Models for command-response interfaces

Peter Hancock, *etc*.

Version of May 5, 2003

It is only relatively recently that in computer science we have begin to exploit the idea that proofs are essentially executable programs, although it emerged from intuitionistic mathematics some decades before the first digital computers ran programs. One application, as it were from logic to computer science, has been in the design of ever more expressive type systems for programming. The situation is currently that type-checkers have been written for a range of experimental functional programming languages in which the type systems are sufficiently rich to express propositions, logical connectives, predicates, quantifiers, relations, predicate transformers, temporal and modal operators, and everything any one has ever asked for to write fully precise mathematical specifications, or the reasoning that underlies the construction of a program to meet a precise specification.

The kind of programs we can write using these type systems are programs that denote mathematical values; they do not of themselves actually *do* anything or exhibit behaviour[1]. Rather, we do something with them, or make practical application of them, or somehow use a mathematical value as a guide to action. Put crudely, the puzzles that underlie this paper are of the following kind. What kind of proof is it that 'does' something, for example one which when set in motion runs an internet server to book plane-flights and hotel rooms, or one which prevents the brakes on a bus from locking in a skid? What kind of proposition does the proof prove, and how is this connected with a specification of the desired behaviour?

In practice, the software components of complex systems are (and forever will be) written in a variety of imperative languages, which exchange information by issuing and responding to invocations of procedures or methods. An important task for computer science is to capture mathematically the notion of an imperative 'interface', or of command-response interaction between components, whatever the technologies with which they are implemented. The problem is to devise mathematical structures with which to model handshaken command-response interfaces or service specifications, and to organise the design of software to meet specifications expressed in terms of those structures.

The question we want to address here, from the perspective of the theory of dependent types, specifically the predicative type system developed by Martin-Löf, is

> What is the logical form of the interface between a component in a system and its environment, where communication is by commands and responses?

By a *command-response* interface I mean an 'imperative' interface, in which communication between a component and its environment takes the form of a succession

[1] Actually it could be argued that there is at least one such system, namely Lennart Augustsson's language Cayenne, with which one can write programs that do more than display values.

of bi-partite command/response, handshaken interactions. One party, agency or agent (called the client, user, master, commander, initiator, stimulator, actor, ...) initiates an interaction by issuing a command, and the other (called the server, system, slave, obeyer, commitor, responder, re-actor, ...) completes it by making a response appropriate to the command. There is a *state*, which determines the interactions that are possible at any point of time; the interface moves or evolves to or is assigned a new state as each interaction takes place, as it were instantaneously and destructively. The state can always be computed from the history of interactions. The combination of sequential execution and destructive assignment is characteristic of imperative programming.

The paradigm of clients and servers pervades the design of computer systems at all levels. No doubt there are many other vitally important patterns of interaction between components, but none as deservedly popular as that between clients and servers.

- In its simplest form, the client (a flesh-and-blood human being to whom computers are a necessary evil) makes a request for a service, or action in a certain repertoire of a device, by clicking on a button perhaps. If the necessary conditions are met, the server returns some results, and moves (as it were in a single instant) to a new state in which the user's bank balance has been debited and goods of some kind have been dispatched. The extreme simplicity of this picture (with its abrupt transition from future to past, initiated by the user and committed by the system) is highly appropriate at the level of user interfaces that can be used by frail and impatient flesh and blood clients.

- At a lower level, the same paradigm accommodates a common kind of procedural interface to an 'object', or component comprising a collection of procedures and data. A call to a procedure passes control and input data values or arguments into a program module which (with luck) subsequently returns control and output data values or results to the caller, after the pattern that calls are eventually followed by matching responses. Here a call means passing control/data, and a response means return of control/data.

- At a lower level still, another application of the same paradigm is to represent the fetch-execute cycle of an interpreter for a certain machine code, or instruction set. Instructions are fetched, then performed, producing results that can influence the effect of later instructions.

It is important to distinguish interfaces from their implementations.

An interface, or a good one should permit a variety of different implementations. The implementation should have latitude in responding to requests, subject only to the constraint that the legality of subsequent requests, and of responses to such requests, should be deducible from the sequence of commands and responses.

By the *logical* form of a specification, I mean to exclude resource requirements, operating conditions, details of coding or marshaling, and other issues like ergonomics, performance and tasteful styling. It seems to be necessary to clarify how one should say *what* is supposed to happen before it is worth talking about the evil necessary to get it to happen, or about *how fast* or *how frequently* it will happen.

Constructivity... The specific interest of constructivity in applied mathematics is that the models of real-world phenomena that we set up and reason about in constructive mathematics are 'without more ado' *working* models. Suitable encoded, they are programs that can in principle be run on a machine – though this might exhaust all the resources of the universe. This is a comprehensive picture of what is fundamentally

going on in writing programs, a recommendation about how to think of that process rather than a recommendation about how to write programs.

The question pursued in this article is this: if proofs are programs, then what kind of proposition is it that is proved by an interactive program – a program that does something more than merely calculate the value of an expression, but actually has an effect on and is affected by its environment? My ambition is not to answer this question in general, but only for those kinds of interactive program which communicate with their environment through command response interfaces.

Even a program which displays the value of an expression has an effect on the device that is used to display it. The communication between the program and its device is (following the command-response paradigm) modelled here as some kind of client-server interface, in which the display is one of the agents.

The model here is that all unsolicited messages are matched one for one by acknowledgements, which is probably not strictly true in real input-output systems.

Predicativity... Dependent type theories may be impredicative (Coq, Lego) or predicative (Alf, Half, Alfa, Agda, Cayenne). The difference this makes is that in the former kind of type system, the quantifier domains are closed under an operation analogous to the power-set operation of classical ZF set theory, that assigns to a set its set of subsets. In the predicative kind the power operator exists, but crosses a size boundary, as the power of a set (the collection of statements that can be made about its elements) is not a (small) set but a (large, proper) type. In these predicative systems this lack of a power-set operator may be addressed to some extent by 'internalised' approximations, expressed using 'universe' sets. The predicative systems incorporate strong principles of inductive definition, taken as primitive rather than as justified by impredicative quantification over power sets.

The restriction to predicative methods has an advantage of conceptual simplicity. One can think of a value, or an element of a set or data type as 'built up from below', or inductively. One can think of a set (or data-type) itself as something inductively defined, as it were with no vicious circularity. This is sometimes referred to as the 'well-foundedness' of standard predicative type theory.

To say that a value is 'built up from below' is of course only to use a metaphor. The underlying idea is perhaps that such a value is a 'fit', or satisfactory argument for a certain definitional scheme, or computation rule – in a sense of 'fit' that needs analysis, but implies termination. This is not a sharp, formal notion, and it may be that there is in fact a mixture of distinct elements[2] Yet it provides a single overall principle or paradigm that is instantiated over and over again in the rules of type-correctness and computation for data-types of all forms, including product types and function spaces[3]. The existence of this overall pattern provides a basis for an implementation of a type-checker that can assist with many combinatorial details of defining functions on data-types.

On the other hand the impredicative notion of subset is of quite a different character. It goes far beyond the idea of something 'built up from below', in any sense. It is fair to

[2]Kreisel remarks [15, p. 171] that the naïve notion of set was a 'crude mixture' containing (three) different elements. Even before the set theoretic paradoxes were discovered, it was criticised for its ambiguity. These criticisms were eventually met by making the necessary distinctions.

[3]In category theory, products and function spaces are usually characterised by 'finality' properties, that is as *destinations* of certain kinds of universal arrow. On the other hand in predicative type theory *all* sets are inductively defined, though Set itself is not. Even products and function spaces are characterised rather by 'initiality' properties, that is as *origins* for a certain kind of universal arrow. This presents a subtle problem in categorical treatments of type theory.

say that it is enormously difficult to attach an effective meaning to statements involving nested impredicative quantification.

Predicativity and imperativity... A particular theoretical problem arises in connecting imperative programming with predicative type theory. One of the most convincing and practical approaches to developing imperative programs from their specifications (Morgan [19], Dijkstra [9] [8], Back and von-Wright [4],..) takes a specification to be a predicate transformer, *i.e.* an operator on the power-set of a state-space. Applied to a predicate representing a goal to be established, such an operator yields another predicate, namely the weakest predicate which ensures both that the program terminates and establishes the desired goal predicate in any state in which it terminates. Programming constructs such as assignment, sequential composition, branching, block structure and procedure calls are defined in terms of operators on predicate transformers. The theory of predicate transformers is based partly on the algebraic structure of subsets of or predicates over a set, and partly on the structure induced by composition or transformers over any type.

To deal with recursive constructions the literature on the refinement calculus exploits particularly the Knaster-Tarski theorem concerning the existence of least and greatest fixed points of monotone functions on a complete lattice. It seems problematic to justify the Knaster-Tarski theorem in its full generality from a predicative point of view. This makes it a challenge to develop a theory of imperative programming in predicative type theory.

1 Modeling command-response interfaces

A complete description of the interface presented by a real device tells you everything you need to know to use it, and therefore to implement it. It is sensible to divide this into parts, a syntactical part and a semantical part.

One 'syntactical' part of the description will be about precisely how things are encoded, such as that a logic '1' is represented by a certain voltage on a certain wire, or that the command is issued by the rising edge of a particular signal, or that numbers are represented by ASCII strings stored in a certain way. This is the evil necessary to make use of the device. I simply assume that ...

The other 'semantical' part of the specification concerns the functionality of the interface. It tells you why you might want to use it, or in other words what it is good for, or its practical value. It is the 'logical' notion, as it applies to command-response interfaces which we want to capture with a mathematical structure.

The fundamental notion on which this analysis is based is *state*. To describe the functionality of an interface, we first of all need a set whose elements represent states or configurations of the device that might arise as interactions take place. (The state is not in general directly visible.) The set of states is called the *state space*. The device moves discretely from state to state, tracing out a trajectory, orbit or path through the state-space.

At each state, the device has a certain potential for interaction. We represent this potentiality, or functionality as a function assigning to each state s a collection of sets of states.

$$\{\, \{\, s[c/r] \mid r \in R(s,c) \,\} \mid c \in C(s) \,\}$$

Here $s[c/r]$ is convenient notation for the new state of the interface, when a command-response interaction $\langle c, r\rangle$ (initiated by c, completed by r, also written c/r) occurs starting in state $s \in S$.

The function C assigns sets to states, and so can be interpreted as a propositional function, or predicate of states. The predicate (which is in some sense a guard) holds if the device can be activated. (This is in a sense of 'activate' according to which a device may be activated in some states for which no reaction is possible; it may be better to say 'an interaction with the system can be initiated'). If the device is properly activated, a proof that the guard predicate holds can be supplied, and we can think of this proof as a command that is issued to the device.

The function R assigns sets to states $s \in S$ and commands $c \in C(s)$. Considered as a predicate, it holds if the command c terminates, or can be performed, obeyed or carried out to termination. We might also say: if it is effective, or feasible. We can think of a proof of that this predicate holds as a response that is returned by the system.

The function assigns to each state s and interaction c/r the next state $s[c/r]$. Note that the next state is fully determined by the sequence of interactions leading from an initial state to the current state. We can think of these interactions as 'bi-partite' events in the history of the interface.

In practice (in Z, VDM, TLA, *etc.*), the state space is most conveniently described as a set of records, with fields or coordinates that divide the state space, or factor it. In general the records are not independent, but are constrained to satisfy an 'invariant' relation with one term (argument place) for each state-variable. For example, if we describe a file system interface (as for example in Morgan and Sufrin [20]), the state-space will comprise all possible configurations of files, directories, and other entities in the interface such as channels, which satisfy the invariant.

Note that the structure

$$\begin{array}{lcl} S & : & Set \\ C & : & (s \in S) \to Set \\ R & : & (s \in S, c \in C(s)) \to Set \\ \_[\_/\_] & \in & (\forall\, s \in S, c \in C(s), r \in R(s,c)) \to S \end{array}$$

is in its first three lines the very paradigm of a dependent context

$$\begin{array}{lcl} A_1 & : & Set \\ A_2 & : & (x_1 \in A_1) \to Set \\ A_3 & : & (x_1 \in A_1, x_2 \in A_2(x_1)) \to Set \\ \ldots & & \end{array}$$

So type-dependency is used in an essential way.

