Building the Access Pointers to a Computation Environment

V.E. Wolfengagen
Building the Access Pointers to a Computation Environment

V.E. Wolfengagen
Vorotnikovskiy per., 7, bld. 4
JurInfoR-MSU Institute for Contemporary Education
Moscow, 103006, Russia
vew@jmsuice.msk.ru

Dedicated to Professor Lev Kuzin

Abstract
A common object technique equipped with the categorical and computational styles is briefly outlined. An object is evaluated by embedding in a host computational environment which is the domain-ranged structure. An embedded object is accessed by the pointers generated within the host system. To assist with an easy extract the result of the evaluation a pre-embedded object is generated. It is observed as the decomposition into substitutional part and access function part which are generated during the object evaluation.

1 Introduction
Recent issues in a data modeling area tend to attract some general algebraic ideas. The most influential to the target data model are the properties of the database domains and their interconnections.

Some useful observations concerning the mappings between the relational database domains [IP94] result in the solutions to integrate a database scheme. But the difficulties were observed when the type considerations occurs: the first-order data model becomes overloaded with the complicated and intuitively unreasonable mappings, especially when attempts to use a category theory are done.

The attempts to apply the same ideas for a conceptual modeling [LH96] are not yet advanced to cover the known effects and models. A gap between the pure reasoning with the objects in a category-style manner (the maps and domains have the similar status) and the realistic data models is indicated every time (see, e.g., [HLF96]) when the researcher put the database concepts together. Nevertheless, the feeling of a category theory usefulness is growing with the rate of accumulating the practical experience in a field [Jac91].

The semantics of database is heavy based on the evaluation of the expressions [BSW94]. The success of the approach is also estimated by the simplicity and intuitive transparency whenever the maps between domains are involved. Information system engineering [JD93] extremely needs to apply the theoretically balanced data models with the higher-order structures.

The observations show the concepts and notions shared by the distinct approaches and the theories. The importance of extracting all the useful feature from the notions of function and type are well understood. Here we will try to rearrange and put together some important ideas concerning the evaluation of expressions. The most of the attention is paid to environment of the evaluation to suit it with the common database models. An environment is assumed to consist of the products of the domains. Thus, the obvious way to get an access to its partitions is to evaluate the projections, each of them being typed according to the positions of the counterparts. This intuitively means, that the pointers to an environment are to be generated. Some unexpected features arise whenever the encapsulation of objects is used relatively the environment prescribed.

To cover the notion in use the most of attention is paid to interrelations and correspondences between types, functions and environment of evaluation. The language is left out of this paper scope, and model theoretic aspects are attracted. The style of reasoning in a category is used along with the equational solutions.
Section 2 covers the minimal amount of a type theory to put the necessary accents. The projections are used almost in a traditional sense. The operation \(^/^\) gives a kind of suit to shift around the variables. In addition, the correspondence between the projections and the identity maps brings in a theory the intuitive ground.

An access to values coupled in environment is discussed in section 3. The main topics are generation of the access pointers and the encapsulation of the objects. The commutative diagram techniques is applied to establish the most important equation. The reasons and solutions are based on the possibilities of the citation. An atomic case is covered by Lemma 3.1. Its generalization leads to the Theorem 3.1 whenever the function constant is applied to an argument. To pass an actual parameter the closure is generated.

## 2 A theory of types

A variety of possible theories of types has been developed with different purposes and with distinct mathematical or logical ideas in use. A category theory gives one of the theories.

To establish a universe of discourse for types we need some kind of primitive frame. Usually we start with a set of generic types and generate the derived types applying some building principles. In a pure category theory we are not given neither building principles nor clear understanding of making new types from old. Entering a category theory we observe the relations between types which hold whenever corresponding mapping statement

\[
f : \text{Env} \rightarrow D_y
\]  

obtains (here: \text{Env} and \(D_y\) are domains).

The way of reading the mapping statement depends on intuitive reasons. Whenever we apply to category theory, a mapping statement is supposed being taken as a statement with one-place functions and operation \(^/^\) of composition with one-place functions. Thus, practical reasons concern the multi-place functions to increase their arity.

The solutions to bring composition with multi-place functions (see, e.g., [Sza78]) are known but give no real suit.

