A category of interfaces and components
⇒ DRAFT ⇐

Peter G. Hancock

Updated Nov 8th:
concatenated with notes on predicate transformers.

Contents

1 Introduction

The notion of 'interface' pervades the use and construction of many kinds of artefact, particularly computer programs. A common form of interface is a command-response or imperative interface. In such an interface the user issues commands, to which the system responds, in strict alternation. In computer software, issuing a command is realised by passing control to a procedure or method, and a response by returning control when the procedure terminates.

Programmers rarely write self-standing programs. Typically, they rely on resources of various kinds, such as libraries[1], the primitives of an operating system or language 'runtime', and low-level interfaces or APIs (Application Programming Interfaces). They may even be *writing* libraries, or device drivers rather than user-level executable programs. They usually collaborate in teams with other programmers, each responsible for some piece of a large system. Even when a single person writes software to run directly 'on the metal', they divide the problem into brain-sized chunks. The boundaries of these chunks are interfaces.

The product of a programmer's effort is in reality always something partial, a *component* of a larger *system*. Commonly this takes the form of a module, exporting one 'high-level' interface (for example files and directory trees) by making use of another 'low-level' interface (for example disk drives and segments of magnetic media).

Sooner or later every programmer is bound to ask themselves "what *is* an interface?", and "what *is* a component?". This paper offers an answer to these questions in the form of a mathematical structure, namely an enriched category in which the objects represent command-resource interfaces, and the morphisms represent components that map (or reduce) one interface to another. The hom-sets are partially ordered, and the order represents a certain kind of refinement or improvement between components.

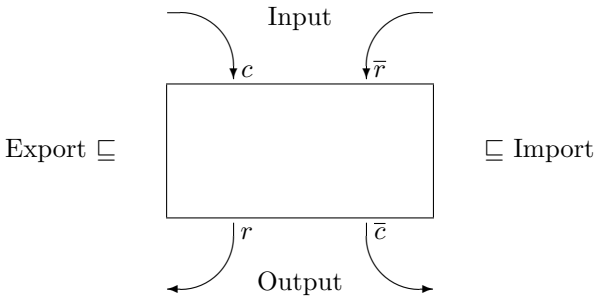
It is characteristic of a program component or partial construction that it has two 'poles', like the poles of a magnet. The poles of a component are *interfaces* – one that is imported, which the component requires or relies on, and one that is exported, which it supplies, provides or implements (on 'top' of the imported interface). There is, as it were, a conditional guarantee: the exported interface will work properly *provided* that the imported one works properly. One may perhaps think of the exported interface as the positive pole (something valuable), and the imported interface as the negative (a cost).

In practice the imported and exported interfaces of a real component commonly have a great deal of structure, and may be subdivided in various ways, and combined with administrative data. In detail, problems may arise in finding suitable components, wiring them up, organising their start-up. There are no solutions to these problems in what follows[2].

Setting many problems aside, a picture for the programmer's task is this:

[1] Commonly shared, dynamically linked libraries.

[2] Subdivision of interfaces is I presume related to products and sums of interfaces.

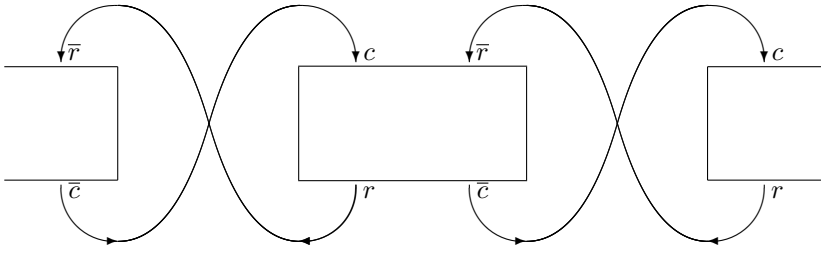


The task is to 'fill the box'. In this picture the horizontal dimension shows interfaces. The exported interface is at the left (the 'outside'), and the imported interface at the right ('the inside'). The vertical dimension shows communication events (calls to and returns from procedures), with data flowing from top to bottom: c and $\overline{r}$ communicate data from the environment, while r and $\overline{c}$ communicate data to the environment. The labels c and r constitute events in the higher level interface, while $\overline{c}$ and $\overline{r}$ are at the lower level. The pattern of communication is that first there is a call c, then some number of repetitions of pairs $\overline{c}\overline{r}$, then finally a return r. One may think of the c's and the r's as opening and closing brackets of two different kinds (*e.g.* round and square brackets) in a language of balanced strings denoted by the regular expression:

$$(c(\overline{c}\overline{r})^*r)^*$$

This regular expression abstracts away the nature of the calls and returns, and the way they depend on the previous history of interactions.

The picture this gives of the assembly of a complete system is that one has a series of to-be-filled boxes, with input arrows linked to output arrows with wiring reminiscent of the Greek letter χ or a cross-bar. This cross-bar stands for composition, or putting together components to form a complete system. It plays a similar role for interactive systems to the piping combinator for Unix 'filter' programs that consume and produce streams.



What is an interface? The notion of an interface pervades the use, design and implementation of devices of all kinds.

The structure below, whose presentation needs to be better organised, is my

best effort so far to define a mathematical structure which can serve as a model (in the sense of applied mathematics) for a particular important common ingredient in the notion of interface.

The word 'interface' is extremely broad, with many uses and shades of meaning covering communication phenomena in the use of artifacts of all kinds. It is inevitable that we have to restrict ourselves to a specific kind of interface, while trying to retain enough generality to cover a significant amount of some important ingredient of the unanalysed intuitive notion. Many phenomena will be out of scope, and indeed some that are important in practice, which we would like to analyse and understand. So be it.

The form of our answer The structure takes the form of a category (with a partial order on the hom-sets), in which the objects model interfaces, and the morphisms model software components, from which entire systems may be built or configured.

What use is a category? Given a little infra-structure, and some syntax, a category serves as the basis for a type-theory. Thus we should have a type-theory in which the types (objects of the category) represent interfaces, and the terms (morphisms between objects) represent components.

The hom-sets represent, so to speak, the design space for a component; they are partially ordered, and have a rich structure in which the order represents the relation of replacebility or refinement between components with the same import and export interfaces. This structure may serve as the basis for a refinement calculus, in which equations and inequations between components are proved by algebraic calculation.

Procedural interfaces The notion of interface to be captured in our model can be called a *procedural*, or *imperative*, or *command-response* interface. In such an interface, there are two parties or agents, that have asymmetric or dual *rôle*s: one is the *client* (who has the initiative in all interactions), and one is the *server* (who speaks only when spoken to). There is an exchange of a pair of messages. The client starts by issuing an instruction (or command), that the server then commits or performs, returning a response or acknowledgement. Issuing an instruction is very often realised by passing control to some procedure in a program module, and commiting the instruction by returning control back to the calling program with output variables set up. My habit is to use the terminology of commands and responses for both the events by which these imperative interactions are initiated or completed, and also for the values or messages that are exchanged.

Atomicity The model described here has a severe shortcoming[3] with respect to the requirements of practical component-based systems: the interaction steps which we shall study are *atomic*. Although calls and returns are different successive events in the evolution of an interface, yet the response to a command must follow it without any intervening event *in that interface*,

[3] which one might hope to remove, or at least to clarify

such as a recursively nested call of some kind, or a call from some other thread of activity in the running program.

Behavioural substance, not syntax It should be emphasised that what we are looking for here is *not* the syntactic 'shape' of a command-response interface. That is in practice coded using an interface description language such as IDL. The information encoded is what is needed for marshalling data for possible remote transmission, for error propagation, *etc.*. This is a matter of parsing and unparsing.

Rather we want to capture is input-output behaviour, abstracting from details of how input arguments and results are encoded. By input output behaviour I mean a contract on which the programmer of a component which imports the interface may rely, and which the programmer of a component which exports the interface must guarantee.

We choose to ignore the way in which messages are encoded in a state [4].

questions of constructivity and predicativity It may be of some interest that the category can be defined in a constructive and predicative setting.

One reason for working in a constructive framework is that a model set up in such a framework for program components is automatically a 'working' model, which in a sense 'goes'. A constructive proof is *ipso facto* a program. In principle, one might write executable program components in constructive type-theory, and exploit type-checking to ensure that they behave correctly.

In practice, one has to write programs in programming languages other than constructive type theory. To propose otherwise would be Quixotic.

The contribution dependent type theory can make to the practice of software engineering may lie more at the level of specification and high level design, than at the level of implementation, code, or verification. Types play no real role when a program is running.

It may be more important that one can *document* the interfaces between which components are programmed (using C, C++ or Java perhaps as a programming language) by exploiting the notion of dependent type as analysed in constructive type theory, with its universes and inductively defined families of sets.

As Goebbels might have said, to gain control over things, one must first gain control of the language in which we specify them.

One can raise the question of equality only in a predicative framework. In such a framework the power of a set need not be a set (over which we may quantify in forming propositions) but can be a 'large' object, akin to a proper class. In an impredicative framework, it is trivial to define singleton predicates, *à la* Leibnitz. $a \,\epsilon\, \{s\} \iff (\forall U : \mathbb{P}(S))\ \ s \,\epsilon\, U \rightarrow a \,\epsilon\, U$

I regard it as an advantage of a predicative framework that one can isolate the operation of forming singleton predicates as a separable logical ingredient of the idea of an inductive definition.

[4] Perhaps this can be studied using simulations, or interface refinements.

It is a disadvantage of a predicative framework that one loses access to the formally straightforward impredicative foundation for both inductive definitions with definition by well-founded recursion, and co-inductive definitions with definition of potentially infinite objects (and relations with which to compare them) by forms of non-wellfounded recursion.

Sources

- Hoare's process calculus CSP[2]. This has a variable-binding constructor 'choice' that resembles the supremum operation of a W-type (apart from well-foundedness). Hoare's book sketches an implementation of certain CSP programs (vending machines, etc) using a lazy functional language. As they run, the user makes choices from successivelt presented menus. The presentation of a menu, followed by selection within it is a pattern of alternating communication
- dependent state machines. One can extend the usual notion of (deterministic) state machine in an interesting direction, by allowing the set of commands to depend on the current state, and the set of responses to depend on the state and input command. If one thinks of the transition function of a (Mealy) state machine as something of type

  $$S \times C \to R \times S$$

  and then allows the set C of commands to depend on the initial state, and the set R of responses to depend on both the initial state and the input command, one arrives at the type

  $$(s : S, c : C(s)) \to R(s, c) \times S$$

  This is the type of functions which map a state and a command to a response and new state. These play a key *rôle* in server programs, as discussed below (somewhere). If one starts instead with Moore machines, one obtains a type that plays an analogous *rôle* for client programs.

  Another point of departure is the notion of non-deterministic state machines, and its analysis with dependent types.
- the 'tree-sets' of Petersonn and Synek [5]. Their trees were introduced as parse trees in certain formal grammar. They provide the right setup for describing interaction structures, and associated notions. This setup can represents a formal grammar in which a number of syntactical categories are defined in terms of unions of sequential patterns:

$S : \mathrm{Set}$	syntactical categories
$C : S \to \mathrm{Set}$	alternates, per category
$R : (s : S) \to C(s) \to \mathrm{Set}$	conjuncts, per cat/alt
$n : (s : S, c : C(s)) \to R(s, c) \to S$	category of conjunct

  They gave a description of the syntax trees induced by such a structure. This is an a simultaneous inductive definition of a indexed family of sets of trees, where the index set is fixed ($= S$). The definition they gave is the 'nub' of the reflexive transitive closure operator * on interaction structures.
- several remarks by Thierry Coquand, particularly concerning the importance and interest of simulation relations.
- formal topology as developed over more than a decade by Giovanni Sambin, recently with several collaborators. Interaction structures provide examples of Sambin's 'basic'-topologies (check!), giving yet more

evidence of the interest of this notion. A 'basic' topology is a little wilder than the ordinary kind of topology. Cite something.

Indicate something about Distributivity? Localisation?

- Back and von Wright's refinement calculus [1]. This is a calculus for reasoning about inclusion between predicate transformers (also relations *etc.*). It has a highly algebraic flavour, and smooth ergonomical features necessary for a working calculus.

  Interaction structures are essentially predicate transformers, presented in a concrete, computational form. Predicate transformers have been linked with the semantics of programming languages since the earliest investigations into program correctness. The key thing a prospective client wants from a specification of an imperative interface is a predicate transformer that lets the client work out what has to be established first, in order to establish (bring about) a desired predicate in a single cycle or operation of the interface.

  I diverge from Back and von Wright's notation in a number of respects, primarily in using "angle brackets" $\langle \_ \rangle$ for angelic update. Back and von Wright use $\{ \_ \}$, using angle brackets for functional updates.

- the insights and collaboration of Anton Setzer. Anton defined a notion of 'redirection' between interfaces that is close if not identical to the notion of simulation defined here. He has studied the situation in which state-dependency is absent. He has studied the question of justifying the use of coinductively defined notions in dependent type theory.

- the insights and collaboration of Pierre Hyvernat. Pierre has connected interaction structures with Sambin's basic topologies: this suggested the idea of coarsening the equivalence relation on simulations from extensional equality to extensional equality of saturations. He has also connected them with certain notions in Girard's ludics, linear logic, and game-theoretical approaches to program semantics. He has established several properties of the category of interfaces and simulations, and shown that this category provides (at least in an impredicative and classical setting) a model for full linear logic.

2 Preliminaries and notation

The following is an incomplete collection of basic notions, notations and terminology needed to present the category.

2.1 Two notions of subset

We distinguish two notions of subset, or more precisely two ways in which a subset of a set can be given. To present them immediately, they are as follows.

$$\{ s : S \mid P(s) \}$$
$$\{ s_i \mid i : I \}$$

The first picks out the subset's elements, according to whether they satisfy a predicate. The second runs through them all, as i various over I.

It is enlightening to present the distinction between these two notions as a contrast between functors of different variance, called here $\mathbb{F}(\_)$ and $\mathbb{P}(\_)$.

In a predicative context, they are endofunctors on a 'large' category of types[5] which has not only normal mathematical sets for its objects, but also the type Set itself, and types built on top of it by dependent function and product types. The morphisms between such objects are functions, that map a domain-type into a codomain type.

Our two endofunctors are as follows.