Because of type-dependency, we can express two different notions of termination; interaction terminates if there are no commands that can be issued in the current state (*i.e.* $C(s)$ is empty), and also when a command c is issued for which no response is possible (*i.e.* $R(s,c)$ is empty). However these are both species of failure, and there is also successful termination in which an interaction is complete, and the new state attained.

The structure $(S, C, R, .[./.])$ is well known, though perhaps not very widely. It has applications in connection with formal grammars (Petersson and Synek [24]), as well as with covering structures in formal topology (See Mac Lane and Moerdijk [17, page 534 Ex. 5] for the notion of covering system, and Coquand, Sambin, Smith, Valentini [7]) for closely related material. No doubt other applications can be cited.

It is well known that the 4-part structure above can also be used to model inferential systems of a certain kind, such as the rule sets of Peter Aczel [3], or generalised[4] forms

[4](The generalisation is that the rules can be infinitary: there are no 'cardinality' restrictions on any of the index sets.)

of Post system. In the inferential applications the elements of S represent statements, while the rest of the structure represents a system of first-order[5] inference rules for inferring statements $s \in S$.

$$\dfrac{\begin{array}{c}\vdots \\ r{:}R(s,c) \\ \cdots \quad s[c/r] \quad \cdots\end{array}}{s}\, c{:}C(s)$$

In this case, the 'commands' $c : C(s)$ index the inferences with conclusion s, and the 'responses' $r : R(s, c)$ index the premises $s[c/r]$ for which proof is required in order to conclude s by inference c.

Another application of the structure is to represent multi-sorted algebras; by the Curry-Howard correspondence, this is really the same as the inferential interpretation. In this more algebraic interpretation the elements of S represent sorts, the commands represent constructors (which partition values of a given sort into their possible alternative forms), and the responses represent selectors (which pick out a field or conjunctive subcomponent from a value formed with a particular constructor). These sorts can be for example syntactical categories in a formal grammar, as in [24].

When we think of inferential structures (proofs) in this way, it is natural to try to model proofs as strategies for winning certain kinds of 'logical' game. The Player in such a game (a server, or service provider) claims a statement, while the Opponent (a client) is sceptical and probes more and more deeply into the reasoning behind the claim. Indeed, connections between proof systems and certain kinds of game or dialogue have had a long history in logic. (Some important papers are: Novikov [23], Hintikka [14], Moschovakis [21], Aczel [2], Coquand, [6], Abramsky [1].)

Since the 4-part structure defined above has so many applications, it isn't surprising that it should also admit applications in programming. One of these seems to be that the structure $(S, C, R, .[./.])$ represents an command-response interface.

2 The programming firmament

In this section we try to map out the structures with which we propose to model command-response interfaces with respect to other entities in a programmer's 'mental constellation'.

The objects with which we deal everywhere in programming are, roughly speaking, either

- functions that transform values in one set into values in another, or
- relations (which may be more or less inclusive) between values, or
- predicate transformers (which may be stronger or weaker).

There are three 'levels' here, as it were of increasing abstraction or latitude. The levels can be thought of as categories in the mathematical sense, and each of the three kinds of entity (functions, relations or predicate transformers) as morphisms in the respective category. They even have extra structure on the hom-sets.

Functions are concrete and deterministic. We can delegate the evaluation of a (constructive) function to a machine. Functions are 'code'[6]. But code is only the precipitate

[5] *i.e.* inference rules which do not discharge hypotheses

[6] In somewhat the sense of Carroll Morgan's text-book on the refinement calculus [19, section 1.5].

of the process of figuring out a program. In the middle of the 'figuring out', there may be considerable latitude in choosing (for example) what value a function should have, and some virtue in postponing the choice. What one wants is as it were, an ideal or virtuous entity, that is 'not yet' a function; and for this, if relations did not exist, we would have at this point to invent them. By themselves, functions are awkward to manipulate algebraically. For example the converse of a function is a relation, seldom a function. For another example, without some form of compatibility restriction (directedness) it may not be possible to combine partial functions should their domains overlap. Converses, conjunctions, and other algebraic operations are important for a smooth and practical calculus.

Relations are amenable to useful operations such as converse and intersection. If they are expressed in a certain form (described below in section 2.3), their textual expression can be used as mechanically executable 'code'[7]. The machine is not self-standing, but relies on an external agency (the outside world or environment) to choose the transition by which it advances to a new state. But relations too are awkward. They admit at most one level of non-determinism, whereas in reality there is often a non-trivial degree of choice on both sides of an interface.

Predicate transformers provide a maximally user-oriented, or environment focussed form of specification, directly connected with the user's goals and concerns. To find out how to use a device, we want to know how we can use it so as to accomplish certain things, and at the same time avoid certain other things. However the specification of a service is represented, the use that we want to get out of it is a calculation with predicates: for example to work out for a given goal predicate what initial state has to hold if the goal is to be established, or to perform other calculations necessary to maintain safe use of a service.

The three levels are systematically related to each other. In a sense, the step from relations to predicate transformers is the same as or analogous to that from functions to relations. A description of this systematic relationship in terms of the theory of enriched categories may be found in de Moor, Gardiner and Martin "An algebraic construction of predicate transformers" [11]. To indicate it briefly, the morphisms in each successive level level can be constructed from *spans* of morphisms in the lower level, while in the other direction, the morphisms in the lower level can be identified with certain morphisms in the upper level that are especially well-behaved. Well-behaved relations are *maps*, while well-behaved predicate transformers are universally conjunctive, meaning that they commute with intersection of arbitrary indexed families of predicates.

This elegant 1-dimensional picture becomes more intricate and 2-dimensional when we examine it from the perspective of type theory (see figure 1).

Note that $\mathbb{P}(B) \to \mathbb{P}(A)$ is isomorphic to $A \to \mathbb{P}(\mathbb{P}(B))$, by transposition of arguments.

2.1 Predicates and families

In type theory as in real life there are two notions of subset, that are reflected by the two different notations in common use for writing subsets of a given set S. They correspond to the separation and replacement axioms of set theory.

separation $\{ s \in S \mid P(s) \}$

where P is a predicate (propositional function) with domain S.

[7] [4, section 9.5])

Functions
$f : A \to B$
$f = (\lambda\, a)b(a)$

Relations
$R : A \to \mathbb{P}\,(B)$
$R(a) = \{\, b \in B \mid a\ R\ b \,\}$

Transition structures
$\phi : A \to \mathbb{F}\,(B)$
$\phi(a) = \{\, a[i]_\phi \mid i \in T_\phi(a) \,\}$

Predicate transformers
$F : \mathbb{P}\,(B) \to \mathbb{P}\,(A)$
$F(P) = \{\, a \in A \mid F(P, a) \,\}$

Interaction structures
$\Phi : A \to \mathbb{F}\,(\mathbb{F}\,(B))$
$\Phi(a) = \{\, \{\, a[c/r]_\Phi \mid r \in R_\Phi(a, c) \,\} \mid c \in C_\Phi(a) \,\}$

Each 'box' represents a category, gives a typical variable for a morphism in that category, and a canonical form for such a morphism.

Figure 1: Three levels.

replacement $\{\, s(i) \mid i \in I \,\}$
where I is an index set and s is an S-valued function with domain I.

These two notions of subset can be expressed as two operators on types, whose values and arguments are *proper* (large) types[8]. The values are proper types, even if the arguments are sets. The two notions (more precisely their set-level analogues) are discussed in Martin-Löf's Bibliopolis book [18, p. 64].

$$\begin{aligned} \mathbb{F}\,(A) &= (I : Set) \times I \to A \\ \mathbb{P}\,(A) &= A \to Set \end{aligned}$$

morphisms The operators $\mathbb{F}$ and $\mathbb{P}$ act not only on types but also on functions between types. With some overloading of notation,

$$\begin{aligned} &\mathbb{F} : (A \to B) \to \mathbb{F}\,(A) \to \mathbb{F}\,(B) \\ &\mathbb{F}\,(f)(\{\, a(i) \mid i \in I \,\}) = \{\, f(a(i)) \mid i \in I \,\} \end{aligned}$$

$$\begin{aligned} &\mathbb{P} : (A \to B) \to \mathbb{P}\,(B) \to \mathbb{P}\,(A) \\ &\mathbb{P}\,(f)(\{\, b \in B \mid P(b) \,\}) = \{\, a \in A \mid P(f(a)) \,\} \quad . \end{aligned}$$

Note that $\mathbb{P}$ is contravariant; $\mathbb{P}\,(f)$ is the inverse image operator f^{-1} or substitution operator $(\_ \cdot f)$ acting on predicates. However $\mathbb{F}\,(\_)$ is covariant.

examples Some simple (if degenerate) examples are provided using

- the empty set [9] $\{\,\}$. We have the empty family $\{\,\} = \{\ \mid \_ \in \{\,\}\,\}$, and the empty predicate $None = None_A = \{\, a \in A \mid \{\,\}\,\}$ which has constant value $\{\,\}$ throughout the set A.

[8] I use $(x : \alpha) \to \beta$ as notation for type level Π, and $(x : \alpha) \times \beta$ for type level Σ. The prefixes $(x : \alpha) \to \ldots$ and $(x : \alpha) \times \ldots$ bind (free) occurrences of x as far to the right as possible. So $(I : Set) \times I \to A$ parses the same as $(I : Set) \times (I \to A)$.

[9] Another notation is N_0.

- the singleton set [10] $\{*\}$ with exactly one element: $*$. We have the singleton families $\{a\} = \{\, a \mid \_ \in \{*\}\,\}$ for each $a : A$, and the vacuously true predicate $All = All_A = \{\, a \in A \mid \{*\}\,\}$ which has constant value $\{*\}$ throughout the set A.

- a propositional equality relation $a =_A a'$ on an underlying set A. One may also write this $Id_A = (=_A) : A \to \mathbb{P}(A)$. If $a : A$, then we can form a predicate, or propositional function on A, written

$$\{a\} = \{\, a' \in A \mid a =_A a' \,\}$$

  Intuitively $\{a\}$ is the strongest statement that can be made about the element a – propositional equality with a. We shall see that use of singleton predicates of this form entails a certain 'loss of innocence'. As it were, the singleton predicates are apples on the tree of knowledge; by using them, you agree to leave the (computational) garden of Eden.

Further examples of predicates and are provided by quantification over indexed families. Both families and predicates are closed under unions of indexed families, with respect the natural notion of inclusion. Predicates (but not families) are also closed under operations such as intersection, complementation, and so forth; in a sense predicates give a more flexible notion of 'subset' than families. Yet, the notion of a predicate is in a certain sense negative: the underlying set A occurs negatively in $\mathbb{P}(A) = A \to Set$. This means that it is not a *computational* notion of subset; whereas families provide a mechanism for computing a subset exhaustively.

conversion between predicates and families If S is a small type or data-type, a predicate $\{\, s \in S \mid P(s)\,\}$ of a set can be written in the form of an indexed subset $\{\, \mathsf{out}_{\mathsf{L}}\, i \mid i \in (\exists\, s \in S)P(s)\,\}$. This makes use of the $(\exists\, s : S)$ construction on state-indexed families of sets and its left projection $\mathsf{out}_{\mathsf{L}}$. I call this *reducing* a predicate to an indexed subset, on the grounds that projection is a kind of reduction.

Conversely, an indexed subset $\{\, t(i) \mid i \in I\,\}$ can be written in the form of the predicate $\bigcup_{i \in I}\{t(i)\}$, which is a union of singleton predicates.

[INNOCENCE.]

In certain respects discussed below, the identity relation is not altogether 'innocent'.

[TERMINOLOGY.] As something not altogether innocent has to be combined with a family to produce a predicate which reduces to it, it seems appropriate to to call the predicate-based notion of subset 'full-blooded'. (This has a nasty fascistic connotation. I should think of some less spin-some terminology to mark the distinction: catholic versus protestant, green versus orange, contravariant versus covariant *etc.*.)

[THE FOLLOWING NEEDS HEAVY REWORK.]

Now I should try to substantiate the allegation that there is something not quite 'innocent' about the notion of a uniform and general substitutive equality relation which makes sense for an arbitrary set. First, there is the question of whether it should be extensional or intensional.

