ISSN 0105-8517

TYPE ALGEBRAS, FUNCTOR CATEGORIES, AND BLOCK STRUCTURE

Frank J. Oles

DAIMI PB-156
January 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

PB-156 F.J. Oles: Type Algebras, Functor Categories, and Block Structure

TRYK: DAIMI/RECAU

This is a preprint of a paper given at the U.S.-French Joint Symposium on the Applications of Algebra to Language Definition and Compilation, June 9-15, 1982, Fontainebleau. References to this paper should cite the Proceedings of that Symposium.

TYPE ALGEBRAS, FUNCTOR CATEGORIES, AND BLOCK STRUCTURE*

Frank J. Oles

Aarhus University

Aarhus, Denmark

ABSTRACT In this paper we outline a category-theoretic approach to the semantics of ALGOL-like languages in which particular attention is paid to the use of functor categories as a mechanism to reflect stack discipline. Also, we explore the idea that implicit conversions can be modelled by making the phrase types of a language into a poset, and we show how any poset freely generates a type algebra.

* Work supported by National Science Foundation grant MCS-8017577.

1. The Utility of Functor Categories

An intuitive grasp of the nature of variable declarations and of what is happening as one enters and exits from blocks is essential to programming in an ALGOL-like language. However, a precise semantic description of the constructs involved is difficult, and it is particularly difficult if one wants semantics for block structure that mesh elegantly with semantics for procedures.

Our goal is to outline in general terms how functor categories can be used to explain the semantics of ALGOL-like languages which possess

1) a rich type structure,

2) higher-order procedures, whose types may be arbitrarily complex,

3) imperative capabilities,

and

4) block structure.

The principal intuition we will try to capture is that during the execution of a program written in a language that obeys a stack discipline not only is the store changed by commands (statements), but also the shape of the store is changed by variable declarations during block entrances and by block exits.

An extended description of the characteristics of the kinds of languages in which we are interested can be found in [6]. We have not worked out the details of the semantics of every languages feature mentioned there, but we do know how to deal with the most important features, and we can tell a complete story about block structure.

On some points our treatment will be sketchy. A detailed discussion of this approach to the semantics of programming languages (including much more on types, implicit conversions, substitution, the algebraic nature of languages, the role of homomorphisms in semantics, and continuation semantics) can be found in the author's Ph.D. dissertation [5], written under the supervision of John C. Reynolds. By generously sharing his ideas and time, he was of invaluable assistance in completing this work.

We have a vision of a method for dealing with the semantics of ALGOL-like languages. Of course, there are many ALGOL-like languages, but at the heart of each is something that we can call an _appropriate setting_. In due time we will precisely define the components of an appropriate setting. Informally, it consists of a collection of _sets of storable values_, _coercion conventions_ that indicate when values of one kind can be used uniformly as values of another kind, and _operations_ on the sets of storable values that we want to implement. An appropriate setting is a basis for creating a typed programming language with both imperative features and arbitrarily complex procedures.

We hold strongly to the view that programming languages should be presented as free algebras of some sort. Nowadays, this is not a very radical position; it only slightly generalizes the point of view in [1]. It is rooted in three observations. First, programming languages are algebras because phrases are formed by algebraically combining other phrases. Second, the constituent subphrases of any phrase are uniquely determined. Third, the process of phrase decomposition eventually results in indecomposable phrases.

It would be nice to have a language for each appropriate setting, which we might term <u>Desugared ALGOL</u> (DA) for that setting, whose features were as general and extensive as possible, and whose semantics were elegantly described. Then giving semantics for an arbitrary ALGOL-like language A might proceed in two steps:

1) identify the underlying appropriate setting for A, thereby determining DA, and
2) explain how to translate all phrases of A into phrases of DA, thereby fixing the semantics of A.

We have in mind a rather precise notion of translation to be used in the second step. In general, A will be presented as a free algebra of one kind, generated by a set G_A, whereas DA will be presented as a free algebra of another kind, generated by a set G_{DA}. The key is then to describe how DA can be viewed as an algebra of the same kind as A; then by defining a function

$$\text{translation} \in G_A \rightarrow DA$$

one automatically obtains a homomorphism

$$\text{Translation} \in A \rightarrow DA.$$

Does DA already exist? Since procedures are so important to an ALGOL-like language, two candidates immediately present themselves. On one hand, perhaps we could use an untyped λ-calculus for DA. The lack of any type structure is an obvious and important drawback. On the other hand, maybe an ordinary typed λ-calculus suffices for DA. Here we raise four objections. A major objection is the lack of implicit conversions between types. Another major objection is that for a typed λ-calculus type-checking is trivial; one cannot say anything meaningful about type-checking if all phrases under consideration are properly typed. A less important, but not insignificant, problem is that all the identifiers are typed at the outset, preventing an adequate reflection of the fact that in a real programming language an identifier can have a variety of types even in single program. Finally, although this is a matter of taste, a typed λ-calculus is inherently a many-sorted algebra. We would prefer to deal with a one-sorted algebra if this is possible.

By the way, we reject an unnatural the idea that giving the semantics of a typed programming language hinges on giving a set of possible meanings for phrases. Just as the study of data types naturally involves many-sorted algebras rather than one-sorted algebras, so also programming language semantics naturally requires some categories with a variety of different meaning objects. Giving semantics for a phrase requires both picking out a meaning object (possibly a set) and also assigning a meaning involving the meaning object. Speaking of the set of meanings for all phrases at the wrong point can mislead our intuition. It turns out that the means of integrating the preference for one-sorted languages with the necessity for many-sorted semantic categories is provided by the activity of type-checking.

Unfortunately we cannot yet describe Desugared ALGOL for each appropriate setting, but we can give an approximation to it that can be called Basic Desugared ALGOL. The major deficiency of Basic Desugared ALGOL is that it does not completely handle generic operators and polymorphism, although it takes some steps in that direction. Technically, each appropriate setting gives rise to a language L (Basic Desugared ALGOL) as a special instance of a family of languages called coercive λ-calculi (called "coercive typed λ-calculi" in [5]) that are obtained by a modified λ-calculus construction. One of their features is that the only binding mechanism is λ-abstraction, and this is important because of the difficulty of handling binding elegantly.

In this section we will be outlining an approach to the nature of denotation for Basic Desugared ALGOL based on functor categories. It is our feeling that, far from being abstract nonsense, functor categories are a natural vehicle for giving mathematical substance to intuitions about stack discipline. It is our aim to draw a sharp distinction between change of state and change of underlying structure. Block entrances and exits in connection with variable declarations change the shape of the store (underlying structure) whereas commands alter the store (state). We hope that this work will eventually find application in constructing implementations of languages and in proving correctness of programs. Also, it leads us to speculate that there may be important results to be found relating classes of languages (with semantic descriptions, of course!) with the "structure of their implementation", much like results relating formal languages to the automata that recognize them.

We will simplify things by avoiding in this paper any contact with language features that involve continuations, even though jumps and labels do not involve theoretical obstacles.

Before going further we must review some definitions and establish some notation. A predomain is a directed-complete poset. A domain is a predomain with a minimal element, usually denoted ⊥ ("bottom"). A function from one predomain to another is *continuous* if it preserves least upper bounds of directed subsets. The category whose objects are predomains (respectively, domains) and whose morphisms are continuous functions, with the usual composition, is denoted *Pdom*(respectively, *Dom*). Let *Set* denote the category of sets. Each set may be regarded a predomain by imposing upon it a discrete partial ordering. Thus, *Set* and *Dom* are full subcategories of *Pdom* . (Recall that a subcategory is *full* if it is the largest subcategory possessing its particular collection of objects.) As general references on category theory we suggest [2], [3], and [4].

From each predomain P we may create a domain $P_\perp$ by simply adding an element not already in P as a minimal element. In most contexts the added minimal element is thought of as an "undefined element of P".

We abhor unnecessary parentheses. Therefore for functional application we write f x rather than f(x). Functional application associates to the left, so that f g h x means (((f g) h) x). Also, application binds more tightly than anything else, so that F a × G b should be read as (F a) × (G b). However, if a few extra parentheses really enhance readability, we shall throw them in.

Suppose $\mathcal{C}$ is a category. The collection of objects of $\mathcal{C}$ is denoted Ob $\mathcal{C}$ and the collection of morphisms (or arrows) of $\mathcal{C}$ is denoted Ar $\mathcal{C}$. The collection of arrows from $X \varepsilon$ Ob $\mathcal{C}$ to $Y \varepsilon$ Ob $\mathcal{C}$ is denoted $X \xrightarrow[\mathcal{C}]{} Y$. We suppress the subscript when $\mathcal{C} = \mathit{Set}$ so that $X \to Y$ is the set of functions from X to Y, and we write $f \ \varepsilon \ X \to Y$ rather than $f : X \to Y$. We also use $\mathcal{C} \to \mathcal{D}$ to denote the collection of all functors from the category $\mathcal{C}$ to the category $\mathcal{D}$. Do not confuse $\mathcal{C} \to \mathcal{D}$ with $\mathcal{C} \Rightarrow \mathcal{D}$, where the latter denotes the category whose objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and whose arrows are natural transformations. We confess to using a plethora of notational arrows, but we claim their meanings are always clear from context.

Let K be a category with finite products specifically given. In [4], K is defined to be a <u>Cartesian closed category</u> if three particular kinds of right adjoints are given. However, in order to establish notation we give the following more verbose, but less abstract, definition. The category K is Cartesian closed if

(1) K has a distinguished terminal object,

(2) K has a distinguished binary product functor, and

(3) for each ordered pair Y, Z ε Ob K there is Y $\Rightarrow$ Z ε Ob K and

$$\mathrm{Ap}_K \langle Y, Z\rangle \ \varepsilon\ (Y \Rightarrow Z) \times Y \underset{K}{\longrightarrow} Z$$

such that whenever X ε Ob K and

$$f \ \varepsilon\ X \times Y \underset{K}{\longrightarrow} Z$$

there exists a unique arrow

$$\mathrm{Ab}_K \langle X, Y, Z\rangle f \ \varepsilon\ X \underset{K}{\longrightarrow} (Y \Rightarrow Z)$$

making the diagram

$$\begin{tikzcd}
X \times Y \arrow[rr, "\mathrm{Ab}_K \langle X, Y, Z\rangle f \times 1_Y"] \arrow[dr, "f"'] & & (Y \Rightarrow Z) \times Y \arrow[dl, "\mathrm{Ap}_K\langle Y,Z\rangle"] \\
& Z &
\end{tikzcd}$$

commute in K.

