Theoretical Computer Science 70 (1990) 159–178
North-Holland

A GENERALIZATION OF THE CONCEPT OF SKETCH

Charles WELLS*

Department of Mathematics and Statistics, Case Western Reserve University, University Circle, Cleveland, OH 44106, USA

Abstract. This paper introduces an extension of the concept of sketch, called a *form*, which allows the specification of entities other than limits and colimits in a model. A form can require that a diagram become (in a model) an instance of any categorial construction specifiable in an essentially algebraic way. Constructions which can be specified in this way include function space objects and reflexive objects in a cartesian closed category, power objects in a topos, and list objects in a locos. This generalization is motivated by the desire to specify functional programming languages by sketches.

Contents

1. Introduction

This paper defines the concept of form (Definition 6.1.1), proves a few basic facts about forms, and gives examples of how they can be used. Forms should have

* Partially supported by NSF Grant CCR-8702425.

applications to the specification of functional programming languages as well as to other categorically-oriented approaches to programming languages such as in [16, 13, 27, 28].

A sketch is an inherently typed, graph-based device for formalizing descriptions and specifications which can take the place of the traditional method of formal language plus axioms plus rules of inference. Familiarity with sketches is assumed here, although pertinent definitions are reviewed. A quick introduction to sketches with some applications to computing is given in [29].

Forms generalize sketches; they allow other things besides cones and cocones to specify special objects in a category. This could be done in an ad hoc way, but the purpose of this paper is to describe a systematic way of doing this for a large class of constructions: namely, those which can be described in an *essentially algebraic* way. The particular kind of sketch corresponding to essentially algebraic theories is the FL sketch. (Freyd [12] advocated using essentially algebraic theories to formalize category theory. They are applied to computer science without using sketches in [26].)

A particular sketch (with cones and/or cocones) can be described in an essentially algebraic way, that is, by an FL sketch (see Section 5). The graph and the diagrams are clearly described by certain pullbacks in the FL theory of categories. These pullbacks require things such as "the domain of *this* arrow is the same as the codomain of *that* arrow", "these two composites are equal", and so on. In fact the graph and diagrams are actual elements of the values of those pullbacks in the initial model of the theory obtained from the FL theory of categories by adjoining a constant to the pullback type describing each diagram. The cones and cocones are more special; they are inhabitants of sorts in some FL theory of those categories which have (at least) limits and colimits of the type of the given cones and cocones.

In this paper, forms are defined in Section 6 using a special type of FL sketch called a "CS sketch" (see Section 4). A CS sketch describes a category with some extra structure: it is a specification of a *kind of category* in a precise sense. The form is an element of a sort called the "description" of the sketch in the corresponding CS theory and is realized as the value in the initial category of that type of a constant adjoined to that sort. Sketches are thus special cases of forms. The theory of the form can then be defined as the whole initial model of the CS sketch with constant adjoined.

To sum up the approach of this paper: One notices that any sketch $\boldsymbol{S}$ can be specified using an FL sketch. $\boldsymbol{S}$ is then an element of an initial model of the FL sketch with constant (designating $\boldsymbol{S}$) adjoined. From that point of view, cones and cocones are only some of many types of constructions which can be specified by an FL sketch. So a form is defined by means of a constant adjoined to some sort in an FL sketch which specifies more general types of categorical constructions.

Some of the ideas in this paper are studied in [19, 11], but the approach and terminology here are different and more restrictive. In particular, this paper is concerned only with finite sketches and forms.

2. Terminology and background

Terminology follows [29]. Here I will amplify some points.

2.1. Notation for limits

I use the square bracket notation described in [5, p. 38] for objects defined by finite limits. For example, the object P in the pullback diagram

$$\begin{array}{ccc} P & \xrightarrow{p_2} & B \\ {\scriptstyle p_1}\downarrow & & \downarrow{\scriptstyle g} \\ A & \xrightarrow[f]{} & C \end{array} \tag{1}$$

in a category $\boldsymbol{C}$ is

$$[(a, b) \in A \times B \mid f(a) = g(b)], \tag{2}$$

the objectification of the set $\{(a, b) \mid f(a) = g(b)\}$ of certain pairs of arrows of $\boldsymbol{C}$.

2.2. Sketches

2.2.1. Definition. An *FL sketch* is a sketch with a finite graph, a finite number of diagrams, a finite number of cones based on finite diagrams, and no cocones. "FL" means finite limits. FL sketches are called LE sketches in [5].

An FL sketch has models in any category with finite limits. A model is a graph homomorphism from the graph of the sketch to the underlying graph of the category which takes diagrams to commutative diagrams and cones to limit cones.

Every FL sketch $\boldsymbol{S}$ has a model M_0 in a finitely complete category $\mathbf{Th}(\boldsymbol{S})$ called the *FL theory* generated by the sketch. This has the property that the category of models of the sketch is equivalent to the category of models of the theory and that the equivalence is obtained by composing with M_0. Various proofs of this result are in [7, 20, 5]. Another proof is mentioned in Section 6.5 below.

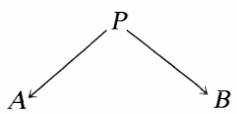
The following theorem has been around in various guises for a long time.

2.2.2. Theorem. *Every FL sketch has an initial model in the category of sets.*

A direct proof is in [4]. By definition, an initial model has exactly one homomorphism of models to each other model. This apparently nonconstructive description is misleading, however, since the initial model for an FL sketch can be constructed Herbrand-style directly from the data in the sketch [29]. It is important for the observations in Section 6.5 to note that the construction does not use the FL theory generated by the sketch.

It is also important for later use to note that constants may be adjoined freely to the sorts of an FL sketch, with the result remaining an FL-sketch. This is because a constant is an operation whose domain is the terminal object, and an FL sketch can specify that a sort become the terminal object by adding a cone over the empty diagram with that sort at the vertex.

2.2.3. Definition (*Potential vs. actual*). One frequently says that a datum C in a sketch is "intended" to be or "becomes" an X, where X stands for an instance of a type of construction which can occur in a category. This means that in any model M of the sketch, $M(C)$ is an X. Similarly, if C "does" something, that means $M(C)$ does it. Thus if one of the cones in a sketch looks like



(3)

then in a model it becomes a product cone.