Extensional equality, as in Martin-Löf's Bibliopolis book [18, p. 61] (see also the Eq relation in [22, pp. 61 – 67]) reflects at the level of sets and propositions the judgemental equality between objects of a type. A consequence of this reflection is that it is no longer possible to characterise all set-forming operators by the validity of a certain

[10] Another notation is N_1 with sole element 0_1

scheme of definition by recursion on the build-up of the elements of a set. (In categorical terms, this is a kind of initiality; the point is that Eq is not associated with any kind of catamorphism.) Moreover equality judgements are not mechanically decidable in extensional type theory; developing programs in such a system (writing well-typed expressions) involves providing explicit proofs of certain equality judgments; the problem of type-checking cannot be delegated to a machine.

> And are universes characterised by initiality properties? Or are they only fixed points? It is interesting that the only two methods we have for the introduction of dependent (small) types are equality, or the use of (recursion into) a universe.
>
> And what *about* the non-mechanical nature of type-checking? Isn't that actually what we implement? What more do we expect than machine *assistance* in writing working programs? To expect anything more is just wishful thinking. At any rate, this is a point of view which it is difficult to criticise.

[INTENSIONAL.] On the other hand, if the equality relation is taken to be intensional (see also the Id relation in [22, pp. 61 – 67]), as in other publications of Martin-Löf's, then it has (especially at function types and in datatypes in which constructors may have functional parts) a strangely unmathematical character.

Whether equality is taken to be extensional or intensional, it requires special treatment in the design and implementation of type-checkers. Some basic choices in the formulation of type-theory hinge on the extent to which we can treat the definition of predicates (propositional functions or set-valued functions) on a par with the definition of functions of other kinds.

Moreover the computational or information content, or practical use of a proof of identity is dubious. What does one do with it?

Propositional identity is a *single* type constructor that is to make sense *uniformly* at arbitrary sets A. Through its set, argument, it somehow comprehends the whole notion of inductively defined set, and yet itself forms sets. (What does this really mean? Does $A \to \ldots$ "somehow comprehend the whole notion of inductively defined set"?)

[ALLEGATIONS GIVEN.] Whether equality is 'guilty' or 'innocent', it is at least interesting from the perspective of type theory to distinguish between the two forms of power-set functor, and to investigate the boundaries of what can be done without resort to a general propositional identity or equality relation. Abstention from unnecessary or profligate use of the equality relation is *not* like denying a boxer the use of their fists[11]. (Perhaps it is a little more like encouraging a southpaw to reserve their left fist for knock-out blows.)

If we abstain entirely from use of an equality relation, we cannot actually *say*, *i.e.* express with a small[12] proposition that a given element occurs in a family, or that one family includes another. However it *is* possible to say two things about a family with respect to a (full-blooded) predicate; firstly that a family is included in a predicate, and secondly that it 'meets' it (*i.e.* overlaps with, or has non-empty intersection with it, written $X \between Y$). These statements can be expressed by means of the following

[11] To allude to Hilbert's famous sound-bite *a propos* of the law of excluded middle.

[12] Of course we can do this using quantification over predicates: one way is to define Leibnitz equality; second order quantification can also be used in other ways than via Leibnitz equality.

propositions.

$$\begin{array}{lcl} \{\, b(i) \mid i \in I \,\} \subseteq \{\, b \in B \mid P(b) \,\} & = & (\forall\, i \in I) P(b(i)) \\ \{\, b(i) \mid i \in I \,\} \between \{\, b \in B \mid P(b) \,\} & = & (\exists\, i \in I) P(b(i)) \end{array}$$

It is then possible to express that one family is included in another by means of a large type, as in the following equivalent definitions. It is of course well-known that one can define an equality *type* between elements of a set by means of quantification over predicates, using so called 'Leibnitz' equality.

$$\begin{array}{rcl} \phi \subseteq \psi & = & (P : \mathbb{P}\,(B)) \to \;\; \psi \subseteq P \to \phi \subseteq P \\ & = & (P : \mathbb{P}\,(B)) \to \;\; \phi \between P \to \psi \between P \end{array}$$

Using a large type in the same way we can represent the notion of one family intersecting with or meeting with another.

$$\begin{array}{rcll} \phi \between \psi & = & (P : \mathbb{P}\,(B)) \to & \psi \subseteq P \to \phi \between P \wedge \\ & & & \phi \subseteq P \to \psi \between P \end{array}$$

[13]

2.2 Relations and transition structures

If we keep the two kinds of powerset distinct, there is a bifurcation or splitting in the notion of a binary relation between two sets A and B. On the one hand, we have 'full-blooded' relations, which are propositional functions (the semantic counterpart of a predicate) defined over the product set $A \times B$, or equivalently functions of type

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

On the other hand we have what might be called 'weak-blooded' relations, but I would prefer to call *transition structures*. These are functions of type

$$A \to \mathbb{F}\,(B)\ .$$

A transition structure can always be written in the form

$$(\lambda\, a)\langle T(a), (\lambda\, t) a[t] \rangle$$

where

$$\begin{array}{lcl} T & : & A \to Set \\ \_[\_] & \in & (\forall\, a \in A) T(a) \to B \quad . \end{array}$$

One can think of elements t of $T(a)$ as 'transitions' with source a and destination $a[t]$.

A transition structure is a 'computational' representation of a relation. We can use it, in combination with an environment that chooses and supplies the indices t, to pass from $a \in A$ for which $T(a)$ is non-empty to $a[t] \in B$. The price paid for this 'computationality' is awkwardness. A transition structure is an asymmetric (or

[13] Another way of expressing the inclusion relation between families is with a transition structure on families themselves; after all, transition structures can often replace full-blooded relations: that is to say, a particularly nice way of expressing a relation is with a transition structure and the equality relation. To avoid size problems, we have to define an isotope of the notion of family, localised to a given universe or family of sets in the sense that the sets which are used as index sets must belong to the universe. The resulting notion of inclusion is thus relativised or localised to such a universe. This is off-topic.

one-way-round) representation of a relation. Transition structures don't have a natural notion of converse; also they are closed under joins, but not meets.

Many everyday relations and operations on them are quite naturally modeled as transition structures and operations on them. When one thing is related to another, the 'reason why' can often be identified with an element of some data-type, whose elements witness the relation holding, and embody the computational core of the proof. For example, a reduction relation between expressions generated by some computation rules (*i.e.* schemas for rewriting expressions) is naturally indexed by the locations within an expression to be rewritten at which a computation rule is applied, together with an indication of the rule (in case there is any ambiguity). Many ordinal notation systems used in proof theory since Gentzen [12] (such as Cantor Normal Form for ordinals below ϵ_0) are naturally described in this way – the notation $\alpha[\_]$ used above for the indexing function of a family was suggested to me by its usage for fundamental sequences assigned to limit ordinals α.

Many operations that one habitually defines for 'full-blooded' relations, such as composition, reflexive and transitive closure, or the well-founded part of a relation make equally good, if not better sense when recast as operations on transition structures. However, some operations cannot be recast in this way, without essential use of singleton predicates. One counterexample is the converse operation; transition systems are 'one-way round'. Another is meets.

When comparing transition structures with full-blooded relations and with each other, the situation resembles comparison of families and predicates, lifted pointwise. We have a way to state (with a small proposition) that the image of a argument for transition structure is included in, or (dually) meets a predicate.

[PTs associated with a TS.] Associated with an transition structure is a pair of dual monotone predicate transformers $\{\phi\}$ and $[\phi]$. The first tells you what has to hold if you can choose the next transition and have to establish a given predicate in the next state, while the second (a form of 'weakest liberal precondition') tells you what has to hold if you have no choice as to the next transition, but have to ensure that it leads only to states satisfying a given predicate – if there are any transitions from the current state at all. The second predicate transformer is thus connected with partial correctness.

$$\begin{array}{l}\{\phi\}, [\phi] : \mathbb{P}\,(B) \to \mathbb{P}\,(A)\\ \{\phi\}(P : \mathbb{P}\,(B)) = \{\, a \in A \mid \phi(a) \between P \,\}\\ [\phi](P : \mathbb{P}\,(B)) = \{\, a \in A \mid \phi(a) \subseteq P \,\}\end{array}$$

We can formulate the idea that one transition structure is included in another, using a large type.

$$\phi \subseteq \psi \quad = \quad (P : \mathbb{P}\,(B)) \to [\psi](P) \subseteq [\phi](P)$$

Equivalently, one could define, again with a large type

$$\phi \subseteq \psi \quad = \quad (P : \mathbb{P}\,(B)) \to \{\phi\}(P) \subseteq \{\psi\}(P)$$

The intuition is that if $\psi \subseteq \phi$, then all predicates that include the image of ϕ for some argument also include the image of ψ for that argument. Similarly, all predicates that intersect with the image of ϕ for some argument also intersect with the image of ψ for that argument.

In fact we can interpret expressions in the following language as transition structures, so that the expected laws[14] hold with respect to the inclusion relation.

[14] Be less vague.

$\cup_i \phi_i$	Union
$\phi\,;\psi$	Sequential composition
$\langle f \rangle$	Map, assignment $f : A \to B$, *e.g.* $\langle \mathsf{id} \rangle = \mathsf{skip}$
$\lvert P \rvert$	Test, condition $P : \mathbb{P}\,(S)$.
ϕ^*	Reflexive-transitive closure

The interpretation is as follows. I defer the treatment of reflexive-transitive closure.

- $(\cup_i \phi_i) : A \to \mathbb{F}\,(B)$ where $\phi_i : A \to \mathbb{F}\,(B)$
  $(\cup_i \phi_i)(a) \triangleq \cup_i (\phi_i(a))$
- $(\phi\,;\psi) : A \to \mathbb{F}\,(C)$ where $\phi : A \to \mathbb{F}\,(B)$, $\psi : B \to \mathbb{F}\,(C)$
  $(\phi\,;\psi)(a) \triangleq \cup_{t \in T_\phi(a)} \psi(a[t]_\phi)$
- $\langle f \rangle : A \to \mathbb{F}\,(B)$ where $A \to B$
  $\langle f \rangle(a) \triangleq \{f(a)\}$
- $\lvert P \rvert : S \to \mathbb{F}\,(S)$ where $P : \mathbb{P}\,(S)$
  $\lvert P \rvert(s) \triangleq \{\, s \mid \_ \in P(s) \,\}$

Reflexive transitive closure $\phi^* : S \to \mathbb{F}\,(S)$ where $\phi : S \to \mathbb{F}\,(S)$

In this case, we define ϕ^* to be the least solution $\psi : S \to \mathbb{F}\,(S)$ of

$$\psi(s) \triangleq \{s\} \cup \bigcup_{t : T_\phi s} \psi(s[t]_\phi) \triangleq (\mathsf{skip} \cup (\phi\,;\psi))(s)$$

Another way of making the essentially same definition is to define first the index predicate to be the least solution $T : \mathbb{P}\,(S)$ of

$$T(s) = \{*\} + \{\phi\}(T, s)$$

Then we define the indexing function by well-founded (structural) recursion on the second argument. (The first is just a parameter.)

$$\begin{array}{lll} \_[\_] \triangleq & \mathsf{rec}\ f : (s : S) \to T(s) \to S\ . & \\ & (\lambda\, s, t)\ \ \mathbf{case}\ t\ \mathbf{of} & \\ & \qquad \mathsf{in}_{\mathsf{L}}\, * & \mapsto \quad s \\ & \qquad \mathsf{in}_{\mathsf{R}}\, \langle t_0, t' \rangle & \mapsto \quad f(s[t_0]_\phi, t') \end{array}$$

Other orders Beside the inclusion relation, there is another partial order of interest between transition structures, namely the following relational refinement relation. The relation $\phi \sqsubseteq \psi$ means that ψ refines ϕ in the sense that it is 'more defined', and 'more deterministic'. It can be defined from the inclusion relation using the domain operator $\mathrm{dom}\,\phi = \{\phi\}(All)$ and the restriction operator $\phi \restriction P = \lvert P \rvert\,;\phi$. (It lacks a general supremum operation, except for directed families.)

$$\begin{array}{l} \mathrm{dom}\,\phi \subseteq \mathrm{dom}\,\psi \\ \psi \restriction \mathrm{dom}\,\phi \subseteq \phi \end{array} \quad .$$

2.3 Predicate transformers and interaction structures

Just as there are two ways of analysing the notion of relation, namely as 'full-blooded' relations *versus* as transition structures, so there are (at least) two ways of analysing the notion of predicate transformer. On the one hand we have 'full-blooded' predicate transformers, that are functions from one power set to another. If we swap the arguments to an operator from predicates over B to predicates over A (nothing is essentially different)[15], what we have is a function from B to the type of predicates of predicates over A.

