Specifying interactions with dependent types

Peter Hancock, Anton Setzer

June 12, 2000

1 Introduction

There are several practical approaches to modelling input−output for functional programs. Two examples are the IO−monad of Haskell, and the uniqueness types of Clean. Some of the basic ideas are surveyed in Gordon's thesis [5]. There has been at least a decade of theoretical and practical work in the functional programming community, not least on the conceptual question of understanding how it is that a 'piece of mathematics', in the form of a functional program can be used to control and bring about changes in the real world. So now we can write[1] windowing systems, systems to play music, web servers, games, and software for robots using functional programs.

For a long time dependent type theories, such as Martin−Löf's ([14]), Luo's system [11], or Coq [4] have been used only to write programs, and to reason about them, but not actually to run them. There has been much recent interest in using these systems for more practical programming, and with Lennart Augustsson's Cayenne system [1] such programs can at last be run. The authors have suggested in [7] that there is great scope for dependent type theory in connection with writing interactive programs. Certain constructions that have been developed in these systems (for example, Martin−Löf's 'W−types': see [15, pages 109–114] and [13, pages 79–86]) seem to have been tailormade for modelling interactive programs. More importantly, there is the prospect that we can really put type theory to work in quite mundane real−world situations, solving important engineering problems.

Dependent type theories were originally conceived (in the late 60's and early 70's: [3], [17], [12]) as a basis for computer checked mathematics, particularly constructive mathematics having a direct computational meaning. Specific to dependent type systems, as opposed to the type systems that emerged in early programming languages (which indeed involved forms of data dependency) is the idea that given a sufficiently rich repertoire of type constructions, a mathematical predicate can be expressed with full precision by a function having data for arguments and types for values: a dependent type.

A mathematical predicate is nothing but an specification – at least, in the static, timeless world of mathematics. Is it possible to express the specifications

[1] This may be to err a little on the side of optimism.

of interactive programs and the primitives from which they are built, with with the same full precision as in constructive mathematics?

In this paper we consider how to express *specifications* of interactions in dependent type theory. The results so far are modest, though we hope we have identified some key structures for describing contracts between independent agents, and shown how to define them in a dependently typed framework. These are called below transition systems (2.2) and interaction systems (2.3). Both are coalgebras; transition systems for a functor $\mathit{Fam}\ \_$, and interaction systems for its composite with itself, $\mathit{Fam}\ (\mathit{Fam}\ \_)$. These structures seems to have interesting connections with predicate transformer semantics for imperative programs, as initiated by Dijkstra, and also with the refinement calculus of Back and von Wright as described in their book [2]. We restrict attention to situations in which the system and its environment communicate by exchanging messages in strict alternation, as with the moves in a two−player game.

Plan of paper The remainder of this section consists of notational preliminaries. The next section begins with a distinction between two manifestations of the powerset functor in dependent type theory. This is followed by two subsections which define transition systems and interaction systems, give examples, and define some predicate transformers over these systems that play a central role in designing an implementation for a given specifications. The next section (3 contains a tentative suggestion for the form of the specification of a single interaction. The last section says how we intend to proceed.

Notation. Our framework is a finite dependent type structure over a collection of ground types, given by a type Set, and a type A for each object $A : \mathrm{Set}$. (We do not bother to distinguish notationally the set $A : \mathrm{Set}$ from the type of its elements.) There are dependent function types $(\, x : \alpha\,) \to \beta$ and dependent product types $(\, x : \alpha\,) \times \beta$, where α is a type and β is a type possibly depending on a variable $x : \alpha$.

The elements of $(\, x : \alpha\,) \to \beta$ are functions $\lambda\, x : \alpha.\, b$, which at arguments $x : \alpha$ determine values $b : \beta$. The typing $: \alpha$ of the bound variable x will sometimes be omitted. Application is expressed using brackets $f(a)$. We also use some infix operators.

The elements of $(\, x : \alpha\,) \times \beta$ are dependent pairs $\langle a, b\rangle$, where $a : \alpha$ and $b : \beta$. We use pattern matching (as in $\lambda\, \langle x, y\rangle.\, \ldots x \ldots y \ldots$) rather than explicit projection functions.

2 Transition systems and interaction systems

We are primarily interested in interaction systems, but begin with transition systems, which are simpler. A transition system is a type−theoretic analogue of the notion of labelled transition system familiar in computer science.