Such terminology is used occasionally here. However, because the forms introduced here have other types of things besides cones and cocones, I am introducing a consistent terminology for this phenomenon.

2.2.4. Definition. If a construction C in a sketch becomes an X in all models, where X is a type of construction which can occur in categories, then C is a *formal X*.

For example, the cone (3) is a "formal product", and a cocone in a sketch over a diagram of some shape S is a "formal colimit" of the diagram.

2.3. Levels of abstraction

This paper sketches various constructions at two levels of abstraction. In particular, it models sketches and forms (abstract ideas) using FL sketches (more abstract). To avoid confusion between the two levels of abstraction, objects and arrows at the higher level of abstraction are given mnemonic names in a sans serif typeface.

As an illustration, the following notation is used for certain of the objects and arrows of the FL sketch for categories [5, Section 4.4]:

- Ob is the object of objects,
- Ar is the object of arrows,
- CP is the object of composable pairs of arrows,
- comp : CP → Ar is the composition operation,
- firstfac : CP → Ar and secondfac : CP → Ar pick out the first and second factors of a composable pair.

3. Functional programming languages

A functional programming language has primitive data types with associated constants, and primitive operations. (Note the remarks in Section 3.1.4 concerning languages with variables.) It also has program-forming operations, often called *constructors*, such as composition of operations, the formation of record types, branching, and recursion.

The language consists of the set of all operations and types derivable from the primitive data types and primitive operations by applying appropriate constructors. It is thus generated by the primitive operations and types in the presence of the constructors: a different choice of constructors would give a different language, even with the same choice of primitive types and operations.

3.1. The category of a language

Under certain assumptions, a category $C(L)$ is naturally associated to a functional programming language L. These assumptions include the following:

(A.1) We must assume that there is a do-nothing operation id_A for each type A (primitive and constructed). When applied, it does nothing to the data.

(A.2) We add to the language an additional type called 1 which has the property that from every type A there is a unique operation to 1. We interpret each constant c of type A as an arrow $c: 1 \to A$. This incorporates the constants into the set of operations; they no longer appear as separate data.

(A.3) We assume the language has a composition constructor: take an operation f which takes something of type A as input and produces something of type B, and another operation g which has input of type B and output of type C; then doing one after the other is a derived operation (or program) typically denoted $f;g$ which has input of type A and output of type C.

(A.4) We assume that the operations are strictly typed; that is, each primitive operation is applicable to exactly one type of input and gives a specific type of output.

Functional programming languages generally have do-nothing operations and composition constructors, so (A.1) and (A.3) fit the concept as it appears in the literature. The language resulting from the change in (A.2) is operationally equivalent to the original language.

Assumption (A.4), unlike the others, is not innocuous and is not often satisfied in real languages. It can be repaired in an ad hoc way by splitting a primitive operation into a set of operations, one for each allowable type of input. A coherent solution to this problem, the problem of polymorphism, is the subject of current research by many workers.

3.1.1. Requirements. The composition constructor must be associative in the sense that, if either of $(f;g);h$ or $f;(g;h)$ is defined, then so is the other and they are the same operation. We must also require that $f;\mathrm{id}_B$ and $\mathrm{id}_A;f$ are defined and are the same operation as f. That is, we impose the equations $f;\mathrm{id}_B = f$ and $\mathrm{id}_A;f = f$ on

the language. Both these requirements are reasonable in that in any implementation, the two operations required to be the same would surely do the same thing.

3.1.2. Definition (*The category*). Under the preceding assumptions, a functional programming language L has a category structure $C(L)$ for which

(C.1) the primitive and derived types of L are the objects of $C(L)$,

(C.2) the primitive and derived operations of L are the arrows of $C(L)$,

(C.3) the source and target of an arrow are the input and output types of the corresponding operation,

(C.4) composition is given by the composition constructor,

(C.5) the identity arrows are the do-nothing operations.

The reader may wish to compare the discussion in [25]. (Michael Barr contributed materially to the present discussion.)

3.1.3. Discussion. $C(L)$ is a *model* of the language, not the language itself. For example, in the category $f;\mathrm{id}_B = f$, but in the language f and $f;\mathrm{id}_B$ are different source programs. This is in contrast to the treatment of languages using context-free grammars: a context-free grammar generates the actual language.

From one point of view, $C(L)$ is a model of the *syntax* of the language; from another point of view, it is arguable at least that $C(L)$ is the *most abstract denotational semantics*. Operational semantics, on the other hand, would presumably require the introduction of rewrite rules.

3.1.4. Variables. This discussion is concerned with functional programming languages in the sense of [2, 3]. Another widely held point of view is that functional programming means no assignment statements: variables may appear but are not assigned to. This is true of the lambda calculus, for example. A typed lambda calculus is well known to be equivalent to a cartesian closed category [24]. More than that a language with variables which are not assigned to can under rather general conditions be modeled as a category; the elimination of the variables can be done systematically as described in [6, Section 7.7].

3.2. Sketching a language

The preceding discussion suggests that the primitive types and operations of a programming language should be given as some type of sketch and that the language would then be the theory of the sketch. This would provide a systematic way of describing the semantics of the language as a model of the sketch; this model extends to a model of the theory, thus providing a uniform and mathematically sound passage from the specification (what the semantics does on the primitive types and operations, that is on the sketch) to the meaning of every legal expression in the language (object or arrow in the theory, which is $C(L)$).

There is no problem doing this for constructors describable via diagrams, cones and cocones. These include record types (products) and free unions (co-products). Unfortunately, there are many other constructors which apparently cannot be described by sketches, such as function space objects (exponentials) in cartesian closed categories, objects such as lists which are defined by recursion, and so on. The goal here is to generalize the concept of sketch to allow for this.

4. Constructor spaces

In this section some preliminary ideas are introduced. The point of view here is that a type of sketch is determined by the kind of category it is modeled in. Here, "kind of category" is translated into "model of a constructor space", which will set the stage for the definition of form in Section 6.1.1.