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

On the other hand we have what I called in section 1 *interaction structures*, which are functions of type

$$A \to \mathbb{F}\,(\mathbb{F}\,(B))\ .$$

An interaction structure can always be written in the form

$$(\lambda\, a)\langle C(a), (\lambda\, c)\langle R(a,c), (\lambda\, r)a[c/r]\rangle\rangle$$

where

$$\begin{array}{lcl} C & : & A \to Set \\ R & : & (\forall\, a \in A)C(a) \to Set \\ \_[\_/\_] & \in & (\forall\, a \in A)(\forall\, c \in C(a))R(a,c) \to B \quad . \end{array}$$

Associated with an interaction structure given as above there is a pair of dual monotone predicate transformers which tell how to ensure or establish a given predicate in the next state, from the user's and system's respective sides of the interface.

$$\begin{array}{l} \Phi^{\circ}, \Phi^{\bullet} : \mathbb{P}\,(B) \to \mathbb{P}\,(A) \\ \Phi^{\circ}(P : \mathbb{P}\,(B)) = \{\, a \in A \mid (\exists\, c \in C(a))(\forall\, r \in R(a,c))P(a[c/r]) \,\} \\ \Phi^{\bullet}(P : \mathbb{P}\,(B)) = \{\, a \in A \mid (\forall\, c \in C(a))(\exists\, r \in R(a,c))P(a[c/r]) \,\} \end{array}$$

In fact the definitions can be written as follows.

$$\begin{array}{l} \Phi^{\circ}(P : \mathbb{P}\,(B)) = \{\, a \in A \mid (\exists\, c \in C(a))Rn(a,c) \subseteq P \,\} \\ \Phi^{\bullet}(P : \mathbb{P}\,(B)) = \{\, a \in A \mid (\forall\, c \in C(a))Rn(a,c) \mathbin{\not\subseteq} P \,\} \end{array}$$

A proof that $\Phi^{\bullet}(All)$ holds in a state a is in essence a function in $(\forall\, c \in C(a))R(a,c)$ that the system can use as a strategy to respond to any command issued in state a. A proof of $\Phi^{\circ}(P, a)$ can be used by the user as a strategy to ensure that the system terminates only in a state satisfying P. On the assumption that the system makes use of a proof of $\Phi^{\bullet}(All, a)$ as a strategy for responding to commands, a proof of $\Phi^{\circ}(P, a)$ can be used by the user as a strategy actually to establish P, not just to ensure that the system terminates only in a state such that P. The predicate transformer $\_^{\circ}$ is the weakest *liberal* precondition operator for the interaction structure Φ. It is much more important than $\_^{\bullet}$. Instead of $\Phi^{\circ}(P)$, I write simply $\Phi(P)$; the interaction system is coerced to a predicate transformer by an implicit $\_^{\circ}$.

We can formulate the idea that one interaction structure is refined by another, using a large type.

$$\Phi \sqsubseteq \Psi \quad = \quad (P : \mathbb{P}\,(B)) \to \ \Phi^{\circ}(P) \subseteq \Psi^{\circ}(P)$$

[15] Precisely this point is discussed in Back and von Wright [4, page 250]

3 Refinement calculus

The refinement calculus is an elegant and for some purposes quite practical 'algebra of contracts' ([4, section 1.4]).

The forms of contract in the refinement calculus that I shall discuss are shown in the table below.

0	– abort	1	– magic
$\Phi \sqcup \Psi$	– binary angelic choice	$\Phi \sqcap \Psi$	– binary demonic choice
$\sqcup_i \Phi_i$	– indexed angelic choice	$\sqcap_i \Phi_i$	– indexed demonic choice
$\{R\}$	– angelic relational update	$[R]$	– demonic relational update
	skip – no-op		
	$\Phi \,;\, \Psi$ – sequential composition		

I shall also discuss two forms of recursion, which are respectively inductive and coinductive. The 'bullet' notation $\Phi^\bullet$, which is a kind of dual or inversion will be discussed later.

Φ^*	– reflexive-transitive closure	$\mu\, \Psi.\, \mathsf{skip} \sqcup (\Phi \,;\, \Psi)$
Φ^∞	– interior	$\nu\, \Psi.\, \mathsf{skip} \sqcap (\Phi^\bullet \,;\, \Psi)$

[CAUTION: The notation in the table above clashes with the notation of Back and von Wright's book on the refinement calculus, where one finds
$\Phi^\infty = \mu\Psi.\, \Phi \,;\, \Psi$,
Φ^* similar to Φ^∞ in my notation,
$\Phi^\omega = \mu\Psi.\, \mathsf{skip} \sqcap \Phi \,;\, \Psi$ which is a 'μ-full' version of Φ^∞ in my notation.]

These expressions are interpreted as predicate transformers. Under this interpretation, they satisfy a host of elegant algebraic laws (something close to, but not quite a quantale – we don't have commutation of join with sequential composition – see [26, definition 12.2.1, p. 183]), which can be expressed in terms of a partial order $\sqsubseteq$ called refinement, interpreted as pointwise inclusion between predicate transformers. By pointwise inclusion (written $\sqsubseteq$) is meant the following universal quantification over predicates.

$$\Phi \sqsubseteq \Psi \equiv \forall X : \mathbb{P}\,(S).\, \Phi(X) \subseteq \Psi(X)$$

The refinement calculus in this abstract form serves as a basis on which to define a number of more familiar programming constructs – local variables, assertions, assignments, conditionals, procedures, recursion, loops and so on. The programming constructs satisfy laws familiar from Hoare logic. They produce *feasible* predicate transformers in a certain sense implying a kind of executability. In fact the forms $\sqcup_i \Phi_i$, $\sqcap_i \Phi_i$, $[R]$, $\{R\}$ and $(\Phi \,;\, \Psi)$ form a sufficient basis.

Interpretation as interaction systems There are two ways to interpret an interaction structure as a predicate transformer. If we are given an interaction structure $\Phi : S \to \mathbb{F}\,(S')$, then we can define two monotone predicate transformers as follows.

$$\begin{aligned}
&\Phi^\circ, \Phi^\bullet : \mathbb{P}\,(S') \to \mathbb{P}\,(S) \\
&\Phi^\circ(P, s) = \exists c \in C(s).\, \forall r \in R(s, c).\, P(s[c/r]) \\
&\Phi^\bullet(P, s) = \forall c \in C(s).\, \exists r \in R(s, c).\, P(s[c/r])
\end{aligned}$$

Of these, the first (Φ°) is more fundamental: it enjoys better properties, and the second can be defined as a special case of the first, using a form of dualisation or inversion.

We show in the following table how to interpret the basic constructions of the refinement calculus as operations on interaction structures. In the case of the relational updates, I have taken the argument to be a transition system, which is the more general course.

$\sqcup_i \Phi_i : A \to \mathbb{F}(\mathbb{F}(B))$
where $\Phi_i : A \to \mathbb{F}(\mathbb{F}(B))$
$C(a) = (\exists\, i \in I) C_i(a)$
$R(a, \langle i, c\rangle) = R_i(a, c)$
$a[\langle i, c\rangle / r] = a[c/r]_i$

$\sqcap_i \Phi_i : A \to \mathbb{F}(\mathbb{F}(B))$
where $\Phi_i : A \to \mathbb{F}(\mathbb{F}(B))$
$C(a) = (\forall\, i \in I) C_i(a)$
$R(a, f) = (\exists\, i \in I) R_i(a, f(i))$
$a[f/\langle i, r\rangle] = a[f(i)/r]_i$

$\{\phi\} : A \to \mathbb{F}(\mathbb{F}(B))$
where $\phi : A \to \mathbb{F}(B)$
$\{\phi\}(a) \triangleq \{\, \{a[t]_\phi\} \mid t \in T_\phi(a) \,\}$

$[\phi] : A \to \mathbb{F}(\mathbb{F}(B))$
where $\phi : A \to \mathbb{F}(B)$
$\{\phi\}(a) \triangleq \{\phi(a)\}$

$(\Phi\,;\Psi) : A \to \mathbb{F}(\mathbb{F}(C))$
where $\Phi : A \to \mathbb{F}(\mathbb{F}(B))$
$\Psi : B \to \mathbb{F}(\mathbb{F}(C))$
$(\Phi\,;\Psi)(a) \triangleq \{\, \cup_{r \in R_\Phi(a,c)} Rn_\Psi(a[c/r]_\Phi, f(r)) \mid \langle c, f\rangle \in \Phi(C_\Psi, a) \,\}$

The interpretation of a refinement calculus expression as a predicate transformers can be factored into the following two steps; first interpret it as an interaction structure, and then apply the operator $\_^\circ$.

Note: an inclusion $\phi \subseteq \psi$ between transition structures is equivalent to a converse inclusion $[\psi] \sqsubseteq [\phi]$ between interaction systems. In fact $\phi^\circ = [\phi]$ and $\phi^\bullet = \{\phi\}$. So

$$\phi \subseteq \psi \equiv [\psi] \sqsubseteq [\phi] \equiv \{\phi\} \sqsubseteq \{\psi\}\ .$$

Note: the refinement calculus notations are very handy, even outside of programming. For example, if $\langle S : Set, \gamma : S \to \mathbb{F}(S)\rangle$ is a homogeneous transition structure, then the predicate $[\gamma]^*(None)$ describes the accessible (well-founded) points of the coalgebra.

Note: there are two interesting monotone predicate transformers which seem to lack a natural representation as Φ-structures. Let $f : A \to B$. Then $\langle f\rangle = f^{-1} : \mathbb{P}(B) \to \mathbb{P}(A)$. If we define $\exists_f, \forall_f : \mathbb{P}(A) \to \mathbb{P}(B)$ by

$$\exists_f(P) = \{\, b \in B \mid f^{-1}\{b\} \not\subseteq P \,\}$$
$$\forall_f(P) = \{\, b \in B \mid f^{-1}\{b\} \subseteq P \,\}$$

then we have

$$P \subseteq f^{-1}(Q) \equiv \exists_f(P) \subseteq Q$$
$$f^{-1}(P) \subseteq Q \equiv P \subseteq \forall_f(Q)$$

Intuitively, $\exists_f(P)$ is the range of values attained by $f \restriction P$, while $\forall_f(P)$ consists of those elements of the range which are attained only as values of f for arguments satisfying P. Neither of these predicate transformers have an obvious representation as an interaction structure (unless we allow singleton predicates, or to put it another way the equality relation).

Its foundations rest on impredicative higher order classical logic[16].

Recursion In the refinement calculus, we may form new expressions using variable binding operations μ and ν, which are defined by recursion; μ returns the least fixed-point of a monotone function, and ν its greatest. I call these *recursive* expressions.

The foundation for the semantics of recursive expressions (involving both μ and ν) is given in two ways in Back and von Wright's book; firstly ([4, pp. 317–321]) μ can be defined as a meet and ν as a join via the Knaster-Tarski theorem; second ([4, pp. 321–322]) via a form of Hartog's Lemma which can be adapted to justify the use of even non-monotone inductive definitions; in this case μ is defined as a join and ν as a meet.

We are instead going to introduce only two special forms of μ and ν.