- The **propositional** power $\mathbb{P}(\_)$. Here one thinks of a subset of a set S as a propositional function, predicate or criterion which distinguishes those elements of S which are members of the subset from those which do not. (Sheep and Goats.)

$$\begin{array}{l}\mathbb{P}(\_) : \mathrm{Type} \to \mathrm{Type} \\ \mathbb{P}(S) \triangleq S \to \mathrm{Set} \\ (\ \textit{typical form}\ ) \quad P = \{\, s : S \mid P(s) \,\} : \mathbb{P}(S)\end{array}$$

$$\begin{array}{l}\mathbb{P}(\_) : (A, B : \mathrm{Type}) \to \hom(A, B) \to \hom(\mathbb{P}(B), \mathbb{P}(A)) \\ \mathbb{P}(f : A \to B)\{\, b : B \mid P(b) \,\} \triangleq \{\, a : A \mid P(f(a)) \,\}\end{array}$$

  Note that this functor is contravariant. Its operation on morphisms is sometimes known as the inverse-image f^{-1} or precomposition $(\cdot f)$ operator. In programming, it is a functional state update or assignment $s := f(s)$. The axiom which corresponds to this notion of subset in ZF set-theory is the separation schema, which guarantees that the subsets of a set picked out by predicates over that set are themselves sets.

- The **parametric** power $\mathbb{F}(\_)$. Here one thinks of a subset of a set or type S as a morphism into S defined on some other 'index' set. Here there is a parametric expression $t(i)$ with a free variable i (the parameter) varying over some index set I.

$$\begin{array}{l}\mathbb{F}(\_) : \mathrm{Type} \to \mathrm{Type} \\ \mathbb{F}(S) \triangleq \text{ the type of pairs } \langle I, t\rangle \text{ with } I : \mathrm{Set}, t : I \to S \\ (\ \textit{typical form}\ ) \quad \langle I, t\rangle = \{\, t(i) \mid i : I \,\} : \mathbb{F}(S)\end{array}$$

$$\begin{array}{l}\mathbb{F}(\_) : (A, B : \mathrm{Type}) \to \hom(A, B) \to \hom(\mathbb{F}(A), \mathbb{F}(B)) \\ \mathbb{F}(f : A \to B)\{\, t(i) \mid i : I \,\} \triangleq \{\, f(t(i)) \mid i : I \,\}\end{array}$$

  Note that the functor $\mathbb{F}(\_)$ is co-variant. Its operation on morphisms is most often known as direct-image $\exists_f(U) = f(U) = \{\, f(u) \mid u \,\epsilon\, U \,\}$, or postcomposition : $(f\cdot)$. The axiom of ZF set-theory that is most relevant to this notion of set is the replacement schema, which guarantees that the images of sets under functional relations are themselves sets.

[5] It is necessary to use such a large category only because we want to allow that the powerset of a set need not be a set. Something else we can do is localise things to a universe.

2.2 Identity

The propositional and the parametric notions of subset of a set S are linked via the relation of equality, or propositional identity between elements of the set S. The relation associates with any element s of a set S the *singleton* predicate $\{s\}$ which holds of just that element.

The singleton predicate 'behaves with respect to other predicates' as follows:

$$\{\, s \,\} \subseteq U \;\triangleq\; U(s) \;\triangleq\; \{\, s \,\} \between U$$

One might also write this predicate using section notation: either $(= s)$ or $(s =)$. In the same notation, the equality relation is written $(=)$.

If we abstain from or confine the use of this operation to regions which are as small as possible, the two forms of the notion of subset fall into relief.

To obtain a predicate from a family requires use of a relation such as equality. To obtain a family from a predicate requires quantification over states.

There are different approaches to propositional identity.

- that it is a reflection of judgment-level equality. This is the approach taken in so called extensional type-theory, in which a proof of an propositional equation can be used to justify an equational judgement.

  Adopting this approach raises problems for the mechanisation of extensional type theory, as type-correctness is then undecidable.

- that if S is a set and if s is an element of S, then the singleton predicate $\{s\}$ is inductively definable as the strongest (least) predicate satisfied by the value s. This is the approach taken in intensional type-theory.

  This approach is not entirely satisfactory, as in many sets (for example sets of functions) intensional equality makes dubious mathematical sense. Moreover proofs of intensional equations have no computational meaning: we have no interest in their canonical form, beyond in its mere existence.

- that sets are intrinsically quotients. Depending on the set, one defines *ad lib* an appropriate relation of equality between elements of that set (*e.g.* equiconvergence between Cauchy sequences of rationals; or whatever seems appropriate, so long as it yields an equivalence relation). Of course, one has then to confine oneself to predicates and operators which depend only on the equivalence class of a value, rather than on the representing value itself.

In some sense, identity (or the operator which produces a singleton predicate) is a source or locus of non-computational phenomena in type theory. I prefer for methodological reasons to confine use of the general notion of singleton predicate (beyond its use in connection with an explicit equivalence relation) to regions which are as small as possible.

The distinction between subsets *à la* predicates and subsets *à la* families is grammatical: a distinction between forms of expression. The grammatical distinction reflects the different use to which predicate and family expressions are put. We use the predicate form in connection with *saying* something

(forming a proposition), whereas we use the latter in connection with *doing* something (computing a chosen element in the subset).

In the examples given later (??), we make use of yet another notion of 'singleton'. Given a set S and an equivalence relation $=$ on S, one can form for each $s \in S$ the set of elements of S that are equivalent to s, written $\{s\}_=$ or $(= s)$. The elements of this set are pairs, consisting of an element s' of S and a proof of the equation $s = s'$. The set $\{s\}$ can be defined using the existential quantifier with its constructive interpretation as $(\exists\, x \in S)\ \ x = s$, or (in terms of the singleton predicate) as $(\exists\, x \in S)\ \ x \,\epsilon\, \{s\}$. Note that an element of a singleton set is a value, together with a proof that it is in the appropriate equivalence class.

If the equivalence is boolean-valued (decidable), the canonical form of the proof of an equation can be of no interest: it will be 'tick', 'ok', the sole element of the standard singleton set – a pure certificate.

The 'interesting' part, if not the whole of an element of a singleton set is the value itself.

Notation for predicate application I sometimes use the symbol ϵ for 'backwards' application of predicates to values, so that if P is a propositional subset of a set S, and s is an element of S, the proposition

$$s \,\epsilon\, P$$

says that P holds of s. The elements of $s \,\epsilon\, P$ are just those of $P(s)$.

Location of families with respect to predicates Even without use of propositional identity, it is possible to state that a parametric subset $t = \{\, t(i) \mid i : I \,\} : \mathbb{F}(S)$ is 'located' with respect to a propositional subset $P = \{\, s : S \mid P(s) \,\} : \mathbb{P}(S)$ either by inclusion or overlapping:

$$\begin{array}{lll} t \subseteq P & \text{means} & (\forall i : I) P(t(i)) \\ t \between P & \text{means} & (\exists i : I) P(t(i)) \end{array}$$

The splendid notation $A \between B$ for overlapping (that hides an existential quantifier in the same way as $A \subseteq B$ hides a universal quantifier) is due to Giovanni Sambin.

Predicates can freely be compared with predicates by $\subseteq$ and $\between$. Other forms of comparision (for example between parametric subsets) require use of the identity relation to replace a family in 'operating position' with a predicate.

Algebraic structure *vs.* **computational content** Predicates are closed under union and intersection of indexed families, forming a distributive lattice, having all set-indexed suprema and infima, notions of complement and implication. On the other hand families are closed only under indexed union.

Predicates are connected with what we say, or specify: they support a rich algebraic structure. Families are connected with what we do or compute: they form an impoverished algebraic structure, which is the price we pay for having something 'computational' (a family) – which in the case consumes an externally selected index (or choice) and produces a value.

2.3 Relations and abstract transition systems

By a relation I mean a binary propositional function, or a predicate 'with a parameter'. A relation between A and B is an element of the (proper) type

$$A \to \mathbb{P}(B) \quad \cong \quad \mathbb{P}(A \times B)$$

The isomorphism will be referred to below as 'Currying', as it is analogous to replacing an ordered pair argument by two separate arguments for the first and second coordinates. A relation is *homogeneous* if its two argument-places have the same type; for example the equality relation on a set is homogeneous.

Roughly speaking, the algebraic structure of relations is that of predicates, and then some. The additional 'some' is essentially:

sequential composition $R_1 \,;\, R_2$, with its identity SKIP.

division R_1/R_2. Together with sequential composition, this satisfies the following adjunction:

$$S \,;\, R_2 \subseteq R_1 \;\text{ iff }\; S \subseteq (R_1/R_2)$$

inversion $R^\sim$, which is an involution.

closure operations $R^?$, R^+, R^*, for (respectively) reflexive, transitive, and reflexive-transitive closure of a homogeneous relation R.

graphs graphs[6] of functions $f : A \to B$ written

$$\mathbf{graph}\, f : A \to \mathbb{P}(B)$$

Note: As relations, the graphs of functions are both deterministic (simple, at-most-one-valued, meaning $R^\sim \,;\, R \subseteq 1$), and total (entire, at-least-one-valued, meaning $1 \subseteq R \,;\, R^\sim$). As predicate transformers $\langle f \rangle = [\, f \,] = (\cdot f) = \mathbb{P}(f)$. (Such a predicate transformer commutes with all intersections and unions.)

tests A test is a relation in which the state does not change (and so the relation must be a subset of the equality relation. Tests are written

$$\mathbf{test}\, U$$

Note: restriction of a relation in the domain or codomain can be effected by composing it with a test on one side or the other.

Spans. Any relation can be put into the form $(\mathbf{graph}\, f)^\sim \,;\, \mathbf{graph}\, g$, that is of first the inverse of a function, then a function. Such a pair of functions (essentially an element of $\mathbb{F}(A \times B)$) is called a span, and a partial order can be defined between spans which reflects inclusion between relations.

A similar trick works at the next level: one can factor any predicate transformer into one which commutes with unions, followed by one which commutes with intersections.

[6] This needs the identity relation. A complete account of assignment involves taking states to be records in which the fields correspond to assignable variables.

Categories of relations The category *rel* whose objects are sets and whose morphisms are relations has a rich structure on its homsets, particularly so when the relations are homogeneous (*i.e.* endomorphisms).

It helps to think first of the category of 'arrows' over *rel*, which we might write $rel^{\downarrow}$. The objects of this category are relations $A \to \mathbb{P}(B)$ for types A and B, and the morphisms from $\alpha : A \to \mathbb{P}(B)$ to $\beta : C \to \mathbb{P}(D)$ are pairs of relations $Q_1 : A \to \mathbb{P}(C)$, $Q_2 : B \to \mathbb{P}(D)$ such that we have a subcommutative square

$$Q_1{}^{\sim} ; \alpha \subseteq \beta ; Q_2{}^{\sim}$$

Note that the subcommutativity condition can also be written as a simple inclusion: $Q_1 \subseteq ((\beta ; Q_2{}^{\sim})/\alpha)^{\sim}$. We call this the category of relations and subcommutative squares between them.

When the domain and the codomain of a relation are the same, the relation is said to be homogeneous. A homogeneous relation is the same thing as an endomorphism in *rel*. These form a category of 'cycles' over *rel*, which we might write $rel^{\circlearrowleft}$. The morphisms from $\alpha : A \to \mathbb{P}(A)$ to $\beta : B \to \mathbb{P}(B)$ are relations $Q : A \to \mathbb{P}(C)$ such that

$$Q^{\sim} ; \alpha \subseteq \beta ; Q^{\sim}$$

We call this the category of homogeneous relations and elementary simulations. The word 'elementary' is supposed to call to mind that a step taken in the simulated homogeneous relation corresponds to a single step in the simulating homogeneous relation.

We also need the notion of a pointed relation, or relation together with a notion of 'current' or 'initial' state. [7]

When the domain and the codomain are equal, and we additionally have a distinguished initial state, the most natural course is to require that simulations relate the initial states. We call this the category of abstract transition systems.

'Abstract' because the transitions are not indexed by a set. An unlabelled transition system is just a binary relation on a set, whereas a labelled transition system is a family of relations indexed by labels. (An 'axiom set' in Giovanni's sense is the same as a function $S \to \mathbb{F}(\mathbb{P}(S))$, which is a version of *labelled* transition system in which one gives for each state s the set $L(s)$ of labels on transitions from that state. Each label $l : L(s)$ determines a predicate holding of states to which there is a transition from s: the transition itself is perhaps the proof that the predicate determined by s and l has solutions.

2.4 Concrete transition structures and systems

Transition structures are the concrete, computational counterpart of relations, and relations the abstract, specificational counterpart of transition structures. Whereas a relation is something whose type has the form $A \to \mathbb{P}(B)$, a transition structure is something whose type has the form $A \to \mathbb{F}(B)$.

[7] Another idea that may need a name is a relation together with a non-empty set of 'start' or initial sets.

I shall use Greek letters ϕ, ψ, ... to denote transition structures. A transition structure $\phi : A \to \mathbb{F}(B)$ is given by two pieces of data: a family of index sets $\{ T_\phi(a) \mid a : A \}$, and a corresponding family of indexing functions

$$a \mapsto \{ a[i]_\phi \mid i : T_\phi(a) \}$$

In a transition structure, a state is interpreted as a 1-dimensional *array*. (The index set may depend on the state.) In an interaction structure it is interpreted as a 2-dimensional entity, or generalised *matrix*. (The generalisation is un-even-ness: that the index set for a row may depend on the column.)

In comparision with relations, the algebraic structure of transition structures is impoverished. They are closed under neither inversion nor intersection. They are nevertheless closed under indexed union, sequential composition, and in the homogeneous case under reflexive, transitive and reflexive-transitive closure.

Tangentially, transition structures support some arithmetic structure, much the same as Cantor's order types. There are natural definitions of zero, ordered addition, successor, multiplication, and ordered sum that make one and multiplication a monoid, and zero and addition a monoid commuting with multiplication on the right. The operations preserve transitivity and well-foundedness.

Not only can transition structures be combined with other transition structures, they can also be combined with relations in interesting ways, yielding relations. That is to say, a transition structure can be used as a relation-transformer. Among these uses are the following; note that the definition of these operators (pre-composition, post-division) does not require any use of the identity predicate.

$$(\phi \,;R) \qquad (R/\phi)$$

The definitions are as follows.

$$\begin{array}{lcll} (\phi \,;R)a & = & \{ c \mid \phi(a) \between R^\sim(c) \} & = & (\langle \phi \rangle \cdot R^\sim)^\sim a \\ (R/\phi)a & = & \{ b \mid \phi(b) \subseteq R(a) \} & = & ([\, \phi \,] \cdot R)a \end{array}$$

In combination with inversion, these operators are sufficient to define directly (*i.e.* without use of identity) the notion of (relational) simulation between concrete transition structures. A simulation is a post-fixed point for a certain operator on relations which can be defined using pre-composition and post-division by the two transition structures.