The easier way is to assume that the category has cartesian products and to select the particular representatives of the product domains. In particular, the cartesian power \(D^n\) for every \(n \geq 0\) gives \(n\)-ary functions as maps

\[
f : D^n \rightarrow D_y.
\]  

### 2.1 A description of products

#### 2.1.1 Empty product.

To make a description of products we bring in the product and start with the assumption that a category hase a special domain \(\emptyset\) as the empty product:

\[
D^0_y = D^0 = \emptyset,
\]  

and for every domain \(D\) a special map:

\[
0_D : D \rightarrow \emptyset
\]  

From an intuitive reason the domain \(\emptyset\) has one element, and map \(0_D\) is unique, i.e. whenever \(f : D \rightarrow \emptyset\) then \(f = 0_D\).
2.1.2 A theory of tuples and multi-ary maps.

This kind of a theory is based on the products. Concerning binary products we have for arbitrary two domains $D_x$ and $D_y$ a special choice of a domain $D_x \times D_y$, and, more generally,

$$D^{n+1} = D^n \times D, \ n > 0.$$ 

A product is equipped with the special maps

$$Fst : D_y \times D_x \rightarrow D_y,$$
$$Snd : D_y \times D_x \rightarrow D_x,$$

which are the projections. As usually, mere existence of maps $Fst$ and $Snd$ does not characterize $D_y \times D_x$ as a product. In addition we assume that there is a chosen pairing operation $< f, g >$ on maps such that types are assigned by the rule:

$$f : Env \rightarrow D_y, \ g : Env \rightarrow D_x$$

$$< f, g > : Env \rightarrow D_y \times D_x$$

The additional property of $Fst$, $Snd$ and $< ., >$ under composition is assumed:

$$Fst \circ < f, g > = f,$$
$$Snd \circ < f, g > = g,$$
$$< Fst \circ h, Snd \circ h > = h,$$

where $f$, $g$ are typed as above, and

$$h : Env \rightarrow D_y \times D_x$$

One-to-one correspondence. It means that there is a one-one correspondence between the pairs of maps $f$, $g$ and the map $h$ into the product.

2.1.3 A theory of functions.

A category usually gives a ‘local’ universe of selected functions. In case of arbitrary functions we need the functional spaces as explicate domains in the category.

Given $D_x$ and $D'_y$ we want to form $(D_x \rightarrow D'_y)$ as a domain in its own right. After adopting the above, the functional space does contain the various maps.

Whenever we have an element $f$ from $(D_x \rightarrow D'_y)$ and the element $x$ from $D_x$ we need to establish the map that will apply element $f$ to element $x$ giving rise to the value of function $f$:

$$\varepsilon : [f, x] \mapsto f(x)$$

This evaluation map $\varepsilon$ is typed as

$$\varepsilon : (D_x \rightarrow D'_y) \times D_x \rightarrow D'_y$$

In addition there has to be a map for shifting around variables. Suppose

$$g : Env \times D_x \rightarrow D'_y$$

is a map with two arguments. In an evaluation

$$g([i, x])$$

we can think of holding $i$ constant and regarding $g([i, x])$ as a function of $x$. We need a name for this function and for correspondence with possible values of $x$:

$$\hat{g} : Env \rightarrow (D_x \rightarrow D'_y)$$
so that the function we are thinking of - given \( x \) - was
\[
\hat{g}(i)(x)
\]

Map \( k \) is one-to-one corresponded to \( g \). All this function value notation is not categorical notation. Nevertheless we are to say that there is a one-one correspondence via \( \hat{\cdot} \) between maps \( g: Env \times D_x \to D'_y \) and maps \( k: Env \to (D_x \to D'_y) \).

This correspondence comes down to the following two equations:
\[
\begin{align*}
\varepsilon \circ (\hat{g} \times id_{D_x}) &= g, \\
\varepsilon \circ (k \times id_{D_x}) &= k,
\end{align*}
\]

where \( (\cdot \times \cdot) \) means a functor product, or, in the neutral to domains form,
\[
\begin{align*}
\varepsilon \circ <\hat{g} \circ Fst, Snrd > &= g, \\
\varepsilon \circ <k \circ Fst, Snrd > &= k,
\end{align*}
\]

where \( <\cdot \circ Fst, Snrd > \) is the same as \( (\cdot \times id_{D_x}) \).

The notation is now wholly categorical and not so suitable. The more sense is added by the language of functors.

### 2.1.4 A system of types within cartesian closed category

Now we give a brief sketch of viewing the cartesian closed category (c.c.c.) as a system of types.

Theory of functions. Each c.c.c represents a theory of functions.

Maps. The maps in the category are certain special functions that are used to express the relations between the types (the domains of the category).