(minimal)	(maximal)
$\dfrac{p : P \cup \Phi(C) \subseteq C}{p^* : \Phi^*(P) \subseteq C}$	$\dfrac{p : C \subseteq P \cap \Phi(C)}{p^\infty : C \subseteq \Phi^\infty(P)}$
$\mathsf{in} : P \cup \Phi(C) \subseteq C$ where $C = \Phi^*(P)$	$\mathsf{ex} : C \subseteq P \cap \Phi(C)$ where $C = \Phi^\infty(P)$

$$c^* \cdot \mathsf{in} = c \cdot (1_P \cup \Phi(c^*)) \qquad \mathsf{ex} \cdot c^\infty = (1_P \cap \Phi(c^\infty)) \cdot c$$

Recursion rather than iteration:

$$\frac{p : P \cup \Phi(\Phi^*(P) \cap C) \subseteq C}{p^* : \Phi^*(P) \subseteq C} \qquad \frac{p : C \subseteq P \cap \Phi(\Phi^\infty(P) \cup C)}{p^\infty : C \subseteq \Phi^\infty(P)}$$

$$c^* \cdot \mathsf{in} = c \cdot (1_P \cup \Phi\langle 1_{\Phi^*(P)}, c^* \rangle) \qquad \mathsf{ex} \cdot c^\infty = (1_P \cap \Phi[1_{\Phi^\infty(P)}, c^\infty]) \cdot c$$

Least fixed points The operator $\_^*$ which makes the reflexive and transitive closure Φ^* of a homogeneous interaction structure $\Phi : S \to \mathbb{F}(\mathbb{F}(S))$.

[WARNING: the $*$ notation I use clashes with the notation in back and von Wright.]

This satisfies the following minimality property.

$$\begin{array}{l} \mathsf{skip} \sqcup (\Phi \,;\, \Phi^*) \sqsubseteq \Phi^* \\ \mathsf{skip} \sqcup (\Phi \,;\, \Psi) \sqsubseteq \Psi \Rightarrow \Phi^* \sqsubseteq \Psi \end{array} \quad .$$

Given $\Phi : S \to \mathbb{F}(\mathbb{F}(S))$, we define $\Phi^* : S \to \mathbb{F}(\mathbb{F}(S))$ in two steps. First we define the index set to be the least solution $C : \mathbb{P}(S)$ of

$$C(s) = \{\, \mathsf{exit} \,\} + \{\, \mathsf{call}\, \langle c, f \rangle \mid \langle c, f \rangle \in \Phi^\circ(C, s) \,\}$$

[16] Only certain parts depend on classical logic; for example the parts connected with the complementation operator, such as the dual of a predicate transformer. Impredicativity, via the Knaster-Tarski theorem is used to handle recursion.

The family of states associated with a state s and a command $p : C(s)$ is then defined by well-founded (structural) recursion on p.

$$\begin{array}{lll} Rn & = & \mu\, X : (s : S) \to C(s) \to \mathbb{F}\,(S)\ . \\ & & (\lambda\, s, p)\ \mathbf{case}\ p\ \mathbf{of} \\ & & \qquad \mathsf{exit} \quad \mapsto \quad \{s\} \\ & & \qquad \mathsf{call}\,\langle c, f\rangle \quad \mapsto \quad \cup_{r:R_\Phi(s,c)} X(s[c/r]_\Phi, f(r)) \end{array}$$

The intuition is to think of Φ as an instruction set. From these, we build up programs in which the instructions are 'composed' sequentially; this is a kind of plugging together, or wiring up. There is an empty composite, which is the exit program.

Think of the commands as 'male', plugging into 'female' sockets, and of each command as containing a family of leads terminating in female sockets: now 'wire together' or compose zero or more of these components, without any looping. Each command (positively) or socket (negatively) has a shape/sort, and these must be the same if a command is to fit a socket.) The 'responses' to these programs are sequences of responses in Φ that lead in the end to an exit command, rather than (as might be the case) to a command for which there is no response. These sequences of responses are logs of complete execution traces, for executions that terminate successfully. (The exit command could well be written with a 'tick' $\surd$, as in Hoare's CSP notation.)

The next state function gives the state in which an exit, leaf or 'leave' command is executed.

We can imagine a programmer-friendly notation in which $\mathsf{call}\,\langle c, f\rangle$ is written with a bound variable resembling the target of an assignment

$$\begin{array}{ll} \mathbf{do} & r \leftarrow \mathsf{call}\, c \\ \ ; & f(r) \end{array}$$

and exit is written $\mathbf{done}$.

Φ^* is a kind of closure under of the instructions in Φ under sequential composition, or formation of transactions. (A transaction is a program which appears to execute in isolation, for which there is a notion of successful completion.)

Greatest fixed points We would like to construct from any $\Phi : S \to \mathbb{F}\,(\mathbb{F}\,(S))$ an interaction structure Φ^∞ analogous to Φ^*, but satisfying a maximality rather than a minimality property. The operation $\_^\infty$ should be an interior rather than a closure operator.

[WARNING: $\_^\infty$ is not Back and von Wright's notation. What I write Φ^∞ they seem to write Φ^* ([4][page 347]), and call it weak iteration. What I write Φ^* they call the dual of weak iteration, or iterative choice ([4][page 378]). The treatment of recursion in the refinement calculus is obscure to me, and I find many things puzzling in it.]

$$\begin{array}{l} \Phi^\infty \sqsubseteq \mathsf{skip} \sqcap (\Phi^\bullet\, ;\, \Phi^\infty) \\ \Psi \sqsubseteq \mathsf{skip} \sqcap (\Phi^\bullet\, ;\, \Psi) \Rightarrow \Psi \sqsubseteq \Phi^\infty \end{array}$$

The blob $\_^\bullet$ refers to the client-server inversion of Φ defined somewhere later. I don't know whether to include it in the definition of Φ^∞, and am probably inconsistent about it.

Unfortunately, I don't see how to do construct Φ^∞ without crossing a 'size' barrier.

At this point, I simply suppose there is some way to do thus. If there is, I probably don't want to know exactly how to do it[17].

[17] People applying mathematics (as in signal processing) cheerfully assume that there'll be some way

Need for coinduction What do we want the 'infinite things' for? A possible answer is that we would prefer to imagine that there is just *one* coalgebra, which is universal, in the sense that 'all' pointed coalgebras, in some safe sense, can be represented as states in it. So we can forget about all the different state spaces S, with all the different C's, R's and n's, and pretend that there is just one. It makes methodological sense.

This pretence should be safe, in the sense that we never need any vicious circularity. (Something as weak as a recursion theorem.) Somehow, we can throw all the instruction sets we can think of into one – there's always room for more. Accumulation.

Similarly (refers to definition of simulation later), we would prefer to imagine that there is just *one* simulation, which is can be treated as if it were maximal. Somehow we throw all the different possibilities into one – there is some kind of unioning going on, but we probably don't want to know exactly what the subterfuge is.

Justification of coinduction One approach to dealing with infinite objects is to use the impredicative existential quantifier in the form

$$\{\, s : S \mid \exists X.(X \subseteq \Phi(X)) \wedge X(s) \,\} = \bigcup \{\, X : \mathbb{P}\,(S) \mid X \subseteq \Phi(X) \,\}\ .$$

The elements of a type $\exists X.(X \subseteq \Phi(X)) \wedge X(s)$ are pointed coalgebras for Φ. (The category here is of predicates over S, with morphisms from A to B functions $f : S \to S$ such that $A \subseteq f^{-1}(B)$, under extensional equality.) There may be a general argument that limited use of the second order existential quantifier is harmless; one thing to discover is what the limitations amount to.

(In a sense, there must be a certain limitation on the use of the second order existential quantifier that renders it harmless to predicativity. The first question is then: what limitation?)

4 System and user

[TRANSACTION.] The 'normal' kind of program is something with a designated entry point, which is run until it exits, that may interact with resources.

Instead of 'exit', we could have 'commit'. (As if the updates to resources were held pending, invisible to others. A transaction.)

Another idea is to treat this as a special case; in the general case that there are many entry-points, and these are activated one after each other in response to a sequence of activation codes (remembering the state) by the run-time system. This is something like a server program or action system: a collection of programs of the first kind in a loop. A transaction server.

[INVERSE.] This is probably connected with the idea that parts of the run-time library are concerned with supporting one or more server-interfaces: get first command, reply to last and get another (or, more handshaken, accept/entry giving commands, reply/return returning results); part is concerned with the client-interface.

---

to make sense of some useful calculus or artefact, such as distribution functions like Dirac's delta whose measure is concentrated at a point. Probably a satisfactory foundation for distribution functions took a long time, maybe a hundred years, and required a deep rethinking of the foundations of measure theory. Meanwhile, the applied mathematicians continue to make use of a formal calculus of distribution functions, in the cheerful expectation that the considerations that make sense of it will be of nightmarish subtlety, of interest only to the truely obsessed. The calculus works so well that there *must* be some way to make sense of it.

Somehow it is $a \oplus b^{\perp}$, where a is the client interface, and $b^{\perp}$ is the inverse of the server interface b. (Many such expressions in papers on game semantics.) But this is not very 'structured': nothing says that things are kicked off by a command to a.

In reality, we don't have a sharp separation between two kinds of programs; most programs have a 'dual' aspect, in that they require one interface, and on this provide or implement another. My idea is that such a program is a simulation, or more precisely, a proof that a simulation relation obtains between the initial states of the two interfaces. Somehow I have to reconcile this with the possibility of inverting an interface, so that the rôles of the agents are exchanged. That means we have one big (but bi-partite) client interface.

[TWO SIDES] There are two sides to an interface of the kind we are considering, namely the user's and the system's. The notion of a program (other words: strategy, script) makes sense on both sides.

On the one hand there are (user-side) programs which when they are run (carried out or performed), the agent issues a command, waits for a response, then when a response is obtained passes control to an appropriate continuation program that in general depends on the response. These are *user programs*[18].

On the other hand there are (system-side) programs which when they are run (carried out or performed), the agent waits for a command from a user program, then after some internal calculation returns a result code to the agent running the user program and passes control to an appropriate continuation program. Both the result code and the destination of control may depend on the command. These are *system programs*[19].

A user program tells the user what commands to issue, and how to continue if/when there results have been returned. A system program tells the system how to deal with commands, meaning what results to return, and how to react to the next command.

It should appear as if returning the response, and moving to the next state are events that occur simultaneously.

(It is possible that the response can be lost.)

User programs The environment for which a user program is written is sometimes called a 'run-time system'. The run-time system makes available a library (*i.e.* organised collection) of procedures to (for example) read and write characters (using paper tape for example), read the time from a time service, read successive entries in a pseudo-random sequence, read the Geiger counter, raise the under-carriage, rotate the rudder through 15 degrees, *etc.*. The question is: what is the logical form of the specification (for someone writing a user program that uses it, or a system program that implements it) of the interface between the user program and the run-time system?

It appears that two predicate transformers are involved. These are akin to the two predicate transformers which Dijkstra suggests (see for example [10, p. 127]) together represent the semantics of an imperative 'batch' program. By a batch program I mean one for a single interaction (or one which appears as an atomic single interaction), where there is an initial state and (perhaps) a final state.

wlp Gives for any postcondition G (goal predicate) the weakest predicate of an initial state which guarantees that execution terminates only in a state satisfying that predicate. This is concerned with (so-called) *partial* correctness[20]. It is not so

[18] or Moore machines

[19] or Mealy machines

[20] Partial correctness of a program c with respect to a precondition predicate P and a postcondition predicate Q, which was originally written by Hoare $\{P\}\, c\, \{Q\}$, is equivalent to $P \Rightarrow \mathbf{wlp}_c(Q)$

much a matter of bringing it about that P, but rather of evading $P^{\complement}$. (There is some kind of double negation here.)

wp Gives for any postcondition (goal predicate) the weakest predicate of an initial state which guarantee that execution terminates in a state satisfying that predicate.

Dijkstra remarks that of these predicate transformers, it is **wlp** which is more basic; if we know it, then all that remains is only the termination condition $\mathbf{wp}(All)$. We can replace $\mathbf{wp}$ by a single predicate $\mathbf{wp}(All)$.

$$\mathbf{wp}(P) = \mathbf{wp}(All) \cap \mathbf{wlp}(P)$$

[RELATE SAFETY AND LIVENESS.] The specification of a command-response interface consists of two parts: a safety specification (for the client), and a liveness specification (for the server).

The safety specification is given by an initial predicate and a next-state relation or (more generally) predicate transformer. It tells the user how to use the system so as to stay out of bad situations. However, the system might get into a deadlocked state, and fail to respond to any command. The interface just stays (and will stay forever) in the launch state.