We make heavy use of two unary type operators, which are in effect different forms of the 'subset of' operator, that assigns to a type the type of 'subsets' of that type.

2.1 Two notions of subset

Definition 1 *The type operators $Pow\,\_$ and $Fam\,\_$ are defined as follows*

$$\begin{array}{l} Pow\,\_, Fam\,\_ : \mathrm{Type} \to \mathrm{Type} \\ Pow\,A = A \to \mathrm{Set} \\ Fam\,A = (\,I : \mathrm{Set}\,) \times I \to A \end{array}$$

If A is a type, an element of $Pow\,A$ is a propositional function or predicate over A, whereas an element of $Fam\,A$ is an indexed family of elements of A. A propositional function has the form $\lambda\,a : A.\,P(a)$, whereas an indexed family has the form of a dependent pair $\langle I, \lambda\,i : I.\,f(i)\rangle$ in which I : Set is an index set, and $f : I \to A$ is an indexing function.

Both these operators act naturally on maps between types, and are what may be called 'pre−functors', meaning functors in the categorical sense, disregarding everything that has to do with equality between morphisms.

Definition 2 *Given types A and B and a function $f : A \to B$, the functions $Pow\,f$ and $Fam\,f$ are defined as follows.*

$$\begin{array}{l} Pow\,f : Pow\,B \to Pow\,A \\ Fam\,f : Fam\,A \to Fam\,B \\ Pow\,f = \lambda\,P : Pow\,B.\,P \cdot f \\ Fam\,f = \lambda\,\langle I, g\rangle : Fam\,A.\,\langle I, f \cdot g\rangle \end{array}$$

So $Fam\,\_$ is contravariant and $Pow\,\_$ is covariant.

Size questions. Both the operators $Pow\,\_$ and $Fam\,\_$ are large, in the sense that they both take types to large types involving (differently) the large type Set. Because they are large, we cannot 'program' with them; for example we cannot form the sequence of their iterates. We can however 'reflect' the type operators with set operators, by taking instead of Set a small universe $\langle U, T\rangle$ of sets, having type $Fam\,Set$. If the universe $\langle U, T\rangle$ is closed under the right set−forming operations, it will serve as a substitute for the large type of sets. If $a : U$ is a substitute for A : Set, then the judgment $\_ : T a$ is a substitute for $\_ : A$. In the paper, we pay no further attention to questions of size, and assume where necessary that the universe is 'large enough'.

Binary relations. By a *relation* between sets A and B one ordinarily means a function $R : A \to Pow\,B$. Sets and relations between them form a category in which the hom−sets have all the structure of predicates over a set, as well as the operations of relational algebra. If one replaces $Pow\,\_$ with $Fam\,\_$, the notion of relation one obtains is in a sense more 'representational'; instead of saying when the relation obtains between arbitrary elements of A and B, one gives for each element of A a set of codes or witnesses of transitions from that element, that index the elements of B to which it is related. Our notion of transition system is essentially that of a binary relation of in this more representational kind.

Predicate transformers. A *predicate transformer* between sets A and B is a function $pt : A \to Pow\,(Pow\,B)$ Note that $A \to Pow\,(Pow\,B)$ is isomorphic to $Pow\,B \to Pow\,A$ so that the predicate transformer transforms predicates over the codomain to predicates over the domain. Sets and predicate transformers between them form an category in which the hom−sets have a rich algebraic structure, as rich as that of predicates on a set. The same is true if one uses instead of $Pow\,\_$ the operator $Fam\,\_$. In this case one obtains a notion of predicate transformer which is in a sense 'set−based'. Our notion of interaction structure is essentially that of a predicate transformer (over a single state space) in this more representational category of predicate transformers.

2.2 Transition structures

In categorical terms, a transition system is a coalgebra for the covariant functor $Fam\,\_$. However we prefer to avoid a heavy categorical presentation of this notion; this would require facing head−on (and perhaps prematurely) some tricky issues connected with the representation of equality between morphisms.