Some authors write Z^Y instead of $Y \Rightarrow Z$, which provides a partial explanation of why the $\Rightarrow$ construction is sometimes called exponentiation. It is not hard to see that, for a Cartesian closed category K, there is a unique way to extend the definition of $\Rightarrow$ to give a functor

$$- \Rightarrow - \in K^{\mathrm{op}} \times K \longrightarrow K.$$

Thus $\Rightarrow$ is sometimes called an "internal" hom functor. When X, Y, and Z are understood, we write

Ab or Ab_K for $\mathrm{Ab}_K\langle X, Y, Z\rangle$ ("abstraction")

and

Ap or Ap_K for $\mathrm{Ap}_K\langle Y, Z\rangle$ ("application").

The categories *Set*, *Pdom*, and *Dom* are all Cartesian closed. For *Set*,

$$Y \Rightarrow Z = Y \to Z.$$

For *Pdom* and *Dom*, $Y \Rightarrow Z$ is the set of continuous functions from Y to Z, partially ordered by $f \leqq g$ if $f\ y \leqq g\ y$ for all $y \in Y$. In all three categories, application is functional application and abstraction is currying.

It is time to turn our attention to a more careful specification of what an appropriate setting is. An appropriate setting has syntactic components and semantic components:

Syntactic components - $\mathcal{D}$, Op, Id

Semantic components - Val, m_{Op}, Σ.

The syntactic component $\mathcal{D}$ is a <u>poset of storable data types</u>. Each relation $\delta \leqq_{\mathcal{D}} \delta'$ indicates an implicit conversion from the data type δ to the data type δ'. Next there is the collection

Op of *operators*, which comes equipped with two functions

$$\text{arity} \; \varepsilon \; \text{Op} \rightarrow \mathcal{D}^*, \qquad \text{res} \; \varepsilon \; \text{Op} \rightarrow \mathcal{D},$$

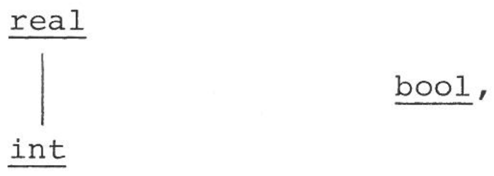
where res abbreviates result. The final syntactic component is Id, an infinite set of *identifiers*.

The semantic component Val ε $\mathcal{D} \rightarrow \mathit{Set}$ is a *value functor* from $\mathcal{D}$, regarded as a category, to the category of sets. Thus a set is given for each storable data type, and an implicit conversion function for each relation $\delta \leq_{\mathcal{D}} \delta'$. By m_{Op} we mean an Op-indexed collection of *interpretations* of operators; if g ε Op, arity g $= \langle \delta_1, \ldots, \delta_n \rangle$, and res g $= \delta$, then

$$m_{\text{Op}} \; g \;\; \varepsilon \; \text{Val} \; \delta_1 \times \cdots \times \text{Val} \; \delta_n \rightarrow \text{Val} \; \delta.$$

The last semantic component is Σ, the *category of store shapes*, whose nature and use we will discuss shortly.

For example, it may be that $\mathcal{D}$ is given by the Hasse diagram



and Op, arity, and res are given by the following table,

g ε Op	arity g	res g
true	<>	*bool*
false	<>	*bool*
and	<*bool*, *bool*>	*bool*
0	<>	*int*
1	<>	*int*
π	<>	*real*
+	<*real*, *real*>	*real*
succ	*int*	*int*

It would then be reasonable for Val to take $\mathcal{D}$ to the diagram

$$\begin{tikzcd} \mathbb{R} & & \\ & & \{\mathit{true}, \mathit{false}\} \\ \mathbb{Z} \arrow[uu, "\text{inclusion}"'] & & \end{tikzcd}$$

and for the interpretations of the operators,

$$m_{\mathrm{Op}}\ \mathrm{true}\ \varepsilon\ \{\langle\rangle\} \to \{\mathit{true}, \mathit{false}\}$$
$$m_{\mathrm{Op}}\ \mathrm{false}\ \varepsilon\ \{\langle\rangle\} \to \{\mathit{true}, \mathit{false}\}$$
$$m_{\mathrm{Op}}\ \mathrm{and}\ \varepsilon\ \{\mathit{true}, \mathit{false}\} \times \{\mathit{true}, \mathit{false}\} \to \{\mathit{true}, \mathit{false}\}$$
$$m_{\mathrm{Op}}\ 0\ \varepsilon\ \{\langle\rangle\} \to \mathbb{Z}$$
$$m_{\mathrm{Op}}\ 1\ \varepsilon\ \{\langle\rangle\} \to \mathbb{Z}$$
$$m_{\mathrm{Op}}\ \pi\ \varepsilon\ \{\langle\rangle\} \to \mathbb{R}$$
$$m_{\mathrm{Op}}\ +\ \varepsilon\ \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$m_{\mathrm{Op}}\ \mathrm{succ}\ \varepsilon\ \mathbb{Z} \to \mathbb{Z}$$

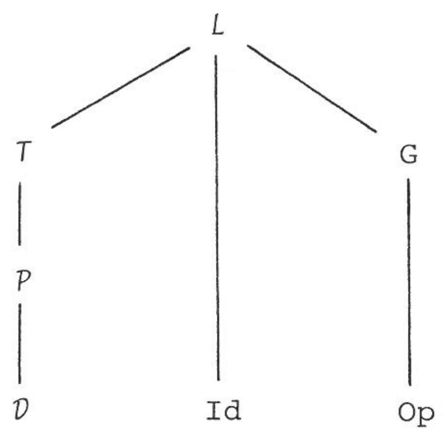
to be the obvious ones.

An important idea is that for each data type $\delta \in \mathcal{D}$, Val δ is a set of storable values.

Anyone familiar with initial algebra semantics, as presented in [1] for instance, can see that Op is a $\mathcal{D}$-sorted signature. However, we maintain that because of implicit conversions there are many more sensible expressions than are found in an initial $\mathcal{D}$-sorted Op-algebra, e.g., $+ \langle 1, \pi \rangle$.

Instead of partially ordering $\mathcal{D}$, why not just add to Op another operation for each implicit conversion? The answer is that implicit conversions exist conceptually on a lower level than operations; they have special properties that are not shared by arbitrary operations. For instance, whereas there can be many operations with argument δ and result δ', there can be at most one implicit conversion from δ to δ' - otherwise it wouldn't be implicit.

The syntactic construction of $\mathcal{L}$ from $\mathcal{D}$, Op, and Id requires some intermediate steps. It may help to contemplate the following diagram



The poset $\mathcal{D}$ of data types is used to construct $\mathcal{P}$, the *poset of primitive phrase types*. For phrase types, $\tau \leqq \theta$ means roughly that phrases of τ can be meaningfully and uniformly used in contexts calling for phrases of type θ.

Each data type $\delta \varepsilon \mathcal{D}$ gives rise to three primitive phrase types:

1) δ-*exp* (for δ-expression),

2) δ-*var* (for δ-variable),

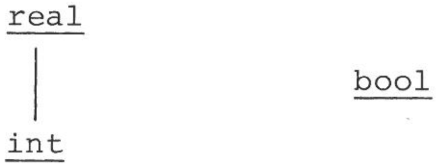
3) δ-*acc* (for δ-acceptor).

Another primitive phrase type is *comm* (for command). The reader probably has an excellent intuitive feeling for the nature of phrases that are assigned the phrase types δ-*exp*, δ-*var*, and *comm*. A phrase has type δ-*acc* if it gobbles up values of type δ and produces commands. Thus, if x and y are δ-variables, then in the assignment command x := y the δ-variable y is used as a δ-expression and the δ-variable x is used as a δ-acceptor. (Hence, even if a language turns out not actually to have any phrases of type δ-*acc*, it is still useful to introduce that phrase type in order to explain δ-*var*.)

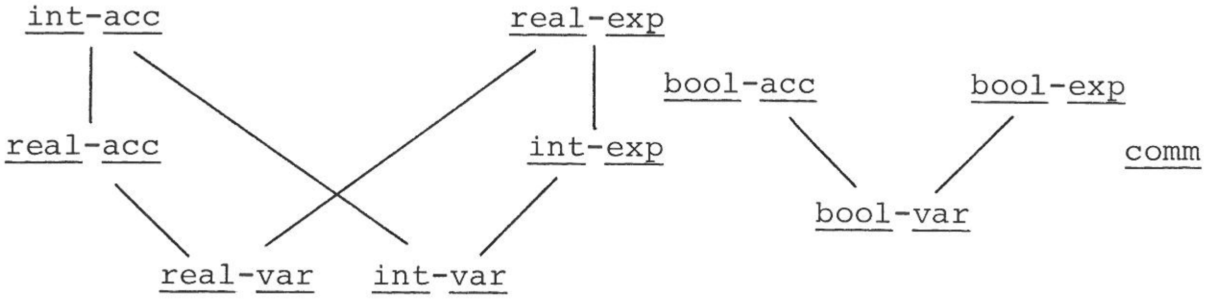
The partial order on $\mathcal{D}$ is given by

$$
\begin{aligned}
\pi \leq_{\mathcal{P}} \pi' \text{ iff } & \pi = \pi', \\
\text{or } & \pi = \delta\text{-}\underline{\text{exp}},\ \pi' = \delta'\text{-}\underline{\text{exp}}, \text{ where } \delta \leq_{\mathcal{D}} \delta', \\
\text{or } & \pi = \delta'\text{-}\underline{\text{acc}},\ \pi' = \delta\text{-}\underline{\text{acc}}, \text{ where } \delta \leq_{\mathcal{D}} \delta', \\
\text{or } & \pi = \delta\text{-}\underline{\text{var}},\ \pi' = \delta'\text{-}\underline{\text{exp}}, \text{ where } \delta \leq_{\mathcal{D}} \delta', \\
\text{or } & \pi = \delta'\text{-}\underline{\text{var}},\ \pi' = \delta\text{-}\underline{\text{acc}}, \text{ where } \delta \leq_{\mathcal{D}} \delta'.
\end{aligned}
$$

Note that *comm* cannot be compared with the other primitive phrase types. For example, if $\mathcal{D}$ were given by



then $\mathcal{P}$ would be



For more discussion, see [8]. (However, unlike [8], we do not permit fully generic operations. As a consequence, the $\delta_1\delta_2$-variable construction is not needed.)

Why is $\mathcal{P}$ so complicated? Why not just use $\mathcal{D}$ or $\mathcal{D} \cup \{\underline{\text{comm}}\}$? Well, $\mathcal{D}$ would be adequate only for a language that lacked imperative features. Furthermore, to catch all errors due to mismatched types of compile time we must distinguish carefully between δ-variables and δ-expressions. Thus $\mathcal{D} \cup \{\underline{\text{comm}}\}$ is inadequate for an imperative language.