4.1. Basic concepts

4.1.1. Definition. A sketch $\boldsymbol{S}$ is a *subsketch* of a sketch $\boldsymbol{T}$ if the graph of $\boldsymbol{S}$ is a subgraph of the graph of $\boldsymbol{T}$ and all the diagrams, cones and cocones of $\boldsymbol{S}$ are diagram, cones, cocones respectively of $\boldsymbol{T}$.

4.1.2. Definition. A *constructor-space sketch* or *CS sketch* is an FL sketch E with the following properties:

(CS.1) E contains a copy of the FL sketch for categories as a distinguished subsketch, and

(CS.2) every object of the graph of E is the vertex of a cone over a diagram whose objects are in the FL theory for categories generated by the subsketch in (CS.1).

It may be useful to replace the distinguished subsketch in this definition with a morphism of sketches, but here the simpler definition will be used.

4.1.3. Definition. If E is a CS sketch, an *E-category* is a model of E, and an E-functor is a morphism of models of E.

The inclusion of the FL sketch for categories as a subsketch of a CS sketch induces an underlying functor U from the category of E-categories to the category of small categories.

It follows that an E-category is a small category, possibly with additional operations, each of whose domains and codomains are equationally defined tuples of objects and arrows. The equations defining these tuples may involve the additional operations (so the tuples are not in general merely diagrams). The additional operations may be required to satisfy certain equations.

An arrow in the category of E-categories is a morphism of models (natural transformation); it can be described somewhat imprecisely as a functor preserving this additional structure on the nose. How to relax this strict preservation property is a subject of current investigation.

4.1.4. Definition. A *constructor space* is the FL theory of a CS sketch.

The idea behind the name is that the sorts and operations of such an FL theory embody all the constructions that are possible in every model of the theory.

4.2. Examples

Most familiar examples of categories with extra structure are models of CS sketches (but functors must preserve the structure on the nose). Examples include categories with binary products, categories with finite limits, categories with finite colimits, and various permutations of these ideas. The FL sketch for categories with finite limits is described in [5, Section 4.4]. In Section 4.3 below, the CS sketch for categories with binary products is worked out in detail.

What is important for our purposes is that many types of categories with second-order structure are models of CS sketches. This includes in particular cartesian closed categories, locally cartesian closed categories, locoses, and toposes. Toposes are treated in [5, Section 4.4]. The way cartesian closed categories and locoses are handled is described in Sections 4.4 and 4.5 below. These are not done in detail, but the details can be filled in using the constructions in Section 4.3 as a guide.

4.2.1. Possible generalizations. The definition of constructor space given here seems adequate to handle the common constructors used or proposed in programming language semantics. It allows constructions based on equationally defined subtypes in one step. Freyd's original idea of essentially algebraic involved defining further subtypes based on those and operations defined on the previous subtypes. This multistep approach may well prove useful in future applications but is not considered here for simplicity's sake (and lack of compelling examples).

4.3. Categories with binary products

A CS sketch **DBP** for categories with distinguished binary products is described here. It is given in some detail as an example of what is involved in specifying a category with extra structure.

Note that a category with distinguished finite products has all finite nonempty products, but only the binary products are distinguished.

Although the presentation which follows is quite detailed, it is still incomplete. I have given the defining sorts and operations; some supplementary operations necessary for the commutative diagrams are only noted in passing. An example is the arrow ip in (C.2) below. The guiding principle is that I have omitted formal

description of those arrows (such as ip) which will exist in the constructor space anyway, given the sorts and operations that *are* defined.

(4.3.1) The graph of the sketch **DBP** has the following constituents:

(G.1) An object Cn, and arrows lproj, rproj : Cn → Ar. Cn is the formal set of cones over two objects and lproj and rproj will point out the left and right arrows of the cone in a model. (This is forced by (C.1) below.)

(G.2) An operation prod : Ob × Ob → Cn, which picks out the product cone for the pair of objects.

(G.3) An object Fi and arrows cone : Fi → Cn and fia : Fi → Ar. In a model, Fi will essentially be the set of pairs (c, h) where c is a cone and h is an arrow, subject to equations requiring in effect that h be a fill-in arrow from the cone c to the product of the base nodes of c. This is accomplished in (C.2) below.

(G.4) An operation ufi : Cn → Fi, which in a model takes a cone c to (c, h), where h is the unique arrow from c to the product of the base of c which commutes with projections. ufi is the formal "unique fill-in".

It is useful to give names to two induced arrows:

$$\mathsf{ver} = \mathsf{dom} \circ \mathsf{lproj} : \mathsf{Cn} \to \mathsf{Ob}$$

which in a model will pick out the vertex of a cone, and

$$\mathsf{ip} = \mathsf{prod} \circ \langle \mathsf{cod} \circ \mathsf{lproj}, \mathsf{cod} \circ \mathsf{rproj} \rangle : \mathsf{Cn} \to \mathsf{Cn}$$

which will give the product cone over the base of a cone. ip stands for "induced product".

(4.3.2) **DBP** has the following cones:

(C.1) A cone

$$\begin{array}{ccc} \mathsf{Cn} & \xrightarrow{\mathsf{lproj}} & \mathsf{Ar} \\ {\scriptstyle \mathsf{rproj}}\downarrow & & \downarrow{\scriptstyle \mathsf{dom}} \\ \mathsf{Ar} & \xrightarrow[\mathsf{dom}]{} & \mathsf{Ob} \end{array} \tag{4}$$

which will become a pullback forcing Cn to be the set of cones as described in (G.1).

(C.2) A pullback cone over a complicated diagram with vertex Fi which forces Fi to be the object described in (G.3).

$$\begin{aligned} [(c, h) \mid & \mathsf{dom}(h) = \mathsf{ver}(c) \ \& \ \mathsf{cod}(h) = \mathsf{ver} \circ \mathsf{ip}(c) \\ & \& \ \mathsf{comp} \circ \langle \mathsf{lproj} \circ \mathsf{ip}(c), h \rangle = \mathsf{lproj} \\ & \& \ \mathsf{comp} \circ \langle \mathsf{rproj} \circ \mathsf{ip}(c), h \rangle = \mathsf{rproj}] \end{aligned} \tag{5}$$

Then cone : Fi → Cn (which picks out c) and fia : Fi → Ar (which picks out h) are two of the projections from this limit. Thus Fi becomes the object of fill-in arrows satisfying the requisite commutativity conditions.