Definition 3 *An transition system is a set S together with a function*

$$\delta : S \to Fam\,S$$

The elements of S are called states*. For a given state s, the general form of $\delta(s)$ is $\langle T(s), \lambda\, t.\, s[t]\rangle$. The set $T(s)$ is called the set of* transitions *leaving state s, and $s[t]$ the* destination state *of transition t. We call a state* deadlocked *if $T(s)$ is empty, meaning that no transition is possible from state s.*

Examples Important examples of a transition system are provided by so−called 'W' types, which are types of wellfounded trees, whose branching is described by a family of types. Given a family of sets $\langle U, T\rangle$, we can form a relativised or 'small' version of the operator $Fam\,\_$, where the index set of a family are restricted to come from the family $\{\, T(u) \mid u : U\,\}$. The set $W(U,T)$ (usually referred to as a W−type, and written $(Wx : U)T(x))$ is the least set closed under a constructor representing this relativised operator. For example if the family $\langle U, T\rangle$ is some standard family of finite sets, then the elements of $W(U,T)$ are exactly the finite trees.

To define a transition system, take S to be $W(U,T)$, so that a state has the typical form $\langle u, \lambda\, x : T(u).\, t(x)\rangle$ where $u : U$ and $t : T(u) \to U$. The function δ is defined by recursion:

$$\delta\,\langle u, \lambda\, x : T(u).\, t(x)\rangle = \langle T(u), \lambda\, x : T(u).\, T(t(x))\rangle$$

Transition systems arising in this way are *terminating*, in the sense that all sequences of transitions starting at a given state must eventually come to a deadlocked state (in which no further transitions are possible). This is because S is defined to be a *least* fixed point. (The initial algebra of $Fam\,\_$ is also a coalgebra for it, by Lambek's Lemma.)

Transition systems provide one possible representation in type theory of a set together with a binary relation on it, or a directed graph (in which the vertices are states, and the edges are transitions). They seem also to provide interesting representations for rewriting systems, in which the transitions are chosen to represent rewriting steps.

Transition systems are closed under sequential composition, several kinds of ordered sum and product, and constructions such as transitive, reflexive and transitive closure, and so on; the definitions are straightforward.

For an example of a transition system which is not terminating, we can take a state space consisting of a single state, with a single transition from that state to itself.

An example of a transition system which is 'the same' as the last one except it is not explicitly cyclic is afforded by taking the natural numbers as states, with a single transition from a number to its successor. The two transition systems are essentially the same in the sense that each simulates the other, or it is impossible to tell them apart by any sequence of transitions. (A precise definition of simulation is given below.)

Predicate transformers. Associated with each transition system is a pair of predicate transformers; they seem to pervade the applications of transition systems. We tentatively use a 'modal' notation to denote these. Thus if $P : \mathit{Pow}\, S$, then $P^{\Box}$ and $P^{\diamond}$ are predicates defined as follows.

Definition 4 $\_^{\Box}, \_^{\diamond} : \mathit{Pow}\, S \rightarrow \mathit{Pow}\, S$

$$P^{\diamond}(s) = (\, t : T(s)\,) \times P(s[t])$$
$$P^{\Box}(s) = (\, t : T(s)\,) \rightarrow P(s[t])$$

A predicate P is progressive *if $P^{\Box} \subseteq P$.*

A state $s : S$ is accessible *if it is in the intersection of all progressive predicates.*

A transition system wellfounded *if all states are accessible.*

There does not seem to be a common term for predicates which satisfy $P \subseteq P^{\diamond}$, or for the greatest such predicate. We call them *invariant*.

The 'universal' modality $\_^{\Box}$ is of particular interest if you are concerned with deadlocked states. Letting *False* denote the predicate which is everywhere empty, a deadlocked state is one in which $False^{\Box}$ holds. An accessible state is one in which deadlock is inevitable. The modality is also of interest in connection with strategies to which ensure that a certain goal predicate is satisfied on the assumption that deadlock is avoided. The least progressive predicate extending a given predicate X (*i.e.* $\mu Y.\, X \cup Y^{\Box} \subseteq Y$) is true of states from which every infinite sequence of transitions must at some point satisfy X.

The 'existential ' modality $\_^{\diamond}$ is of interest if you are concerned with deadlock avoidance. Letting *True* denote the predicate which is everywhere a singleton, a state in which $True^{\diamond}$ holds is a non$-$deadlocked state. The greatest invariant predicate included in a given predicate X is true of states from which there is an infinite sequence of transitions, starting with one that satisfies X.