From $\mathcal{P}$ we construct $\mathcal{T}$, the *poset of all phrase types*. Technically, $\mathcal{T}$ is the *free type algebra* generated by $\mathcal{P}$, and the details of its construction are the subject of the second part of this paper. In particular, we defer discussion of the partial order on $\mathcal{T}$. An important phrase type of $\mathcal{T}$ is $\mathcal{T}_{ns}$, the nonsense type; it is the type assigned to phrases which contain errors due to mismatched types, thereby allowing us to meaningfully discuss type-checking. Also, for all elements τ, θ of $\mathcal{T}$, there is a phrase type $\tau \Rightarrow \theta$ that is assigned to procedures accepting arguments (i.e., having formal parameters) of type τ and producing results (i.e., having calls) of type θ. The use of the same symbol for both procedural phrase types and exponentiation in a Cartesian closed category is, of course, intentional.

The entity G is a *typed set of generators*; it is derived from Op. As the terminology indicates, G comes equipped with a function

$$\text{type} \ \varepsilon \ G \to \mathcal{T}$$

which assigns a phrase type to each generator. Structurally G is a disjoint union of Op and another set Op'. For g ε Op let

$$\text{type } g = \delta_1\text{-}\underline{\text{exp}} \Rightarrow \delta_2\text{-}\underline{\text{exp}} \Rightarrow \cdots \Rightarrow \delta_n\text{-}\underline{\text{exp}},$$

where arity g = $\langle\delta_1,\ldots,\delta_n\rangle$ and res g = δ. So, partly because product types are not in $\mathcal{T}$, G contains curried versions of operators in Op. This works out very neatly. The typed set Op' is given by the following table, where $\delta \ \varepsilon \ \mathcal{D}$ is a data type and $\tau \ \varepsilon \ \mathcal{T}$ is a phrase type.

$\underline{g \ \varepsilon \ \mathrm{Op}'}$	$\underline{\mathrm{type}\ g}$
skip	$\underline{\mathrm{comm}}$
;	$\underline{\mathrm{comm}} \Rightarrow \underline{\mathrm{comm}} \Rightarrow \underline{\mathrm{comm}}$
$:=_\delta$	$\delta\text{-}\underline{\mathrm{acc}} \Rightarrow \delta\text{-}\underline{\mathrm{exp}} \Rightarrow \underline{\mathrm{comm}}$
$\mathrm{ifthenelse}_\tau$	$\underline{\mathrm{bool}}\text{-}\underline{\mathrm{exp}} \Rightarrow \tau \Rightarrow \tau \Rightarrow \tau$
rec_τ	$(\tau \Rightarrow \tau) \Rightarrow \tau$
newvar_δ	$(\delta\text{-}\underline{\mathrm{var}} \Rightarrow \underline{\mathrm{comm}}) \Rightarrow \underline{\mathrm{comm}}$

It is the presence of Op' as a subset of G that entitles us to regard L as Basic Desugared ALGOL. Intuitively, skip is the "do nothing" command, the operator ; concatenates commands, $:=_\delta$ is assignment for the data type δ, $\mathrm{ifthenelse}_\tau$ is the conditional for the phrase type τ, rec_τ is used in the desugared denotation of recursively defined entities, and newvar_δ is used to desugar declarations of δ-variables.

We are now in a position to describe how L comes from T, Id, and G. The idea is that for each g ε G, x ε Id, $\tau \ \varepsilon \ T$ and for arbitrary phrases ℓ, $m \ \varepsilon \ L$, we want

$$g, \ [\![x]\!] \ , [\![\ell \ m]\!], \text{ and } [\![\lambda x{:}\tau.\ell]\!]$$

to be in L. Thus we create a signature Λ (for 1-sorted algebras) such that there is a 0-ary operator for each identifier, there is a unary operator $\lambda x{:}\tau$ for each identifier x and each phrase type τ, and there is a binary operator ap. Then L is the free Λ-algebra generated by G. So the language L is generated by operations rather than identifiers; the identifiers are incorporated into the signature. This is not just a technical trick. It is in keeping with the intuition that how identifiers are used in programming languages never varies; what varies are the operations and their meanings. Note also that despite the appearance of phrase types in Λ, L is a 1-sorted algebra like the untyped λ-calculus. Thus there is a single set of parseable phrases that we seek to analyze. This is simple and satisfying. (But as we said before, we reject the idea that there is a single set of meanings for phrases. Having a single set of meanings for phrases in a typed language is as unnatural as positing the existence of a set of "all elements" as a basis for set theory.) Of course, since L is 1-sorted, is has in it many strange phrases, such as

$$[\![\, [\![\, + \; :=_{\underline{\mathrm{int}}}]\!] \quad \mathrm{ifthenelse}_{\underline{\mathrm{comm}}}]\!] \, .$$

However the nonsensical semantics of such expressions can be handled elegantly and simply. Furthermore, we need such phrases to understand the nature of type-checking. The idea is that the algebra L consists not necessarily of those phrases meaningful at run-time, but rather is the collection of parseable phrases, and that the compiler can check if a parseable phrase will be meaningful at run-time.

On the subject of binding, note that identifiers are untyped except in the scope of a $\lambda x{:}\tau$ operator. The intended intuitive semantics of $[\![\lambda x{:}\tau . \ell]\!]$ is that it represents a procedure which can be applied to any argument that can be coerced into type τ.

As an aid to understanding L, we offer the following table of popular ALGOL phrases and their informal translation into L.

Favorite Phrase	Translation
$c_1 ; c_2$	$[\![\ [\![\ ;\ c_1\ ]\!]\ c_2\ ]\!]$
if b then c_1 else c_2	$[\![\ [\![\ [\![\ \text{ifthenelse}_{\text{comm}}\ b]\!]\ c_1]\!]\ c_2]\!]$
while b do c	$[\![\ \text{rec}_{\text{comm}}\ [\![\lambda x{:}\text{comm}.\ell\ ]\!]\ ]\!]$ where ℓ = translation of if b then $(c;x)$ else skip
let $x{:}\tau$ be m in n	$[\![\ [\![\ \lambda x{:}\tau.n]\!]\ m\ ]\!]$
letrec $x{:}\tau$ be m in n	$[\![\ [\![\ \lambda x{:}\tau.n\ ]\!] [\![\ \text{rec}_\tau [\![\ \lambda x{:}\tau.m]\!]\ ]\!]\ ]\!]$
begin δ-var x; c end	$[\![\ \text{newvar}_\delta [\![\ \lambda x{:}\delta\text{-var}.c\ ]\!]\ ]\!]$

Observe that the informal meaning of $[\![\text{rec}_\tau [\![\ \lambda x{:}\tau.\ell]\!]\ ]\!]$ is "the most natural solution of $x = \ell$". Note that we make explicit in L that variable declaration involves two steps theoretically: (1) the binding of an identifier to a phrase type, thereby creating a λ-expression, and (2) the creation of a command from the λ-expression.

A syntactic matter of fundamental importance is type-checking. We have a function type $\varepsilon\ G \rightarrow T$. Can we extend it to a function Type $\varepsilon\ L \rightarrow T$? For a simple reason, the answer is no. The problem is the typeless nature of unbound occurrences of identifiers. A quick way out might be to assign types only to phrases in which all identifiers are bound, but this is not good because we would not be able to express the type of a phrase in terms of the types of its components. A better solution is to recognize that the function Type requires another argument whose purpose if to give typing information about identifiers. Perhaps Type $\varepsilon\ L \rightarrow (T^{\text{Id}} \rightarrow T)$, i.e., maybe the extra argument for Type is a function from Id to T.

This is a workable idea, but not a desirable one. There are two objections. The mathematical objection, which can only really be appreciated after a careful study of the semantics of L, is that using functions defined on all of Id eventually leads to the requirement that certain infinite products ("infinite environments", loosely speaking) exist in categories used for semantics, thus limiting the generality of this approach. More accessible is the programming language objection: using functions defined on all of Id means all identifiers at all times have a type, an assertion at odds with intuition. Before program execution begins, no identifier has a type; during execution some have types, and some don't. Keep in mind also that the assignment of types to identifiers has a dynamic aspect in that an identifier has the ability to be bound to different phrase types in different blocks. So we are led to introduce A, the poset of phrase type assignments. A phrase type assignment α is a function

$$\alpha \in \mathrm{I} \longrightarrow T$$

where $\mathrm{I} = \mathrm{dom}\ \alpha$ is a finite set of identifiers. Thus,

$$\mathrm{emp}_T \in \phi \rightarrow T,$$

the phrase type assignment with empty domain, is the phrase type assignment most appropriate to the beginning of program execution. The partial order on A is such that $\alpha \leqq \beta$ is interpreted as "α can be used for β", i.e.

$$\alpha \leqq \beta \text{ iff } \begin{cases} (1) \ \mathrm{dom}\ \beta \subseteq \mathrm{dom}\ \alpha, \text{ and} \\ (2) \ \alpha\ x \leqq \beta\ \ x \text{ for all } x \in \mathrm{dom}\ \beta. \end{cases}$$

Thus the proper functionality of Type is

$$\text{Type} \in L \to (A \Rightarrow T),$$

where $A \Rightarrow T$ denotes the monotone functions from A to T, although the reader may here ignore monotonicity if he finds it unpalatable. When $\ell \in L$, $\alpha \in A$, we read Type $\ell\alpha$ as "the type of ℓ in the context of a phrase type assignment α." The astute reader may feel that we have not vanquished the problem of typeless identifiers because the domain of a phrase type assignment is finite; possibly even empty. However the judicious use of the nonsense type T_{ns} for those identifiers saves the day. We cannot here delve into the nature of Type except to assert that it is a Λ-algebra homomorphism.

This completes the discussion of the syntactic nature of L. Next we turn to the nature of the semantic component Σ and how Val, m_{Op}, and Σ jointly determine the semantics of L.

An integral part of the semantics of a typed language is the assignment to each phrase type τ of a meaning, denoted Mng τ. Perhaps with the plan of arranging matters so that program fragments of type τ denote elements of Mng τ, one might suppose Mng τ is a set. However, the possible existence of nonterminating programs, which lead to an "undefined" state, provides an inducement to partially order Mng τ, where the relation $x \leqq y$ means x is "more undefined" than y. For instance see [7] and [9]. Following this line of reasoning, a command which never terminates, such as while true do skip, denotes the minimal element in Mng comm. The need to give meanings to recursively defined expressions of type τ causes us to require that directed subsets of Mng τ have least upper bounds, i.e., that Mng τ is a predomain. (Also, we generally want Mng τ to have a minimal element, but we must tread cautiously at this point to avoid becoming overcommitted to the use of *Dom* rather than *Pdom*. As we shall see later, *Dom* is technically inadequate.)