Note that the type of the expression $\langle \mathsf{lproj} \circ \mathsf{ip}(c), h \rangle$ in equation (5) is context-sensitive. Normally the angle brackets would make the expression denote an element of $\mathsf{Ar} \times \mathsf{Ar}$, but the presence of "comp" implies that it is an element of CP. This is acceptable since the square bracket notation for limits is only semi-formal; a machine implementation would require either more notation or compile-time type determination. Compare the way this problem is handled in equation (16) below.

(4.3.3) DBP has three diagrams. One is

$$\mathsf{Cn} \underset{\text{cone}}{\overset{\text{ufi}}{\rightleftarrows}} \mathsf{Fi} \tag{6}$$

which forces the arrows $\mathsf{ufi} : \mathsf{Cn} \to \mathsf{Fi}$ and $\mathsf{cone} : \mathsf{Fi} \to \mathsf{Cn}$ to be formal inverses to each other, thus ensuring the uniqueness of the fill-in arrow.

The other two diagrams express the formal equations

$$\mathsf{cod} \circ \mathsf{lproj} \circ \mathsf{prod} = p_1 : \mathsf{Ob} \times \mathsf{Ob} \to \mathsf{Ob}$$

and

$$\mathsf{cod} \circ \mathsf{rproj} \circ \mathsf{prod} = p_2 : \mathsf{Ob} \times \mathsf{Ob} \to \mathsf{Ob}$$

which say that the product cone should be over the correct pair of objects.

4.3.4. Remarks. In connection with this, it should be clear that another presentation involving different sorts or operations could give a category of models which is equivalent, but not in general isomorphic, to the category of models of the theory called **DBP** here. Thus to describe a category with binary products as a **DBP**-category (following Definition 4.1.3) is an invariant description if you cannot distinguish equivalent categories, but is presentation-dependent if you can distinguish categories that are nonisomorphic.

A presentation satisfying a stronger form of requirement (CS.2) of Definition 4.1.2 (namely that the arrows as well as the objects of the base diagram be in the FL theory for categories) will prove that the category of models of **DBP** is tripleable (monadic) over the category of categories. This follows from a theorem of Lair [22], who shows how to produce such presentations for categories with any particular kind of limits. Knowing that the models are monadic in cases like this may perhaps provide theoretical advantages when the subject is studied more deeply. On the other hand, the essentially algebraic presentation given here is considerably less complicated.

4.4. Cartesian closed categories

A cartesian closed category has finite products and exponential objects. Thus a CS sketch **CCC** can be built by adding new sorts and operations to the sketch for categories with binary objects described in Section 4.3.

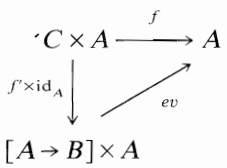
4.4.1. Finite products. To get all finite products you need to add an object Trm (the formal object of terminal objects) and an equationally-defined object U of all arrows

with codomain in Trm. You also need an operation $u: \mathsf{Ob} \to U$ picking out the unique arrow from an object to a terminal object. Its uniqueness can be ensured by adding an operation $v: U \to \mathsf{Ob}$ and equations making it the formal inverse of the image of u.

4.4.2. Function spaces. A function space object $[A \to B]$ in a cartesian closed category comes equipped with an arrow

$$\mathrm{ev}: [A \to B] \times A \to B \tag{7}$$

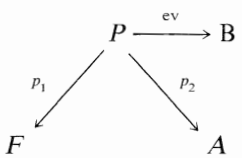
whose defining property is that for any arrow $f: C \times A \to B$ there is a unique arrow $f': C \to [A \to B]$ such that



(8)

commutes. We will call a system such as (7) a *function space system.*

(4.4.3) Thus to construct **CCC**, one needs an equationally defined object H of diagrams of the form



(9)

together with equations forcing P with p_1 and p_2 to be the product of F and A.

One needs two operations to force F to be a formal function space object. One is $\mathsf{fss}: \mathsf{Ob} \times \mathsf{Ob} \to H$ which constructs the formal function space system corresponding to the pair of objects. (An appropriate projection from H to Ob formally selects $[A \to B]$ itself.) The other operation is analogous to ufi in Section 4.3 (G.4); it picks out the arrow f' in diagram (8).

You may wish to contrast the more general treatment of closed categories in [19].

4.5. Locoses

A locos is a type of category studied by Cockett [8, 9] in which one can make a fairly general type of definition by recursion. One of the desirable properties of a locos is that if it is generated by decidable objects, then all its objects are decidable.

Here I will give an informal description of how to construct a CS sketch **LOC** whose models are locoses.

Let $\boldsymbol{C}$ be a **DBP** category. For each object A there is a category act(A) which is the category of algebras for the functor $A \times -$ with an underlying functor $U_A: \mathrm{act}(A) \to C$.

4.5.1. Definition. A **DBP** category $\boldsymbol{C}$ is *recursive* if for every object A the underlying functor U_A has a left adjoint $F_A: \boldsymbol{C} \to \mathrm{act}(A)$.

For an object B, denote $U_A(F_A(B))$ by $\mathrm{rec}(A, B)$; for $f: B \to C$, we obtain an arrow $\mathrm{rec}(A, f) = U_A(F_A(f)): \mathrm{rec}(A, B) \to \mathrm{rec}(A, C)$. The algebra $F_A(B)$ is an algebra structure $r(A, B): A \times \mathrm{rec}(A, B) \to \mathrm{rec}(A, B)$. In particular, if $\boldsymbol{C}$ has a terminal object, $\mathrm{rec}(1, 1)$ is a natural numbers object.