Two transition structures $\phi : A \to \mathbb{F}(B)$ and $\psi : C \to \mathbb{F}(D)$ determine a certain 'backwards' relation transformer from $B \to \mathbb{P}(D)$ to $A \to \mathbb{P}(C)$, whose value at a relation R is the relation which maps $a : A$ to the predicate:

$$\{ c : C \mid (\forall i : T_\phi(a))(\exists j : T_\psi(c)) R(a[i]_\phi, c[j]_\psi) \}$$

The relation transformer can also be written $R \mapsto ((\psi \,; R^\sim)/\phi)^\sim$, or even $((^\sim) \cdot (/\phi) \cdot (\psi \,;) \cdot (^\sim))$ where '$\cdot$' stands for composition of relation transformers.

An *elementary simulation* of transition structure $\phi : A \to \mathbb{F}(A)$ by $\psi : B \to \mathbb{F}(B)$ is then a relation $Q : A \to \mathbb{P}(B)$ such that

$$Q^\sim \,; \phi \subseteq \psi \,; Q^\sim$$

A concrete transition system consists of a set S, a transition structure $\phi : S \to \mathbb{F}(S)$, and an initial state $s_0 : S$. A elementary simulation between transition systems is defined to be an elementary simulation of the transition structure of one by the transition structure of the other, which relates the two initial states. A (general) simulation (also called a refinement) is defined to be an elementary simulation of one transition system by the reflexive and transitive closure of the other.

$$Ref[\tau_1 \to \tau_2] = Sim[\tau_1 \to \tau_2^*]$$

One could perhaps use the terminology "linear" instead of "elementary". Then it is natural to call a morphism in $Sim[\tau_1 \to \tau_2{}^?]$ an "affine" simulation, and $Sim[\tau_1 \to \tau_2{}^*]$ a general simulation.

The terminology isn't good.

3 Objects: interfaces

To recapitulate, we have two functors $\mathbb{P}(\_)$ and $\mathbb{F}(\_)$, and from these two subtly different notions of relation: 'true' relations $A \to \mathbb{P}(B)$ and transition maps $A \to \mathbb{F}(B)$.

Relations (of either kind) can be presented as objects in a category of 'arrows' over the category of sets and relations, in which the morphisms are sub-commutative squares.

When $A = B$, the arrows are in fact cycles, and we have something that can be iterated, or combined with identity (equality) in various ways. We say that we have a *structure*. When we also have a designated initial state, we say we have a *system*. Between systems (of either kind) we have a category in which the morphisms are simulation relations that hold between the initial states.

A transition structure models a program whose progress from state $s : A$ to state $s[t] : B$ requires choice of an index $t : T(s)$ (as it were requires some external agency to supply the t's in a one-way form of interaction). A relation (in the propositional sense) models a specification for a function $f : A \to B$, and need not be executable or constructive in any sense.

So much for relations and transition structures, where there is one level of choice; now we come to predicate transformers and interaction structures. These introduce a second level (or dimension) of choice.

An interaction structure from a set S to a set S' is an object of type

$$S \to \mathbb{F}(\mathbb{F}(S'))$$

It is given by the following data:

- For each $s : S$ a set $C(s)$ of commands that may be issued in state s
- For each $s : S$ and $c : C(s)$ a set $R(s, c)$ of responses to command c issued in state s
- For each $s : S$, $c : C(s)$ and $r : R(s, c)$ a state $n(s, c, r) : S'$.

For comparision, a predicate transformers from a set A to a set B is an object of the type

$$\begin{array}{ll} & A \to \mathbb{P}(\mathbb{P}(B)) \\ \cong & \quad (\ \textit{Curry}\ ) \\ & \mathbb{P}(A \times \mathbb{P}(B)) \\ \cong & \quad (\ \textit{flip arguments}\ ) \\ & \mathbb{P}(\mathbb{P}(B) \times A) \\ \cong & \quad (\ \textit{Curry, in reverse}\ ) \\ & \mathbb{P}(B) \to \mathbb{P}(A) \end{array}$$

Between predicate transformers[8], we define a partial order, by lifting the inclusion relation between predicates "pointwise": $F \sqsubseteq G$ means $(\forall\, X : \mathbb{P}(B))\ \ F(X) \subseteq G(X)$. Equality is then pointwise extensional equality. Note that the definitions of inclusion and equality use quantification over predicates $X : \mathbb{P}(B)$. In a predicative setting this raises some difficulties.

An interaction structure $\Phi : S \to \mathbb{F}(\mathbb{F}(S))$ determines two monotone predicate transformers : $\mathbb{P}(S') \to \mathbb{P}(S)$, which I call disjunctive and conjunctive normal form respectively. Also angelic and demonic. Also SigmaPi and PiSigma.

$$\begin{array}{l} \Phi^\circ, \Phi^\bullet : \mathbb{P}(S') \to \mathbb{P}(S) \\ \Phi^\circ(U) \ \triangleq\ \{\, s : S \,|\, (\exists\, c : C(s))\ \ \{\, s[c/r] \,|\, r : R(s,c) \,\} \subseteq U \,\} \\ \Phi^\bullet(U) \ \triangleq\ \{\, s : S \,|\, (\forall\, c : C(s))\ \ \{\, s[c/r] \,|\, r : R(s,c) \,\} \between U \,\} \end{array}$$

The former ($\_^\circ$, that is dnf) is more fundamental in the sense that latter ($\_^\bullet$, that is cnf) can be re-expressed in angelic form by applying the axiom of choice. We mention $\_^\bullet$ again only in connection with inversion.

We do not explicitly write the function which coerces an interaction structure Φ to the predicate transformer Φ°, where the ambiguity can be resolved by the context.

Observation: interaction structures are morphisms of a category IS in which the objects are sets, and the homsets are lattices, closed under indexed suprema and infima.

Composition in the category is defined as follows.

$$\begin{array}{l} C_{\Phi;\Psi} \ \triangleq\ \{\, s : S \,|\, (\exists\, c : C_\Phi(s))\ \ (\forall\, r : R_\Phi(s,c))\ \ C_\Psi(s[c/r]) \,\} \\ \qquad = \Phi(C_\Psi) \\ R_{\Phi;\Psi}(s, \langle c, f\rangle : \Phi(C_\Psi, s)) \ \triangleq\ (\exists\, r : R_\Phi(s,c))\ \ R_\Psi(s[c/r]_\Phi, f(r)) \\ s[\langle c, f\rangle / \langle r, r'\rangle]_{\Phi;\Psi} \ \triangleq\ (s[c/r]_\Phi)[f(r)/r']_\Psi \end{array}$$

The unit of composition we call SKIP, which maps a state s to the singleton family of the singleton family of s. (The point is that we should have SKIP° the identity with respect to ;.)

$$\begin{array}{l} C_{\text{SKIP}}(s) \ \triangleq\ N_1 \\ R_{\text{SKIP}}(s, \_) \ \triangleq\ N_1 \\ s[\_/\_]_{\text{SKIP}} \ \triangleq\ s \end{array}$$

[8] We are interested here in monotone predicate transformers only

Suprema are defined as follows.

$$\begin{array}{l} C_{\sqcup_{i\in I}\Phi_i} \triangleq \{\, s : S \mid (\exists\, i \in I)\ \ C_i(s)\,\} \\ \qquad = \bigcup_{i\in I} C_i \\ R_{\sqcup_{i\in I}\Phi_i}(s, \langle i, c\rangle) \triangleq R_i(s,c) \\ s[\langle i,c\rangle/r]_{\sqcup_{i\in I}\Phi_i} \triangleq s[c/r]_i \end{array}$$

The supremum of the empty family is 'abort', which is the worst possible program for the angel. Thought of as a contract one might make use of, it is entirely useless. Nothing is guaranteed.

Infima are defined as follows.

$$\begin{array}{l} C_{\sqcap_{i\in I}\Phi_i} \triangleq \{\, s : S \mid (\forall\, i \in I)\ \ C_i(s)\,\} \\ \qquad = \bigcap_{i\in I} C_i \\ R_{\sqcap_{i\in I}\Phi_i}(s, f) \triangleq (\exists\, i \in I)\ \ R_i(s, f(i)) \\ s[f/\langle i,r\rangle]_{\sqcap_{i\in I}\Phi_i} \triangleq s[f(i)/r]_i \end{array}$$

The empty infimum is sometimes called 'magic', or 'miracle'. Such a resource could be used to accomplish the impossible. (Pierre calls it 'stop' .. check.)

When the codomain and the domain of an IS are the same (ie. it is homogeneous, an endomorphism), there is a lot of further structure: closure operations: $\_^?$ (reflexive closure), $\_^*$ (transitive and reflexive closure), $\_^+$ (transitive closure). These jointly satisfy

$$\begin{array}{l} \Phi^? = \text{SKIP} \sqcup \Phi \\ \Phi^* = (\Phi^+)^? \\ \Phi^+ = \Phi\,;\Phi^* \end{array}$$

They can be constructed entirely at the level of C's and R's. The following comprises an inductive definition of the S-indexed family of sets

$$C_{\Phi^*} : \mathbb{P}(S)$$

This is followed by a definition of a function of type $(\forall\, s : S)\ \ C(s) \to \mathbb{F}(S)$ by (well-founded) recursion on the second argument c. Note that strictly speaking, we presuppose a universe of sets which contains N_1 and is closed under $(\exists \ldots)$.

$$\begin{array}{l} C_{\Phi^*} \triangleq (\mu\, U : S \to \text{Set})\ \ (\forall\, s : S)\ \ N_1 + \Phi(U,s) \subseteq U(s) \\ \qquad = \Phi^*(\mathit{True}) \\ R_{\Phi^*}(s, \mathbf{i}\,1) \triangleq N_1 \\ R_{\Phi^*}(s, \mathbf{j}\,\langle c, f\rangle) \triangleq (\exists\, r : R_\Phi(s,c))\ \ R_{\Phi^*}(s[c/r]_\Phi, f(r)) \\ s[\mathbf{i}\,1/1]_{\Phi^*} \triangleq s \\ s[\mathbf{j}\,\langle c,f\rangle/\langle r,r'\rangle]_{\Phi^*} \triangleq (s[c/r]_\Phi)[f(r)/r']_{\Phi^*} \end{array}$$

For each state s, the elements of $C_{\Phi^*}(s)$ can be seen as 'client'-programs. An agent running it issues a sequence of commands. After each command, it waits for a response The program then tells the agent what to do next depending on the response. This is the behaviour of a client. Eventually, there are no more commands to be issued, there is an exit from the program and the run

terminates. (This the behaviour of a client which terminates.) Given such a program c, the elements of $R_{\Phi^*}(s, c)$ can be seen as traces, logs are histories of responses to the program c with the property that they lead to an exit point of the program. They are witnesses to the fact that there are 'holes' or exits in the program. (A 'hole' should not be thought of as a defect; in fact 'exit' is the same as successful completion.) Each such trace r determines a state $s[c/r]_{\Phi^*}$, which is the state of the interface when the program exits.

angelic and demonic update Relations (of either the propositional or computational variety) lift to predicate transformers – in two ways. If Q is a relation of type $A \to \mathbb{P}(B)$, then we define two predicate transformers, for which we adapt the notation of Back and von Wright.

$$\begin{array}{l}
\langle _ \rangle, [\,_\,] : \mathbb{P}(B) \to \mathbb{P}(A) \\
\langle Q \rangle (U : \mathbb{P}(B)) \triangleq \{\, a : A \mid Q(a) \between U \,\} \qquad (\textit{ angelic update, assertion }) \\
[\,Q\,](U : \mathbb{P}(B)) \triangleq \{\, a : A \mid Q(a) \subseteq U \,\} \qquad (\textit{ demonic update, assumption })
\end{array}$$

$$\begin{array}{l}
C_{\langle Q \rangle} \triangleq \langle Q \rangle (\mathit{True}) \\
R_{\langle Q \rangle}(s, \langle s', _ \rangle) \triangleq N_1 \\
s[\langle s', _ \rangle / _]_{\langle Q \rangle} \triangleq s'
\end{array}$$

$$\begin{array}{l}
C_{[\,Q\,]}(s) \triangleq N_1 \\
R_{[\,Q\,]}(s, _) \triangleq s \in \langle Q \rangle (\mathit{True}) \\
s[_ / \langle s', _ \rangle]_{[\,Q\,]} \triangleq s'
\end{array}$$

Note that $\langle \phi \rangle$ and $[\,\phi\,]$ make perfectly good sense where ϕ is a transition structure. These operations can be defined without use of singleton predicates.

$$\begin{array}{l}
C_{\langle \phi \rangle} \triangleq T_\phi \\
R_{\langle \phi \rangle}(s, t) \triangleq N_1 \\
s[t/_]_{\langle \phi \rangle} \triangleq s[t]_\phi
\end{array}$$

$$\begin{array}{l}
C_{[\,\phi\,]}(s) \triangleq N_1 \\
R_{[\,\phi\,]}(s, _) \triangleq T_\phi(s) \\
s[_ / t]_{[\,\phi\,]} \triangleq s[t]_\phi
\end{array}$$

The notation '$\langle \_ \rangle$' is used instead of Back and von Wright's brace notation $\{\_\}$, which collides with our use of set theoretic use of braces. The notations $\langle \_ \rangle$ and $[\,\_\,]$ are intended to be reminiscent of modal operators.

Assertions and assumptions of predicates An assertion of a predicate U is an angelic update[9] where the relation does not change the state. (A 'stuttering' step, in Lamport's terminology.) The relation which constrains the angel's choice is the reflexive relation $(U \restriction \textsc{eq})$. The angel has therefore merely to establish that U holds – there is no choice for the next state. Dual to assertions of predicates, we also have assumptions, where the burden of establishing the predicate is instead on the demon. The generalisation of assertion and assumption from predicates to general relations is clear.

[9]The angel is anyone on the test team, who has to ensure that a step takes place which is described by the relation.