The liveness specification is given by a predicate representing those states in which (any proposed) interaction is guaranteed to terminate – or at any rate, a failure to terminate is the fault of the computer, not the program. It tells the user how to use the system to bring about good situations.

The point of separating the specification into a safety part and a liveness part: they have different logical forms; different techniques are used to deal with them.

The point of separating the specification into system and environment is that we can then describe both the behaviours in which the system behaves correctly (misbehaves only after the user misbehaves) as well as the behaviours in which both the system and environment behave correctly.)

We will denote the two predicate transformers IO and OI.

IO is reflexively and transitively closed, which is equivalent to $IO^* \sqsubseteq IO$.

The operator IO is a closure operator.

$$\begin{array}{l} X \subseteq IO(X) \\ \wedge\ X \subseteq IO(Y) \Rightarrow IO(X) \subseteq IO(Y) \end{array}$$

The operator OI is, dually, an interior operator.

$$\begin{array}{l} OI(X) \subseteq X \\ \wedge\ OI(X) \subseteq Y \Rightarrow OI(X) \subseteq OI(Y) \end{array}$$

It is not required that IO is inductively defined (*i.e.* Φ^* for some interaction structure Φ), nor that OI is coinductively defined (*i.e.* Φ^∞ for some Φ). It is not even required that these operators 'sandwich' a Φ[21].

The first predicate transformer IO tells the user how to make *safe* use of the resource. Applied to a postcondition P, it gives the strongest (least) predicate Q weaker than (including) P which ensures that an interaction initiated in a state satisfying Q terminates successfully (effectively) only in a state satisfying P. It does not guarantee that the interaction *will* terminate successfully; $IO(None)$ need not be $None$; $IO(None)$

[21] This means something like: $\Phi^* \subseteq IO$, $OI \subseteq \Phi^\infty$.

holds precisely when there are (non-terminating, or 'winning') commands for which the system has no response; the system 'loses', or deadlocks. IO is not strict.

The second predicate transformer OI tells the user how to make *effective* use of the resource. It gives for any predicate P the weakest invariant Q stronger than P which ensures that an interaction initiated in a state satisfying Q terminates successfully (effectively) in a state satisfying P. It does not guarantee that there are any such commands. There may be no commands that the user can issue in a given state – for example a state reached when the user has signalled that it has no further use for the resource. So $OI(None)$ need not be $None$. The user can lose, or be deadlocked.

Each individual predicate transformer is not very exciting; it is too weak. It is the conjunction that is important: state regions in which 'nobody loses'.

Here, an invariant is a predicate that holds perpetually.

Deadlocking is connected with some kind of concavity in the state space. But states from which it is possible to deadlock the system are convex.

[FOLLOWING: NEEDS INITIALISING GUARDS.]

User programs The user program is something of type $P \subseteq IO(Q)$, *i.e.* something which when run from a state in which P holds terminates only (if at all) in states which satisfy Q. As it were, it does not necessarily establish Q, only evades $Q^{\complement}$ (something bad we want to avoid), where $Q^{\complement}$ is the complement of Q. If P is an invariant maintained by the system, in the strong sense that the system executes a program which responds to any command so that the invariant holds in the new state (which must exist), if it held in the old (so there is a kind of liveness guarantee, if the mechanism executing the system eventually performs all performable instructions), then the user program can actually be used to establish Q, so long as it is carried out to the bitter end.

To show $P \subseteq IO(Q)$ is to show that for all predicates X which satisfy $Q \cup \Phi(X) \subseteq X$ we have $P \subseteq X$,

To show $P \not\subseteq OI(Q)$ is to provide a predicate X which satisfies $X \subseteq Q \cap \Phi(X)$ and a state which satisfies both P and X.

System programs A system program is something of type $P \not\subseteq OI(Q)$, *i.e.* a predicate X such that $X \subseteq Q \cap \Phi(X)$ together with a proof that $P \not\subseteq X$.

Initialising guards

$$A : \mathbb{P}(S)$$
$$B : (\forall\, s : S) \to A(s) \to \mathbb{P}(S)$$

We call A the guard and B the effect. Then

$$client : (\forall\, s : S, p : A(s))\Phi^*(B(s,p))$$
$$server : (\exists\, s : S, p : A(s))\Phi^\infty(B(s,p))$$

Note that the effect depends on (can refer to) the proof that the guard holds. This will typically be an existential statement, so that the effect can refer to 'things in the initial state', by projecting them out from the proof of the guard holding. We have to support the common practice of distinguishing initial values by a $\__0$ or to final values by a $\_'$.

Now what happens to the execution rule?

5 Running system programs as user programs

There are two ways of running system programs as user programs.

- It is possible, by grace of the the axiom of choice, to 'invert' system programs, so that they too can be regarded as issuing commands. (This is, we can run them as user programs.) The inverse (in this sense) of the interaction structure $\Phi : A \to \mathbb{F}\,(\mathbb{F}\,(B))$ is the following interaction system $\Phi^\bullet : A \to \mathbb{F}\,(B)$.

$$\begin{array}{lcl} C & = & \{\, a : A \mid (\forall\, c : C_\Phi(a)) R_\Phi(a, c) \,\} \\ R(a, \_) & = & C_\Phi(a) \\ a[f/c] & = & a[c/f(c)]_\Phi \end{array}$$

  The commands of this interaction structure are in effect entire arrays containing the 'precomputed' responses of Φ for any possible command, while the responses (which are the commands of Φ) select one of these precomputed responses.

  We have classically that $\Phi^\bullet$ is dual to Φ.

  The bullet operator $\_^\bullet$ provides a way to run system programs as user programs. It is not fully satisfactory, because it is (at any rate *prima facie*) unrealistically 'eager': the response of the server to any request is precomputed, albeit lazily, by the runtime. It is also unsettling that the response sets do not depend on the commands. However, one thing to be said in favour of this 'inversion' is that the state space is unaltered.

- Another 'inverse' construction that is closer to what one does in practice (in operating systems) enriches the state space visible to the system to contain a copy of the last command, if there was one. Starting with an interaction structure $\Phi : S \to \mathbb{F}\,(\mathbb{F}\,(S))$, one enlarges the state space to contain, either an indication that no command has been issued, or the last such command. More precisely, one changes to the state-space $S' = S + (\exists\, s \in S) C_\Phi(s)$ (which is equivalent to $(\exists\, s \in S)(\{*\} + C_\Phi(s))$, which can be written $(\exists\, s \in S) Maybe(C_\Phi(s)))$, and constructs a new interaction structure with type $S' \to \mathbb{F}\,(\mathbb{F}\,(S'))$ as follows.

$$\begin{array}{lcl} C(\mathsf{in}_{\mathsf{L}}\, s) & = & \{\, \mathsf{GetFirst} \,\} \\ R(\mathsf{in}_{\mathsf{L}}\, s, \mathsf{GetFirst}) & = & C_\Phi(s) \\ \mathsf{in}_{\mathsf{L}}\, s[\mathsf{GetFirst}/c] & = & \mathsf{in}_{\mathsf{R}} \langle s, c \rangle \\ C(\mathsf{in}_{\mathsf{R}} \langle s, c \rangle) & = & \{\, \mathsf{GetNext}\, r : R_\Phi(s, c) \,\} \\ R(\mathsf{in}_{\mathsf{R}} \langle s, c \rangle, \mathsf{GetNext}\, r) & = & C_\Phi(s[c/r]_\Phi) \\ \mathsf{in}_{\mathsf{R}} \langle s, c \rangle[\mathsf{GetNext}\, r/c'] & = & \mathsf{in}_{\mathsf{R}} \langle s[c/r]_\Phi, c' \rangle \end{array}$$

  The initial state of the new system should be $\mathsf{in}_{\mathsf{L}}\, s_0$.

  In this way we can run a system program as a user program with a slightly different interface.

  It is worth noticing that this introduces state-dependency in an essential way: the set of commands depends on the current state, since the commands are actually responses to a pending command.

  The system gets going by requesting the first request from the client; after that each request for the next request from the client "piggy-backs" the response to the one that was delivered last.

The piggy-backing might be regarded as an optimisation. Here is a 4-message de-optimisation. There is something canonical about this. We have a 'hello', the inverse of the server interaction (2 messages), then a 'goodbye'. There are two 'handshakes'.

$$\begin{array}{lcl} C(\mathsf{in}_{\mathsf{L}}\, s) & = & \{\, \mathsf{Accept} \,\} \\ R(\mathsf{in}_{\mathsf{L}}\, s, \mathsf{Accept}) & = & \{\, \mathsf{Call}\, c \mid c \in C_\Phi(s) \,\} \\ \mathsf{in}_{\mathsf{L}}\, s[\mathsf{Accept}/\mathsf{Call}\, c] & = & \mathsf{in}_{\mathsf{R}}\langle s, c\rangle \\ C(\mathsf{in}_{\mathsf{R}}\langle s, c\rangle) & = & \{\, \mathsf{Return}\, r \mid r \in R_\Phi(s, c) \,\} \\ R(\mathsf{in}_{\mathsf{R}}\langle s, c\rangle, \mathsf{Return}\, r) & = & \{\, \mathsf{Ack} \,\} \\ \mathsf{in}_{\mathsf{R}}\langle s, c\rangle[\mathsf{Return}\, r/\mathsf{Ack}] & = & \mathsf{in}_{\mathsf{L}}\, s[c/r]_\Phi \end{array}$$

In fact it would probably be better to give the state-space as a data-type with constructed forms $\{\, \mathsf{Idle}\, s \mid s \in S \,\}$ and $\{\, \mathsf{Pending}\, s\, c \mid s \in S, c \in C(s) \,\}$

$$\begin{array}{lcl} C(\mathsf{Idle}\, s) & = & \{\, \mathsf{Accept} \,\} \\ R(\mathsf{Idle}\, s, \mathsf{Accept}) & = & \{\, \mathsf{Call}\, c \mid c \in C_\Phi(s) \,\} \\ (\mathsf{Idle}\, s)[\mathsf{Accept}/\mathsf{Call}\, c] & = & \mathsf{Pending}\, s\, c \\ C(\mathsf{Pending}\, s\, c) & = & \{\, \mathsf{Return}\, r \mid r \in R_\Phi(s, c) \,\} \\ R(\mathsf{Pending}\, s\, c, \mathsf{Return}\, r) & = & \{\, \mathsf{Ack} \,\} \\ (\mathsf{Pending}\, s\, c)[\mathsf{Return}\, r/\mathsf{Ack}] & = & \mathsf{Idle}\, s[c/r]_\Phi \end{array}$$

There is something to prove about this inverse construction; I don't quite see what it is. It is about the relationship between $\Phi^\bullet$ and $(\Phi' ; \Phi')^\circ$, the second of which has a richer state-space, which is projectible onto the first. Perhaps it is this:

$$\Phi^\bullet = i^{-1} ; \Phi' ; \Phi' ; p^{-1}$$

Here $i : S \to S'$ maps s to $\langle s, * \rangle$ and $p : S' \to S$ is a projection of $S' = (\exists\, s \in S)\{*\} + C(s)$ to S, such that $p \cdot i$ is the identity on S.

'Janus' interfaces Somehow related to the question of running servers as clients or vice versa is the notion of an interface as a *pair* of states $\langle a_0, b_0\rangle \in A \times B$, given two state spaces $A, B : Set$ and maps $\phi : A \to \mathbb{F}(B)$ and $\psi : B \to \mathbb{F}(A)$. One can think of such an object as a pair of interfaces – it certainly determines a pair of pointed interaction structures $\langle A, \Phi, a_0\rangle$ and $\langle B, \Psi, b_0\rangle$ where

$$\begin{array}{l} \Phi(a) = \{\, \{\, \psi(t') \mid t' \in T_\psi(a[t]_\phi) \,\} \mid t \in T_\phi(a) \,\} \\ \Psi(b) = \{\, \{\, \phi(t') \mid t' \in T_\phi(b[t]_\psi) \,\} \mid t \in T_\psi(b) \,\} \end{array}$$

Such a structure is reminiscent of a Conway game (in which it is not yet known whether 'left' or 'right' is to begin); but while with Conway games evolution of states must eventually terminate, with Janus-structures such a sequence of states may proceed indefinitely.

