1 Introduction

One of the tasks for computer scientists is to devise effective mathematical models that capture central programming concepts. These models should gain us intellectual leverage that we can bring to bear on difficult practical problems that defy unguided ingenuity. We by bringing these concepts under simple and uniform rules. One of the central concepts of programming, brought into prominence by component-based systems, such as X, Y, and Z is the notion of an interface: a boundary across which components interact. The notions of interface and interaction are fundamentally the same.

The task of modelling interaction has had a lot of attention. It is scarcely necessary to mention the pioneering work of Milner and Hoare in the 1970's developing process algebras, CCS and CSP respectively. The focus in process algebra is on events in which independent components or agents engage at the same time, that are so to speak shared synchronisation points in the curricula vitae of those agents. The models of process algebra are labelled transition systems, in which the transitions represent instances of atomic actions, private or shared.

Most of the work of a programmer is to write code (a component or modules in a larger system) that implements or provides one or more procedural (call and return) interfaces, given and relying on other such interfaces, or a hardware interface. The notion of a contract or specification to be fulfilled by an implementation of an interface is of pivotal practical importance. How should we model procedural interfaces in the framework of process algebra?

- At a coarse level of granularity, we might represent the entire execution of a procedure as an atomic action, in which the calling and the called program both participate, exchanging output results for input data.

- At a finer level, we might represent entry to a procedure as one action, and return from it as another.

It is difficult to argue that process algebra provides a model that captures the notion of a procedural interface particularly well. To view the execution of a procedure as an atomic action makes it difficult to model the occurrence of deadlock, in which the call takes place but no return is possible. To view (at a finer granularity) the execution of a procedure as a sequence of actions bracketed by a call and a return action requires some extra structure on the labels to tie the elements of the pair together. This structure is something extraneous, a complication on top of straightforward labelled transition systems.

We propose a model of procedural interfaces (and command-response interaction) in which there is an intrinsic "bracketing" between actions. The model is directly inspired by ideas developed in formal topology, particularly by Sambin and his cohorts. The model takes the form of an order enriched 'pre-category', in which the objects (analogous to formal spaces) represent interfaces, and the morphisms (analogous to continuous maps) represent program modules. There is an associative form of composition, which is moreover monotone in both the preponent and postponent.

2 Two notions of powerset

We work in a form of predicative type theory without any general identity relation: intensional or extensional. The reasons for this are partly ideological (a proof of identity has no computational meaning), partly psychiatric (traumatisiation of one of the authors), and partly experimental (we wanted to see how far we could get without it). In effect, what this amounts to is a distinction between two forms of (predicative) power set, that we distinguish with the words 'family' and 'predicate'[1]. If S is a set (or even a type), a family of S's is an indexed family of S's, and a predicate of S's is a propositional (set-valued) function with domain S.

$$\begin{array}{rcl} \mathrm{Fam}(S) & = & (\Sigma\, I : \Omega) S^I \\ \mathrm{Pow}(S) & = & S \rightarrow \Omega \end{array}$$

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

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

We'll sometimes use the operator 'ε' for backwards application of a predicate to a value in its domain: $s\ \varepsilon\ P =_{\mathrm{def}} P(s)$.

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

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

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

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

and for any set

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

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

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

One slight difference is that the passage from a predicate to a family (with small index set) fails when S is not a set but a type. A more conspicuous

[1] It may be objected that a predicate is something syntactical, and the terminology is therefore inappropriate. That may well be, but the terminology in which 'predicate' is synonymous with boolean-valued (or propositional) function has been used by computer scientists for about 4 decades, starting with Dijkstra. Language has changed.

difference is that if we consider Pow(_) and Fam(_) as functors from the category of types (and functions between them) to the category of types, then Pow(_) is contravariant, and Fam(_) is co-variant. The actions of the two functors on morphisms $f : A \to B$ are as follows.

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

One can also suggest that a family is something computational – a function which can be used to compute a element or datum given an index in the family's domain – whereas a predicate is something that belongs to the realm of specification.

Although we distinguish between families and predicates, they have a lot in common. For example the notion of an empty family, a singleton family, and the union of an indexed family of families make perfect sense. (Over and above this, predicates are closed under intersections of families.) Moreover one can make good sense of the notions of a family being included in a predicate, and having non-empty intersection with a predicate. (We use Sambin's infix operator $\_\between\_$ for this notion.)

$$\begin{array}{lll} f \subseteq P & =_{\mathrm{def}} & (\forall i \in f_{\mathrm{D}}) P(f[i]) \\ f \between P & =_{\mathrm{def}} & (\exists i \in f_{\mathrm{D}}) P(f[i]) \end{array}$$

Varieties of relation If we distinguish predicates from families, then there are several subtly different notions of relation, of which the most important are

$$\begin{array}{lll} A & \to & \mathrm{Pow}(B) \\ A & \to & \mathrm{Fam}(B) \end{array}$$

We shall call a relation of the first kind an 'honest-to-God' relation from A to B, and a relation of the second, more computational kind a 'transition structure' from A to B.

Both kinds of relation are closed under indexed unions, and in general the structure of families and predicates lifts 'pointwise' to relations. Both have in addition a monoid structure of composition (which we denote with a semi-colon), though the unit for composition of honest-to-God relations is the problematic relation of equality. There is also an operation of reflexive transitive closure.

An important difference is that honest-to-God relation is invertible: given $R : A \to \mathrm{Pow}(B)$ we have $R^- : B \to \mathrm{Pow}(A)$ defined by $R^-(a, b) = R(b, a)$.

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

$$\mathrm{Pow}(B) \to \mathrm{Pow}(A)$$

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

$$A \to \mathrm{Pow}(\mathrm{Pow}(B))$$

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

$$\begin{array}{l} A \to \mathrm{Fam}(\mathrm{Fam}(B)) \\ A \to \mathrm{Fam}(\mathrm{Pow}(B)) \end{array}$$

These first of these is the central type which is studied in this paper. The second (which is closer to Sambin's formulation of a covering relation in formal topology) appears to be of equal interest; it may also be more realistic (true to life, or relevant to real software) as a model of interface specifications.

3 Modelling interaction

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

Syntactically, a procedural interface is usually a record: a mapping from names to data items or entry points. For current purposes, it is sufficient to suppose that an interface consists of just one entry point.

To explain the service to be provided, we posit a service state which is sufficient to explain the effect of successive procedure calls. We try to pick a state space which allows a description of the observable effect of a procedure call which is as simple to understand as possible. A component that implements this interface has to behave as if it 'contained' a state in this statespace, in so far as it may be indirectly observed, through calling procedures with particular input arguments, and observing the result values when the calls have returned.

To describe the semantics of a procedure, we have first to say which calls are legitimate (for the caller) in a given state, and then say which return states are legitimate (for the procedure) for a given call. Here a call can be thought of as a pair of an input state together with a vector of input arguments chosen by the caller, or something derived from such a pair.