Suppose we try to give the semantics of programs which do not contain block entrances and exits. We start by positing the existence of a set S of possible stores. Regard sets as discretely ordered predomains. Since a command is a transformation of S that possibly may not terminate, and a function from S to $S_\perp$ is the same as a continuous function from S to $S_\perp$, we expect

$$\text{Mng}\ \underline{\text{comm}} = S \Rightarrow S_\perp .$$

Here $\Rightarrow$ is the internal hom functor for *Pdom*.
Also, for each data type δ,

$$\begin{aligned}\text{Mng}\ \delta\text{-}\underline{\text{exp}} &= S \Rightarrow (\text{Val}\ \delta)\ ,\\ \text{Mng}\ \delta\text{-}\underline{\text{acc}} &= \text{Val}\ \delta \Rightarrow \text{Mng}\ \underline{\text{comm}},\\ \text{Mng}\ \delta\text{-}\underline{\text{var}} &= \text{Mng}\ \delta\text{-}\underline{\text{acc}} \times \text{Mng}\ \delta\text{-}\underline{\text{exp}}.\end{aligned}$$

In other words, a δ-expression attempts to compute a value of type δ from the current store, a δ-acceptor uses a value of type δ to update the current store, and a δ-variable may be used as either a δ-expression or a δ-acceptor. Finally, for all $\tau, \theta \in T$, we expect

$$\text{Mng}\ (\tau \Rightarrow \theta) = \text{Mng}\ \tau \Rightarrow \text{Mng}\ \theta ,$$

i.e., the predomain of meanings for the procedural phrase type $\tau \Rightarrow \theta$ is the predomain of continuous functions from Mng τ to Mng θ.

Although the approach of the preceding paragraph is attractively comprehensible, it is inadequate for the semantics of block structure because the set S is fixed throughout the exposition. The whole point of block structure is to permit S to vary during program execution. For instance, if we view stores as being functions from finite sets of locations in memory to the set of data-type values, then the domains of those functions may be regarded as store shapes. Variable declarations at the start of a block alter the shape of the store by adding locations to it, whereas block exit restores the shape of the store to its condition at block entrance. The semantics of a language obeying a stack discipline should reflect this dynamic behavior.

Therefore, let Σ be the collection of all store shapes. To each $X \in \Sigma$, there is a set St X of stores of that shape. Since the meaning of $\tau \in T$ varies with the store shape, Mng τ is not a predomain, but is rather a Σ-indexed collection of predomains. For instance we might arrange matters so that

$$\begin{aligned}
\text{Mng } \underline{\text{comm}}\ X &= \text{St } X \Rightarrow (\text{St } X)_{\perp},\\
\text{Mng } \delta\text{-}\underline{\text{exp}}\ X &= \text{St } X \Rightarrow (\text{Val } \delta)_{\perp},\\
\text{Mng } \delta\text{-}\underline{\text{acc}}\ X &= \text{Val } \delta \Rightarrow \text{Mng } \underline{\text{comm}}\ X,\\
\text{Mng } \delta\text{-}\underline{\text{var}}\ X &= \text{Mng } \delta\text{-}\underline{\text{exp}}\ X \times \text{Mng } \delta\text{-}\underline{\text{acc}}\ X,
\end{aligned}$$

where $X \in \Sigma, \delta \in \mathcal{D}$..

It is important to realize that for $\tau \in T$ and $X, Y \in \Sigma$ the predomains Mng τ X and Mng τ Y cannot be arbitrarily different. After all, we want the notion of command to have a uniform meaning for all store shapes or else the definition of operations like ; (concatenation of commands) will be bizarrely complicated. For instance, consider the program skeleton

```
          begin int-var x;
                .
                .
                .
                x := 3;
                .
                .
                .
                begin bool-var y;
                      .
                      .
                      .
                      x := 3;
                      .
                      .
                      .
                end;
                .
                .
                .
          end.
```

Suppose X is the store shape corresponding to the outer block and Y is the store shape corresponding to the inner block. Then Mng comm X is relevant to the first occurrence of the assignment command x := 3, while Mng comm Y is relevant to the second occurrence. However, both occurrences are meant to alter the contents of the same location. Roughly speaking, the fact that X can be "expanded" to give Y induces a function from Mng comm X to Mng comm Y. So it becomes important to contemplate the notion of an expansion from a store shape X to a store shape Y. Certainly, expansions ought to be composable. The composition ought to be associative. For each store shape X, there ought to be an identity expansion which involves "doing nothing" to X. In short, we assert that we erred in letting Σ be the collection of store shapes. From now on, take Σ to be the category of store shapes. The morphisms of Σ are called expansions. Furthermore, for each phrase type τ we should require that Mng τ be a functor

$$\text{Mng } \tau \in \Sigma \longrightarrow \mathcal{P}dom;$$

this will elegantly take care of how an expansion

$$\sigma \in X \xrightarrow[\Sigma]{} Y$$

induces a function

$$\mathrm{Mng}\ \tau\ \sigma \in \mathrm{Mng}\ \tau\ X \xrightarrow[\mathit{Pdom}]{} \mathrm{Mng}\ \tau\ Y.$$

In this paper we shall not be more explicit about the nature of Σ, but a more comprehensive treatment would require that Σ be a precisely determined category. Actually, there are a number of candidates for Σ. A very general one can be found in [5], and another can be found in [6].

Procedural phrase types are a bit tricky. Let $\tau, \theta \in T$. Recall that in the simpler setting the predomain of meanings of type $\tau \Rightarrow \theta$ was the set of continuous functions from Mng τ to Mng θ. One might hope that in the more sophisticated setting, where Mng $(\tau \Rightarrow \theta)$ is to be a functor, that the set of procedural meanings of type $\tau \Rightarrow \theta$ in the context of a store shape X would be the set of continuous functions from Mng τ X to Mng θ X, i.e.,

$$\mathrm{Mng}\ (\tau \Rightarrow \theta)\ X = \mathrm{Mng}\ \tau\ X \Rightarrow \mathrm{Mng}\ \theta\ X.$$

Alas, this does not define a functor because $\Rightarrow$ in *Pdom* is contravariant in its first argument. (For the same reason, one cannot define a functor from *Set* to *Set* by diagonalizing the hom functor, i.e., by letting X to go X → X.) Another idea might be to recall that Mng τ and Mng θ are objects of a functor category and to try letting Mng $(\tau \Rightarrow \theta)$ be the set of natural transformations from Mng τ to Mng θ. That's plain nonsense, because there is no way to regard such a set of natural transformations as a functor. We are, however, getting closer to the heart of the matter. In [5] we prove that the functor category $\Sigma \Rightarrow \mathit{Pdom}$ is Cartesian closed. Therefore, the appropriate equation governing meanings of procedural types is just

$$\text{Mng } (\tau \Rightarrow \theta) = \text{Mng } \tau \Rightarrow \text{Mng } \theta,$$

where the heavy arrow on the right is exponentiation in the functor category $\Sigma \Rightarrow \mathit{Pdom}$. Further evidence of the essential rightness of this equation will be presented at the conclusion of this section.

The reader may wonder why we didn't engineer this discussion so as to end up with the functor category $\Sigma \Rightarrow \mathcal{D}om$. Unfortunately, contrary to the claim in [6], it does not appear that $\Sigma \Rightarrow \mathcal{D}om$ is a Cartesian closed category, in spite of the fact that $\mathcal{D}om$ is Cartesian closed.

This all sounds nice enough, but we have lost somewhere the idea that phrases of type τ should have denotations which are elements of Mng τ, because Mng τ doesn't seem to have any elements if it is a functor. The solution is to reject even in the simple setting where block structure is ignored the intuition that a phrase of type τ denotes an element of Mng τ. Actually, this is a rather conventional notion. This is where environments enter semantics. Each function $A \in \text{Id} \rightarrow T$ determines a predomain E A of environments by taking E A to be the product in Pdom of the Id-indexed collection $\{\text{Mng } (A\ x) \mid x \in \text{Id}\}$. Thus, each element of E A associates a meaning of the right type to each identifier. Then even in the simpler setting we conceive of the denotation of a phrase of type τ as being a (continuous) function from E A to Mng τ. We have thus been led to the idea that the semantics of a phrase is a morphism in a category from an environment object to a meaning object. We will give more details shortly, but at least we need not worry that Mng τ is a functor rather than a set of elements.

The question is how to describe environments in a way that meshes smoothly with our intuitions about store shapes and about the connections between identifiers and phrase types. The answer is that each phrase type assignment α determines an environment Env α, which is a functor in $\Sigma \Rightarrow \mathit{Pdom}$, defined by taking the product in $\Sigma \Rightarrow \mathit{Pdom}$ of the (dom α)-indexed collection of functors $\{\text{Mng } (\alpha\ x) \mid x \in \text{dom } \alpha\}$. If X is a store shape, then Env α X is the product in Pdom of the (dom α)-indexed collection of predomains $\{\text{Mng } (\alpha\ x)\ X \mid x \in \text{dom } \alpha\}$, which is a "conventional environment". In fact, if we regard the poset A as a category, then

$$\text{Env} \in A \rightarrow (\Sigma \Rightarrow \mathit{Pdom})$$

is a functor.