(4.5.2) With this notation, we define a CS sketch with a binary operation $\mathsf{rec}: \mathsf{Ob} \times \mathsf{Ob} \to \mathsf{Ob}$ which will take A and B to $\mathrm{rec}(A, B)$, and a unary operation from Ob to Ar yielding an arrow $\eta B: B \to \mathrm{rec}(A, B)$, with diagrams forcing ηB to have the correct domain and codomain.

The universal property of η is that for any arrow $f: B \to C$ and any algebra $c: A \times C \to C$, there is a unique arrow $g: \mathrm{rec}(A, B) \to C$ such that

$$\begin{array}{ccc} A \times \mathrm{rec}(A, B) & \xrightarrow{A \times g} & A \times C \\ {\scriptstyle r(A, B)}\downarrow & & \downarrow{\scriptstyle c} \\ \mathrm{rec}(A, B) & \xrightarrow[g]{} & C \end{array} \tag{10}$$

commutes (that is, g is the underlying arrow of an algebra map) and also

$$\begin{array}{ccc} B & \xrightarrow{f} & C \\ {\scriptstyle \eta B}\downarrow & \nearrow_{g} & \\ \mathrm{rec}(A, B) & & \end{array} \tag{11}$$

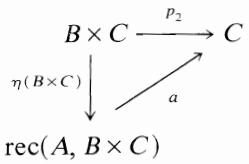
To obtain g requires another unary operation in the CS sketch **LOC**; its domain is the object representing the set of triplets (A, f, c), where A is an object and f and c are arrows with the same codomain C such that the domain of c is $A \times C$. The domain of g must be $\mathrm{rec}(A, B)$, where B is the domain of f. All these are equational conditions in the CS sketch being constructed, as are the commutativity of diagrams (10) and (11).

4.5.3. Remark. It is clear (and not a new observation) that any left adjoint F to a functor U with the property that each object $U(F(A))$ and each arrow $U(F(f))$ is a model of an FL sketch (one FL sketch for all objects and one for all arrows) can be prescribed by a sketch in an analogous way. One example of this is the construction of dependent products as a right adjoint to a slice functor. The constructions using adjoints in [18] are sketchable in this way.

(4.5.4) For objects A, B, C of a recursive category $\boldsymbol{C}$, the map $p_2: B \times C \to C$ produces a unique arrow

$$a: \mathrm{rec}(A, B \times C) \to C$$

with the property that



(12)

commutes. There is then an arrow

$$c(A, B, C) = \langle \mathrm{rec}(A, p_1), a \rangle : \mathrm{rec}(A, B \times C) \to \mathrm{rec}(A, B) \times C.$$

This arrow is determined by a universal property, and the object of all such arrows can clearly be specified in the same way as the object Fi in (G.3) above.

4.5.5. Definition. Let $\boldsymbol{C}$ be a recursive category. If for all A, B and C, $c(A, B, C)$ is an isomorphism, then $\boldsymbol{C}$ has *local recursion.* A *locally recursive category* is a recursive category with local recursion for which every slice category $\boldsymbol{C}/C$ is recursive.

Making $c(A, B, C)$ be an isomorphism requires a simple diagram in the CS sketch **LOC**. To make a slice category $\boldsymbol{C}/A$ recursive requires redoing the constructions described previously for making a category recursive, with all the constructions providing for an extra arrow to A for each object and for the commutativity of the diagrams defining morphisms in the slice.

4.5.6. Definition. A **DBP** category $\boldsymbol{C}$ is a *locos* if it has disjoint finite sums and is locally recursive.

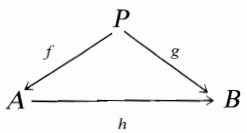
Disjoint finite sums are defined using pullbacks and the initial object, both of which can be specified in a CS sketch.

5. Sketches as elements of initial models

All the data of a sketch $\boldsymbol{S}$ can be described using operations and equationally defined objects in some CS sketch $\boldsymbol{E}$, with the objects and arrows of the graph adjoined as constants to the sketch. Equations must be added to the sketch to force the objects and arrows of the diagrams, cones and cocones to be specific objects and arrows of the graph. For this reason, the sketch $\boldsymbol{S}$ can be identified as an element of the initial model of the resulting sketch $\boldsymbol{E}$.

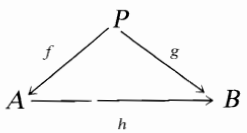
5.1. Example

For example, the (silly) sketch $\boldsymbol{P}$ with graph



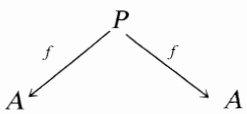
(13)

diagram



(14)

and one cone C



(15)

is an element of the initial model of a certain sketch E obtained from **DBP** by adjoining certain objects, operations and equations which will now be described.

$\boldsymbol{P}$ will be a constant of type Γ, where Γ is a cone determining a subobject of

$$\mathsf{Ob} \times \mathsf{Ob} \times \mathsf{Ob} \times \mathsf{Ar} \times \mathsf{Ar} \times \mathsf{Ar} \times \mathsf{CP} \times \mathsf{Cn}.$$

The subobject is defined equationally as follows:

$$\begin{aligned}\Gamma = [(x, y, z, u, v, w, r, s) \mid & \mathsf{dom}(u) = \mathsf{dom}(v) = x \\ & \& \ \mathsf{dom}(w) = \mathsf{cod}(u) = y \ \& \ \mathsf{cod}(w) = \mathsf{cod}(v) = z \\ & \& \ \mathsf{firstfac}(r) = u \ \& \ \mathsf{secondfac}(r) = w \ \& \ \mathsf{comp}(r) = v \\ & \& \ \mathsf{lproj}(s) = u \ \& \ \mathsf{rproj}(s) = u \ \& \ \mathsf{ver}(s) = x \ \& \ \mathsf{prod}(y, y) = s].\end{aligned} \tag{16}$$

Thus Γ is the limit of a certain diagram in the theory of **DBP**. As is customary, Γ will denote both the cone and the vertex of the cone.

Using the labels in (13), we can say that the constant is

$$(x, y, z, u, v, w, r, s) = (P, A, B, f, g, h, \langle h, f \rangle, C), \tag{17}$$

where C is the cone in (15).