Commuting with ... Angelic update, being 'existential', commutes with arbitrary unions (is strict and defined by its action on singletons). (This is a kind of degeneracy.) On the other hand demonic update, being 'universal', commutes with arbitrary intersections. (Also degenerate.)

Interestingly, although demonic update does not in general commute with unions, it is continuous (*i.e.* commutes with unions of *directed* families) under the restriction that the relation is total and image-finite (finite but non-empty range per element).

Continuity is an interesting property of predicate transformers, which seems be close to the idea of *implementability* for interactive programs. Back and von Wright.

To restrict ourselves to continuous predicate transformers, we have only to restrict demonic updates to such relations, and abstain from infinite $\sqcap$.

Accessibility and its dual The following fundamental properties of predicates with respect to a relation can be defined as pre-fixed or post-fixed points of these update commands.

$$\begin{array}{ll} U \text{ is } \phi\text{-progressive} & [\,\phi\,]U \subseteq U \\ U \text{ is } \phi\text{-invariant} & U \subseteq \langle\phi\rangle U \end{array}$$

The predicate transformer that assigns to a predicate U the least (*i.e.* strongest) ϕ-progressive predicate $\mathcal{A}(U)$ that includes U, which is of course a closure operator, is called here the *accessibility* transformer. When U is empty, the accessible elements are also called well-founded. Dually, the predicate transformer that assigns to a predicate U the greatest (*i.e.* weakest) ϕ-invariant predicate $\mathcal{I}(U)$ that is still contained in U, which is of course an interior operator is (here) called the *safety* transformer. It picks out from a set of states those elements from which there is an infinite stream of ϕ-transitions entirely in that set. When U is the trivial predicate which holds universally, the safe states are non-wellfounded, in a 'positive' sense.

Relational composition and division obtained from updates Sequential composition and division are closely connected to angelic and demonic update respectively, *via* inversion:

$$\begin{array}{ll} (R_1\,;R_2) & = (\langle R_1\rangle \cdot (R_2{}^\sim))^\sim \\ (R_1\,;R_2)a & = \langle R_2{}^\sim\rangle(R_1 a) \\ (R_1/R_2)a & = [\,R_2\,](R_1 a) \\ (R_1 \setminus R_2) & = ([\,R_1{}^\sim\,] \cdot (R_2{}^\sim))^\sim \end{array}$$

In pointfree form:

$$\begin{array}{l} (R_1\,;) = (^\sim) \cdot (\langle R_1\rangle\cdot) \cdot (^\sim) \\ (\,;R_2) = (\langle R_2{}^\sim\rangle\cdot) \\ (/R_2) = ([\,R_2\,]\cdot) \\ (R_1\backslash) = (^\sim) \cdot ([\,R_1{}^\sim\,]\cdot) \cdot (^\sim) \end{array}$$

Verification of the equations involving angelic update is straightforward,

starting from the following equations.

$$\begin{array}{ll} (R_1\,;R_2)a & = \{\, c \mid R_2{}^{\sim}(c) \between R_1(a) \,\} \\ & = \langle R_2{}^{\sim}\rangle(R_1 a) \\ (R_1\,;R_2)^{\sim}c & = \{\, a \mid R_1(a) \between R_2{}^{\sim}(c) \,\} \\ & = \langle R_1\rangle(R_2{}^{\sim}c) \end{array}$$

The match-up with Giovanni's[10] notations seems to be as follows:

Giovanni		Back and von Wright	
r^-	(*existential anti-image*)	$\langle r \rangle$	(*angelic update, assertion*)
r^*	(*universal anti-image*)	$[\, r \,]$	(*demonic update, assumption*)
r	(*existential image*)	$\langle r^{\sim} \rangle$	
r^{-*}	(*universal image*)	$[\, r^{\sim} \,]$	

The two notation styles do not harmonize nicely[11]. (It does however seem to be safe to steal from Giovanni and write just $R(U)$ for $\langle R^{\sim}\rangle\, U$, and hence $R^{\sim}(U)$ for $\langle R\rangle\, U$; and this is a large part of the charm of Giovanni's notation: the overloading of the notation for application to suit the common case.)

Our category has for its objects triples:

$$\langle S, \Phi : S \to \mathbb{F}(\mathbb{F}(S)), s_0 : S \rangle$$

which we call *interaction systems*, or (tendentiously) *interfaces*.

Next we turn to its morphisms.

4 Morphisms: components

What is the appropriate notion of morphism between interaction systems? It should be a "partial" implementation of its exported interface, that can make use of an implementation of its imported interface. We analyse this notion as follows:

- first we define an elementary simulation to be a relation between states equipped with a certain "back and forth" mapping which translates export-commands to import commands and import responses to export responses in such a way that the illusion of an implementation of the domain interface can be perpetually maintained. Elementary simulations also relate the initial states.
- It turns out that the operation of transitive and reflexive closure is (among other interesting operations) a monad in the category of interaction systems and elementary simulations.
- It may be that a command in the upper-level interface requires zero, one or more interactions with the lower-level interface before a high-level

[10] beastly, horrid, ...

[11] To put it mildly. The discord between of * and the usual notation for reflexive and transitive closure is ear-splitting.

response is ready. We model this by moving to the Kleisli category for the monad $\_^*$. This is our 'real' category. A (general) simulation[12] of interface A by interface $\langle B, \Phi : B \to \mathbb{F}(\mathbb{F}(B)), b_0\rangle$ is an elementary simulation of A by the interaction system $\langle B, \Phi^*, b_0\rangle$

– we define a poset structure between simulations. This again makes use of the transitive and reflexive closure, this time to close the set of low-level states that can simulate a high-level state inductively – this closure operation is called saturation. We treat each simulation as equal to its saturation. We place on simulations the associated partial order, *i.e.* extensional inclusion of saturations). This order is coarser than extensional inclusion. Simulations form a complete lattice with respect to this partial order.

Definition Let $\Phi : A \to \mathbb{F}(\mathbb{F}(A))$ and $\Psi : B \to \mathbb{F}(\mathbb{F}(B))$ be two interaction structures. An elementary simulation of Φ by Ψ is a relation $Q : A \to \mathbb{P}(B)$ which satisfies

$$Q(a) \subseteq \bigcap_{c:C_\Phi(a)} \Psi(\bigcup_{r:R_\Phi(a,c)} Q(a[c/r]_\Phi))$$

We now tease apart what this says. The relation Q should be such that for all a and b,

$$\begin{aligned} Q(a,b) \to \\ &(\forall\, c : C_\Phi(a)) \\ &(\exists\, c' : C_\Psi(b)) \\ &(\forall\, r' : R_\Psi(b,c')) \\ &(\exists\, r : R_\Phi(b,r')) \\ &\qquad Q(a[c/r]_\Phi, b[c'/r']_\Psi) \end{aligned}$$

This can, by using the axiom of choice, be pulled into a yet more "computational" form: there should exist, for all pairs of states $a : A$ and $b : B$ such that $Q(a,b)$, functions

$$\begin{aligned} &\gamma : C_\Phi(a) \to C_\Psi(b) \\ &\delta : (c : C_\Phi(a)) \to R_\Psi(a, \gamma(c)) \to R_\Phi(b,c) \end{aligned}$$

such that

$$(\forall\, c : C_\Phi(a), r : R_\Psi(\gamma(c)))\ Q(a[c/\delta(c,r)]_\Phi, b[\gamma(c)/r]_\Psi)$$

If we have a simulation relation, then the proof that it is a simulation can be used as a program to translate Φ-commands to Ψ-commands, and to translate the Ψ-responses so evoked to Φ-responses to the original command.

Properties of simulation There are a number of key properties of this notion of simulation. The following shows that a simulation enjoys a "sub-commutativity" property (Lamport has the terminology 'right-moving') in analogy with the notion of simulation between relations.

$$\langle Q^\sim \rangle \,;\Phi \sqsubseteq \Psi \,;\langle Q^\sim \rangle$$

[12] This was called a refinement in my notes with Pierre.

The following (which amounts to the same thing) states that a simulation preserves the covering structure of an interface.

$$U \subseteq \Phi(V) \to \langle Q^{\sim}\rangle(U) \subseteq \Psi(\langle Q^{\sim}\rangle(V))$$
(*which looks better in Giovanni-speak:*)
$$U \lhd_{\Phi} V \to Q(U) \lhd_{\Psi} Q(V)$$

We compare elementary simulations by relational inclusion and extensional equality.

Monads for elementary simulation It then follows that the operations of reflexive and transitive closure $\_^*$, reflexive closure $\_^?$, and transitive closure $\_^+$ are all monads in the category of interaction structions and elementary simulations. These correspond to different kinds of "flexibility" we can add to elementary simulations.

General simulations So much for elementary-simulations between interaction structures (from and to the same state-space). We now define a simulation between interaction *systems* (with an initial state) to be an elementary simulation of the exported interface by the *reflexive and transitive closure* of the imported interface (allowing zero, one or more calls to the imported interface). We should also have $Q(a_0, b_0)$.

That is, we take for our monad the reflexive and transitive closure, which offers maximum flexibity. Furthermore, since we are dealing with structures with designated initial states that we require to be related, we do not have empty simulations.

Their ordering However the partial order we put on simulations is weaker (more coarse) than relational inclusion. We define $Q \sqsubseteq Q'$ (between simulations by a saturated interface $\Psi = \Psi^*$) to mean

$$(\forall\, a : A)\ \ \Psi^*(Q(a)) \subseteq \Psi^*(Q'(a))$$

In Giovanni's terminology, we replace relations by their saturations. We use the monad to define the partial order.

5 Examples of interfaces

The following sketched examples concern memory interfaces. These are in a sense the simplest non-trivial (history sensitive) interfaces. They are extremely well-behaved. They can be 'idempotent' in a certain sense: meaning that repetition of writes (or replaying sequences of writes) is harmless. Memory interfaces are 'universal', in the sense that an arbitrary state machine can be implemented by storing its state in a memory.

Sketch of :

- a simple memory cell.
- a memory cell with 'bad' writes.

– an addressible array of (simple) memory cells.
– an addressible array of memory cells with atomic update.
– a 'unix' memory array, with interruptible operations on sets of memory cells.

Coming attractions (maybe): memory cells with write timestamps; to make a change from memory cells, an arbitrary (Mealy) state machine. There are some interesting examples of cached memory in in [4]. They are presented in a form not too far from the below.

Although there are examples of interfaces, there ought also to be examples of simulations.

5.1 Conventions for interfaces

My habit is to use constructors to represent 'opcodes' of a command set. The operands are the input arguments for that opcode. If C is an opcode, then I use $\overline{C}$ as a constructor for replies to an instruction with opcode C. (This is a slight notational gesticulation towards CCS.)

In certain cases, one of the components of the input or output operands is a proof, typically of a decidable (*i.e.* boolean) statement. We may suppose that no computational use is made of such an operand: it is a sheer 'certificate', that can be quoted in the preparation of other such certificates. We know perfectly well what the form of a proof of a boolean statement is.

5.2 Partial functions

Many examples require some machinery for partial functions. Partial functions are relations which are single-valued on their domain.

When there are several memory cells, it may be that one has access to certains sets of addresses simultaneously. We usually have a total order on the addresses and may have access to all addresses in the union of certain sets of disjoint intervals simultaneously.

We have not only a set A of addresses but also a family I of (decidable) subsets of A. I should be closed under intersection and symmetric ?? difference[13]. If a set of addresses is in I, then the locations with those addresses can be accessed simultaneously.

In the following I write $A \mathrel{+\!\!\!\rightarrow} V$ for some implementation of the partial functions from A into V, where the domain of the partial function is in I. Equality on V is written $\sim$).

One may for example define $A \mathrel{+\!\!\!\rightarrow} V$ to be $A \to V^+$, where V^+ is the set

$$\{\,\mathrm{Unknown}\,\} \cup \{\,\mathrm{Known}\, v \mid v : V\,\}$$

Each such function $p : A \mathrel{+\!\!\!\rightarrow} V$ has a domain $\mathrm{dom}(p)$ which is a decidable subset of A. Between partial functions $p, q : A \mathrel{+\!\!\!\rightarrow} V$ we have an operation of

[13] *i.e.* I should be a ring of subsets of A.

overwriting defined by

$(p \text{ overwritten by } q) \triangleq [a \in \mathrm{dom}(p) \cup \mathrm{dom}(q) \mapsto \textbf{if } a \in \mathrm{dom}(q) \textbf{ then } q[a] \textbf{ else } p[a]]$

I shall use $\sim$ for equality between partial functions.

5.3 Memory cell

First we give an example of a simple memory cell. The part that says that we have a memory cell is the singleton output set for a read command (defined here to have the stored value as its only element).

states: $s : V$. V is the set of storable values. Let $\sim_V$ denote equality on V. It is reasonable to take this to be decidable.

commands: The set of commands is the same in every state

$$C(s) = \{ \mathrm{Read} \} \cup \{ \mathrm{Write}\ v \mid v : V \}$$

responses to Read: In any state, there are the following (correct) responses

$$R(s, \mathrm{Read}) = \{ \overline{\mathrm{Read}}\ v\ x \mid v : V, x : v \sim_V s \}$$
$$s[\mathrm{Read}/\_] = s$$

responses to (Write v): There is only one reply:

$$R(s, \mathrm{Write}\ v) = \{ \overline{\mathrm{Write}} \}$$
$$s[\mathrm{Write}\ v/\_] = v$$

5.4 Faulty memory cell

First we give an example of a faulty memory cell. A bad write can 'trash' the stored value.

states: $\{ \mathrm{Bad} \} \cup \{ \mathrm{Ok}\ v \mid v : V \}$.

commands: The set of commands is the same in every state

$$C(s) = \{ \mathrm{Read} \} \cup \{ \mathrm{Write}\ v \mid v : V \}$$

Note that a read is always legal.

responses to Read: In any state, there is at most one (correct) response

$$R(s, \mathrm{Read}) = \{ \overline{\mathrm{Read}}\ v\ x \mid v : V, x : s \sim_V \mathrm{Ok}\ v \}$$
$$s[\mathrm{Read}/\_] = s$$

Note that read can return only if the last write was good; otherwise a read may hang.

responses to (Write v): There are two possible replies.

$$R(s, \mathrm{Write}\ v) = \{ \overline{\mathrm{Write}} \} \cup \{ \overline{\mathrm{BadWrite}} \}$$
$$s[\mathrm{Write}\ v/\overline{\mathrm{Write}}] = \mathrm{Ok}\ v$$
$$s[\mathrm{Write}\ v/\overline{\mathrm{BadWrite}}] = \mathrm{Bad}$$

5.5 Array of memory cells

Next we consider an array of memory cells which can be individually read or written. (Basically we use $\sqcap$ here.)