Products. In order to be able to deal with multi-ary functions, we assume we can form and analyze products.

Function spaces. In order to be able to work with transformations of arbitrary functions (arbitrary within the theory) we assume we can form function spaces. Note, that the higher types enter the theory, e.g., as the sequence of domains:
\[
D, (D \to D), ((D \to D) \to D), \ldots
\]

Operations \( \varepsilon \) and \( \hat{\cdot} \). To be able really to view these domains as function spaces, certain operations, \( \varepsilon \) and \( \hat{\cdot} \), with characteristic equations have to be laid down.

Cartesian closed category. C.c.c is a theory of functions, and the higher type functions are included. Hence, the theory of c.c.c.'s is the theory of types. It is only one such theory.

'Bigger' theories. 'Bigger' theories could be obtained by demanding more types, e.g., by axiomatizing coproducts (disjoint sums) and \( D_x + D_y \)

Type \([\cdot]\). We could throw in type \([\cdot]\) of propositions so that higher types like \((D^n \to [\cdot])\) correspond to \( n \)-ary predicates.
3 Environment and an access to values

To build the typed language we need to think of the values of the variables. The values of the variables are available via access functions from an environment \( Env \). The representation of an environment is given by the domains \( D_y, D_x, \ldots \) which are ranges of possible values of \( y, x, \ldots \). The domains \( D_y, D_x \) give the explicit part of an environment \( Env \), and its implicit rest \( E \), not detailed for current consideration, is separated from \( D_y, D_x \):

\[
Env = (E \times D_y) \times D_x
\]

3.1 Updating an environment

Whenever we want to update \( Env \) the restriction is imposed to its counterparts:

\[
Env = \text{Implicit part} \times \text{Explicit part},
\]

e.g.,

\[
Env = E \times D_x
\]

with ‘\( E \)’ for implicit part and ‘\( D_x \)’ for explicit part.

In fact, within \( Env \) we have an ‘old’ value of \( x \) which ranges \( D_x,\text{Old} \), and \( E \) which does not depend on \( x \). The description of updating an environment \( Env \) has to include both its explicit part \( D_x \) and implicit rest \( E \). An outline of updating process is given below.

Step 1: Building the old environment. To combine \( Env,\text{Old} \) we construe the product

\[
Env,\text{Old} = E \times D_x,\text{Old}.
\]

Step 2: Bringing in a range of values. The product of \( Env,\text{Old} \) and \( D_x \) is generated as

\[
Env,\text{Old} \times D_x = (E \times D_x,\text{Old}) \times D_x.
\]

Step 3: Establishing an Update-function. The An Update\(_x\) function is established to enable the transformation from \( Env,\text{Old} \) to \( Env,\text{New} \):

\[
\text{Update}_x : (E \times D_x,\text{Old}) \times D_x \rightarrow E \times D_x,\text{New},
\]

or,

\[
\text{Update}_x : Env,\text{Old} \times D_x \rightarrow Env,\text{New}.
\]

At this stage we are to compare the properties of the domains \( D_x, D_x,\text{Old}, D_x,\text{New} \):

1. \( D_x \) is an unrestricted range for free variable \( x \);
2. \( D_x,\text{Old} \) is some existing (i.e., before evaluation) restriction of \( D_x \);
3. \( D_x,\text{New} \) differs from \( D_x,\text{Old} \) exclusively in a point \( x \).

All of this could be implemented in a particular kind of Update\(_x\), which would be referred as substitution, or Subst\(_x\):

\[
\text{Subst}_x : Env,\text{Old} \times D_x \rightarrow Env,\text{New}.
\]

The description of its behaviour by the elements gives the following:

\[
\text{Subst}_x : [i, d] \mapsto i(d/x),
\]

where \( d \in D_x, i \in Env,\text{Old}, \) and \( i(d/x) \in Env,\text{New} \). [Here: the new instance \( i(d/x) \) of environment is the same as its old instance \( i \) excepting the point \( x \), which is replaced by \( d \).

One could imagine that there is a pointer from ‘\( x \)’ to its possible values ‘\( d \)’. 

Advances in Databases and Information Systems, 1997
3.2 Viewing $\text{Subst}_x$ as a pointer

Now we discuss the possibility to construe a pointer to the partitions of an environment. At first, we would try the equation

$$\text{Subst}_x = F_{st} \times \text{id}_{D_x}$$

where $F_{st} \times \text{id}_{D_x}$ is a functor product, its linear notation, and

$$\text{Subst}_x : \text{Env}_{old} \times D_x \rightarrow \text{Env}_{new}$$

for $\text{Env} = (E \times D_y) \times D_x$.