5.1.1. Remarks. A diagram in a category is a graph homomorphism from a certain shape graph to the underlying graph of the category. (More generally, a diagram is a functor from a certain shape *category* to the category, but we do not need the greater generality here. The diagram-as-graph is a special case of diagram-as-functor via the free category generated by the graph.)

The cone Γ plays the role of the shape graph in the present context, although in a more sophisticated way. It is constructed as the limit of a diagram which pastes together formal arrows and objects so that they will have the correct domain and codomain relationships in a model. In this respect it is like the shape graph, and as in the case of ordinary diagrams, a particular constant of type Γ might well have certain of the objects or the arrows repeated (for example, one could have $P = A$ in equation (17)).

The cone Γ in Section 5.1, however, is over a diagram containing the nodes CP and Cn, which have no analog in a shape graph. They require certain diagrams in the graph to be commutative diagrams or cones respectively. Γ is best thought of as a *description* of the sketch (a *graph-based* rather than *linguistic* description).

This motivates the formal definition of the description of a form in Section 6.1.1.

6. Forms

The discussion in Section 5 shows how an informal description of a sketch can be formalized in such a way that the sketch becomes an element of a certain initial model. This is the basis of our definition of form.

6.1. Descriptions and forms

Let E be a finite CS sketch with theory $\mathbf{Th}(E)$ and let Γ be a limit cone in $\mathbf{Th}(E)$. Let $\mathbf{Th}(E, f)$ be the theory obtained from $\mathbf{Th}(E)$ by freely adjoining a constant f of type Γ. In a model of the sketch, which is a category with extra structure, Γ represents a particular collection of entities in that category; for example, the graph, diagram and cone of the sketch $\boldsymbol{P}$ in Section 5.1 was an element of Γ as described there. In a model of $\mathbf{Th}(E, f)$, f becomes a particular element of the collection represented by Γ.

Forgetting f provides an underlying functor U_f from models of $\mathbf{Th}(E, f)$ to models of $\mathbf{Th}(E)$, that is, to E-categories.

6.1.1. Definition. With the notation of the preceding paragraph, the *form* $\boldsymbol{F}$ *of type* E determined by Γ is the value of the constant f in the initial model of $\mathbf{Th}(E, f)$, and Γ is called the *description* of $\boldsymbol{F}$.

Since the initial model can be constructed inductively, this definition of the concept of form does not in fact depend on knowing the whole category of models, as is apparently implied by the fact that it is specified as an element of an initial model. The form is actually a term in a type of Herbrand model with a purely finitistic definition.

6.2. Models of forms

Suppose $\boldsymbol{F}$ is a form of type E with description Γ.

6.2.1. Definition. Let $\boldsymbol{F}$ be a form with f as in Definition 6.1.1. A *model* of $\boldsymbol{F}$ in an E-category $\boldsymbol{C}$ is the image of f in a model M of $\mathbf{Th}(E, f)$ for which $U_f(M) = \boldsymbol{C}$.

The image of f mentioned in the definition is uniquely determined because $\boldsymbol{F}$ is an element of an *initial* $\mathbf{Th}(E, f)$-model.

This definition of model of a form is holistic and not obviously constructive. However, as we shall see, a form has a graph just as a sketch has and the model can be defined in terms of the graph.

6.3. Morphisms of models

A homomorphism of sketches is defined to be a natural transformation between the graphs of the sketches. The definition of form does not require that a graph be specified. However, a form determines a graph.

6.3.1. Definition. Let $\boldsymbol{F}$ be a form with description Γ. The set $\mathcal{N}(\boldsymbol{F})$ of *nodes* of $\boldsymbol{F}$ consists of all the values of $p \circ f$ in the initial model of $\mathbf{Th}(E, f)$ for each operation $p : \Gamma \to \mathsf{Ob}$ of $\mathbf{Th}(E, f)$. The set $\mathcal{A}(\boldsymbol{F})$ of *arrows* of $\boldsymbol{F}$ consists of all the values of $q \circ f$ in the initial model of $\mathbf{Th}(E, f)$ for each operation $q : \Gamma \to \mathsf{Ar}$ of $\mathbf{Th}(E, f)$. The *graph* of $\boldsymbol{F}$ is the graph with objects $\mathcal{N}(\boldsymbol{F})$ and arrows $\mathcal{A}(\boldsymbol{F})$.

There are plenty of arrows $p : \Gamma \to \mathsf{Ob}$ and $q : \Gamma \to \mathsf{Ar}$ as described in this definition. By Definition 4.1.2, every object of the graph of E is the vertex of a cone over a diagram whose nodes are in the FL theory for categories. Because E is a *finite* sketch, the description Γ of the form $\boldsymbol{F}$ is a cone in $\mathbf{Th}(E, f)$ over a base diagram whose objects are defined as limits over finite diagrams with projections ultimately to objects of the FL sketch for categories, and so via dom, cod and comp to Ob and Ar.

If an arrow is the value of one of these projections, its domain and codomain are values of projections too, because of dom and cod. So the result is really a graph: if an arrow is in the graph, so is its source and target.

In practice, it is expected that a form of type E will be defined in the way a sketch is defined, by giving a graph and some entities which are elements of various sorts in the CS sketch E. Thus a form of type **DBP** is given by specifying a graph, some diagrams, and some discrete binary cones (elements of Cn), so is essentially an FP sketch (meaning all cones are over discrete diagrams). Note that this approach means that we specify the type of form ahead of time, and that for example a sketch (form) of type **DBP** is not the same as the sketch with the identical graph, diagrams and cones but which is specified as an FL sketch. Indeed, a sketch of type **DBP** from this point of view is not the same as a sketch of type **FP** where **FP** is some FL sketch whose models are categories with finite products (for example obtained by adjoining a sort for the terminal object).