Let A be the set of addresses: it too should have decidable equality.

states: $s : A \to V$

commands: The set of commands is the same in every state

$$C(s) = \{\, \text{Read } a \mid a : A \,\} \cup \{\, \text{Write } a\ v \mid a : A, v : V \,\}$$

responses to (Read a): In any state, there is only one (correct) response

$$R(s, \text{Read } a) = \{\, \overline{\text{Read}}\ v\ x \mid v : V, x : s(a) \sim_V v \,\}$$
$$s[\text{Read } a/\_] = s$$

responses to (Write $a\ v$): There is only one reply:

$$R(s, \text{Write } a\ v) = \{\, \overline{\text{Write}} \,\}$$
$$s[\text{Write } a\ v/\_] = (a := v)s$$

Here $a := v$ stands for the operation which overwrites a memory-state (whose domain must be equipped with a decidable equality $\sim_A$) with a new value v at a given argument a.

5.6 Atomic array of memory cells

Next we consider an array of memory cells that can be simultaneously (atomically) read and written. For example, we can write a cell and return its immediately prior value.

Let I be a family (a ring) of decidable subsets of A, closed under finite unions and relative complement (for example finite sets of disjoint intervals in a linear order). We intend that if a set of addresses is in I, the corresponding locations may be accessed simultaneously.

states: $s : A \to V$.

commands: The set of commands is the same in every state

$$C(s) = \{\, \text{RW } w\ p \mid w : I, p : A \rightharpoonup V \,\}$$

responses to (RW $w\ p$):

$$R(s, \text{RW } w\ p) = \{\, \overline{\text{RW}}\ p'\ x \mid p' : A \rightharpoonup V, x : \text{dom}(p') = w \wedge w \restriction s \sim_V p' \,\}$$
$$s[\text{RW } w\ p/\_] = s \text{ overwritten by } p$$

5.7 Unix memory array

This example is an abstraction of the unix system calls 'read' and 'write' to read and write portions of files. These are interruptible, and so may terminate 'incompletely' or abnormally, having read or written only part of what was asked for. Several cells can (with luck) be simultaneously read, or simultaneously written. If they are incomplete, writes "trash" the area of uncertainty, so that we are not even allowed to ask to read a trashed cell until it has been satisfactorily re-written. (Unix does not have such an extreme notion of 'trashing': trashed cells may safely be read, though the values are constrained only to be sequences of bytes.)

states: $A \pfun V$

commands: The set of commands is the same in every state

$$C(s) = \{\, \text{Read } w \mid w : I \,\} \cup \{\, \text{Write } p \mid p : A \pfun V \,\}$$

responses to (Read w):

$$R(s, \text{Read } w) = \{\, \overline{\text{Read}}\ p\ x \mid p : A \pfun V, x : \mathrm{dom}(p) \subseteq w \wedge (\mathrm{dom}(p) \restriction s) \sim p \,\}$$
$$s[\text{Read } w/\_] = s$$

Note: unknown cells must be written before being read.

responses to (Write p):

$$R(s, \text{Write } p) = \{\, \overline{\text{Write}}\ w \mid w \subseteq \mathrm{dom}(p) \,\}$$
$$s[\text{Write } p/\overline{\text{Write}}\ w] = \ ((\mathrm{dom}(s) \setminus \mathrm{dom}(p)) \restriction s) \text{ overwritten by } (w \restriction p)$$

Note that it is perfectly legitimate for an implementation (the demon) to trash any write request. To rule this out, we might give a fairness guarantee, *e.g.* that persistent requests to write any specific cell will eventually succeed.

6 Loose ends

Saturation and hiding Undiscussed: the point of passing to the saturation.

Finitely many 'silent' interactions may occur in the lower-level interface, and they are in a sense 'hidden' in a single response. We want to allow this kind of 'silencing'.

Saturation and universal properties Connected with simulation: since the equality induced with $\_^*$ is coarser than extensional equality, we should have more universal objects, more 'uniqueness' modulo saturation.

Unresolved: is SKIP initial? Weakly initial? There may be many simulations, but are any distinguished in some way? What is the appropriate notion of universality in an enriched category? Generally, what limits and colimits do we have? We have a supremum operation for saturated simulations.

Pierre Hyvernat has established that predicate transformers and simulations provide a model for full linear logic, in which a linear formula is interpreted by a predicate transformer, and a proof of it by a post-fixed point. The linear implication is in effect the relation transformer whose post-fixed points are simulation relations. Investigations are continuing.

Continuity of ;? Unresolved: we know that simulations are closed under sequential composition, which is monotone in both arguments (with respect to extensional inclusion).

What is known about sequential composition and the lattice structure appears to be this:

$$\begin{array}{l}(\sqcap_{i:I}\Phi_i)\,;\Psi = \sqcap_{i:I}(\Phi_i\,;\Psi)\\ (\sqcup_{i:I}\Phi_i)\,;\Psi = \sqcup_{i:I}(\Phi_i\,;\Psi)\\ \Phi\,;(\sqcap_{i:I}\Psi_i) \sqsubseteq \sqcap_{i:I}(\Phi\,;\Psi_i)\\ \Phi\,;(\sqcup_{i:I}\Psi_i) \sqsupseteq \sqcup_{i:I}(\Phi\,;\Psi_i)\end{array}$$

The last two inequations (which are just expressions of monotonicity) can be strengthened to equalities for certain classes of Φ.

But does sequential composition, perhaps, commute (in the right-hand argument) with suprema in the saturation order? Suprema in the saturation order are not unions, but closures of unions.

In general the role of saturated simulations needs to be clarified. It may have something to do with continuity: commuting with suprema of directed families.

According to Back and von Wright sequential composition preserves continuity in both arguments.

Wipe out all differences below skip The topic here is (perhaps) an embarrassment. Where ABORT is the predicate transformer than maps all predicates to the empty predicate (and MAGIC is the predicate transformer that maps all predicates to the trivial predicate) we have that ABORT is isomorphic to SKIP. (This is because they have the same reflexive and transitive closure, namely SKIP.) We abstract from the difference, because as an import interface, both ABORT and SKIP are equally useful (*i.e.* entirely useless).

Division Unresolved: is there a division operator for interaction structures analogous to that for relations?

Inversion Undiscussed: inversion - this concerns switching imported and exported interfaces. Servers and clients exchange roles. This involves $\_^\circ$ and $\_^\bullet$. There is certainly an operator $\Phi^- : A \to \mathbb{F}(\mathbb{F}(B))$ for $\Phi : A \to \mathbb{F}(\mathbb{F}(B))$ (note: there is no contravariance, in contrast with relational inversion) such that $\Phi^\bullet = \Phi^{-\circ}$.

$$\begin{array}{lcl}C^-(s) & \triangleq & (\forall\, c : C(s))\;\; R(s,c)\\ R^-(s,f) & \triangleq & C(s)\\ s[f/c]^- & \triangleq & s[c/f(c)]\end{array}$$

This inversion does not seem to be an involution. The result of iterating it twice is:

$$\begin{array}{ll} C^{--}(s) & \triangleq\ C^{-}(s) \to C(s) \\ R^{--}(s,\_) & \triangleq\ C^{-}(s) \\ s[F/f]^{--} & \triangleq\ s[F(f)/f(F(f))] \end{array}$$

There is a more subtle kind of inversion, which uses a kind of 4-way handshake to invert the direction of commands and responses. This is found in practice.

The situation is that any interaction map $\Phi : A \to \mathbb{F}(\mathbb{F}(B))$ can be expressed in the form $\langle\phi\rangle;[\,\psi\,]$ where $\phi : A \to \mathbb{F}(A')$, $\psi : A' \to \mathbb{F}(B)$.

$$\begin{array}{ll} A' & \triangleq\ (\exists\, s : S)\ \ C(s) \\ T_\phi(s) & \triangleq\ C(s) \\ s[c] & \triangleq\ \langle s, c\rangle \\ T_\psi\langle s, c\rangle & \triangleq\ R(s,c) \\ \langle s, c\rangle[r] & \triangleq\ s[c/r] \end{array}$$

We have $\Phi^{-} = [\,\phi\,];\langle\psi\rangle$. The 4-way inversion opens this up, forming $\langle\psi\rangle\,;[\,\phi\,]$ with some suitable initialisation and finalisation. (A proper description is lacking.)

Fish and foul Undiscussed: there are actually two dual closure operators (or 3 dual pairs if we consider analogues of $\_^{+}$ and $\_^{?}$):

$$\begin{array}{l} \Phi^{*} = (\mu\Psi : S \to \mathbb{F}(\mathbb{F}(S)))\ \textsc{skip} \sqcup (\Phi\,;\Psi) \\ \Phi^{\infty} = (\nu\Psi : S \to \mathbb{F}(\mathbb{F}(S)))\ \textsc{skip} \sqcap (\Phi^{-}\,;\Psi) \end{array}$$

The operator $\_^{\infty}$[14] is directly related to Giovanni's 'fish' relation $\ltimes$.

G-notation		H-notation
$s \ltimes_\Phi U$	means	$s \,\epsilon\, \Phi^{\infty}(U)$.
$s \lhd_\Phi U$	means	$s \,\epsilon\, \Phi^{*}(U)$.
$s \,\epsilon_\Phi\, U$	means	$s \,\epsilon\, \Phi(U)$.

We have

$$\begin{array}{ll} \Phi^{*}(P : \mathbb{P}(S)) & = \bigcap\{\, Q : \mathbb{P}(S) \mid\ P \cup \Phi(Q) \subseteq Q \,\} \\ \Phi^{\infty}(P : \mathbb{P}(S)) & = \bigcup\{\, Q : \mathbb{P}(S) \mid\ Q \subseteq P \cap \Phi^{-}(Q) \,\} \end{array}$$

Here we have an ambiguity in the adjective 'complete' in 'complete lattice'. We mean: complete for set indexed families – not necessarily for predicates. The expressions above use a second order predicate (a quantifier) in the body of the comprehension term to pick out the sets in the intersection or union. By allowing ourselves the operator $\_^{*}$, we commit ourselves to saying that the second order property (being a pre-fixedpoint of a certain operator, or a post-fixedpoint of another) can be in some sense set-indexed.

By appeal to transfinite induction, we turn the impredicative intersection into a union of a transfinitely indexed sequence of progressively weaker predicates. One can think of the indexing elements as predecessors of a certain ordinal

[14] The ν binding needs a general analysis of greatest fixed points and corecursion.

(the closure ordinal), and hence as of the indexing as having for its domain a genuine set.

By allowing ourselves the operator $\_^\infty$, we seem to turn the impredicative union into an intersection of an ω-sequence of progressively stronger predicates. The contrast between a transfinitely ordered union in the inductive case, and the ω-ordered intersection in the coinductive case is very striking.

win and sin The two operators fish $(\ltimes\_)$ (aka $\_^\infty$) and (what else?) fowl $(\lhd\_)$ (aka $\_^*$) appear to be closely connected with the predicate transformers *win*, and *sin* introduced by Lamport[3] which give the weakest respectively strongest invariant[15] stronger than respectively weaker than a given predicate. These predicate transformers have been used to analyse concurrent algorithms such as the 'bakery' algorithm that do not require a given grain of atomicity for access to shared variables. The exact relationships need to be clarified.

Lamport's *win* and *sin* operators take relations to predicate transformers. For fixed relation ϕ, we have the following.

$$win_\phi = \langle\phi\rangle^\infty \quad (1)$$

$$sin_\phi = [\,\phi\,]^* \quad (2)$$

The operators $\_^*$ and $\_^\infty$ seem to generalise Lamport's operators from relations to predicate transformers. The novelty here is the use of inversion in the definition of Φ^∞.

Points Undiscussed: the counterpart of topological points. The interactive counterpart of the notion of a point seems to be that of an inhabited predicate α which is indefinitely extensible, sustainable, or refinable in a certain sense. We can express this impredicatively, quantifying over predicates U. It is a closure property on α.

$$\alpha \text{ is inhabited, and forall } U,\ \alpha \between (\lhd U) \to \alpha \between U$$

Note that in formal topology, one usually has a relation $\leq$ on the states (which must be a self-simulation[16] with respect to the interaction structure), and a point is further required to be a filter with respect to this relation. For example any inhabited predicate of the form $(\ltimes V)$ (a closed set according to Giovanni, formally an open set) has this closure property.

Such an α ought (perhaps) to be (roughly) the same thing as a simulation of the interface by SKIP (or equivalently, ABORT). A bit tendentiously, we call a simulation of an interface by SKIP an *implementation* of the interface. This amounts to a predicate α over the interface states which satisfies the above closure property.

The proof that α is inhabited is the 'active ingredient'. It can be used as a deadlock free or non-stop server program (a perpetuum mobile).

[15] One can take an invariant to be a predicate which is true initially and stable (*i.e.* preserved under interaction).

[16] Is this true? Palmgren has the extra properties in the form $(\lhd U) \cap (\lhd V) \subseteq U_\leq \cap V_\leq$ (where $U_\leq$ is the $\leq$-downward closure of U), and $(\leq b) \subseteq (\lhd\{\,b\,\})$.

Specifications, client-server contracts Undiscussed: There are two kinds of specification: a server program satisfies a specification[17] of the form

$$(\exists\, s_0 : S)\ \ \mathrm{Init}(s_0) \wedge s_0 \ltimes \mathrm{Inv}$$
$$\textit{i.e.}\quad \mathrm{Init} \between (\ltimes \mathrm{Inv}) \quad \text{written} \quad \mathrm{Init} \ltimes \mathrm{Inv}$$

whereas a client program (which terminates accomplishing a goal predicate – an atomic transaction) satisfies a specification[18] of the form

$$(\forall\, s : S)\ \ \mathrm{Init}(s) \rightarrow s \lhd \mathrm{Goal}$$
$$\textit{i.e.}\quad \mathrm{Init} \subseteq (\lhd \mathrm{Goal}) \quad \text{written} \quad \mathrm{Init} \lhd \mathrm{Goal}$$

The logic of the interaction between clients and servers is contained in the following rule.

$$\frac{\mathrm{Init} \lhd \mathrm{Goal} \qquad \mathrm{Init} \ltimes \mathrm{Inv}}{\mathrm{Goal} \ltimes \mathrm{Inv}}$$ Giovanni will have a name: compatibility?