It is not entirely obvious how to deal with greatest fixed points in type theory, in which historically most of the development has focussed on least fixed points, and on wellfounded rather than infinite structures. Perhaps what one wants is to identify schemes for introducing coinductively defined sets and predicates, and functions defined by corecursion into such sets. These schemes should preserve the most important metamathematical properties, such as decidability of type−checking. However, one would also like to really understand these rules, perhaps by finding an interpretation or model of this extended type theory back into the well−founded fragment. To make such an interpretation, perhaps one approach is to make systematic use of inverse−limit constructions, such as occur in the Lindström's definition of bisimulation in [10]. These constructions are based on an idea that occurs in work on non−wellfounded set by Lars Hallnäs ([6]).

Simulation. The modal predicate transformers introduced above can be used to define a notion of a simulation relation between states. (We have taken the notion from Gordon [5].) The existence of a simulation relation means that there is a state dependent mapping from transitions to transitions, that can be used to translate a sequence of transitions from the first state step by step to a sequence of transitions from the second.

We first of all define this property in a traditional way, as a postfixed point for a certain operation on relations.

Definition 5 *A* simulation relation *between two transition systems $\langle S, \delta \rangle$ and $\langle S', \delta' \rangle$ is a relation $\preccurlyeq : S \to Pow\, S'$ such that the following holds for all $s : S$ and $s' : S'$.*

$$s \preccurlyeq s' \to (\, t : T(s)\,) \to (\, t' : T'(s')\,) \times s[t] \preccurlyeq s'[t']'$$

If $s \preccurlyeq s'$ then the state s is said to be simulated by *the state s'. The first system is simulated by the second if there is a total relation (from states of S to non−empty subsets of S') which is a simulation relation. A* similarity *is a simulation relation on a single transition system which is reflexive and transitive. A* bisimilarity *is a simulation relation which is an equivalence relation.*

The definition of a simulation relation can be expressed using the modal predicate transformers together with 'section' notation for relations. If $R : A \to Pow\, B$ is a relation, then let $R^{\sim}$ denote its converse $\lambda\, s, s'.\, R(s', s) : B \to Pow\, A$. A relation $(\preccurlyeq) :: S \to Pow\, S'$ is a simulation relation if for all $s' : S'$ the following inclusion holds in $Pow\, S$.

$$\begin{aligned}(\preccurlyeq) \subseteq \mathbf{let}\ \ & R = ({}^{\diamond}\cdot(\preccurlyeq)) \\ \mathbf{in}\ \ & ({}^{\square}\cdot R^{\sim})^{\sim}\end{aligned}$$

It is often the case that one simulates one transition system with the *transitive closure* of another, in which transitions of the first are simulated by finite chains of transitions in the other. Occasionally the reflexive and transitive closure is useful, when transitions can be simulated with an empty sequence.

2.3 Interaction systems

An interaction system is an abstraction for a device or resource of some kind, which interacts with a user or environment. Two kinds of examples which are intended to be captured by this notion are devices that monitor and control machinery (for example an aircraft aileron), and servers that provide some service such the ability to order books or other goods. We want to propose a definition to capture the idea of the 'legal' contract between the device (or its makers) and its user. (The notion of a contract statement underlies the refinement calculus presented in [2].) We suppose that each interaction is initiated by the user (which issues a 'command'), and completed by the system (which delivers a 'response'). The user is active, while the system is reactive or passive.

Initiation. For an interaction to be legally initiated, certain conditions $I(s)$ must be met, depending on the current state s of the system, specifically on the syntax of a command or request issued by the user. For example to open a named file for reading, it must be currently registered in the file system under the given name. One can think of a proof that these conditions hold as an issued command, or evidence that the interaction has been properly initiated.

We call I the *guard* predicate of the interaction, and an element of $I(s)$ a *guard for* s.

Termination. Once the guard has been established and an interaction is initiated, certain conditions $J(s, i)$ have to be met for it to become legally complete. These can depend not only on the state s in which the interaction was initiated (whether enough money exists in a certain bank balance for example), but also on evidence $i : I(s)$ that the interaction was legally initiated. The conditions J will typically involve the presence of a response by the system to the users command. One may think of a proof that these conditions are met as evidence that an interaction has terminated.

We call J the *outcome* predicate of the interaction, and an element of $J(s, i)$ an *outcome* of the interaction.

