

Hyperindexical Pandimensional Beings

Bill Wadge
University of Victoria
wadge@csr.uvic.ca

The modern incarnation of the language *Lucid* (the pure declarative subset of *GLU*) has many novel and unusual features, such as intensionality, the use of arbitrary, even temporary, new dimensions, dimensionally abstracted definitions of variables and functions, and a serendipity principle that makes everything mean more than it appears at first sight. So what does this all really mean? What is its semantics?

The semantics of *Lucid* is denotational rather than operational. Rather than specify a conventional denotational semantics, we will indicate how the semantics has grown from the semantics of *Lucid*'s previous incarnations.

The latest version of *Lucid* is a generalization of *Field Lucid* (fixed space and time dimensions), itself a generalization of original *Lucid* (time only), which in turn was the result of adding temporal operators to *ISWIM*, Landin's generic functional language. We can therefore present the denotational semantics of *Lucid* in terms of a series of generalizations of the semantics of *ISWIM*.

All of these languages, *ISWIM* included, are really families of languages, each member determined by the choice of data objects and operations. The syntax of each family member is determined by the set of constants and operations symbols available, and the semantics is determined by an interpretation for these symbols. An interpretation in this sense consists of a domain D of data objects and a collection of operations (of appropriate arity) indexed by the set of operations symbols. These interpretations are just algebras; therefore every particular language is determined by the choice of the underlying data algebra. Given an algebra A , $\text{Lucid}(A)$ is the member of the *Lucid* family based upon A .

Let us assume we have chosen a fixed algebra A . In a sense, all the languages discussed are extensions of the simple expression language $\text{Exp}(A)$. We begin by outlining the formal syntax and semantics of $\text{Exp}(A)$.

Syntactically, $\text{Exp}(A)$ "programs" are expressions built up from variables using the operation symbols in the signature of A (we will consider constants to be nullary operations). Thus, the meaning of an $\text{Exp}(A)$ program is determined by the meanings of the variables; so we define an $\text{Exp}(A)$ *interpretation* to be a function that assigns an element of D (the universe or data domain of A) to every variable.

Since $\text{Exp}(A)$ has only two construction rules, the denotational semantics also has only two rules. The first rule says that the meaning of an expression consisting of a single variable is the value the given interpretation assigns to the variable. For example, the meaning of the expression x with respect to an interpretation I is $I(x)$.

The second rule of $\text{Exp}(A)$ semantics is that the meaning of an expression consisting of an operation symbol applied to operands is the result of applying the operation A associates with that symbol to the meanings of the operand expressions. For example, if D consists of the integers and A interprets the symbol $+$ in the usual way, the basic rule implies that the meaning of a sum $P+Q$ is the (numeric) sum of the meanings of P and Q .

The second language, *ISWIM*, extends *Exp* by adding function variables, *where* clauses, and recursive definitions (of both individual and function variables). Since there are two more construction rules, the denotational semantics has two more rules.

The first says that the meaning of an expression that consists of a function variable applied to actual parameters is the result of applying the meaning I associates with the function variable to

the meanings of the actual parameters. For example, the meaning of $f(P, Q)$ is the result of applying $I(f)$ to the meanings of P and Q .

The second rule says that the meaning of a *where* clause is the meaning of the subject in the least interpretation I' that (1) makes all the definitions in the body true and (2) differs from I at most in the values assigned to the local variables of the *where* clause (those variables with definitions in the clause). Here the definitions are considered as *literally* being equations—an interpretation satisfies a definition if and only if both sides have the same meaning, for all values of the formal parameters.

Of all those interpretations that make the definitions true (there may be more than one) we select the one that is *least* in the domain-theoretic sense. Roughly speaking, this is the solution that can be calculated by using the definitions as computation rules. (This *least fixpoint* semantics is described in more detail in [1]). For least fixpoints to be defined, the original algebra A has to be a continuous algebra, which implies that its universe must be a complete partial order with a least element \perp ("bottom").

The original (temporal) Lucid extends ISWIM by adding the temporal operators *first*, *next* and *fb* and by reinterpreting expressions as time sequences. Functions and operations map sequences to sequences, with the operations of A reinterpreted as acting pointwise on sequences.