A proof of the left hand premise is a program for a client. A proof of the right hand premise is a program for a server. The proof that we get of the conclusion (having run the client program against the server program, or having 'made the inference') is a new server program, after the goal of the client program has been accomplished. (It has the same invariant, but a new initial predicate equal to the client's goal.) (The client program is all used up.)

References

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

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

[3] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on programming languages and systems*, 12(3), June 1990.

[4] Butler W. Lampson. Implementing coherent memory. In A.W.Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R.Hoare*, pages 259–275. Prentice Hall, 1994.

[5] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *LNCS*, volume 389. Springer-Verlag, 1989.

7 syntax

The usual setting for Back and von Wright's refinement calculus is higher order classical logic, with quantification over predicate variables, and a

[17] As it were, written on the box or packaging in which the server program comes

[18] Which tells you when you can use it, and what you can use it to bring about

complement operator. Here instead we define certain of these constructions in predicative type theory in which quantification over predicates is not allowed in propositions.

The basic judgements in which we are interested are (primarily) $U \subseteq V$ and (secondarily, the dual judgement) $U \between V$ which says that U and V are compatible. It is necessary to have a separate judgement form such as $U \between V$ in the absence of the complement operator. Here U and V are predicate expressions that may contain variables of various sorts: states, state-predicates, and relations. The variables are implicitly universally quantified. If the predicate expressions depend on a single predicate variable X, we have a (pointwise) relation between predicate transformers. If the predicate expressions depend on a state variable s, we have the inclusion relation between relations. In combination with predicate transformers Φ and iterated form, the basic relations give rise to relations of interest in client-server programming such as the following

$$U \lhd_\Phi V = U \subseteq \Phi^*(V)$$
$$U \ltimes_\Phi V = U \between \Phi^\infty(V)$$

As for predicativity, all reasoning should be essentially 'point-free', or algebraic. That is to say, proofs should be algebraic manipulations in which free subset variables never officially appear.

In the general case we may have beside predicates also relations with various arities, and beside unary transformers of unary predicates also transformers with arity of the form $\langle n_1, .., n_k \rangle \to n$

Two base types (for the two kinds of variables): S (states – s, s', s_1, ...)) and P (predicates – P). We have one binary relation $s \epsilon P$, meaning that state s satisfies predicate P. This gives rise to 3 kinds of statement:

$s \epsilon U$	
$s \epsilon Q(s')$	Q a relation
$s \epsilon F(U)$	F a predicate transformer

The following higher types:

$f, g, \ldots$	$: S \to S$	STATE TRANSFORMER	$f = g$
$R, Q, \ldots$	$: S \to P$	STATE RELATIONS	$R \subseteq Q$
$F, G, \ldots$	$: P \to P$	PREDICATE TRANSFORMERS	$F \sqsubseteq G$

The first kind of comparison is equality between state expressions that may have free state-variables. Might want apartness.

The second kind of comparison is equivalence and implication between statements that may have free state-variables. Might want overlap.

The third kind of comparison is equivalence and implication between statements that have free occurences of both state-variables and predicate-variables. Again, might want overlap.

8 predicates and families

Predicates over a set form a distributive lattice: closed under sup (empty, binary, set-indexed) and inf (empty, binary, set-indexed). By distributivity I mean that binary sups distribute over binary infs and vice-versa. We also have implication, forms of relative complement etc. Binary infs distribute over arbitrary sups. We also have singleton predicates.

Families on the other-hand are merely a sup-lattice with singletons. .

An obscure point to be explained: the link between families and predicates (or transition structures and relations) involves not just equality (which gets us one way), but also existential quantification over states, which gets us a family from a predicate, in which the function is first projection. There is a question of size here: S may be 'large' compared to the universe of sets we are working in.

9 grammar

states	s, s'	$::=$	$f(s)$
state transformers	f, g	$::=$	$id \mid f \cdot g$
families	α, β	$::=$	$\{ s \} \mid \sqcup_i \alpha_i \mid \phi(s)$
predicates	U, V	$::=$	$\alpha \mid \bigcup_i U_i \mid \bigcap_i U_i \mid \Phi(U) \mid R(s)$
transition structures	ϕ, ψ	$::=$	$\mathbf{graph}\, f$ $\mid U \rightharpoonup \phi$ $\mid \sqcup_i \phi_i$ $\mid \phi \,;\, \psi \mid \textsc{skip}$ $\mid \phi^* \mid \phi^+ \mid \phi^?$
relations	Q, R	$::=$	ϕ $\mid U \rightharpoonup Q$ $\mid \bigcup_i Q_i \mid \bigcap_i Q_i$ $\mid R \,;\, Q \mid \textsc{skip} \mid Q / R$ $\mid \phi \,;\, Q \mid Q / \phi$ $\mid \Phi \cdot Q$ $\mid Q^* \mid Q^+ \mid Q^?$ $\mid Q^\sim$
interaction structures	Φ, Ψ	$::=$	$\mid \langle \phi \rangle \mid [\, \phi \,]$ $\mid \sqcup_i \Phi_i \mid \sqcap_i \Phi_i$ $\mid \Phi \,;\, \Psi \mid \textsc{skip}$ $\mid \mathbf{assign}\, f$ $\mid \Phi^* \mid \Phi^+ \mid \Phi^?$ $\mid \Phi^\infty$ $\mid \Phi^\perp$
predicate transformers	F, G	$::=$	Φ $\mid \langle R \rangle \mid [\, R \,]$ $\mid \sqcup_i F_i \mid \sqcap_i F_i$ $\mid F \,;\, G \mid \textsc{skip}$ $\mid \mathbf{assign}\, f$ $\mid F^* \mid F^+ \mid F^?$ $\mid F^\infty$

10 types and definitions

10.1 State transformers

1. composition of state transformers $\dfrac{f : A \to B \qquad g : B \to C}{g \cdot f : A \to C}$

   $(f.g)(s) \triangleq f(g(s))$.

2. identity of state transformers: $\textsc{skip} : A \to A$

   $\textsc{skip}(a) \triangleq a$.

3. base state transformers

   Often, the state space is a product $S = \Pi_{v:V} S_v$, where V is a set of variable-names (with decidable equality) and S_v is a factor of the state space corresponding to variable v: the type of v. Then a state is a function which assigns to a variable-name v a value of the type S_v appropriate to v. This can also be thought of as a record with field-names or coordinates V, with field $v : V$ having type S_v.

   If e is an expression built up from variable-names using function constants, we define by recursion on its build up the value $|e|_s$ of e in the start state s – in the obvious way (see below). We then define the update function $(v := e) : S \to S$.

   $$(v := e) : S \to S$$
   $$(v := e)(s) \triangleq (\lambda\, v' : V)\ \textbf{if}\ v' = v\ \textbf{then}\ |e|_s\ \textbf{else}\ s(v)$$

   Thus an *assignment* statement can be interpreted as a state transformer.

   We may obviously extend the definition to *simultaneous* assignment, where we have a finite partial map from variable names to expressions. We can represent this in the usual way with a vector $\vec{v}$ of distinct variables, and a vector $\vec{e}$ the same length, giving for each variable the corresponding expression.

   $$x_1, \ldots, x_n := e_1, \ldots, e_n$$

   Example : $x, y := y, x$ – atomically swap contents of variables x and y.

   The syntax of an assignment statement is essentially a function from names of assignable variables to expressions built over those and other 'read-only' variables. The semantics of an assignment statement is a state-transformer. To define the semantics of an assignment statement, all that seems to be necessary is that the domain of the syntax-function should be decidable.

   Tangentially, if we are interested in (statically) *typed* variables, then one can represent the type system in the form of an interaction structure – a set of sorts or types, and for each sort σ a family of families of sorts:

   $$\{\, \{\, \sigma[c/r] \mid r : R(\sigma, c) \,\} \mid c : C(\sigma) \,\}$$

Here $C(\sigma)$ describes the constructors that can be used to form an expression of a given sort σ, and for each such constructor $c : C(\sigma)z$ an element r of $R(\sigma, c)$ selects the location of an immediate subexpression. The sort of the subexpression must equal $\sigma[c/r]$.

An expression of sort σ is now a well-founded tree, in which there are 'leaves' labelled by state variables. (If there are none, the expression is closed – the tree has finished growing.) We might use some notation such as $\sigma \lhd \gamma$ for the set of expressions which have sort σ with respect to type assignment $\gamma : V \to S$.

We can define the value of an expression $e : \sigma \lhd \gamma$ by wellfounded recursion on the structure of e. (Assumes a meaning is given to the constructors.)

An assignment statement is well-typed with respect to a type-assignment γ if the sort assigned to a variable on the left hand side of the '`:=`' equals the sort assigned to the corresponding expression on the right.

We can now define the state transformer corresponding to a well-typed assignment statement.

10.2 Category of relations and simulations

Predicates have one parameter – what the predicate is about. Similarly families have one parameter – the index of the general term. By considering a further state-parameter, we pass from predicates to relations, and from families to transition structures. We thereby add to the lattice operations a 'sequencing' monoid. The unit of sequential composition is the identity relation, or in other words the graph of the identity function.

Survey: Closure properties. Other notes.

- graphs of functions are transition structures.
- transition structures are closed under restriction by predicates (guarding).
- transition structures are closed under sup.
- transition structures are closed under composition, eq *etc.*.
- in the homogeneous case, we have the usual closure operators (reflexive, transitive, etc).
- transition functions are *not* closed under converse, nor intersection, nor division. (That is, without specific use of the equality relation.)

Transition structures are 'regular' – form a sup-lattice (with set-indexed sups), and have an iteration star operation. (We get back the infs with interaction structures, and two notions of iteration.)

Transition structures form a Kleene (regular) algebra in the following sense. It has an associative and commutative binary sup $\cup$ with unit 0 (the empty transition structure); associative binary sequencing with unit SKIP, distributing over sup. 0 is absorbing. An iteration operator star, satisfying ϕ^* is a solution of the equations $\textsc{skip} \cup (\phi\,;x) \subseteq x$ and $\textsc{skip} \cup (x\,;\phi) \subseteq x$; and if $\phi\,;x \subseteq x$ or $x\,;\phi \subseteq x$, then $\phi^*\,;x \subseteq x$ or $x\,;\phi^* \subseteq x$ respectively. Transition structures also include 'tests' somewhat in Kozen's sense, except that they needn't form a Boolean algebra (but a Heyting algebra).

Each transition structure determines two relation transformers $(\phi\,;)$ and $(/\phi)$. The definition of these does not use equality. These are closely related to the predicate transformers $\langle\phi\rangle$ and $[\,\phi\,]$. We have $(\phi\,;) = (\tilde{}) \cdot (\langle\phi\rangle\cdot) \cdot (\tilde{})$ and $(/\phi) = ([\,\phi\,]\cdot)$.

10.2.1 Relations and transition structures

1. *st*'s as *rel*'s $\quad \dfrac{f : A \to B}{\mathbf{graph}\, f : A \to \mathbb{P}(B)}$

$$b \,\epsilon\, (\mathbf{graph}\, f)a \;\triangleq\; b = f(a) \tag{3}$$

2. *st*'s as *ts*'s $\quad \dfrac{f : A \to B}{\mathbf{graph}\, f : A \to \mathbb{F}(B)}$

$$T(a) \triangleq N_1 \quad ; \quad s[\_] \triangleq f(s) \tag{4}$$

3. predicates as *rel*'s $\dfrac{U : \mathbb{P}(A)}{\mathbf{test}\, U : A \to \mathbb{P}(A)}$

$$b \,\epsilon\, (\mathbf{test}\, U)a \triangleq a \,\epsilon\, U \wedge b = a \tag{5}$$

4. predicates as *ts*'s $\dfrac{U : \mathbb{P}(A)}{\mathbf{test}\, U : A \to \mathbb{F}(A)}$

$$T \triangleq U \quad ; \quad s[\_] \triangleq s \tag{6}$$

5. domain restriction of *rel*'s $\dfrac{Q : A \to \mathbb{P}(B) \qquad U : \mathbb{P}(A)}{U \rightharpoonup Q : A \to \mathbb{P}(B)}$

$$b \,\epsilon\, (U \rightharpoonup Q)a \triangleq a \,\epsilon\, U \wedge b \,\epsilon\, Q(a) \tag{7}$$

Note: some redundancy. $U \rightharpoonup R = \mathbf{test}\, U \,;\, R$.

6. domain restriction of *ts*'s $\dfrac{\phi : A \to \mathbb{F}(B) \qquad U : \mathbb{P}(A)}{U \rightharpoonup \phi : A \to \mathbb{F}(B)}$

$$T \triangleq U \cap T_\phi \quad ; \quad s[\langle \_, t \rangle] \triangleq s[t]_\phi \tag{8}$$

7. mapping *rel*'s by a st $\dfrac{R : A \to \mathbb{P}(B) \qquad f : C \to B}{\mathbb{P}(f) \cdot R : A \to \mathbb{P}(C)}$

$$c \,\epsilon\, (\mathbb{P}(f) \cdot R)a \triangleq f(c) \,\epsilon\, R(a) \tag{9}$$

8. mapping *ts*'s by a st $\dfrac{\phi : A \to \mathbb{F}(B) \qquad f : B \to C}{\mathbb{F}(f) \cdot \phi : A \to \mathbb{F}(C)}$

$$T(a) \triangleq T_\phi(a) \quad ; \quad a[t] \triangleq f(a[t]_\phi) \tag{10}$$

Note: redundant. $\mathbb{F}(f) \cdot \phi = \phi \,;\, \mathbf{graph}\, f$.

9. union, *and* intersection of *rel*'s

$$\frac{Q_i : A \to \mathbb{P}(B)}{(\cup_i Q_i), (\cap_i Q_i) : A \to \mathbb{P}(B)}$$

$$(\cap_i Q_i)(a) \triangleq \cap_i (Q_i(a)) \tag{11}$$
$$(\cup_i Q_i)(a) \triangleq \cup_i (Q_i(a)) \tag{12}$$

Note, no counterpart to intersection on *ts*'s.

10. union of *ts*'s $\dfrac{\phi_i : A \to \mathbb{F}(B)}{(\sqcup_i \phi_i) : A \to \mathbb{F}(B)}$

$$T(a) \triangleq (\exists_i)\ iT_{\phi_i}(a) \quad ; \quad a[\langle i, t\rangle] \triangleq a[t]_i \tag{13}$$

Note, no counterpart of intersection.