Next state function. As part of completing an interaction (for example, processing instructions to move funds from one bank account to another), the system may move to a new state. The state should be a function of the start state, and the evidence i/j now available. I write it $s[i/j]$.

The fundamental rôle played by the notion of the state of an interaction system is to determine at each point in its evolution what counts as evidence of initiation, and what then counts as evidence of termination. The state is itself determined by the history of completed interactions since the system was started in a known initial state.

As an alternative to giving a termination predicate and a next−state function, one may instead give for each guarded state (that is, each pair $\langle s, i\rangle$ where s is a state and $i : I(s)$ is a guard for s) a family of states: $(J(s, i), \lambda j.\, s[i/j])$.

This family might perhaps be called the *effect*, or potential effect of the guarded state.

In a transition system, the transitions are atomic in the sense that they have no internal structure, such as initiation and subsequent completion: they just occur. To capture the notion of interaction we use the $Fam\,\_$ operator twice.

Definition 6 *An* interaction system *is a set S together with an* interaction structure on S, *which is a function*

$$\delta : S \to Fam\,(Fam\,S)$$

A value $\delta(s)$ of this function has the form $\langle I(s), \lambda\, i.\, \langle J(s,i), \lambda\, j.\, s[i/j]\rangle\rangle$ for a given state s. We call $I(s)$ the set of inputs*, $J(s,i)$ the set of* outputs *for input i, and $s[i/j]$ as the* destination state *of the interaction $\langle i, j\rangle$. When $I(s)$ is empty, we say that the* user is deadlocked*. When $i : I(s)$ is such that $J(s,i)$ is empty, we say that i* deadlocks the system

Examples. Many board games in which players make alternating moves can be represented as interaction systems. A state is a state of the board, or something summarising the relevant state of play, and the interaction system assigns to each even state (where it is the turn of the first player) a family of families of even states, namely for each move made by the first player, the family of states that might result.

Many programming interfaces can be represented as interaction systems. It is often the case that the most convenient way to specify the services available from a device is in terms of a notion of "state of the interface", perhaps one that is not maximally abstract. Roughly speaking, this is the approach taken in a number of practical specification formalisms, such as Z ([18]), or Lamport's TLA$^+$ ([8], [9]).

Predicate transformers. Associated with an interaction system are a pair of predicate transformers which play a key role in using program specifications. I shall use 'bullet' notation to denote these. Thus if $P : Pow\,S$, then P° and $P^\bullet$ are predicates, defined as follows.

Definition 7 *The definition of the* 'white' *and* 'black' *predicate transformers associated with an interaction system is as follows.*

$$\begin{aligned} P^\circ(s) &= (\, i : I(s)\,) \times (\, j : J(s,i)\,) \to P(s[i/j]) \\ P^\bullet(s) &= (\, i : I(s)\,) \to (\, j : J(s,i)\,) \times P(s[i/j]) \end{aligned}$$

The $\_^\circ$ transform is of interest to the user, who is the agency responsible for initiating an interaction; we may think of the user as having the rôle of white (who goes first) in chess. The $\_^\bullet$ transform is of interest to system, who is the passive or reactive agency in the transaction.

If the user's goal is to bring about a state which satisfies the predicate P, then P° holds exactly when the user can initiate an interaction such that P holds in the next state, if there is one. In particular it holds in those states where an interaction can be initiated which cannot be completed. One might call such a state 'winning' for white in the sense of game theory, but it is a calamity for the user of a device, in the sense that the device has become useless.

If the system's goal is to satisfy the predicate P, then $P^\bullet$ holds if there is some way to complete any interaction initiated by the user in such a way that P holds in the next state, if there is one. In particular it holds in those states where no interaction can be initiated.

Note that if both P° and $P^\bullet$ hold, then the user can actually ensure that P is brought about, on the assumption that the system keeps working.

Extreme fixed points. Given these two predicate transformers, we can consider their extreme fixed points. I shall use the names *Bar* and *Pos* for these. *Bar* is the least state predicate such that $\mathit{Bar}^\circ \subseteq \mathit{Bar}$. *Pos* is the greatest such that $\mathit{Pos} \subseteq \mathit{Pos}^\bullet$.

