interaction structures is that the former are "subset based" whereas the later are "family based". An axiom set is an element of the (proper) type $S \to \mathbb{F}\,(\mathbb{P}\,(S))$. (Whereas an interaction structure is in $s \to \mathbb{F}^2 S$.)
Since one can safely go from subsets to families, an axiom set is a special case of an interaction system.

Translating the above reflexivity and infinity rules from axiom sets to interaction structures, one gets the rules (1) and (2) for $\mathcal{A}$.

3.3 The balanced $(S, \mathcal{A}, \mathcal{J})$

As already mentioned, one obtains a real (distributive) topology when the axiom set is localized. This means that the set S comes with a preorder $\sqsubseteq$ which is a simulation on S.

Interaction structures come with a natural notion of "precedence". We write $s \sqsubseteq s'$ when s comes "before" s': when $s = s'$ or when there is a sequence $(s_i, a_i, d_i)_{0 \le i \le n}$, s.t. $s = s_0$, $a_i \in A(s_i)$, $d_i \in D(s_i, a_i)$, $s_{i+1} = n(s_i, a_i, d_i)$ and $s' = n(s_n, a_n, d_n)$.

Writing the simulation condition for this preorder gives:

$$s \sqsubseteq s' \quad \Rightarrow \quad \big(\forall\, a' \in A(s')\big)\,\big(\exists\, a \in A(s)\big)\,\big(\forall\, d \in D(s,i)\big)\,\big(\exists\, d' \in D(s',i')\big)\, n(s,a,d) \sqsubseteq n(s',a',d')$$

which says that "all later states can simulate earlier states"

In term of interaction, requiring $\sqsubseteq$ to be a simulation is difficult to justify: it means the systems cannot "forget" things!

Suppose however that we are given such a world. Then we can inductively generate a formal topology $(S, \mathcal{A})$ using the rules (infinity) and (reflexivity) given above and the rule

$$\frac{s \sqsubseteq s' \qquad s \varepsilon \mathcal{A}(U)}{s' \varepsilon \mathcal{A}(U)} \ (\sqsubseteq\text{-left}) \ .$$

Its only purpose is to close the subset U upward to extend the simulation to the degenerate atomic experiment "$i(\_)$".

Theorem 3.6 of [3] guarantees that the result will be a (distributive) formal topology.

The analysis of interaction is probably the first example arising "in nature" which demonstrates that some non-distributive topologies (*i.e.* basic topologies) can be just as interesting as distributive ones (*i.e.* formal topologies).

3.4 Inductive continuity

As argued above, basic topologies and interaction structures are quite similar. The real test is now to see whether the notion of continuity translates into the realm of interaction structures.

Rather than considering "full" continuity, we will concentrate on the purely inductive fragment of basic topologies (without the operator $\mathcal{J}$.) The main reason for doing so is that "inductive continuity" can be fully formulated inside traditional versions of Martin Löf's type theory. Also, since the second continuity condition (condition (2) on page 22) is classically equivalent to the first condition, "inductive continuity" is as good a candidate for constructive continuity as Sambin's formal continuity.

3.4.1 Continuous relations revisited

Definition 23. We put:

- an inductive basic topology is a pair $(S, \mathcal{A})$ where S is a set, and $\mathcal{A}$ is a closure operator on $\mathbb{P}(S)$;
- if $(S_1, \mathcal{A}_1)$ and $(S_2, \mathcal{A}_2)$ are inductive basic topologies, a relation $R : S_1 \to \mathbb{P}(S_2)$ is inductively continuous if $R^-\big(\mathcal{A}_2(U)\big) \subseteq \mathcal{A}_1\big(R^-(U)\big)$ for any $U \subseteq S_2$.

Inductive continuity is close to the usual formulation of continuity in point-set topology, according to which a function is continuous if the inverse image of an open set is open.

In the sequel, we will drop the adjective inductive, except where the context is not clear.

Thus, any interaction structure with experiments (or processes) is an example of such a basic topology.

Lemma 3.1. *Let w_1 and w_2 be two interaction structures, let R be a simulation from w_1 to w_2, then $R\big(\mathcal{A}(U)\big) \subseteq \mathcal{A}\big(R(U)\big)$ for all $U \subseteq S_1$.*

Proof. Similar (but simpler) to the proof of proposition 4. □

Thus, modulo reversing the direction, both simulations and refinements (see lemma 4) are continuous relations between interaction structures.

Continuity is thus a very practical, down to earth notion!

note: this "reversing the direction" is familiar in the field of formal topology, as it is found in the frames versus locales debate. (Jonstone's terminology.)
The category of frames (like our interaction structures with simulations/refinements) has a more algebraic feeling but the category of locales is more "topology-like".

3.4.2 Equivalence.

The actual category of basic topologies and continuous relation **BFTop** also has a notion of equality that is more subtle than crude extensional equality of relations.

Transposing this equality in our context (by reversing the arrows) we get: $R \approx T$ if and only if $\mathcal{A}(R(s_1)) = \mathcal{A}(T(s_1))$ for all $s_1 \in S_1$, which is easily shown equivalent to $\mathcal{A}(R(U)) = \mathcal{A}(T(U))$ for all $U \subseteq S_1$.[15]

This is exactly the equivalence relation on refinements defined in 2.4.