There is a natural 'converse' operation on such objects, reminiscent of the representation of (signed) integers by pairs of naturals, and the minus operation. The definition of converse is a form of unfold, or corecursion.

These 'Janus' interfaces may be of interest, because the components we usually have to write have two interfaces (high-level and low-level). At one interface the component appears to be a server; at the other it appears to be a client.

The chief problem with this notion is that it is far from clear what we should require for initial states (one? two?). What too about morphisms?

6 Execution rule

What will the programming environment (ie. run-time library) look like when we are writing a component program in type theory? By a component program I mean one which makes use of certain services in a run-time library or system call interface, and makes available a service or services of some other 'added value' kind. So it has a positive and a negative interface - as it were two poles, or an anode and cathode.

Some speculations..

We might abstract the entire programming environment (or library interface) to a pair of predicate transformers IO and OI which are formally closure and interior operations respectively. The predicate transformers IO and OI typically arise as Φ^* and Φ^∞ for some Φ.

In Sambin's words, the gist of $IO(A)$ is 'I want to bring about A' (a kind of goal, or liveness requirement, that one positively wants to bring about), while the gist of $OI(B)$ is 'I want to stay within B' (a kind of safety requirement, which one wants to stay inside). Violation of a requirement of the first kind is a sin of omission, while violation of the second is a sin one actually commits.

The system programmer (who implements the run-time interface) writes a program of type $A \between OI(B)$. The predicates A and B are respectively the *initial* predicate and the *invariant*[22] predicate. An implementation of a run-time interface is a proof of such a proposition.

The application programmer writes a program of type $A \subseteq IO(B)$. The predicates A and B are respectively the *initial* and the *final* predicate. An application of a run-time interface (API) is a proof of such a proposition.

[NOTION OF INVARIANT.] The notion of invariant here is strong in the sense that if the invariant holds, there should actually *be* a next state. It has some connection with liveness and/or deadlock freedom. (Thus, if we were to run the program on a fault-tolerant (hardware) computer, or an immortal and perfectly diligent (human) computer, we can count on actually getting to a next state that also satisfies the invariant.)

When we run a program we put together a program of type $C \subseteq IO(A)$ with a program (the run-time system) of type $C \between OI(B)$. If the current state of the interface is s, then we have an object of type $C(s) \to IO(A, s)$, and objects of type $C(s)$ and $OI(B, s)$. Putting the first together with the second, we get a proof of $IO(A, s)$. Together, the proofs of $IO(A, s)$ and $OI(B, s)$ can be jointly executed. We wind up in a state which satisfies A (as well as $OI(B)$). This is Sambin's compatibility rule[23]

$$\frac{IO(A) \between OI(B)}{A \between OI(B)}$$

The compatibility rule above says that if the closure of a set A intersects with the interior of a set B, or in other words there is a state (the 'current' state) which lies in the intersection, then the set A itself must in fact intersect with that open set, or in other words there is a state (the 'future' state) which lies in the intersection of the original set with the open set. We obtain the future state by running simultaneously the proofs that $IO(A)$ and $OI(B)$ hold in the current state. (To 'run' a proof means to interpret it as, or use it as, or to put it into service as a strategy, either (client) for issuing commands and reading responses or (server) for reading commands and issuing responses.)

[22] This may be an awful word. It *includes* the invariant, whittled away from B by OI.

[23] See for example [25]. Sambin's notation is $s \between X$ or $X \between s$ for $IO(X, s)$, and $s \between X$ or $X \between s$ for $OI(X, s)$.

In some sense Sambin's compatibility law is the basis for applying the refinement calculus. It is the point of connection between the mathematical values, and real actions, events and eventualities. We give the logical inference a temporal interpretation.

Forgetting for the moment about initialising guards, and thinking of a specific interface Φ, we can recast Sambin's rule by using a couple of definitions:

$$\begin{aligned} \mathit{Client}_\Phi(A,B) &= A \subseteq \Phi^*(B) \\ \mathit{Server}_\Phi(A,B) &= A \between \Phi^\infty(B) \end{aligned}$$

The compatibility rule is then something like this

$$\frac{\mathit{Client}_\Phi(A,B) \quad \mathit{Server}_\Phi(A,C)}{\mathit{Server}_\Phi(B,C)}$$

Note that we have (weakening of initial predicate)

$$\frac{A \subseteq B \quad \mathit{Server}_\Phi(A,C)}{\mathit{Server}_\Phi(B,C)}$$

(weakening of invariant)

$$\frac{\mathit{Server}_\Phi(A,B) \quad B \subseteq C}{\mathit{Server}_\Phi(A,C)}$$

and a number of other rules ....

7 Use of refinement calculus, feasibility

The party line, if I understand it, about how one uses the refinement calculus is that one constructs a chain of refinement steps, starting with a monotone predicate transformer that expresses the specification, and finishing with a predicate transformer which is not merely monotone, but *executable*, or *feasible*. These adjectives can be compared with *effective*; executability doesn't carry the mechanical connotation of effectivity. Feasible doesn't carry the connotation of polynomially bounded resource consumption, still less reasonable, affordable consumption in practice. The connotation of 'feasible' is just 'performable', non-miraculous, logically possible in a very weak sense (especially with classical logic). Nevertheless, feasible specifications are 'code', as Carroll Morgan calls it. They are executable, performable, followable, can serve as guides to action.

Some 'healthiness' conditions are placed on feasible specifications. A feasible predicate transformer is (I think) characterised as being at least strict, $\Phi(\mathit{None}) = \mathit{None}$ (which is Dijkstra's law of the excluded miracle, that rules out a magic device to accomplish the impossible). Strictness means commutation with empty unions. More generally, we can consider commutation with some class (empty, finite, countable, directed, totally ordered, unrestricted) of intersections and/or unions.

In the refinement calculus literature, the concept of feasibility is also given ostensively, by examples; we consider *these* forms (assignment, guarded 'if' and 'do', etc) to be feasible.

Perhaps the above is 'feasibility' for batch programs. Feasibility is not so often discussed in connection with interactive programs. (Not in Morgan.) I think it means: continuity. An interactive program need not be conjunctive. And continuity means commuting with directed limits.

Whatever the correct characterisation, if a predicate transformer is feasible (executable as a batch program), this should imply that the text of some expression for this predicate transformer is a kind of pseudo-code that can be translated into the syntax of some programming language and run autonomously (ie. without need of anything else, any intervention or input, beyond being properly installed/loaded) by a machine.

While the final, executable specification is what Morgan calls 'code', the initial specification frequently takes the form of a 'specification statement', or 'transition specification' $\{pre^{\sim}\} \; ; [post]$, where pre and $post$ are (full-blooded) relations such that for some set C, $pre : C \to \mathbb{P}(A)$ and $post : C \to \mathbb{P}(B)$, and $\_^{\sim}$ denotes the converse operator on (full-blooded) relations. One can also write this

$$\langle C, \langle pre, post \rangle \rangle : \mathbb{F}(\mathbb{P}(A) \times \mathbb{P}(B))$$

It is a *span* in the (poset enriched) category of sets and binary relations. The (hidden) middle terms are thought of as remaining constant while the state changes from the initial to the final value. The relation pre is usually a restriction of the equality relation.

(There is a lot of discussion of whether a specification specification has a 'frame', which exhibits the variables which may change, or another construction which exhibits the constants.) [RELATED TO INITIALISING GUARDS.] This is how one refers to the initial state in the postcondition.

The manner in which one might use the refinement calculus as a calculus supported by type-theory seems to be rather different. The type-theoretical view is that programs are proofs, not predicate transformers. On the other hand, the view from the refinement calculus literature is that programs are something propositional, like predicates, relations or predicate transformers. The relation between these two may be that for certain predicate transformers which are (for example) sufficiently healthy, the relevant proofs[24] are immediately apparent from the form of those predicates.

[FORMAL SYSTEM]. Need something formal, citable as 'the' refinement calculus.

The refinement calculus is a formal system for establishing inclusions $A \subseteq B$, where the A and the B are predicate expressions, that may contain variables of certain kinds – predicate variables in particular, perhaps also predicate transformers. We establish such inclusions in a context: a sequence of hypotheses that themselves are inclusions.

If we use it with intuitionistic logic, then to inclusions we may need to add statements of non-emptiness, such as Sambin's statement form $A \between B$. This is to be thought of as making a positive, existential statement from two predicates. (Note: there is a similar existential import connected with relational composition; also non emptiness of a predicate P can be rendered by $P \between P$.).

The refinement calculus is used to prove judgements of one of these two forms, primarily between predicate transformers, but also lifted versions between relations and predicates.

If the refinement calculus is used with intuitionistic logic, it seems useful to extend the judgement forms to incorporate claims that an intersection is non-empty (in the positive, existential sense), as with Sambin's $\between$ notation.

As it were, we try never to mention the 'points', or individual states.

The development of a program largely consists of demonstrations of such statements, and related statements (pointwise inclusions, and maybe extensions of $\between$ to predicate transformers and relations). That is an abominably vague statement, and anyone

[24] What might those be?

would like to know precisely what more is involved, and precisely what role is played by proofs of inclusion and overlap in the development/delivery of the final program.

One idea may be this: there are two kinds of programs. A program which runs as a client of an interface Φ is a proof of an inclusion of the form $A \subseteq \Phi^*(B)$. A program which provides the service Φ is a proof of the 'overlaps' statement $A \between \Phi^\infty(B)$. In the case of a client program A is the precondition, or conditions under which the program is required to terminate, while B is a predicate which must hold in the final state. In the case of a server program A is the initial condition of a system, while B is an invariant.

8 Simulation

8.1 Transition systems

This is a run-through for the interesting case, which is interaction structures.

If we are given two transition structures

$$\begin{array}{l} \phi : A \to \mathbb{F}\,(A') \\ \psi : B \to \mathbb{F}\,(B') \quad , \end{array}$$

then we can derive a relation transformer $[\phi, \psi] : \mathbb{P}\,(A' \times B') \to \mathbb{P}\,(A \times B)$ as follows:

$$[\phi, \psi](R, a, b) \;\triangleq\; (\forall\, t \in T_\phi(a))(\exists\, t' \in T_\psi(b))R(a[t]_\phi, b[t']_\psi)$$

One can also write this as follows:

$$[\phi, \psi](R, a, b) \;\triangleq\; [\phi](\{\, a' \in A' \mid \{\psi\}(R(a'), b)\,\}, a)\ .$$

That doesn't seem very enlightening, but if we give a very point free version it is, using $\mathit{flip}R$ for the operation 'converse' of swapping the argument places of a binary function R, we get

$$[\phi, \psi](R) = \mathit{flip}([\phi] \cdot \mathit{flip}(\{\psi\} \cdot R))$$

That is at least prettier. (Another pretty form is $((/\phi) \cdot (\psi;))R$.)

The intuition is that if $[\phi, \psi](R)$ holds between a and b, then choices for transitions from a can be mapped to choices for transitions from b in such a way that R holds between the destination states.

If $A = A'$ and $B = B'$, a *simulation relation* is a relation R such that

$$R \subseteq [\phi, \psi](R)\ .$$

Because simulations are post-fixed points (coalgebras, invariants), they are closed under arbitrary $\bigsqcup$ (disjunction).

A particularly important case is that in which $A = B$, and $\psi = \phi^*$.

Because of the alternating quantifiers, there is an interaction structure (on pairs of states) connected with a simulation relation. If this is equipped with a point (*i.e.* a pair of states), then what we have is a pair of pointed interaction systems. [25]

[25] Definitions.

1. A transition *structure* from set A to set B is an element of $A \to \mathbb{F}\,(B)$. I reserve ϕ, ψ, ϕ', ϕ_i, *etc.* as typical variables for transition structures. So a transition structure is a morphism in the Kleisli category for the functor $\mathbb{F}\,(\_)$ (which takes types to types). I pick out the components of ϕ as follows: $\phi = (\lambda\, a)\langle T_\phi(a), a[\_]_\phi\rangle$.