The predicate *Bar* holds in those states in which the user has a strategy to drive the device into deadlock. A proof that the predicate holds in some state can be pictured as a wellfounded tree, which can be used as a strategy or program by following which the user can ensure that the system is driven into a deadlock, provided that it keeps responding so long as some response is legal.

The predicate *Pos* (which is disjoint from *Bar*) holds in those states in which the device has a strategy to evade deadlock. (This includes states in which the *user* is deadlocked.) When *Pos* holds, P° is the weakest precondition for the user to obtain P by means of one interaction.

The predicate *Bar* was first identified by Petersson and Synek in [16]. There is further discussion and illustration of their construction in the chapter 'General Trees' of [15, pages 115–121].

The predicate *Pos* is a greatest fixed point; as already remarked (in 2.2) there is as yet no entirely satisfactory way of providing a foundation for such predicates in type theory.

A useful generalisation of *Bar* is the predicate transformer *IO* which to a predicate P and a state s assigns the set of strategies for the user which will drive the machine into a state in which P holds, on the assumption assuming that the system avoids deadlock, and eventually completes any legally initiated interaction.

Simulation. There is a natural way to extend the notion of a simulation relation to interaction systems. That a simulation relation holds between two states s and s' means that there is a history sensitive way of translating back and forth between commands and responses which can systematically be used to 'fake' s by using s'.

We first of all define this property in a traditional way, as a postfixed point for a certain operation on relations.

Definition 8 *A* simulation relation *between two interaction systems* $\langle S, \delta \rangle$ *and* $\langle S', \delta' \rangle$ *is a relation* $(\preceq) : S \to \mathit{Pow}\, S'$ *such that the following holds for all* $s : S$

and $s' : S'$.

$$\begin{array}{rl} s \preccurlyeq s' \rightarrow & (\, i : I(s)\,) \rightarrow (\, i' : I'(s')\,) \times \\ & (\, j' : J'(s', i')\,) \rightarrow (\, j : J(s, i)\,) \times \\ & \quad s[i/j] \preccurlyeq s'[i'/j']' \end{array}$$

If $s \preccurlyeq s'$ *then the state* s *is said to be* simulated by *the state* s'. *The first system is simulated by the second if there is a total relation (from states of* S *to non−empty subsets of* S'*) which is a simulation relation. A* bisimulation *is a simulation relation which is reflexive. Two systems are bisimilar if there is a total bisimulation between them.*

The definition of a simulation relation can be partially expressed using the white predicate transformers together with 'section' notation for relations. A relation $(\preccurlyeq) :: S \rightarrow Pow\, S'$ is a simulation relation if for all $s' : S'$ the following inclusion holds in $Pow\, S$.

$$(s \preccurlyeq) \subseteq \bigcap_{i:I(s)} (\bigcup_{j:J(s,i)} (s[i/j] \preccurlyeq))^{\circ}$$

3 Syntax and semantics for terminating programs

Syntax. In [7] it was suggested that the *syntax* of a user−program which makes calls on services made available by its environment can be represented by a family of sets $\langle C, R\rangle$, called a *world* in the terminology of that paper. This family of sets represents as it were the 'instruction set' or repertoire of basic instructions available to the user. Based on this representation we can define various important notions such as the following.

- the set of terminating programs $W\langle C, R\rangle$ (which eventually issue a command to which there is no response).
- the programs *Term*(*Results*) (or 'IO−trees' in the terminology of [7]) that finally terminate yielding a result of a particular type

  $$\mu X\,.Result + (\, c : C\,) \times R(c) \rightarrow X$$

  This functor can be equipped with a monad structure, so that the monad laws hold with respect to an inductively defined equivalence relation between programs. It seems that there may be other ways to construct a monad. For example one could also use the following functor, which is a monad with respect to extensional equality

  $$\begin{array}{ll} \textbf{let} & Prog = W(C, R) \\ \textbf{in} & (Result \rightarrow Prog) \rightarrow Prog \end{array}$$

- the programs that may or may not terminate, but when they terminate do so yielding a value of a certain type. One may represent such programs as graphs.

  $$G(Result) = (\, S : Set\,) \times S \times (S \rightarrow Result + (\, c : C\,) \times R(c) \rightarrow S)$$

Such a graph has the form $\langle S, \langle s_0, g \rangle \rangle$ where S is the set of (labels or addresses of) nodes in the graph, s_0 is the root node, and $g : S \to \mathit{Result} + (\, c : C\,) \times R\,c \to S$ determines for each node whether it is terminal (if so giving the value yielded), or not (and in that case giving the family of nodes to which it is directly connected). A path through such a graph is a stream of pairs $\langle c, r \rangle : (\, c : C\,) \times R\,c$, which may lead to a final state for which $g\,s : \mathit{Result}$.