The (pre-) category $(\mathbf{IntRef}, \approx)^{op}$ is thus the exact inductive counterpart of Giovanni Sambin's category **BFTop**.

What it also tells, is that simulations are not an appropriate definition of general continuity, they are too "constrained". The fact that the saturation of a simulation is itself not a simulation (but a refinement) shows that simulations are not natural. (There is no canonical representative for the equivalence class under $\approx$.)

This came as a relief because the intuition is that while (the inverse of) a refinement transforms basic opens into open sets ($R^- : w_1 \to w_2^*$), (inverses of) simulations transforms basic opens into basic opens ($R^- : w_1 \to w_2$). Somehow, the simulation-continuity depends on what we choose as basic opens, *i.e.* on the basis we choose for the topology!

note: It is perhaps worth recalling the following result (see [8]):

Write $\mathcal{O}(w)$ for the type of "saturated" subsets of S (*i.e.* the type of all subsets U such that $U = \mathcal{A}(U)$).

$\mathcal{O}(w)$ is a sup lattice by defining $\bigvee_{i\in I} U_i \equiv \mathcal{A}(\bigcup_{i\in I} U_i)$.

Proposition 8. *Refinements from w_1 to w_2 "are" sup-lattice morphisms from $\mathcal{O}(w_1)$ to $\mathcal{O}(w_2)$.*

What this says is that for any refinement R from w_1 to w_2, there is an associated sup-lattice morphism $\widetilde{R}$ from $\mathcal{O}(w_1)$ to $\mathcal{O}(w_2)$. It is defined as $\widetilde{R}(U) = R\big(\mathcal{A}(U)\big)$...

The converse[16] also holds, but requires the identity relation to be present.

Concluding remarks

This work has brought into light an application of formal topology to modelling real-world situations, which the authors did not expect to find. What is most striking is the almost perfect fit between the topological and the computational notions.

The question arises of whether and how far the theory of formal topology can be brought to bear on the notions of component and interface in programming. A component, or at least a component of a very pervasive kind, seems to be nothing other than a continuous function.

From the perspective of formal topology, it is perhaps interesting to have a concrete example of a basic topology in which "non-distributivity" is natural. Even better, the example may have some practical relevance! This raises the interest of gaining a detailed view of where distributivity is actually required in formal topology, and what for.

[15] This relies heavily on $\mathcal{A}$ being a closure operator.
[16] any sup-lattice morphism can be presented as a refinement

For the software engineer, our work suggests a qualitatively new way to model component based systems, with some advantages over approaches based on process algebra. This may contribute something to a better understanding of the notion of a handshaken interface.

We have mentioned co-inductive notions at various points, which seem to make good sense from a computational point of view, but we do not know how these can be supported in type theory. Nevertheless, the interpretation of the co-inductive notions is of more than esoteric interest. A reactive strategy —*i.e.* a closed set— is nothing but a server program.

acknowledgments: We would particularly like to thank Giovanni Sambin for inviting the second author to the Formal topology meeting in Venice, and Thierry Coquand for saving us from a deadlock by some remarks.

References

[1] *Basic Concepts of Enriched Category Theory*. Cambridge Univ. Press, 1982.

[2] R-J. Back and J. von Wright. *Refinement Calculus: a systematic introduction*. Graduate texts in computer science. Springer-Verlag, New York, 1998.

[3] Thierry Coquand, Giovanni Sambin, Jan Smith, and Silvio Valentini. Inductively generated formal topologies, 2000. to appear.

[4] Willem-Paul de Roever and Kai Engelhart. *Data Refinement: Model-Oriented Proof Methods and their Comparision*. 47, Cambridge Tracts in Theoretical Computer Science. CUP, Cambridge, UK, 1998.

[5] P. H. B. Gardiner, C.E. Martin, and O. de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22:21–44, 1994.

[6] Silvia Gebellato and Giovanni Sambin. A preview of the basic picture: a new perspective on formal topology. In *Types for proofs and programs (Irsee, 1998)*, pages 194–207. Springer, Berlin, 1999.

[7] Sylvia Gebellato and Giovanni Sambin. The essence of continuity (the Basic Picture, II), 2001. Preprint n. 27, Dipartimiento di matematica, Università di Padova.

[8] Sylvia Gebellato and Giovanni Sambin. Pointfree continutity and convergence (the Basic Picture, III), 2002. draft.

[9] Jean-Yves Girard. Locus solum: from the rules of logic to the logic of rules. *Math. Structures Comput. Sci.*, 11(3):301–506, 2001.

[10] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[12] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[13] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory*. The Clarendon Press Oxford University Press, New York, 1990. An introduction.

[14] Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, UK*, volume 389 of *LNCS*. Springer Verlag, 1989.

[15] Giovanni Sambin. The Basic Picture, a structure for topology (the Basic Picture, I), 2001. Preprint n. 26, Dipartimiento di matematica, Università di Padova.

[16] Giovanni Sambin. Basic topologies, formal topologies, formal spaces (the Basic Picture, III), 2002. draft.

[17] Giovanni Sambin. Some points in formal topology, 2002. to appear?

[18] Giovanni Sambin and Silvio Valentini. Building up a toolbox for Martin-Löf's type theory: subset theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, pages 221–244. Oxford Univ. Press, New York, 1998.