What we have seen, then, is that the meanings of types and consequently the semantics of L depends on the category Σ of store shapes and the functor Val $\in \mathcal{D} \rightarrow \mathcal{Set}$. Of course, we must still tackle the basic question of what phrases of L actually denote. Whatever denotations are, on the most concrete level determination of the denotation of $\ell \in L$ requires a phrase type assignment $\alpha \in A$ and a store shape X $\in$ Ob Σ. Let Den be the function that assigns denotations to phrases of L. It is then intuitively satisfying to assert

$$\text{Den } \ell\ \alpha\ X \in \text{Env } \alpha\ X \underset{\mathit{Pdom}}{\longrightarrow} \text{Mng } (\text{Type } \ell\ \alpha)\ X,$$

i.e., the denotation of ℓ in the context α when the store shape in X is a continuous function from the conventional environment for this situation to the predomain of meanings for ℓ for this situation. Now, in order to incorporate the intuition that when there is an expansion from the store shape X to the store shape Y the denotations Den ℓ α X and Den ℓ α Y must be consistent with one another, we assert

$$\text{Den } \ell\ \alpha \in \text{Env } \alpha \xrightarrow[\Sigma \Rightarrow Pdom]{} \text{Mng (Type } \ell\ \alpha),$$

i.e., Den $\ell\,\alpha$ is a natural transformation from the functor Env $\alpha \in \Sigma \to Pdom$ to the functor Mng (Type ℓ $\alpha) \in \Sigma \to Pdom$. How does m_{Op} enter into this? In [5] we show that m_{Op} canonically determines denotations for all the generators of the free Λ-algebra L; then it can be shown that there is in certain sense a uniquely determined function Den which extends the denotations of the generators to all phrases of L. A pleasant byproduct of the development in [5] is that Den ℓ is itself a natural transformation:

$$\text{Den } \ell \in \text{Env} \xrightarrow[A \Rightarrow (\Sigma \Rightarrow Pdom)]{} \text{Mng} \circ \text{Type } \ell.$$

Our final topic in this section is a discussion of how this approach lends itself to an explanation of the interactions between procedures and block structure. However, we must preface this with the definition of the functor

$$\text{hom}^{\Sigma} \in \Sigma^{op} \to K, \text{ where } K = \Sigma \Rightarrow Pdom.$$

The usual hom functor is

$$\text{hom}_{\Sigma} \in \Sigma^{op} \times \Sigma \to \mathcal{Set}.$$

Let

$$\text{E} \in \mathcal{Set} \longrightarrow Pdom$$

be the embedding functor which gives a discrete partial order to each set. By composing, we get

$$E \circ \hom_{\Sigma} \in \Sigma^{op} \times \Sigma \longrightarrow \mathit{Pdom}.$$

Then curry to get

$$\hom^{\Sigma} \in \Sigma^{op} \longrightarrow (\Sigma \Rightarrow \mathit{Pdom}).$$

Thus, for store shapes X and Y, $\hom^{\Sigma}$ X Y is $\hom_{\Sigma}\langle X,Y\rangle = X \xrightarrow[\Sigma]{} Y$, equipped with a discrete partial order.

Let's take a look at a phrase $\ell \in L$, which in the context of $\alpha \in A$, has a procedural type; say

$$\text{Type } \ell\ \alpha = \tau \Rightarrow \theta$$

where $\tau, \theta \in T$. Recall that we have an equality of functors:

$$\text{Mng } (\tau \Rightarrow \theta) = \text{Mng } \tau \Rightarrow \text{Mng } \theta.$$

Therefore,

$$\text{Den } \ell\ \alpha \in \text{Env } \alpha \xrightarrow[K]{} \text{Mng } \tau \Rightarrow \text{Mng } \theta.$$

Suppose the procedure ℓ is defined in a program in a block whose store shape is X. In this context, the denotation of the procedure is

$$\text{Den } \ell\ \alpha\ X \in \text{Env } \alpha\ X \xrightarrow[\mathit{Pdom}]{} (\text{Mng } \tau \Rightarrow \text{Mng } \theta)\ X.$$

A peek at the proof in [5] that K is a Cartesian close category shows that the predomain (Mng $\tau \Rightarrow$ Mng θ) X is obtained by partially ordering the set of natural transformations

$$(\hom^{\Sigma}\ X) \times (\text{Mng } \tau) \xrightarrow[K]{} \text{Mng } \theta.$$

Let $e \in \text{Env } \alpha$ X be the conventional environment existing when ℓ is defined. Then

$$\text{Den } \ell\ \alpha\ X\ e \in (\hom^{\Sigma}\ X) \times (\text{Mng } \tau) \xrightarrow[K]{} \text{Mng } \theta.$$

Now suppose the procedure ℓ is called in an interior block whose store shape is Y. Thus, there is an expansion

$$\sigma \varepsilon X \xrightarrow[\Sigma]{} Y.$$

The denotation of ℓ constructed at the time of its definition should be clearly connected with an element of the predomain Mng τ Y $\Rightarrow$ Mng θ Y, which is the proper collection of conventional meanings at the time of the call. Note that

$$\text{Den } \ell \ \alpha \ X \ e \ Y \ \varepsilon \ (X \xrightarrow[\Sigma]{} Y) \times (\text{Mng } \tau \ Y) \xrightarrow[Pdom]{} (\text{Mng } \theta \ Y).$$

Using the fact that *Pdom* is a Cartesian closed category we obtain

$$\text{Ab}_{Pdom} \ (\text{Den } \ell \ \alpha \ X \ e \ Y) \ \varepsilon \ (X \xrightarrow[\Sigma]{} Y) \xrightarrow[Pdom]{} (\text{Mng } \tau \ Y \Rightarrow \text{Mng } \theta \ Y).$$

Finally, apply this function to the specific expansion connecting the shape X of the store at the time of definition to the shape Y of the store at the time of the call, and we obtain

$$\text{Ab}_{Pdom} \ (\text{Den } \ell \ \alpha \ X \ e \ Y) \ \sigma \ \varepsilon \ \text{Mng } \tau \ Y \Rightarrow \text{Mng } \theta \ Y,$$

a conventional meaning for the procedure at the time of call. We have captured here two important intuitions about the meanings of procedures:

(1) The environment used to determine the meaning of a procedure is not the environment existing at the moment of the call, but is rather the environment existing when the procedure is defined. (An intuition described by other approaches as well.)

(2) The meaning of a procedure is also dependent on the store existing at the moment of call, the store existing at the moment of definition, and the expansion connecting them.

2. Type Algebras

Our aim in this section is the creation of mathematical objects that define collections of phrase types assignable to meaningful program fragments. These objects are certain "type algebras" freely generated by posets whose elements are viewed as primitive phrase types. These free type algebras are adequate for the developments in [5], but they don't provide the final answer to the question "What is a type?" In particular we don't delve into the nature of recursively defined phrase types.

Some points to keep in mind are these. The phrase types should be partially ordered and constructed in a natural way from a poset of primitive phrase types. Also it should be possible to define a functor Mng to some semantic category ($\Sigma \Rightarrow \mathcal{P}dom$ for Basic Desugared ALGOL; perhaps $\mathcal{S}et$ or $\mathcal{P}dom$ for some simpler language) by specifying its restriction to the primitive phrase types.

We will need to be a bit more careful about our notations for posets in this section. Our terminology is as follows. A partial order on a set X is a relation on X, i.e., a subset of X×X, which is reflexive, antisymmetric, and transitive. An ordered pair

$$\mathcal{P} = \langle \text{Ob } \mathcal{P}, \text{ Ar } \mathcal{P} \rangle$$

is a poset if Ob $\mathcal{P}$ is a set and Ar $\mathcal{P}$ is a partial order on Ob $\mathcal{P}$. The assertion $\langle x, y \rangle \in \text{Ar } \mathcal{P}$ is usually written $x \leq_{\mathcal{P}} y$. In addition the expression $x \leq_{\mathcal{P}} y$ is also used to denote the ordered pair $\langle x, y \rangle$ when it is true that $\langle x, y \rangle \in \text{Ar } \mathcal{P}$. The correct reading of $x \leqq_{P} y$ will be clear from context. The poset P becomes a category if we make the canonical definitions:

(1) $\mathrm{dom}_{P}\ (x \leqq_{P} y) = x$,

(2) $\mathrm{cod}_{P}\ (x \leqq_{P} y) = y$,

(3) $(y \leqq_{P} z) \circ (x \leqq_{P} y) = (x \leqq_{P} z)$, and

(4) $1_{P}\ x = (x \leqq_{P} x)$.

For posets P and Q, the object part of any functor

$$F \in P \rightarrow Q$$

is a monotone function, i.e., $x \leqq_{P} y$ implies $F\ x \leqq_{Q} F\ y$. Conversely, any monotone function $F \in \mathrm{Ob}\ P \rightarrow \mathrm{Ob}\ Q$ has a unique extension to a functor $F \in P \rightarrow Q$. The terms "embedding" and "full" in the context of posets have the following meanings. Following [3] rather than [4], a functor is an embedding if it is injective on both objects and arrows. A functor $F \in P \rightarrow Q$ is full if for all $x, y \in \mathrm{Ob}\ P$,

$$x \leqq_{P} y \quad \text{iff} \quad F\ x \leqq_{Q} F\ y.$$

Let R be the poset whose objects are monotone functions from P to Q and whose partial order is given by

$$f \leqq_{R} g \quad \text{iff} \quad f\ x \leqq_{Q} g\ x \quad \text{for each } x \in \mathrm{Ob}\ P.$$

We may compare the canonical category derived from R with the functor category $P \Rightarrow Q$ created from the canonical categories derived from P and Q. It is easy to see that they are isomorphic. Indeed, the clarity of the subsequent exposition is not harmed by writing $R = P \Rightarrow Q$ and $f \leqq_{P \Rightarrow Q} g$ rather than $f \leqq_{R} g$.

We use ϕ to denote both the empty set and the poset whose underlying set of objects is empty. Similarly we use $\{a\}$ to denote both the singleton set and the singleton poset whose sole object is a. We say that P is a subposet of Q (notation: $P \subseteq Q$) if Ob $P \subseteq$ Ob Q and Ar $P \subseteq$ Ar Q; in this case the inclusion mapping is monotone and is consequently a functor. If the inclusion functor is full, i.e. for all x, y ε Ob P

$$x \leqq_P y \quad \text{iff} \quad x \leqq_Q y,$$

then P is a full subposet of Q (notation: $P \sqsubseteq Q$). The intersection $P \cap Q$ of posets P and Q is defined by Ob $(P \cap Q)$ = (Ob P) $\cap$ (Ob Q) and Ar $(P \cap Q)$ = (Ar P) $\cap$ (Ar Q). Thus

$$x \leqq_{P \cap Q} y \quad \text{iff} \quad x \leqq_P y \text{ and } x \leqq_Q y.$$

Of course, $P \cap Q$ is a subposet of both P and Q. If $P \cap Q = \phi$, then the posets P and Q are disjoint.

There are two natural ways of constructing binary products of posets. First, consider $P \times Q$, the product of the ordered pair $\langle P,Q \rangle$ of posets viewed as categories. Alas, we cannot in general take <Ob $(P \times Q)$, Ar $(P \times Q)$> and obtain a poset because Ar $(P \times Q)$ = Ar $P \times$ Ar Q is a subset of (Ob $P \times$ Ob P) $\times$ (Ob $Q \times$ Ob Q) which implies Ar $(P \times Q)$ is not a relation on Ob $(P \times Q)$. Second, consider the poset $P \underline{\times} Q$ given by

(1) Ob $(P \underline{\times} Q)$ = Ob $P \times$ Ob Q

(2) $\langle x, y \rangle \leqq_{P \underline{\times} Q} \langle x', y' \rangle$ iff $x \leqq_P x'$ and $y \leqq_Q y'$.