Semantics. To give the *semantics* of a service (*i.e.* in effect an application programmers manual for it) one has to provide two things:

- A state space S. These states are 'specification entities', and needn't be expressed directly in the states of an implementation. The states of a simple file system might consist of a partial function from certain pathnames to sequences of bytes.
- A function assigning each command $c : C$ its interpretation, which is the following:
  - an interaction structure δ_c: *i.e.* a guard I_c and an effect

    $$\lambda\, s, i.\, (J_c(s, i), \lambda\, j.\, s[i/j]_c) : (\, s : S\,) \to I_c(s) \to \mathit{Fam}\ S$$

    Together these give the 'action' of the command.

    The ability to have commands executed allows control over the specification state.
  - a function $|\_|_c : R(c) \to (\, s : S, g : I_c(s)\,) \to J_c(s, g)$. This function gives for each $r : R(c)$ its interpretation $|r|_c$ as a function from guarded states to outcomes. Note that the value of this function may depend on the guard for the state. So knowing only the result, you may not be able to predict the next specification state.

Given these things we obtain an interaction system which describes the overall action of the system: (S, δ) where

$$\delta(s) = (I(s), \lambda\, i.\, (J(s, i), \lambda\, j.\, s[i/j]))$$

where

$$\begin{array}{lcll} I(s) & = & (\, c : C\,) \times I_c(s) & s : S \\ J(s, \langle c, g \rangle) & = & R(c) & s : S, c : C, g : I_c(s) \\ s[\langle c, g \rangle / r] & = & s[g / |r|_c(s, g))]_c & s : S, c : C, g : I_c(s), r : R(c) \end{array}$$

It should be stressed that this proposal is extremely tentative.

4 Conclusion

It is clear that the ideas discussed in this paper need to be tested and sharpened in the context of specific examples. Although we believe that the notions of transition and interaction system, with their associated predicate transformers will play a central role in a practically useful approach to the to the specification and development of interactive programs, the current situation is that we have (at least some of) the right ingredients, but have yet to understand the recipe for baking the cake.

One reason to be hopeful for the prospects of carrying out a programme to develop a theory of specification for interactive systems in type theory along the lines indicated above is that there is fundamentally nothing particularly novel about our approach. We advocate a straightforward state−based approach to the description of interactions, familiar from a number of specification formalisms and frameworks for reasoning about the correctness and refinement of programs. What is new is only that we hope to exploit the expressive power of dependent types, and the constructions that have been developed within it over roughly three decades to the description and analysis of interactive programs.

References

[1] L. Augustsson. Cayenne −− a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

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

[3] N. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, Dec. 1968. Springer−Verlag LNM 125.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin−Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[5] A. Gordon. *Functional programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[6] L. Hallnäs. An intensional characterization of the largest bisimulation. *Theoretical Computer Science*, 53:335–343, 1987.

[7] P. Hancock and A. Setzer. Interactive programs in dependent type theory. Technical Report 5, Department of Mathematics, Uppsala University, 2000. ISSN 1101−3591.

[8] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[9] L. Lamport. Specifying concurrent systems with TLA^{+}. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, pages 183–247. IOS Press, 1999.

[10] I. Lindström. A construction of non−well−founded sets within martin−löf's type theory. Technical Report 15, Department of Mathematics, Uppsala University, 1986.

[11] Z. Luo. *Computation and Reasoning.* Clarendon Press, Oxford, 1994.

[12] P. Martin−Löf. An intuitionistic theory of types: predicative part. In H. Rose and J. Shepherdson, editors, *Logic colloquium 1973*, pages 73–118, Amsterdam, 1975. North−Holland.

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

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

[15] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin−Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

[16] K. Petersson and D. Synek. A set constructor for inductive sets in martin−löf's type theory. In *LNCS*, volume 389. Springer−Verlag, 1989.

[17] D. Scott. Constructive validity. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 237–275, Versailles, France, Dec. 1968. Springer−Verlag LNM 125.

[18] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.