11. argument swapping

$$\frac{Q : A \to \mathbb{P}(B)}{Q^{\sim} : B \to \mathbb{P}(A)}$$

$$a \in Q^{\sim}(b) \triangleq b \in Q(a) \tag{14}$$

Notes

- no counterpart operation on *ts*'s
- prime example of a relation transformer. Contravariant functor on the category of sets and relations. An involution. Called converse, inverse, reverse, interchange, twist, flip, swap, and so on.
- determines a notion of $(^{\sim})$-duality for relation transformers. The $(^{\sim})$-dual of a relation transformer Φ is $(^{\sim}) \cdot \Phi \cdot (^{\sim})$. For example, the division $(Q\backslash)$ is $(^{\sim})$-dual to $(/(Q^{\sim}))$.

12. *rel*'s closed under sequential composition

$$\frac{R : A \to \mathbb{P}(B) \qquad Q : B \to \mathbb{P}(C)}{(R\,;Q) : A \to \mathbb{P}(C)}$$

$$c \,\epsilon\, (R\,;Q)(a) \;\triangleq\; R(a) \between Q^{\sim}(c) \tag{15}$$

13. identity *rel*: $\textsc{skip} = \mathbf{graph}\,\textsc{skip} : A \to \mathbb{P}(A)$

$$a \,\epsilon\, \textsc{skip}(a') \;\triangleq\; a = a' \tag{16}$$

14. *ts*'s closed under sequential composition

$$\frac{\phi : A \to \mathbb{F}(B) \qquad \psi : B \to \mathbb{F}(C)}{(\phi\,;\psi) : A \to \mathbb{F}(C)}$$

$$T(a) \;\triangleq\; (\exists_{\,)}\;\; t_1 : T_\phi(a) T_\psi(a[t_1]_\phi) \quad ; \quad a[\langle t_1, t_2\rangle] \;\triangleq\; (a[t_1]_\phi)[t_2]_\psi \tag{17}$$

15. identity *ts*: $\textsc{skip} = \mathbf{graph}\,\textsc{skip} : A \to \mathbb{F}(A)$

$$T(\_) \;\triangleq\; N_1 \quad ; \quad a[\_] \;\triangleq\; a \tag{18}$$

16. closures of *rel*'s $\dfrac{R : A \to \mathbb{P}(A)}{R^?, R^+, R^* : A \to \mathbb{P}(A)}$

$$\begin{array}{rcl} R^* & \triangleq & \cap\{\, T : A \to \mathbb{P}(A) \mid (\textsc{skip} \cup (R\,;T)) \subseteq T \,\} \\ & = & \cup\{\, (R\,;)^n \textsc{skip} \mid n = 0, 1, \ldots \} \end{array} \tag{19}$$

$R^? \triangleq R \cup \textsc{skip}, \qquad R^+ \triangleq R\,;R^*.$

17. closures of *ts*'s $\dfrac{\phi : A \to \mathbb{F}(A)}{\phi^*, \phi^?, \phi^+ : A \to \mathbb{F}(A)}$

ϕ^*:

$$\begin{array}{rcl} T & \triangleq & (\mu\, X : A \to \mathrm{Set}) \\ & & (\forall\, a : A) \\ & & \quad \{\, \mathrm{nil} \,\} \\ & & \quad \cup\{\, \mathrm{cons}(t_0, t') \mid t_0 : T_\phi(a), t' : X(a[t_0]_\phi) \,\} \\ & & \quad \subseteq X(a) \\ a[\mathrm{nil}] & \triangleq & a \\ a[\mathrm{cons}(t_0, t')] & \triangleq & (a[t_0]_\phi)[t'] \end{array} \tag{20}$$

$\phi^? \triangleq \phi \sqcup \textsc{skip}, \qquad \phi^+ \triangleq \phi\,;\phi^*.$

18. *rel*'s closed under post-division

$$\frac{Q : A \to \mathbb{P}(B) \qquad R : C \to \mathbb{P}(B)}{(Q/R) : A \to \mathbb{P}(C)}$$

$$c \,\epsilon\, (Q/R)(a) \;\triangleq\; R(c) \subseteq Q(a) \tag{21}$$

19. pre-composition of *ts*'s to *rel*'s

$$\frac{\phi : A \to \mathbb{F}(B) \qquad Q : B \to \mathbb{P}(C)}{(\phi\,;Q) : A \to \mathbb{P}(C)}$$

$$c \,\epsilon\, (\phi\,;Q)(a) \;\triangleq\; \phi(a) \between Q^{\sim}(c) \tag{22}$$

$(\phi\,;Q)(a) = \cup\{\, Q(a[t]_\phi) \mid t : T_\phi(a) \,\}$
Note that this lets us lift a *ts* ϕ to the *rel* $(\phi\,;\textsc{skip})$.

20. post-division of *rel*'s by *ts*'s

$$\frac{Q : A \to \mathbb{P}(B) \qquad \psi : C \to \mathbb{F}(B)}{(Q/\psi) : A \to \mathbb{P}(C)}$$

$$c \,\epsilon\, (Q/\psi)(a) \;\triangleq\; \psi(c) \subseteq Q(a) \tag{23}$$

Note that this gives us a 'reciprocal' lift of *ts* ϕ to *rel* $(\textsc{skip}/\phi)$. The reciprocal of a relation is completely different from its converse. For what relations are the converse and reciprocal the same?

21. *pt*'s as operations on *rel*'s $\dfrac{\Phi : \mathbb{P}(A) \to \mathbb{P}(B) \qquad Q : C \to \mathbb{P}(A)}{(\Phi \cdot Q) : C \to \mathbb{P}(B)}$

$$b \,\epsilon\, (\Phi \cdot R)(c) \;\triangleq\; b \,\epsilon\, \Phi(R(c)) \tag{24}$$

10.2.2 Morphisms between relations and transition structures

Consider the Kleisli category for the monadic functor $\mathbb{F}(\_)$. The arrows in this category (diagrams of shape $A \to \mathbb{F}(B)$, which we picture as vertical arrows) are called transition structures. A morphism between two such arrows $\phi : A \to \mathbb{F}(B)$ and $\psi : C \to \mathbb{F}(D)$ is a pair of horizontal relations $Q_1 : A \to \mathbb{P}(C)$ and $Q_2 : B \to \mathbb{P}(D)$ which form a "sub-commuting" square:

$$Q_1{}^{\sim}\,;\phi \subseteq \psi;Q_2{}^{\sim}$$

This is equivalent to

$$Q_1{}^\sim \subseteq (\psi; Q_2{}^\sim)/\phi$$

or even

$$\begin{array}{rcl} Q_1 \subseteq & (\sim) \cdot (/\phi) \cdot (\psi;) \cdot (\sim)\ Q_2 \\ = & (\sim) \cdot ([\,\phi\,]\cdot) \cdot ((\sim) \cdot (\langle\psi\rangle\cdot) \cdot (\sim)) \cdot (\sim)\ Q_2 \\ = & (\sim) \cdot ([\,\phi\,]\cdot) \cdot (\sim) \cdot (\langle\psi\rangle\cdot)\ Q_2 \end{array}$$

One composes morphisms between arrows "horizontally", as relations. (Pairs of relations are compared pointwise, for inclusion and equality.)

If instead of arrows we restrict ourselves to cycles (*i.e.* homogeneous transition structures, *i.e.* endomorphisms in the Kleisli category, the appropriate notion of morphism is a simulation.

10.3 Category of predicate transformers and simulations

Mumble mumble.

10.3.1 Predicate transformers (and interaction structures)

1. Relational update, lifting *rel*'s to *pt*'s.

$$\frac{R : A \to \mathbb{P}(B)}{\langle R\rangle, [\,R\,] : \mathbb{P}(B) \to \mathbb{P}(A)}$$

$$a \,\epsilon\, \langle R\rangle(U) \quad \triangleq \quad R(a) \between U \tag{25}$$
$$a \,\epsilon\, [\,R\,](U) \quad \triangleq \quad R(a) \subseteq U \tag{26}$$

2. angelic and demonic lifting of a *ts* to an *is*

$$\frac{\phi : A \to \mathbb{F}(B)}{\langle\phi\rangle, [\,\phi\,] : A \to \mathbb{F}((\mathbb{F}(B)))}$$

angel $\langle\phi\rangle$

$$\begin{aligned} C &\triangleq T_\phi \\ R(\_,\_) &\triangleq N_1 \\ a[t/\_] &\triangleq a[t]_\phi \end{aligned} \tag{27}$$

demon $[\,\phi\,]$

$$\begin{aligned} C(\_) &\triangleq N_1 \\ R(a,\_) &\triangleq T_\phi(a) \\ a[\_/t] &\triangleq a[t]_\phi \end{aligned} \tag{28}$$

3. *is*'s as *pt*'s $\quad \dfrac{\Phi : A \to \mathbb{F}(\mathbb{F}(B))}{\Phi : \mathbb{P}(B) \to \mathbb{P}(A)}$

$$a \,\epsilon\, \Phi(U) \quad \triangleq \quad (\exists\, c : C_\Phi(a))\ \{\, a[c/r]_\Phi \mid r : R_\Phi(a,c) \,\} \subseteq U \tag{29}$$

Note: $\Phi : A \to \mathbb{F}(\mathbb{F}(B))$ can always be written $\langle\phi\rangle\,;[\,\psi\,]$ for certain $\phi : A \to A'$, $\psi : A' \to \mathbb{F}(B)$, as follows. Write Φ_{pre},Φ_{post} for ϕ,ψ .

$$\begin{aligned} A' &= \{\, \text{pending}(a,c) \mid a : A, c : C_\Phi(a) \,\} \\ \phi(a) &= \{\, \text{pending}(a,c) \mid c : C_\Phi(a) \,\} \\ \psi(\text{pending}(a,c)) &= \{\, a[c/r]_\Phi \mid r : R_\Phi(a,c) \,\} \end{aligned}$$

4. infima, suprema of *pt*'s $\dfrac{F_i : \mathbb{P}(A) \to \mathbb{P}(B)}{(\sqcap_i F_i), (\sqcup_i F_i) : \mathbb{P}(A) \to \mathbb{P}(B)}$

$$
\begin{array}{rcll}
(\sqcap_i F_i)(U) & \triangleq & \cap_i (F_i(U)) & (30) \\
(\sqcup_i F_i)(U) & \triangleq & \cup_i (F_i(U)) & (31)
\end{array}
$$

5. infima, suprema of *is*'s $\dfrac{\Phi_i : A \to \mathbb{F}(\mathbb{F}(B))}{(\sqcap_i \Phi_i), (\sqcup_i \Phi_i) : A \to \mathbb{F}(\mathbb{F}(B))}$

angelic $\sqcup$:

$$
\begin{array}{rcll}
C & \triangleq & \cup_i C_i & (32) \\
R(s, \langle i, c \rangle) & \triangleq & R_i(s, c) & \\
s[\langle i, c \rangle / r] & \triangleq & s[c/r]_i &
\end{array}
$$

demonic $\sqcap$:

$$
\begin{array}{rcll}
C & \triangleq & \cap_i C_i & (33) \\
R(s, f) & \triangleq & (\exists\, i)\ R_i(s, f(i)) & \\
s[f / \langle i, r \rangle] & \triangleq & s[f(i)/r]_i &
\end{array}
$$

6. sequential composition of *pt*'s

$$\frac{F : \mathbb{P}(A) \to \mathbb{P}(B) \qquad G : \mathbb{P}(B) \to \mathbb{P}(C)}{(F\,;G) : \mathbb{P}(A) \to \mathbb{P}(C)}$$

$$(F\,;G)(U) \quad \triangleq \quad F(G(U)) \tag{34}$$

7. sequential composition of *is*'s

$$\frac{\Phi : A \to \mathbb{F}(\mathbb{F}(B)) \qquad \Psi : B \to \mathbb{F}(\mathbb{F}(C))}{(\Phi\,;\Psi) : A \to \mathbb{F}(\mathbb{F}(C))}$$

$$
\begin{array}{rcll}
C & \triangleq & \Phi(C_\Psi) & (35) \\
 & = & \{\, a : A \mid (\exists\, c : C_\Phi(s))(\forall\, r : R_\Phi(a, c))\ C_\Psi(a[c/r]_\Phi) \,\} & \\
R(a, \langle c, f \rangle) & \triangleq & (\exists\, r : R_\Phi(a, c))\ R_\Psi(a[c/r]_\Phi, f(r)) & \\
a[\langle c, f \rangle / \langle r_0, r' \rangle] & \triangleq & (s[c/r_0]_\Phi)[f(r_0)/r']_\Psi &
\end{array}
$$

8. identity *pt*: $\textsc{skip} = \langle \textsc{skip} \rangle = [\,\textsc{skip}\,] : \mathbb{P}(A) \to \mathbb{P}(A)$

$$b \,\epsilon\, \textsc{skip}(U) \quad \triangleq \quad b \,\epsilon\, U \tag{36}$$

9. identity *is*: $\textsc{skip} = \langle \textsc{skip} \rangle = [\,\textsc{skip}\,] : A \to \mathbb{F}(\mathbb{F}(A))$

$$\begin{array}{rcl} C(\_) & \triangleq & N_1 \\ R(\_,\_) & \triangleq & N_1 \\ a[\_/\_] & \triangleq & a \end{array} \tag{37}$$

10. dual of an *is*

$$\frac{\Phi : A \to \mathbb{F}((\mathbb{F}(B)))}{\Phi^{\perp} : A \to \mathbb{F}((\mathbb{F}(B)))}$$

$$\begin{array}{rcl} C(a) & \triangleq & (\forall\, c : C_{\Phi}(a)) R_{\Phi}(a, c) \\ R(a, \_) & \triangleq & C_{\Phi}(a) \\ a[f/c] & \triangleq & a[c/f(c)]_{\Phi} \end{array} \tag{38}$$

Note: this doesn't have very good properties constructively. One can however calculate duals formally, by changing suprema to infima, angels by demons, *etc.*.