The functor product when being applied to ordered pair generates an access separately to the first and to the second its members. This feature makes it possible to bring in the following maps as the pointers to the partitions of the environment.

(Pointer to the part independent on '$x$'. This is a composition of $F_{st}$'s which ranges the product $E \times D_y$, i.e. and implicite - and independent, - part of the environment:

$$F_{st} \circ F_{st} : \text{Env}_{old} \times D_x \rightarrow E \times D_y$$

(Pointer to the part of new values for '$x$'. This is a second projection $\text{Snd}$ which ranges over the desired domain $D_x$:

$$\text{Snd} : \text{Env}_{old} \times D_x \rightarrow D_{x,new}$$

Coupling the new environment. Now we generate an access to the new environment. Taking into account the pointers for both the partitions, we need to construe their couple to obtain the pointer to the new environment:

$$< F_{st} \circ F_{st}, \text{Snd }>: \text{Env}_{old} \times D_x \rightarrow \text{Env}_{new}$$

Getting started with a new environment

$$\text{Env}_{new} = (E \times D_y) \times D_{x,new}$$

we can evaluate the arbitrary fuctions. The process of extracting the pointers to $D_y$, $D_{x,new}$ and generating the values from $D_y'$ whenever $D_y = (D_y')^{D_x}(= D_y' \rightarrow D_x)$, i.e. for the fuction space $D_y$, comes down to the following steps.

Step 1: Access to $D_y$. We take the first partition ($F_{st}$) of $\text{Env}_{new}$ and after that construe the pointer to its second ($\text{Snd}$) partition:

$$\text{Snd} \circ F_{st} : \text{Env}_{new} \rightarrow D_y$$

Step 2: Access to $D_{x,new}$. An effect of applying $\text{Snd}$ to $\text{Env}_{new}$ gives the pointer

$$\text{Snd} : \text{Env}_{new} \rightarrow D_{x,new}$$

Step 3: Coupling an access to $D_y'$ for $D_y = (D_y')^{D_x}$ by $\varepsilon$. We take the subpartions of $\text{Env}_{new}$ as above and restore the pointer:

$$< \text{Snd} \circ F_{st}, \text{Snd }>: \text{Env}_{new} \rightarrow ((D_y')^{D_x} \times D_{x,new})$$

taking in mind that $\varepsilon : (D_y')^{D_x} \times D_{x,new} \rightarrow D_y'$.

Now we are able to take a function $f$ from $(D_y')^{D_x}$ and the argument $d$ from $D_{x,new}$ and apply $f$ to $d$ using $\varepsilon$. Thus, the equation $\varepsilon[f,d] = f(d)$ is valid giving rise to the values $f(d)$ from $D_y'$
3.3 Encapsulation of an object

In particular, an evaluation process may result in capturing the object being evaluated by an environment.

Lemma 3.1 (Citation) For any given environment $Env = (E \times D_y) \times D_x$ and the domain $D_y = (D_y)^{D_x}$ the constant $c \in D_x$ and the function constant $f \in D_y$ are described by the maps $\{1_{\{c\}} \circ Snd\}$ and $\{f \circ Snd\}$ respectively.

Proof. For any given instance $i \in Env$, e.g., $i = [e, y], x$ whenever $x, d \in D_x, y, f \in D_y$ then:

1. $(1_{\{c\}} \circ Snd)i c = (1_{\{c\}} \circ Snd)[i, c] = 1_{\{c\}}c = c$.
2. $(f \circ Snd)i d = (f \circ Snd)[i, d] = f(d)$.

Thus, this proof is straightforward and elementary.

Canonical evaluation of a constant is according the commutative diagram in Figure 1. The reasons are as follows. Let $i$ be an instance of environment $Env$, thus, $i \in Env$. Each occurrence of ‘a’ canonically is replaced by the same ‘a’, i.e. $i_{\{a\}}$ means the instance of environment which captured the constant $a$, also means the substitution of domain for $a$ by $a$. We need a closure to trigger the evaluation process, and $1_{\{a\}}$ (identity map as a canonical evaluation) for the evaluated constant is generated. Roughly speaking, this identity map evaluates a constant and whenever a closure is not the identity map then the constant is not canonically evaluated.