6.3.2. Definition. Let $\boldsymbol{F}$ be a form determined by a constant f with description Γ, with models $M(\boldsymbol{F})$ and $N(\boldsymbol{F})$ in the same E-category $\boldsymbol{C}$. A *morphism of models* from

$M(\boldsymbol{F})$ to $N(\boldsymbol{F})$ is a graph homomorphism ψ from the graph of $M(\boldsymbol{F})$ to the graph of $N(\boldsymbol{F})$ in $\boldsymbol{C}$ for which

(MM.1) $\psi(M(p \circ f)) = N(p \circ f)$ for each operation $p : \Gamma \to \mathsf{Ob}$ of $\mathbf{Th}(E, f)$.

(MM.2) $\psi(M(q \circ f)) = N(q \circ f)$ for each operation $q : \Gamma \to \mathsf{Ar}$ of $\mathbf{Th}(E, f)$.

The models of a form $\boldsymbol{F}$ of type E in an E-category $\boldsymbol{C}$, together with their morphisms, form a category $\mathrm{Mod}(\boldsymbol{F}, \boldsymbol{C})$. Note that a morphism of models of a form of type **DBP**, for example, will automatically take limit cones to limit cones since the definition implies that the morphism takes cones to cones and the cones become limit cones in every model.

6.4. Examples of forms

It should be clear from the constructions in Section 5 that every finite sketch can be presented as a form. Every diagram can be factored into triangles, so that diagrams can be handled just as in Example 5.1. The description of the sketch in that example is the object Γ defined in equation (16). The following theorem is then clear, given the discussion in Example 5.1.

6.4.1. Theorem. *For every sketch $\boldsymbol{S}$ of type E there is a form $\boldsymbol{F}$ with an equivalent category of models.*

However, a cone by itself is also an example of a form. For example, a cone such as (3) is given by the simple description $\Gamma = \mathsf{Cn}$, where Cn is defined in (G.1).

Ordinary finite sketches are in some sense equivalent in expressive power to first order logic [17]. Forms are a type of higher-order sketch. For example, there is a form **FS** which constructs function spaces. Its "graph" is just the graph (9). In its description Γ will be an equation forcing that graph to become a function space system by forcing it to be the value of fss on the pair (A, B).

6.4.2. Reflexive objects. One can describe a form whose models in a cartesian closed category are objects D with $[D \to D]$ a retract of D. Thus one could use this in a programming language specification to ensure that a given data type allowed recursive definitions by fixed points. (This provides a methodology for *syntax*; it does not settle questions of semantics!)

I will use the notation in Section 4.4.3.

H contains a subobject D of function space systems of the form

$$\begin{array}{ccc} & & P \xrightarrow{\ \mathrm{ev}\ } D \\ & \overset{p_1}{\swarrow} & \qquad \searrow \overset{p_2}{} \\ [D \to D] & & \qquad D \end{array} \tag{18}$$

Let $\mathsf{fs} : D \to \mathsf{Ob}$ be the projection which takes a function space system of this type to $[D \to D]$, and let $\mathsf{bs} : D \to \mathsf{Ob}$ take the system to the base space D. Let $\mathsf{id} : \mathsf{Ob} \to \mathsf{Ar}$

be the operation which takes an object to its identity arrow. All these operations exist in the theory of **CCC**.

The description of the form is

$$\begin{aligned}\Gamma = [(s, i, j, r) \mid & \mathsf{dom}(i) = \mathsf{cod}(j) = \mathsf{fs}(s) \\ & \& \ \mathsf{cod}(i) = \mathsf{dom}(j) = \mathsf{bs}(s) \\ & \& \ \mathsf{firstfac}(r) = j \ \& \ \mathsf{secondfac}(r) = i \\ & \& \ \mathsf{comp}(r) = \mathsf{id}(\mathsf{fs}(s))].\end{aligned} \tag{19}$$

In a model M, this forces $M(j) \circ M(i) = \mathrm{id}_{[D \to D]}$. (Here, s is the diagram (18) and r is the formal composite $j \circ i$.)

One could describe this form this way. Its graph is given in (18) with two additional arrows $I : [D \to D] \to D$ and $J : D \to [D \to D]$. The part of the graph without these extra arrows is required to be of type D. A diagram forces $J \circ I = \mathrm{id}_{[D \to D]}$. This way of defining a form makes it look more like a true generalization of sketch than Definition 6.1.1.

6.5. Theories as initial models

The preceding definitions and observations provide a way of constructing the theory of an arbitrary sketch.

Let $\boldsymbol{S}$ be a sketch of a particular type. It specifically may have cocones as well as cones. Present it as a form in an essentially algebraic way analogous to Example 5.1, using a description Γ in the theory of a suitable sketch E for categories with the types of finite limits and/or colimits suitable to the type of sketch being considered. Γ will be equationally defined using objects $\mathsf{Cn}(\mathscr{I})$ and $\mathsf{Ccn}(\mathscr{J})$ of cones over diagrams of a specific shape $\mathscr{I}$ and cocones over diagrams of a specific shape $\mathscr{J}$ respectively, for various $\mathscr{I}$ and $\mathscr{J}$. Adjoin a constant e of type Γ to E. Then we have the following theorem.

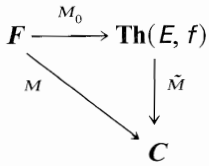
6.5.1. Theorem. *The initial model of* $\mathbf{Th}(E, e)$ *is equivalent to* $\mathbf{Th}(\boldsymbol{S})$, *and the universal model* M_0 *is inclusion.*

This is because a model of $\boldsymbol{S}$ is the image of the initial model in the model category. Thus the initial model satisfies the defining property of $\mathbf{Th}(\boldsymbol{S})$: Every model of $\boldsymbol{S}$ extends to a model of $\mathbf{Th}(\boldsymbol{S})$ which is determined uniquely up to natural isomorphism.

Following this, we make the following definition.

6.5.2. Definition. Let $\boldsymbol{F}$ be a form of type E determined by a constant f with description Γ. The *Theory* of $\boldsymbol{F}$ is the initial model of $\mathbf{Th}(E, f)$. The *universal model* M_0 of $\boldsymbol{F}$ is the inclusion of the value of the constant f in the initial model of $\mathbf{Th}(E, f)$.