Of course, the definition of $\underline{\times}$ has an obvious generalization to more than two factors. If we consider the second product $P \underline{\times} Q$ as a category in the canonical way, then it is isomorphic to the first product $P \times Q$ with the isomorphism being

$$I \ \varepsilon \ P \times Q \rightarrow P \underline{\times} Q,$$

the functor which is the identity function on objects and which on arrows satisfies

$$I\ \langle x \leqq_{P} x',\ y \leqq_{Q} y'\rangle = (\langle x, y\rangle \leqq_{P \underline{\times} Q} \langle x', y'\rangle).$$

Given two disjoint posets, the utility of connecting them together by providing a common top, i.e., maximal element, will soon be seen. So suppose P and Q are disjoint posets and $t \notin (\mathrm{Ob}\ P) \cup (\mathrm{Ob}\ Q)$. The poset $P{*}t{*}Q$ is defined by

$$\mathrm{Ob}\ (P{*}t{*}Q) = (\mathrm{Ob}\ P) \cup \{t\} \cup (\mathrm{Ob}\ Q)$$

and

$$x \leqq_{P*t*Q} y \quad \text{iff} \quad x \leqq_{P} y \text{ or } y = t \text{ or } x \leqq_{Q} y.$$

Observe that one way to completely specify a functor $F \in P{*}t{*}Q \to C$ is to let $F\ t$ = term where term is a terminal object of C and to define functors in $P \to C$ and $Q \to C$ that are the restrictions of F to P and Q.

The first proposition summarizes some evident properties of full subposets.

<u>Proposition 2.1</u>: Suppose $P \sqsubseteq Q$ and $P' \sqsubseteq Q'$.

(1) $P^{\mathrm{op}} \sqsubseteq Q^{\mathrm{op}}$.

(2) $P \underline{\times} P' \sqsubseteq Q \underline{\times} Q'$

(3) If Q and Q' are disjoint, and $t \notin (\mathrm{Ob}\ Q) \cup (\mathrm{Ob}\ Q')$ then Q, Q', and $P{*}t{*}P'$ are all full subposets of $Q{*}t{*}Q'$.

<u>Proof</u>: Trivial. □

A moment's thought shows that the union of two posets P and Q cannot be defined by mimicking the definition of their intersection because $(\mathrm{Ar}\ P) \cup (\mathrm{Ar}\ Q)$ is not in general a partial order on $(\mathrm{Ob}\ P) \cup (\mathrm{Ob}\ Q)$. However, this simple-minded approach is entirely adequate for dealing with unions of ascending chains of posets, a matter we now take up.

Consider an ascending chain

$$\mathcal{P}_0 \subseteq \mathcal{P}_1 \subseteq \mathcal{P}_2 \subseteq \cdots$$

of posets, which gives rise to a diagram

$$(*) \qquad \mathcal{P}_0 \xrightarrow{J_0} \mathcal{P}_1 \xrightarrow{J_1} \mathcal{P}_2 \xrightarrow{J_2} \cdots$$

where each J_i is an inclusion mapping. We can form the poset

$$\mathcal{P} = \cup \{\mathcal{P}_i \mid i \varepsilon \mathbb{N}\}$$

by letting OB $\mathcal{P} = \cup \{\text{Ob } \mathcal{P}_i \mid i \varepsilon \mathbb{N}\}$ and Ar $\mathcal{P} = \cup \{\text{Ar } \mathcal{P}_i \mid i \varepsilon \mathbb{N}\}$. Thus

$$x \leqq_{\mathcal{P}} y \quad \text{iff} \quad x \leqq_{\mathcal{P}_i} y \text{ for some } i \varepsilon \mathbb{N}.$$

Of course, $\mathcal{P}_i \subseteq \mathcal{P}$ for each i. Let $K_i \varepsilon \mathcal{P}_i \longrightarrow \mathcal{P}$ be the inclusion mapping, so that

$$\begin{tikzcd}
\mathcal{P}_i \arrow[rr, "J_i"] \arrow[dr, "K_i"'] & & \mathcal{P}_{i+1} \arrow[dl, "K_{i+1}"] \\
& \mathcal{P} &
\end{tikzcd}$$

commutes for all i. Then $<\mathcal{P}, \{K_i\}>$ is a direct limit (or colimit) of the diagram (*) above, i.e., for each category $\mathcal{C}$ and each $\mathbb{N}$-indexed collection $F_i \varepsilon \mathcal{P}_i \to \mathcal{C}$ of functors such that

$$\begin{tikzcd}
\mathcal{P}_i \arrow[rr, "J_i"] \arrow[dr, "F_i"'] & & \mathcal{P}_{i+1} \arrow[dl, "F_{i+1}"] \\
& \mathcal{C} &
\end{tikzcd}$$

commutes for all i, there exists a unique functor $F \in \mathcal{P} \rightarrow \mathcal{C}$ such that

$$\begin{tikzcd}
 & \mathcal{P} \arrow[dd, "F"] \\
\mathcal{P}_i \arrow[ur, "K_i"] \arrow[dr, "F_i"'] & \\
 & \mathcal{C}
\end{tikzcd}$$

commutes for all i. Note that if each $\mathcal{P}_i$ is a full subposet of $\mathcal{P}_{i+1}$, then each $\mathcal{P}_i$ is a full subposet of $\mathcal{P}$. Also, if

$$\mathcal{Q}_0 \subseteq \mathcal{Q}_1 \subseteq \mathcal{Q}_2 \subseteq \cdots$$

is another ascending chain of posets and

$$\mathcal{Q} = \cup \{\mathcal{Q}_i \mid i \in \mathbb{N}\}$$

then

$$\mathcal{P}_0 \underline{\times} \mathcal{Q}_0 \subseteq \mathcal{P}_1 \underline{\times} \mathcal{Q}_1 \subseteq \mathcal{P}_2 \underline{\times} \mathcal{Q}_2 \subseteq \cdots$$

is an ascending chain and

$$\mathcal{P} \underline{\times} \mathcal{Q} = \cup \{\mathcal{P}_i \underline{\times} \mathcal{Q}_i \mid i \in \mathbb{N}\}.$$

We have seen that the object $\mathcal{T}$ containing the types for a language having coercion among its features should be a poset. If the language is to have a general procedure-definition facility, then for any phrase types τ and θ there should be a type $\tau \Rightarrow \theta$ for procedures which accept arguments (i.e., parameters) of type τ and produce results (i.e., have calls) of type θ. How should $\Rightarrow$ interact with the partial order on $\mathcal{T}$? A procedure of type $\tau \Rightarrow \theta$ can accept arguments of type τ' if $\tau' \leq \tau$; the result of such a procedure can be used in a context calling for a result of type θ' if $\theta \leq \theta'$. Thus if $\tau' \leq \tau$ and $\theta \leq \theta'$, then we want $\tau \Rightarrow \theta \leq \tau' \Rightarrow \theta'$. So $\Rightarrow$ should be antimonotone in its first argument and monotone in its second, or in category-theorectic language $\Rightarrow$ is like a hom functor in that it is contravariant in its first argument and covariant in its second.

We have argued that Mng should be a functor:

$$\mathrm{Mng} \in T \rightarrow K,$$

where K is a category with appropriate structure. One thing that K should have is a functor contravariant in its first argument and covariant in its second; we can call this functor $\Rightarrow$, too. We want Mng to preserve $\Rightarrow$, i.e.,

$$\mathrm{Mng}\ (\tau \Rightarrow \theta) = \mathrm{Mng}\ \tau \Rightarrow \mathrm{Mng}\ \theta$$

for all phrase types τ and θ. To see how plausible this is, consider the case $K = \mathit{Set}$ with $\Rightarrow$ equal to the ordinary hom functor; then the set of meanings for phrases of type $\tau \Rightarrow \theta$ should be the set of functions from Mng τ to Mng θ.

In order to talk about type-checking we must be able to assign a nonsense type T_{ns} to parseable expressions which contain errors due to mismatched types. Since any expression can be used in a context for which nonsense suffices, the nonsense type should be the top, i.e., the terminal object, of T. What should be the nature of Mng T_{ns}? Knowing that a phrase has a nonsense meaning amounts to asserting that its meaning contains no information. Thus Mng T_{ns} should be a singleton set when $K = \mathit{Set}$, and it should be the analogue of a singleton set, i.e., a terminal object, for other possible K.

To make everything elegant, both T and K should be the same sort of entities, and Mng should be a homomorphism from T to K. This serves to motivate the following definitions.

An ordered triple

$$\mathcal{A} = <|\mathcal{A}|, \mathcal{A}_{ns}, \mathcal{A}_{\Rightarrow}>$$

is a <u>type algebra</u> if

(1) $|\mathcal{A}|$ is a category, called the <u>carrier</u> of $\mathcal{A}$,

(2) $\mathcal{A}_{ns}$ is a terminal object of $|\mathcal{A}|$, called the <u>nonsense</u> object of the algebra, and

(3) $\mathcal{A}_{\Rightarrow} \varepsilon |\mathcal{A}|^{op} \times |\mathcal{A}| \to |\mathcal{A}|$.

Alternative notations for $\mathcal{A}_{\Rightarrow} <a,b>$ are $a \underset{\mathcal{A}}{\Longrightarrow} b$ and, when $\mathcal{A}$ is readily determined from context, $a \Rightarrow b$. Cartesian closed categories give the most accessible examples of type algebras. Thus, from a Cartesian closed category $\mathcal{K}$ we obtain a type algebra $\mathcal{A}$ by letting $|\mathcal{A}| = \mathcal{K}$, $\mathcal{A}_{ns} = \mathcal{K}_{term}$, and $\mathcal{A}_{\Rightarrow} = \mathcal{K}_{\Rightarrow}$.

A <u>type algebra homomorphism</u> consists of a domain $\mathcal{A}$ and a codomain $\mathcal{B}$, both of which are type algebras, and a functor F (notation: $F \varepsilon \mathcal{A} \xrightarrow[\mathit{Type\ Alg}]{} \mathcal{B}$) such that

(1) $F \varepsilon |\mathcal{A}| \to |\mathcal{B}|$,

(2) $F\, \mathcal{A}_{ns} = \mathcal{B}_{ns}$, and

(3) the diagram

$$\begin{tikzcd}
|\mathcal{A}|^{op} \times |\mathcal{A}| \arrow[r, "\mathcal{A}_{\Rightarrow}"] \arrow[d, "F^{op} \times F"'] & |\mathcal{A}| \arrow[d, "F"] \\
|\mathcal{B}|^{op} \times |\mathcal{B}| \arrow[r, "\mathcal{B}_{\Rightarrow}"] & |\mathcal{B}|
\end{tikzcd}$$

commutes.