More formally, the algebra $Lu(A)$ assigns to each variable an element of D^N , N the set of natural numbers.

The only new rules we need are the familiar rules for the Lucid operators. For example, the rule for *next* says that the meaning of *next* P is $\lambda t \pi(t+1)$, where π is the meaning of P . Another way of stating this rule, more in the spirit of intensionality, is to say that the value of *next* P at time t is that of P at time $t+1$.

The rule about the interpretations of the operations of A must be altered to reflect the fact that these now work pointwise on intensions. For example, the modified rule implies that the value of $P+Q$ at time t is the sum of the value of P at time t and the value of Q at time t .

The other rules for ISWIM carry over *unchanged* to temporal Lucid (we are omitting the old *is* current construct, whose semantics entailed a departure from that of the ISWIM *where* clause).

Field Lucid and similar extensions added space as well as time dimensions. These languages were still dimensionally static in the sense that have a fixed, predefined set K of dimensions and a corresponding fixed collection of operations over these dimensions.

In each of these dimensions a coordinate is still a natural number; the context space is therefore N^K , the set of all K -indexed families of natural numbers. Since an intension is a map from the context space to the data domain; the space of intensions is therefore D^{N^K} .

Data operations still work pointwise, so that the value of $P+Q$ at coordinate c is the sum of the value of P at coordinate c and the value of Q at coordinate c .

The temporal operations like *next* have to be reinterpreted to map D^{N^K} to itself. In fact most of the intensional operations affect only a few dimensions but have to formally map the entire intension space to itself. We do this by making them work pointwise on the other dimensions. For example, the *realign* operation is naturally defined on to space dimensions (say *row* and *column*). If there is also a time dimension, we treat a three-dimensional intensions as a time stream of matrices, and we apply the *realign* operation *pointwise* to every element of the stream.

The other semantic rules are still unchanged.

Finally, we can consider the semantics of Lucid itself. The crucial difference is that there is no longer a predefined set of dimensions over which everything varies. We can assume the existence of a large set Δ of *possible* dimensions, so that variables and expressions denote elements of D^{N^Δ} . We need to make Δ large to make sure there are plenty of unused dimensions available for assignment to local dimension variables.

The main task is to give a precise and semantics-based definition of the notion "unused" - and we formalize it through the notion of the *dimensionality* or the *rank* of an object (an intension or a filter).

Informally, we say that an object is K -dimensional (K a subset of Δ) iff the object does not 'use' or 'know about' any of the dimensions not in K — iff these other dimensions play no special role.

If X is an intension, X is K -dimensional iff X is constant in the dimensions outside of K . This means that to determine the value of X at a context c , it is enough to know the components of c

for dimensions in K .

If F is a filter, F is K -dimensional iff F acts pointwise over the dimensions not in K . In other words, a K -dimensional filter treats an element of D^{N^Δ} as a $\Delta - K$ dimensional array of elements of D^{N^K} and applies the same operation to each of them.

We can define the *dimensionality* of an object as the smallest set K for which the object is K -dimensional.

We can now describe how to modify the ISWIM semantics of `where` to take declared dimensions into account. Assume we have an interpretation I , a context c , and that d and e are the new dimensions declared in the clause. We choose two arbitrary elements d and e in Δ that do *not* appear in any the dimensionality of any of the meanings assigned by I to global variables of the clause, and alter I to assign d and e to d and e . Next, we *remove* the declarations of d and e from the clause and find the denotation of the altered clause in the altered interpretation, using the standard ISWIM semantics rule. Finally, we find the value of this intension at the context altered by setting the d and e components to 0. This value, the value of the altered clause in the altered interpretation at the altered context, is (by definition) the value of the original clause in the original interpretation at the original context.

(It is crucial to note that the least fixed point of the `where` clause is finite dimensional, provided all the globals denote finite dimensional entities. So recursion cannot by itself produce infinite dimensional entities.)

The remaining definitions (importantly, function application) remain unchanged. In particular, dimensionally abstract functions pose no particular problem—we can treat “dimension” as a separate type.

References

- [1] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, U.K., 1985.