11. closures of pt's:

$$\frac{F : \mathbb{P}(A) \to \mathbb{P}(A)}{F^?, F^+, F^*, F^\infty : \mathbb{P}(A) \to \mathbb{P}(A)}$$

$$\begin{array}{rcll} F^* & \triangleq & \cap\{\, T : \mathbb{P}(A) \to \mathbb{P}(A) \mid (\textsc{skip} \cup (F\,;T)) \subseteq T \,\} & (39) \\ & = & U \mapsto \{\, a : A \mid (\forall\, V : \mathbb{P}(A))\ ((U \cup F(V)) \subseteq V) \to V(a) \,\} & \\ & = & \cup_\alpha (F\,;)^\alpha(\textsc{skip}) & \\ F^\infty & \triangleq & \cup\{\, T : \mathbb{P}(A) \to \mathbb{P}(A) \mid T \subseteq (\textsc{skip} \cap (F^\perp\,;T)) \,\} & (40) \\ & = & U \mapsto \{\, a : A \mid (\exists\, V : \mathbb{P}(A))\ V \subseteq (U \cap (F^\perp(V)) \wedge V(a) \,\} & \\ & = & \cap_n\{\, F_n \mid F_0 = \textsc{skip}; F_{n+1} = F_n \cap (F^\perp\,; F_n) \,\} & \end{array}$$

12. closures of is's $\dfrac{\Phi : A \to \mathbb{F}(\mathbb{F}(A))}{\Phi^*, \Phi^?, \Phi^+, \Phi^\infty : A \to \mathbb{F}(\mathbb{F}(A))}$

Φ^*

$$\begin{array}{rcll} C & \triangleq & (\mu\ X : A \to \mathrm{Set}) & (41) \\ & & (\forall\ a : A) & \\ & & \quad \{\, \mathrm{exit} \,\} & \\ & & \cup\{\, \mathrm{call}(c, f) \mid c : C_\Phi(a), f : (\forall\ r : R(a, c))\ X(a[c/r]_\Phi) \,\} & \\ & & \quad \subseteq X(a) & \end{array}$$

$$\begin{array}{rcl} R(a, \mathrm{exit}) & \triangleq & \{\, \mathrm{nil} \,\} \\ R(a, \mathrm{call}(c, f)) & \triangleq & \{\, \mathrm{cons}(r_0, r') \mid r_0 : R_\Phi(a, c), r' : R(a[c/r_0], f(r_0)) \,\} \\ a[\mathrm{exit}/\mathrm{nil}] & \triangleq & a \\ a[\mathrm{call}(c, f)/\mathrm{cons}(r_0, r')] & \triangleq & (a[c/r_0]_\Phi)[f(r_0)/r'] \end{array}$$

$\Phi^? \triangleq \Phi \sqcup \textsc{skip}, \qquad \Phi^+ \triangleq \Phi\,;\Phi^*.$

Φ^∞

$$\begin{array}{rcll} C & \triangleq & (\nu\ X : A \to \mathrm{Set}) & (42) \\ & & (\forall\ a : A) & \\ & & X(a) \subseteq \{\, \mathrm{srv}(f, g) \mid & f : (\forall\ c : C_\Phi(a)) R_\Phi(a, c), \\ & & & g : (\forall\ c : C_\Phi(a)) X(a[c/f(c)]_\Phi) \,\} \end{array}$$

$$\begin{array}{rcl} R(a, \mathrm{srv}(f, g)) & \triangleq & \{\, \mathrm{nil} \,\} \\ & & \cup\{\, \mathrm{cons}(c_0, c') \mid c_0 : C_\Phi(a), c' : R(a[c_0/f(c_0)], g(c_0)) \,\} \\ a[\mathrm{srv}(f, g)/\mathrm{nil}] & \triangleq & a \\ a[\mathrm{srv}(f, g)/\mathrm{cons}(c_0, c')] & \triangleq & (a[c_0/f(c_0)]_\Phi)[g(c_0)/c'] \end{array}$$

13. functional assignment as a *pt*

$$\frac{f : A \to B}{\mathbf{assign}\, f : \mathbb{P}(B) \to \mathbb{P}(A)}$$

$$s \,\epsilon\, (\mathbf{assign}\, f)(P) \;\triangleq\; f(s) \,\epsilon\, P$$

Note: $\mathbf{assign}\, f = (\cdot f) = f^{-1} = \mathbb{P}(f)$.

Note: $\mathbf{assign}\, f \,;\mathbf{assign}\, g = \mathbf{assign}\, g \cdot f$.

Redundant.

$$\mathbf{assign}\, f = \langle \mathbf{graph}\, f \rangle = [\, \mathbf{graph}\, f \,]$$

14. functional assignment as an *is*

$$\frac{f : A \to B}{\mathbf{assign}\, f : A \to \mathbb{F}(\mathbb{F}(B))}$$

$$\begin{aligned} C(\_) &\triangleq N_1 \\ R(\_,\_) &\triangleq N_1 \\ a[\_/\_] &\triangleq f(a) \end{aligned}$$

10.3.2 Morphisms between predicate transformers

In analogy with transition structures, we could define a morphism between interaction structures to be a pair of predicate transformers that satisfy the appropriate sub-commutativity property. However, in this case we insist that the predicate transformer is *angelic*, that is commutes with all disjunctions, that is is determined by its value at singletons, that is is an angelic relational update.

Give definition.

Give it for interaction structures. Note: really between an is and a pt.

11 laws

Laws for (inferring inclusion and equality between) relations.

1. Predicate transformers determine relation transformers. However, the effect of certain predicate transformers can sometimes be expressed merely from sequential composition, and division. The following are equations between relation transformers.

$$\begin{array}{lcllcl} (Q\,;) & = & (^\sim)\cdot(\langle Q\rangle\cdot)\cdot(^\sim) & (/Q) & = & ([\,Q\,]\cdot) \\ (;Q) & = & (\langle Q^\sim\rangle\cdot) & (\backslash Q) & = & (^\sim)\cdot([\,Q^\sim\,]\cdot)\cdot(^\sim) \end{array} \tag{43}$$

   To a certain extent, we are interested in representing relations with transition structures – we may represent a relation as a transition structure, or as the converse of a transition structure, or as the reciprocal of a transition structure (and so on and on).

2. I might have introduced binary operators for relative complement and implication. Then the adjunctions for relations are:

$$(R_1\,;R_2)\subseteq Q \iff R_1\subseteq(Q/R_2) \tag{44}$$
$$(R_1-R_2)\subseteq Q \iff R_1\subseteq(R_2\cup Q) \tag{45}$$
$$(R_1\cap R_2)\subseteq Q \iff R_1\subseteq(R_2\Rightarrow Q) \tag{46}$$

   Some laws (need to check):

$$Q-(R_1\cup R_2) = (Q-R_1)\cap(Q-R_2)=((Q-R_1)-R_2) \tag{47}$$
$$Q-(R_1\cap R_2) \supseteq (Q-R_1)\cup(Q-R_2) \tag{48}$$
$$(R_1\cup R_2)\Rightarrow Q = (R_1\Rightarrow Q)\cap(R_2\Rightarrow Q) \tag{49}$$
$$(R_1\cap R_2)\Rightarrow Q = R_1\Rightarrow(R_2\Rightarrow Q) \tag{50}$$

   Non-binary sups and infs?

3. laws for sups and infs.

$$(\cup R_i)\subseteq Q \iff (\forall\, i)\; R_i\subseteq Q \tag{51}$$
$$Q\subseteq(\cap R_i) \iff (\forall\, i)\; Q\subseteq R_i \tag{52}$$

   For relations, sequential composition commutes with union (on both sides).

4. The laws for relational converse $^\sim$. This commutes with infima and suprema (and closures), is monotone, and

$$Q^{\sim\sim}=Q \tag{53}$$
$$(\mathbf{graph}\,f)^\sim\,;(\mathbf{graph}\,f)\subseteq \textsc{skip};\quad \textsc{skip}\subseteq(\mathbf{graph}\,f)\,;(\mathbf{graph}\,f)^\sim \tag{54}$$
$$\textsc{skip}^\sim=\textsc{skip} \tag{55}$$
$$(Q\,;R)^\sim = R^\sim\,;Q^\sim \tag{56}$$

5. What are right laws for domain restriction?

$$\mathbf{test}\, U = U \rightharpoonup \text{SKIP} \tag{57}$$
$$U \rightharpoonup R = \mathbf{test}\, U \,;\, R \tag{58}$$
$$\mathbf{test}\, U \,;\, \mathbf{test}\, V = \mathbf{test}\, (U \cap V) \tag{59}$$
$$R_1 \subseteq R_2 \Rightarrow (U \rightharpoonup R_1) \subseteq (U \rightharpoonup R_2) \tag{60}$$
$$U \subseteq V \Rightarrow (U \rightharpoonup R) \subseteq (V \rightharpoonup R) \tag{61}$$

6. Dedekind's law (modular law). This is what governs sequential composition, converse and intersection.

$$(Q \,;\, R) \cap S \subseteq Q \,;(R \cap (Q^{\sim} \,;\, S)) \tag{62}$$

   This law is used to prove that if Q is a partial function ('deterministic' relations) then $Q \,;(R_1 \cap R_2) = (Q \,;\, R_1) \cap (Q \,;\, R_2)$. In other words $(Q \,;)$ is conjunctive (it is always disjunctive).

7. Intersection of relations.

$$\begin{aligned}
\mathbf{test}\, U \cap \mathbf{test}\, V &= \mathbf{test}\, U \cap V && (63)\\
(Q_1 \cap Q_2) \cap Q_3 &= Q_1 \cap (Q_2 \cap Q_3) && (64)\\
Q \cap \text{void} &= \text{void} && (65)\\
Q \cap \text{chaos} &= Q && (66)\\
Q_1 \cap Q_2 &= Q_2 \cap Q_1 && (67)\\
Q \cap Q &= Q && (68)\\
Q \cap (Q \cup R) &= Q && (69)\\
Q \cap (\cup_i R_i) &= \cup_i (Q \cap R_i) && (70)\\
Q \cap (\cap_i R_i) &= \cap_i (Q \cap R_i) && (71)\\
Q \cap (R_1 \,;\, R_2) &\subseteq R_1 \,;(R_2 \cap (R_1{}^{\sim} \,;\, Q)) && (72)\\
Q \cap R^{\sim} &= (Q^{\sim} \cap R)^{\sim} && (73)
\end{aligned}$$

8. Union of relations.

$$\begin{aligned}
\mathbf{test}\, U \cup \mathbf{test}\, V &= \mathbf{test}\, U \cup V && (74)\\
(Q_1 \cup Q_2) \cup Q_3 &= Q_1 \cup (Q_2 \cup Q_3) && (75)\\
Q \cup \text{void} &= Q && (76)\\
Q \cup \text{chaos} &= \text{chaos} && (77)\\
Q_1 \cap Q_2 &= Q_2 \cap Q_1 && (78)\\
Q \cup Q &= Q && (79)\\
Q \cup (Q \cap R) &= Q && (80)\\
Q \cup (\cup_i R_i) &= \cup_i (Q \cup R_i) && (81)\\
Q \cup (\cap_i R_i) &\subseteq \cap_i (Q \cup R_i) && (82)\\
Q \cup (R_1 \cap R_2) &= (Q \cup R_1) \cap (Q \cup R_2) && (83)\\
Q \cup (R_1 \,;\, R_2) &\ldots && (84)
\end{aligned}$$

9. Sequential composition of relations.

$$\begin{aligned}
(Q_1 \,;Q_2)\,;Q_3 &= Q_1 \,;(Q_2 \,;Q_3) && (85)\\
\textsc{skip}\,;Q &= Q && (86)\\
Q\,;\textsc{skip} &= Q && (87)\\
Q\,;(\cap_i R_i) &\subseteq \cap_i(Q\,;R_i) && (88)\\
(\cap_i Q_i)\,;R &\subseteq \cap_i(Q_i\,;R) && (89)\\
Q\,;(\cup_i R_i) &= \cup_i(Q\,;R_i) && (90)\\
(\cup_i Q_i)\,;R &= \cup_i(Q_i\,;R) && (91)\\
Q\,;\mathrm{void} &= \mathrm{void} && (92)\\
\mathrm{void}\,;Q &= \mathrm{void} && (93)\\
\mathbf{test}\,U\,;\mathbf{test}\,V &= \mathbf{test}\,(U\cap V) && (94)\\
\textsc{skip} &= \mathbf{test}\,\mathrm{chaos} && (95)\\
\mathbf{graph}\,f\,;\mathbf{graph}\,g &= \mathbf{graph}\,(g\cdot f) && (96)\\
\textsc{skip} &= \mathbf{graph}\,\textsc{skip} && (97)
\end{aligned}$$

Laws for (inferring inclusion and equality between) predicate transformers.

1. For predicate transformers, sequential composition commutes with sups and infs in its left-hand argument.(Unlike the case of relations, where we don't commute with inf's, but commute with sups on both sides.)

$$\begin{aligned}
(F_1\,;F_2)\,;F_3 &= F_1\,;(F_2\,;F_3) && (98)\\
\textsc{skip}\,;F &= F\\
F\,;\textsc{skip} &= F\\
(\cup_i F_i)\,;G &= \cup_i(F_i\,;G)\\
(\cap_i F_i)\,;G &= \cap_i(F_i\,;G)\\
F\,;(\cup_i G_i) &\supseteq \cup_i(F\,;G_i)\\
F\,;(\cap_i G_i) &\subseteq \cap_i(F\,;G_i)
\end{aligned}$$

   The last 2 semi-equations are just by monotonicity.

   They can be strengthened to equality for certain F.

$$\begin{aligned}
\langle\phi\rangle\,;(\cup_i G_i) &= \cup_i(\langle\phi\rangle\,;G_i)\\
[\,\phi\,]\,;(\cap_i G_i) &= \cap_i([\,\phi\,]\,;G_i)
\end{aligned}$$

2. Special cases of sequential composition. Should be associative with unit SKIP.

$$\begin{aligned}
\langle Q\rangle\,;\langle R\rangle &= \langle Q\,;R\rangle && (99)\\
[\,Q\,]\,;[\,R\,] &= [\,Q\,;R\,]\\
\textsc{skip} &= \mathbf{assign}\,\textsc{skip}\\
\mathbf{assign}\,f &= \langle\mathbf{graph}\,f\rangle = [\,\mathbf{graph}\,f\,]
\end{aligned}$$

3. Are these true?

$$F;\langle Q\rangle = \langle F\cdot Q\rangle \tag{100}$$
$$F;[\,Q\,] = [\,F\cdot Q\,] \tag{101}$$

An interaction structure is something of the form $\langle\phi\rangle\,;[\,\psi\,]$.