6.5.3. Theorem. *In the notation of Definition* 6.5.2, *let* $\mathbf{C}$ *be an* E*-category. Every model* $M : \mathbf{F} \to \mathbf{C}$ *induces an* E*-functor* $\bar{M} : \mathbf{Th}(E, f) \to \mathbf{C}$ *for which*



(20)

commutes. Moreover, $\mathbf{Th}(E, f)$ *is uniquely determined up to equivalence of categories by this fact.*

6.6. Conclusion

The finite specification given by the form automatically generates a category (the theory) which contains the ingredients in the form and which is closed under all the constructions available in an E-category. Moreover, the objects and arrows of the theory can be constructed recursively from the objects and arrows of the form. The last theorem then says that any functorial semantics defined on the form is automatically defined on the theory.

Forms thus provide a uniform method for defining primitive types and operations to which specified constructors are to be applied, in a way which allows consistent semantics to be defined by giving it only on the primitive part. The only restriction is that the constructors be specifiable in an essentially algebraic way.

Acknowledgment

I have learned much discussing these ideas with Michael Barr, J.R.B. Cockett, C. Lair, François Lamarche, Colin McLarty and John Power. I am grateful to Michael Barr, John Power and the referees for many helpful suggestions and corrections.

References

[1] P. Ageron, Categorical semantics of higher order type constructors, Lecture at Journées d'Etude *Esquisse, Logique et Informatique Théoretique*, July, 1988.

[2] J. Backus, The algebra of function programs: function level reasoning, linear equations, and extended definitions, in: J. Diaz and I. Ramos, eds., *Formalization of Programming Concepts*, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981) 1–43.

[3] J. Backus, Is computer science based on the wrong fundamental concept of 'program'?, in: J. W. de Bakker and J.C. van Vliet, eds., *Algorithmic Languages* (North-Holland, Amsterdam, 1981).

[4] M. Barr, Models of sketches, *Cahiers Topologie Géom. Différentielle* **27** (1986) 93–107.

[5] M. Barr and C. Wells, *Toposes, Triples and Theories*, Grundlehren der Mathematischen Wissenschaften **278** (Springer, Berlin, 1985).

[6] M. Barr and C. Wells, *Category Theory for Computing Science*, Preliminary version of textbook (1989).

[7] A. Bastiani and C. Ehresmann, Categories of sketched structures, *Cahiers Topologie Géom. Différentielle* **10** (1968) 104–213.
[8] J.R.B. Cockett, Locally recursive categories, Preprint, Department of Computer Science, University of Tennessee, Knoxville, TN, 1987.
[9] J.R.B. Cockett, On the decidability of objects in a locos, in: J.W. Gray and A. Scedrov, eds., *Categories in Computer Science and Logic*, Contemporary Mathematics (American Mathematical Society, Providence, RI, 1989).
[10] L. Coppey, Catégories de Peano et catégories algorithmiques, recursivité, *Diagrammes* **12** (1984).
[11] L. Coppey and C. Lair, Algébricité, monadicité, esquissabilité et non-algébricité, *Diagrammes* **13** (1985) 1–111.
[12] P. Freyd, Aspects of topoi, *Bull. Austral. Math. Soc.* **7** (1972) 1–72.
[13] J. Gray, A categorical treatment of polymorphic operations, in: M. Main, A. Melton, M. Mislove and D. Schmidt, eds., *Mathematical Foundations of Programming Language Semantics*, Lecture notes in Computer Science **298** (Springer, Berlin, 1988) 2–22.
[14] J. Gray, Categorical aspects of data type constructors, *Theoret. Comput. Sci.* **50** (1987) 103–135.
[15] J. Gray, The category of sketches as a model for algebraic semantics, in: J.W. Gray and A. Scedrov, eds., *Categories in Computer Science and Logic*, Contemporary Mathematics (American Mathematical Society, Providence, RI, 1989).
[16] R. Guitart, On the geometry of computations, *Cahiers Topologie Géom. Différentielle* **27** (1986) 107–136.
[17] R. Guitart and C. Lair, Limites et co-limites pour representer les formules, *Diagrammes* **7** (1982) 1–112.
[18] T. Hagino, A typed lambda calculus with categorical type constructors, in: D. Pitt, A. Poigné and D. Rydeheard, eds., *Category Theory and Computer Science*, Lecture Notes in Computer Science **283** (Springer, Berlin, 1987) 140–157.
[19] G.M. Kelly, Examples of non-monadic structures on categories, *J. Pure Appl. Algebra* **18** (1980) 59–66.
[20] G.M. Kelly, On the essentially-algebraic theory generated by a sketch, *Bull. Austral. Math. Soc.* **26** (1982) 45–56.
[21] C. Lair, Foncteurs d'omission de structures algébriques, *Cahiers Topologie Géom. Différentielle* **12** (1971) 447–486.
[22] C. Lair, Condition syntaxique de triplabilité d'un foncteur algébrique esquissé, *Diagrammes* **1** (1979).
[23] C. Lair, Trames et sémantiques catégoriques des systèmes de trames, *Diagrammes* **18** (1987).
[24] J. Lambek and P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge Studies in Advanced Mathematics **7** (Cambridge University Press, Cambridge 1986).
[25] D. Pitt, Categories, in: D. Pitt, A. Poigné and D. Rydeheard, eds., *Category Theory and Computer Programming*, Lecture Notes in Computer Science **240** (Springer, Berlin, 1986).
[26] H. Reichel, *Initial Computability, Algebraic Specifications and Partial Algebras* (Clarendon Press, Oxford, 1987).
[27] E.G. Wagner, Categories data types and imperative languages, in: D. Pitt, A. Poigné and D. Rydeheard, eds., *Category Theory and Computer Programming*, Lecture Notes in Computer Science **240** (Springer, Berlin, 1986).
[28] E.G. Wagner, Algebraic theories, data types and control constructs, *Fund. Inform.* **9** (1986) 343–370.
[29] C. Wells and M. Barr, The formal description of data types using sketches, in: M. Main, A. Melton, M. Mislove and D. Schmidt, eds., *Mathematical Foundations of Programming Language Semantics*, Lecture Notes in Computer Science **298** (Springer, Berlin, 1988) 490–527.