8.2 Interactive structures

Suppose we are given interaction structures

$$\Phi : U \to \mathbb{F}\,(\mathbb{F}\,(U'))$$
$$\Psi : V \to \mathbb{F}\,(\mathbb{F}\,(V'))\ .$$

Then we can derive a relation transformer $[\Phi, \Psi]$ (read Φ mediated by Ψ) as follows.

$$\begin{array}{ll} [\Phi, \Psi] : \mathbb{P}\,(U' \times V') \to \mathbb{P}\,(U \times V) & \\ [\Phi, \Psi](Q, u, v) \stackrel{\Delta}{=} & (\forall\, c \in C_\Phi(u))(\exists\, t \in C_\Psi(v)) \\ & (\forall\, p \in R_\Psi(v, t))(\exists\, r \in R_\Phi(u, c)) \\ & \quad Q(u[c/r]_\Phi, v[t/p]_\Psi) \end{array}$$

This can also be written as follows.

$$[\Phi, \Psi](Q, u) = \ \textstyle\bigcap_{c \in C_\Phi(u)} \Psi(\bigcup_{r \in R_\Phi(u,c)} Q(u[c/r]_\Phi))$$

The intuition here is that of a 'one-step delayed' preimage of Q. Any Φ-interaction from u can be simulated by a Ψ-interaction from v, leaving the initial states related. This means that we can translate a command for Φ into a command for Ψ in such a way that a response for Ψ can be translated back to a suitable response for Φ. We have a 'jacket' round calls to Ψ, which is made to appear like Φ.

If $U = U'$ and $V = V'$, then we say that Q is a *simulation relation* provided that $Q \subseteq [\Phi, \Psi](Q)$. Because simulation relations are post-fixed points (co-algebras, invariants) of a certain monotonic operator, they are closed under arbitrary unions (co-limits). A simulation of u by v is a simulation relation Q together with a proof that $Q(u, v)$.

This surely has to do with implementing one interface using another. (Φ is a "high-level" interface for which we are a server; Ψ is the 'low-level" interface of which we are a client.)

A conjecture: the following is a category (in which the homsets have a Heyting algebra structure).

objects are triples of the form

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

A transition *system* is a set S (whose elements are called *states*) together with a transition structure from S to S. (One should also define a notion of 'large' system, in which we do not have a set but a proper type of states.) A transition system can be written $\langle S, \phi \rangle$.

A *pointed* transition system is an transition system $\langle S, \phi \rangle$ with a state s_0, called the *initial* state. It can be written $\langle S, \phi, s_0 \rangle$.

2. an interaction *structure* from set A to set B is an element of $A \to \mathbb{F}\,(\mathbb{F}\,(B))$. I reserve Φ, Ψ, Φ', Φ_i *etc.* as typical variables for interaction structures. I pick out the components of Φ as follows:

$$\Phi(a) = \langle C_\Phi(a), Rn_\Phi(a) \rangle : \mathbb{F}\,(\mathbb{F}\,(B))$$

where $Rn_\Phi(a, c) = \{\, a[c/r]_\Phi \mid r \in R_\Phi(a, c) \,\} : \mathbb{F}\,(B)$.

An interaction structure is a morphism in the Kleisli category for $\mathbb{F}\,(\mathbb{F}\,(\_))$.

An interaction *system* is a set S (whose elements are called *states*) and an interaction structure from S to S. So a system is 'homogeneous', *i.e.* from and to the same set S, called its state-space. It can be written $\langle S, \Phi \rangle$.

A *pointed* interaction system is an interaction system $\langle S, \Phi \rangle$ with a state s_0, called the *initial* state. It can be written $\langle S, \Phi, s_0 \rangle$.

In other words, the objects are interaction systems.

morphisms from (S, Φ, s_0) to (T, Ψ, t_0) are relations $R : \mathbb{P}(S \times T)$ such that

$$R \subseteq [\Phi, \Psi]\, R$$
$$R(s_0, t_0)$$

In other words, the morphisms are simulations of the domain's initial state by the codomain's. The morphisms, as relations, are partially ordered by extensional inclusion. Composition is relational composition and therefore monotone in both components. For identity morphisms we have to take the equality relation between states.

In some sense this category is the one in which most programmers work. Programs are morphisms in it.

[QUESTIONS.]

Is reflexive and transitive closure $\_^*$ (of an interaction structure) a monadic construction? (What is the multiplication?) What can be said about the Kleisli category? What about $\_^\infty$ and comonads?

[TO DO.] This needs to be related to Sambin's 'basic picture', which focuses on binary relations, and a notion of continuous morphism. What does the basic picture look like from the perspective of the refinement calculus? Many things are recognisable .... How does the basic picture look if we keep families distinct from predicates? Then one probably has to reconsider Sambin's way of writing 'infinitary relations', which is something like $A \to \mathbb{F}(\mathbb{P}(B))$.

[SNIPPETS.]

9 Twice iterated powerset

Structures in which the powerset operator is iterated twice are perhaps not so common in mathematics (apart from the quantifiers!), though the main examples are of enormous interest. Part of the interest is the foundational problem of formalisation of these notions in a predicative setting.

point-set topology [16] Usually, a topological space is defined to be a pair $\langle S, \Omega \rangle$ where S is a set (of points), and $\Omega \subseteq \mathbb{P}(S)$, *i.e.* $\Omega : \mathbb{P}(\mathbb{P}(S))$ is a set of subsets of S – the open sets of the space. Ω must be closed under finite intersections and arbitrary unions – there is yet a third level of powerset involved in the notion of an arbitrary union.

A number of structures related to topological spaces feature a twice-iterated powerset. For example, there is the notion of a covering system [17, page 534 Ex. 5], which has the form $\langle S, \mathrm{Cov}, \leq \rangle$, where S is a set, and $\mathrm{Cov} : S \to \mathbb{F}(\mathbb{F}(S))$ and $(\leq)$ together satisfy the axioms

1. $\leq$ is a partial order.
2. $X : \mathrm{Cov}(s), X(s') \to s' \leq s$.
3. (Stability) $X : \mathrm{Cov}(s), s' \leq s \to \exists Y : \mathrm{Cov}(s').\, Y \leq X$ where $X \leq Y$ means $\forall s.\, X(s) \to \exists s'.\, Y(s') \wedge s \leq s'$.

A covering system is essentially an interactive structure $\Phi : S \to \mathbb{F}(\mathbb{F}(S))$ together with a partial order $\leq$ that (among other things) is a simulation relation with respect to Φ (stability).

probability space [13] A probability space is a structure $\langle S, E, \mu \rangle$ where S is a set of outcomes, $E : \mathbb{P}(\mathbb{P}(S))$ is a σ-ring of subsets of S whose elements are called events, and $\mu : E \to [0,1]$ satisfies $\mu(All_S) = 1$ and is additive on countable sequences of disjoint sets.

10 Right factors

If R and S are proper relations, say with

$$\begin{array}{l} R : A \to \mathbb{P}(C) \\ S : B \to \mathbb{P}(C) \end{array}$$

we might be interested in relations T with the following property.

$$\begin{array}{l} T : A \to \mathbb{P}(B) \\ T \,;\, S \subseteq R \end{array}$$

There is a weakest such relation, and it can be calculated as follows. We have

$$\begin{array}{ll} & \langle a, b \rangle \in T \\ \equiv & \forall c.\, \langle b, c \rangle \in S \Rightarrow \langle a, c \rangle \in R \\ \equiv & S(b) \subseteq R(a) \end{array}$$

So we can define T, which is usually written R/S and called the weakest pre-component of S within R or R right-divided by S by

$$T(a) = \{\, b \in B \mid S(b) \subseteq R(a) \,\}\ .$$

(Right) division of a relation by a transition structure From this it is evident that the operation makes good sense when S (the thing that comes last) is a transition relation. In other words, if $\phi : B \to \mathbb{F}(C)$, then $(/\phi)$ is an operator on (proper) relations, taking relations in $A \to \mathbb{P}(C)$ to relations in $A \to \mathbb{P}(B)$. It is defined by

$$\begin{array}{ll} (/\phi)(R, a, b) & = \phi(a) \subseteq R(b) \\ & = [\phi](R(b), a) \end{array}$$

We may compare this with the operator $(\phi \,;\,)$ which takes relations in $C \to \mathbb{P}(D)$ to relations in $B \to \mathbb{P}(D)$. It is defined by

$$\begin{array}{ll} (\phi \,;\,)(R, b, d) & = \phi(b) \between R^{\sim}(d) \\ & = \{\phi\}(R^{\sim}(d), b) \end{array}$$

What is astonishing about $(/\phi)$ is that in the sequential composition the transition system comes *second*; one might have expected it to come first, since only that way round does a transition structure compose 'well' with a proper relation.

The 'right division' operator comes from Conway's theory of factors [5].

The important point is that one way in which we can refine specifications to programs using sequential composition is to postulate that they are to be accomplished in two successive parts, where the second part of the computation (which 'finishes off') takes the form of a transition structure. (Of course another way of breaking down the task is to postulate something computational to be done first.) The right factor here

is something computational, and we 'chase' the non-computational propositional part back and back, or back and forth, until it becomes the identity, and so vanishes.

One question arising is, can we do right division of an arbitrary predicate transformer by an interactive structure? To what extent do predicate transformers support division?

Division and simulations A common way to describe simulation relations with respect to transitions system $\rightarrow_\phi$ (simulated) and $\rightarrow_\psi$ (simulator) is to say that a relation $\mathcal{S}$ is a simulation relation provided:

$$(\mathcal{S} \; ; \rightarrow_\phi) \subseteq (\rightarrow_\psi \; ; \mathcal{S})$$

Thus $\mathcal{S}(a)$ is the predicate that holds of something simulated by a. Using rightdivision, we can re-express this as

$$\mathcal{S} \subseteq (\rightarrow_\psi \; ; \mathcal{S})/(\rightarrow_\phi) \; .$$

This makes it clear, if there was ever any doubt, that a simulation relation is a coalgebra for a transformer on binary relations. A proof that a simulation relation obtains between a pair of states is something that can be used to perform a simulation of one state by the other.

References

[1] S. Abramsky. Semantics of interaction. In A. Pitts and P. Dybjer, editors, *Semantics and Logics of Computations*, pages 1–31. Cambridge University Press, Cambridge, 1997.

[2] P. Aczel. Quantifiers, games and inductive definitions. In *Proceedings 3rd Scandinavian Logic Symposium*. North-Holland, 1975.

[3] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

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

[5] J. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[6] T. Coquand. A semantics of evidence of classical arithmetic. *Journal of Symbolic Logic*, 6-:325–338, 1995.

[7] T. Coquand, J. Sambin, G. Smith, and S. Valentini. Inductively generated formal topologies. `http://www.cs.chalmers.se/~coquand/silvio.ps`, December 2000.

[8] E. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *CACM*, (18):453–457, 1975.

[9] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[10] E. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1989.

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

[12] G. Gentzen. The consistency of elementary number theory. In *The Collected Papers of Gerhard Gentzen*, pages 132–213. North-Holland, Amsterdam, 1969.

[13] P. Halmos. *Measure Theory*. Van Nostrand, 1969, 13th printing.

[14] J. Hintikka. *Language Games and Information*. Clarendon Press, London, 1972.

[15] G. Kreisel and J. Krivine. *Elements of Mathematical Logic (model theory)*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1967.

[16] K. Kuratowski. *Introduction a la Théorie des Ensembles et a la Topologie*. L'enseignement Mathématique de l'Université de Genève, 1966.

[17] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[18] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[19] C. Morgan. *Programming from Specifications, 2nd edition*. Prentice Hall, 1994.

[20] C. Morgan and B. Sufrin. *Specification Case Studies, 2nd edition*, chapter Specification of the Unix filing system, pages 49–78. Prentice-Hall, 1993.

[21] Y. Moschovakis. The game quantifier. *Proceedings of the American Mathematical Society*, 31(1):245–250, January 1972.

[22] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Clarendon Press, Oxford, 1990.

[23] P. Novikov. On the consistency of a certain logical calculus. *Matématicésky sbovnik*, 12(3):353–369, 1943.

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

[25] G. Sambin and S. Gebellato. A preview of the basic picture: a new perspective in formal topology. `http://www.math.unipd.it/~logic/ftp/BPP.ps`.

[26] S. Vickers. *Topology via Logic*. Cambridge University Press, 1989.