For the purpose of assigning phrase types to program fragments we need a type algebra whose carrier is a poset. Generally we expect to start with a poset $\mathcal{P}$ of primitive phrase types. The following theorem asserts that each poset $\mathcal{P}$ generates a free type algebra $\mathcal{T}$ whose carrier is a poset. It is the free type algebra generated by the poset of primitive types which provides the phrase types for program fragments.

A few comments on this theorem are in order. In the theorem $\mathcal{Q}$ is the poset of procedural phrase types. The theorem says that the set underlying the carrier of T is the disjoint union of the set of primitive phrase types, the singleton consisting of the nonsense type, and the procedural types. If τ is a procedural phrase type, then there are uniquely determined phrase types θ and θ' such that $\tau = \theta \Rightarrow \theta'$ because $T_{\Rightarrow}$ is injective on objects. Therefore, if we ignore the partial order, T is a free (universal) algebra generated by Ob P where the signature (i.e., ranked set of operators) consists of a single operator ns of rank 0 and a single operator $\Rightarrow$ of rank 2. In a different vein, notice how easy it will be to define the type algebra homomorphism

$$\mathrm{Mng} \in T \xrightarrow[\mathit{Type\ Alg}]{} K;$$

we need only give the functor which is the restriction of Mng to P.

The proof of the theorem is an exercise in the exploitation of the properties of unions of ascending chains of posets. We really do have to go through the construction given. Because of the mixed covariance and contravariance of $T_{\Rightarrow}$, we cannot get Theorem 2.2 by the general methods of constructing solutions to recursive domain equations given by Smyth and Plotkin in [10].

<u>Theorem 2.2</u>: Let P be a poset. There is a type algebra such that

(1) $|T|$ is a poset, $T \in |T|^{\mathrm{op}} \times |T| \to |T|$ is a full embedding, $|T| = P * T_{\mathrm{ns}} * \mathcal{Q}$, where $\mathcal{Q}$ is the image of $T_{\Rightarrow}$, and

(2) T satisfies the following universal mapping property: let K be a type algebra, let $f \in P \to |K|$, and let $incl \in P \to |T|$ be the inclusion functor; then there exists a unique type algebra homomorphism $F \in T \xrightarrow[\mathit{Type\ Alg}]{} K$ such that

$$\begin{tikzcd}
 & {|T|} \arrow[dd, "F"] \\
P \arrow[ur, "incl"] \arrow[dr, "f"'] & \\
 & {|K|}
\end{tikzcd}$$

commutes.

Proof: Let top be such that $\underline{top} \notin \text{Ob } P$, and let arrow be such that no element of $(\text{Ob } P) \cup \{\underline{top}\}$ is an ordered triple $\langle x, \underline{arrow}, y\rangle$ whose second component is arrow.

We start by constructing inductively a sequence

$$Q_0, T_0, Q_1, T_1, Q_2, T_2, \ldots$$

of posets such that, for all i, $P \cap Q_i = \phi$. Let

$$Q_0 = \phi.$$

Assuming that the sequence is constructed through T_i, let

$$T_i = P * \underline{top} * Q_i.$$

Assuming that the sequence is constructed through T_i, let

$$Q_{i+1} = T_i^{op} \underline{\times} \{\underline{arrow}\} \underline{\times} T_i.$$

By using Proposition 2.1 it is easy to see that

$$Q_0 \sqsubseteq Q_1 \sqsubseteq Q_2 \sqsubseteq \cdots$$

and

$$T_0 \sqsubseteq T_1 \sqsubseteq T_2 \sqsubseteq \cdots \quad .$$

Let $|T| = \cup \{T_i \mid i \in \mathbb{N}\}$ and $Q = \cup \{Q_i \mid i \in \mathbb{N}\}$. Clearly $Q = |T|^{op} \underline{\times} \{\underline{\text{arrow}}\} \underline{\times} |T|$. By the choice of arrow, $P \cap Q = \phi$, and, by the choice of top, $\underline{\text{top}} \notin (\text{Ob } P) \cup (\text{Ob } Q)$. It is now apparent that Ob $|T|$ is a disjoint union:

$$\text{Ob } |T| = (\text{Ob } P) \cup \{t\} \cup (\text{Ob } Q).$$

Also,

$$\begin{aligned} \tau \leq_{|T|} \theta \quad & \text{iff} \quad \tau \leq_{T_i} \theta \quad \text{for some i} \\ & \text{iff} \quad \tau \leq_P \theta \text{ or } t = \theta \text{ or } \tau \leq_{Q_i} \theta \quad \text{for some i} \\ & \text{iff} \quad \tau \leq_P \theta \text{ or } t = \theta \text{ or } \tau \leq_Q \theta \end{aligned}$$

Therefore, $|T| = P * \underline{\text{top}} * Q$.

Next we will define $T_{\Rightarrow} \in |T|^{op} \times |T| \to |T|$; we want to arrange matters so that, for $\tau, \theta \in \text{Ob } |T|$,

$$T_{\Rightarrow} \langle \tau, \theta \rangle = \langle \tau, \underline{\text{arrow}}, \theta \rangle.$$

This is very easy. Let

$$H \in |T|^{op} \underline{\times} \{\underline{\text{arrow}}\} \underline{\times} |T| \to |T|^{op} \times |T|$$

be the canonical isomorphism. Then let $T_{\Rightarrow}$ be the unique functor such that

$$\begin{tikzcd} |T|^{op} \times |T| \arrow[r, "T_{\Rightarrow}"] & |T| \\ |T|^{op} \underline{\times} \{\underline{\text{arrow}}\} \underline{\times} |T| \arrow[u, "H"] \arrow[r, equal] & Q \arrow[u, "\text{incl}"'] \end{tikzcd}$$

commutes. Clearly $T_{\Rightarrow}$ is a full embedding whose image is Q.

Since top is a terminal object of $|T|$, we may let $T_{ns} = \underline{\text{top}}$. Therefore $|T| = P * T_{ns} * Q$. This proves the first part of the theorem.

For the second part, let $\mathcal{K}$ be a type algebra, and let $f \in \mathcal{P} \to |\mathcal{K}|$. We will define a sequence of functors $G_0, F_0, G_1, F_1, G_2, F_2, \ldots$ such that

$$G_i \in \mathcal{Q}_i \to |\mathcal{K}| \text{ and } F_i = \mathcal{T}_i \to |\mathcal{K}|.$$

Since $\mathcal{Q}_0 = \phi$, the functor G_0 is uniquely determined. Suppose we have constructed all functors through G_i. Recall $\mathcal{T}_i = \mathcal{P} * \underline{\text{top}} * \mathcal{Q}_i$, and let F_i be the unique functor satisfying

(1) $F_i \; \underline{\text{top}} = \mathcal{K}_{\text{ns}}$,

(2) the restriction of F_i to $\mathcal{P}$ is f, and

(3) the restriction of F_i to $\mathcal{Q}_i$ is G_i.

Suppose we have constructed all functors through G_i. Our intention is that, for $\tau, \theta \in \text{Ob}\; \mathcal{T}_i$,

$$G_{i+1} \langle \tau, \underline{\text{arrow}}, \theta \rangle = \mathcal{K}_{\Rightarrow} \langle F_i^{\text{op}}\, \tau, F_i\, \theta \rangle.$$

Thus, let G_{i+1} be such that

$$\begin{tikzcd}
\mathcal{Q}_{i+1} \arrow[r, "G_{i+1}"] \arrow[d, "H_i"'] & {|\mathcal{K}|} \\
\mathcal{T}_i^{\text{op}} \times \mathcal{T}_i \arrow[r, "F_i^{\text{op}} \times F_i"] & {|\mathcal{K}|^{\text{op}} \times |\mathcal{K}|} \arrow[u, "\mathcal{K}_{\Rightarrow}"']
\end{tikzcd}$$

commutes, where $H_i \in \mathcal{T}_i^{\text{op}} \underline{\times} \{\underline{\text{arrow}}\} \underline{\times} \mathcal{T}_i \to \mathcal{T}_i^{\text{op}} \times \mathcal{T}_i$ is the canonical isomorphism.

Next we claim that the diagrams in the sequence

$$\begin{tikzcd}[column sep=small]
\mathcal{Q} \arrow[rr, "\text{incl}"] \arrow[dr, "G_0"'] & & \mathcal{Q}_0 \arrow[dl, "G_1"] & , & \mathcal{T}_0 \arrow[rr, "\text{incl}"] \arrow[dr, "F_0"'] & & \mathcal{T}_1 \arrow[dl, "F_1"] & , & \mathcal{Q}_1 \arrow[rr, "\text{incl}"] \arrow[dr, "G_1"'] & & \mathcal{Q}_2 \arrow[dl, "G_2"] & , & \mathcal{T}_1 \arrow[rr, "\text{incl}"] \arrow[dr, "F_1"'] & & \mathcal{T}_2 \arrow[dl, "F_2"] & , \ldots \\
& {|\mathcal{K}|} & & & & {|\mathcal{K}|} & & & & {|\mathcal{K}|} & & & & {|\mathcal{K}|} & &
\end{tikzcd}$$

are all commutative. The first diagram commutes because $\mathcal{Q}_0 = \phi$. Suppose all diagrams commute through

$$\begin{tikzcd}
\mathcal{Q}_{i-1} \arrow[rr, "\text{incl}"] \arrow[dr, "G_{i-1}"'] & & \mathcal{Q}_i \arrow[dl, "G_i"] \\
& |\mathcal{K}| &
\end{tikzcd} ;$$

then the commutativity of

$$(**) \qquad \begin{tikzcd}
\mathcal{T}_{i-1} \arrow[rr, "\text{incl}"] \arrow[dr, "F_{i-1}"'] & & \mathcal{T}_i \arrow[dl, "F_i"] \\
& |\mathcal{K}| &
\end{tikzcd}$$

is immediate from the definitions of F_{i-1} and F_i. Suppose all the diagrams through (**) commute; then from the commutativity of all the inner diagrams in

$$\begin{tikzcd}
\mathcal{Q}_i \arrow[rrrr, "\text{incl}"] \arrow[dr, "H_{i-1}"] \arrow[dddrr, "G_i"'] & & & & \mathcal{Q}_{i+1} \arrow[dl, "H_i"'] \arrow[dddll, "G_{i+1}"] \\
& \mathcal{T}^{op}_{i-1} \times \mathcal{T}_{i-1} \arrow[rr, "\text{incl}"] \arrow[dr, "F^{op}_{i-1} \times F_{i-1}"] & & \mathcal{T}^{op}_i \times \mathcal{T}_i \arrow[dl, "F^{op}_i \times F_i"'] & \\
& & |\mathcal{K}|^{op} \times |\mathcal{K}| \arrow[d, "\mathcal{K}_{\Rightarrow}"] & & \\
& & |\mathcal{K}| & &
\end{tikzcd}$$

we get the commutativity of the outer triangle.

By applying the earlier remarks on direct limits of ascending chains of posets, we see there exist unique functors G and F such that for all i the diagrams

$$\begin{tikzcd}
 & \mathcal{Q} \arrow[d, "G"] \\
\mathcal{Q}_i \arrow[ur, "\text{incl}"] \arrow[r, "G_i"'] & {|\mathcal{K}|}
\end{tikzcd} \qquad \begin{tikzcd}
 & \mathcal{T} \arrow[d, "F"] \\
\mathcal{T}_i \arrow[ur, "\text{incl}"] \arrow[r, "F_i"'] & {|\mathcal{K}|}
\end{tikzcd}$$

commute. The functors G and F are related by two commutative diagrams. To obtain the first diagram, note that for each i>0 the diagram

$$\begin{tikzcd}
 & & \mathcal{Q} \arrow[d, "H"] \\
 & & {|\mathcal{T}|^{op} \times |\mathcal{T}|} \arrow[dd, "F^{op} \times F"] \\
\mathcal{Q}_i \arrow[uurr, "\text{incl}"] \arrow[r, "H_{i-1}"] \arrow[ddrr, "G_i"'] & \mathcal{T}^{op}_{i-1} \times \mathcal{T}_{i-1} \arrow[ur, "\text{incl}"] \arrow[dr, "F^{op}_{i-1} \times F_{i-1}"'] & \\
 & & {|\mathcal{K}|^{op} \times |\mathcal{K}|} \arrow[d, "\mathcal{K}_{\Rightarrow}"] \\
 & & {|\mathcal{K}|}
\end{tikzcd}$$

commutes, and

$$\begin{tikzcd}
 & \mathcal{Q} \arrow[d, "K_{\Rightarrow} \circ (F^{op} \times F) \circ H"] \\
\phi = \mathcal{Q}_0 \arrow[ur, "incl"] \arrow[r, "G_{\emptyset}"'] & {|\mathcal{K}|}
\end{tikzcd}$$

is trivially commutative. Therefore the composite vertical arrow must equal G, i.e.,

$$\begin{tikzcd}
{|\mathcal{T}|^{op} \times |\mathcal{T}|} \arrow[d, "F^{op} \times F"'] & \mathcal{Q} \arrow[l, "H"'] \arrow[d, "G"] \\
{|\mathcal{K}|^{op} \times |\mathcal{K}|} \arrow[r, "K_{\Rightarrow}"] & {|\mathcal{K}|}
\end{tikzcd}$$

commutes. Also, for each i the diagram

$$\begin{tikzcd}
 & & \mathcal{Q} \arrow[d, "incl"] \\
\mathcal{Q}_i \arrow[urr, "incl"] \arrow[r, "incl"] \arrow[drr, "G_i"'] & \mathcal{T}_i \arrow[r, "incl"] \arrow[dr, "F_i"] & {|\mathcal{T}|} \arrow[d, "F"] \\
 & & {|\mathcal{K}|}
\end{tikzcd}$$

commutes. Again, the composite vertical arrow must equal G, and we get a commutative diagram

$$\begin{tikzcd}
\mathcal{Q} \arrow[r, "incl"] \arrow[dr, "G"'] & {|\mathcal{T}|} \arrow[d, "F"] \\
 & {|\mathcal{K}|}
\end{tikzcd}$$

To conclude that F is a type algebra homomorphism, there are three conditions that must be checked. First, from the definition of F it is clear that $F \varepsilon |\mathcal{T}| \to |\mathcal{K}|$. Second, since F extends $F_0 \varepsilon \mathcal{T}_0 \to |\mathcal{K}|$ we easily compute

$$F\ \mathcal{T}_{ns} = F_0\ \underline{top} = \mathcal{K}_{ns}.$$

Finally, we must verify that in the diagram

$$\begin{tikzcd}
|\mathcal{T}|^{op} \times |\mathcal{T}| \arrow[rr, "\mathcal{T}_{\Rightarrow}"] \arrow[dd, "F^{op} \times F"'] \arrow[dr, "H^{-1}"'] & & |\mathcal{T}| \arrow[dd, "F"] \\
& \mathcal{Q} \arrow[ur, "incl"] \arrow[dr, "G"'] & \\
|\mathcal{K}|^{op} \times |\mathcal{K}| \arrow[rr, "\mathcal{K}_{\Rightarrow}"] & & |\mathcal{K}|
\end{tikzcd}$$

the outer square is commutative; this follows because all the inner diagrams commute. Thus

$$F\ \varepsilon\ \mathcal{T} \xrightarrow[\mathit{Type\ Alg}]{} \mathcal{K}.$$

Notice that

$$\begin{tikzcd}
& |\mathcal{T}| \arrow[dd, "F"] \\
\mathcal{P} \arrow[ur, "incl"] \arrow[dr, "f"'] & \\
& |\mathcal{K}|
\end{tikzcd}$$

commutes because F extends F_0 which in turn extends f.

Finally, we must verify the uniqueness of F. Let

$$F'\ \varepsilon\ \mathcal{T} \xrightarrow[\mathit{Type\ Alg}]{} \mathcal{K}$$

be an arbitrary type algebra homomorphism whose restriction to $\mathcal{P}$ is f, and let

$$G'_0,\ F'_0,\ G'_1,\ F'_1,\ G'_2,\ F'_2,\ \ldots$$

be the restriction of F' to

$$\mathcal{Q}_0,\ \mathcal{T}_0,\ \mathcal{Q}_1,\ \mathcal{T}_1,\ \mathcal{Q}_2,\ \mathcal{T}_2,\ \ldots,$$

respectively. We will show by induction that

$$G'_0 = G_0,\ F'_0 = F_0,\ G'_1 = G_1,\ F'_1 = F_1,\ G'_2 = G_2,\ \ldots$$

Since $\mathcal{Q}_0 = \phi$, $G'_0 = G_0$. Suppose all the equalities through $G'_i = G_i$ are true. We make three observations.

(1) $F'_i\ \mathcal{T}_{ns} = F'\ \mathcal{T}_{ns} = \mathcal{K}_{ns}$ since F' is a homomorphism.

(2) The restriction of F'_i to $\mathcal{P}$ is the restriction of F' to $\mathcal{T}$, which is f.

(3) The restriction of F'_i to $\mathcal{Q}_i$ is the restriction of F' to $\mathcal{Q}_i$, which is $G'_i = G_i$.

By uniqueness of the construction of F_i, we get $F'_i = F_i$. Now suppose all the equalities through $F'_i = F_i$ are true. Observe that in the following diagram all inner diagrams commute.

$$\begin{tikzcd}
\mathcal{Q}_{i+1} \arrow[rrrrrr, "G'_{i+1}"] \arrow[rrd, "\mathrm{incl}"] \arrow[dddd, "H_i"'] & & & & & & {|\mathcal{K}|} \arrow[dddd, "\mathcal{K}"] \\
& & \mathcal{Q} \arrow[rr, "\overset{(2)}{\mathrm{incl}}", "(5)"'] \arrow[rd, "H"'] & & {|\mathcal{T}|} \arrow[rru, "F'"] & & \\
& (1) & & {|\mathcal{T}|^{op} \times |\mathcal{T}|} \arrow[ru, "\mathcal{T}_{\Rightarrow}"'] \arrow[rrrdd, "F'^{op} \times F'"] & & (3) & \\
& & & (4) & & & \\
\mathcal{T}_i^{op} \times \mathcal{T}_i \arrow[rrruu, "\mathrm{incl}"] \arrow[rrrrrr, "F'^{op}_i \times F'_i = F^{op}_i \times F_i"'] & & & & & & {|\mathcal{K}|^{op} \times |\mathcal{K}|}
\end{tikzcd}$$

The reasons for commutativity of various diagrams is as follows:

(1) definition of H and H_i,

(2) G'_{i+1} is the restriction of F' to $\mathcal{Q}_{i+1}$,

(3) F' is a type algebra homomorphism,

(4) F'_i is the restriction of F' to $\mathcal{T}_i$, and

(5) definition of $\mathcal{T}_{\Rightarrow}$.

Thus the outer square commutes. By the definition of G_{i+1}, we see that $G'_{i+1} = G_{i+1}$. Therefore all the equalities in the sequence are valid. Since $|\mathcal{T}| = \cup \{\mathcal{T}_i \mid i \in \mathbb{N}\}$ and the restriction of F' to $\mathcal{T}_i$ in all cases equals the restriction of F to $\mathcal{T}_i$, we conclude F' = F. □

References

[1] ADJ: J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, J. ACM 24 (1) (1977) 68-95.

[2] M.A. Arbib and E.G. Manes, *Arrows, Structures, and Functors - The Categorial Imperative* (Academic Press, New York, 1975).

[3] H. Herrlich and G.E. Strecker, *Category Theory* (Allyn and Bacon, Boston, 1973).

[4] S. MacLane, *Categories for the Working Mathematician*, (Springer-Verlag, Berlin, 1971).

[5] F.J. Oles, *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph.D. Dissertation, Syracuse University, 1982.

[6] J.C. Reynolds, The essence of ALGOL, in: J.W. de Bakker and J.C. van Vliet (Eds.), *Algorithmic Languages* (North-Holland, Amsterdam, 1981) 345-372.

[7] J.C. Reynolds, Semantics of the domain of flow diagrams, J. ACM 24 (3) (1977) 484-503.

[8] J.C. Reynolds, Using category theory to design implicit conversions and generic operators, in: N.D. Jones (Ed.), *Semantics-Directed Compiler Generation*, LNCS 94 (Springer-Verlag, Berlin, 1980) 211-258.

[9] D.S. Scott, The lattice of flow diagrams, in: E. Engeler (Ed.), *Symposium on Semantics of Algorithmic Languages*, LNCS 188, (Springer-Verlag, Berlin, 1971) 311-372.

[10] M.B. Smyth and G.D. Plotkin, The category-theoretic solution of recursive domain equations, CSR-102-82, Dept. of Computer Science, University of Edinburgh (Feb., 1982).