Now we describe the evolution of an environment when encapsulation of the constant occurs. The environment in Figure 2 is treated as the cartesian product of the range domains. The notations naturally reflects the ranges, and $D_a$ is a range of $a$-compatible objects, i.e. those with the same type. For simplicity we assume $D_a = \{a\}$, and this singleton $\{a\}$ is the encapsulated constant. The map $Encapsulate_a$ builds a renewed environment by setting up the product of implicit partition of the environment with the singleton $\{a\}$.

3.4 Building a pointer to values

3.4.1 Evaluation of a variable

For single free variable the element-wise reasons for the evaluation are described by the commutative diagram in Figure 3. To read this diagram we use the additional notations: $d \in D_x$ for an element being substituted; $1_{D_x} : D_x \rightarrow D_x$ for an identity map.
We try to ‘solve’ this diagram relatively \( g \) and \( \text{Subst}_x \).

**Solution for \( g \).** For every \( i \in \text{Env} \) the maps

\[
\begin{align*}
\hat{g}(i) : & \quad D_x \to D_x; \\
g : & \quad \text{Env} \to (D_x \to D_x); \\
g : & \quad \text{Env} \times D_x \to D_x;
\end{align*}
\]

are valid, hence the following is a ‘solution’:

\[
g = \text{Snd}
\]

The value of a free variable is represented by an identity map. Note that this diagram corresponds to some idea of **closure**: free variable is supposed to be closed under the environment of its evaluation.

**Solution for \( \text{Subst}_x \).** For every \( i = [e, x] \) from ‘old’ environment the map \( \text{Subst}_x \) gives \([e, d]\) as an instance of ‘new’ environment:

\[
\text{Subst}_x : [e, x], d \mapsto [e, d],
\]

hence,

\[
\text{Subst}_x = < \text{Fst} \circ \text{Fst}, \text{Snd} >
\]

Therefore, the ‘solution’ of diagram in Figure 3 for \( g \) and \( \text{Subst}_x \) in case we evaluate a single free variable is given by diagrams in Figure 4 and in Figure 5.

### 3.4.2 Evaluation of a constant function

Evaluation of a **constant function** gives the most typical sample to encapsulate the object of general nature. To observe the effects we describe an applying of the constant function to the argument. All the counterparts - both function and argument, - from the category theory view are the **objects**.
The environment is changed whenever the application of the function $f$ to the argument $x$ occurs, i.e. the triggering event is $(fx)$, or similarly, $f(x)$. In the environment an evaluation is triggered whenever the value of argument `$d$' is passed to argument `$x$'.

The following Theorem 3.1 reflects the computational ideas in use.

**Theorem 3.1 (Citation of the function)** (1) The equation
\[
  f \circ (\varepsilon \circ \langle \hat{g}_d \times \hat{id}_{D_x} \rangle) = \varepsilon \circ \langle \hat{g}_{f} \times \hat{id}_{D_x} \rangle
\]
describes the object $f$ as a functional constant parameterized by $g_d$ and $g_f$.

(2) The equation in (1) has the solution
\[
  g_d = 1_{D_x} \circ \text{Snd}, \\
  g_f = f \circ \text{Snd},
\]

thus, the pointers to an environment are generated.

**Proof.** (1) The equation above is commented as follows:

Left part:
\[
  f \circ (\varepsilon \circ \langle \hat{g}_d \times \hat{id}_{D_x} \rangle)
\]
right part:
\[
  \varepsilon \circ \langle \hat{g}_f \times \hat{id}_{D_x} \rangle
\]
eval of `$x$' within env `$i$'
when actual parameter `$d$'
is passed to argument `$x$'

The premise of the sentence is described by the commutative diagrams $(a)$, $(b)$, and $(c)$ in Figure 6. The equation is valid due to the existence of commutative diagram $(abc)$, thus the conclusion is valid.

(2) The existence of the pointers is due to Lemma 3.1. Hence, the commutative diagram in Figure 7 gives the needed pointers.

4 Conclusions

A common object technique equipped with the categorical and computational styles is outlined. As was shown, an object can be represented by embedding in a host computational environment. An embedded object is accessed by the laws of the host system. A pre-embedded object is observed as the decomposition into substitutional part and access function part which are generated during the object evaluation. They assist to easy extract of the result.

Acknowledgements

The author is indebted to JurInfoR-MSU Institute for Contemporary Education for stimulating the research. This research is supported in part by the Russian Foundation for Basic Research (project 96-01-01923).
Figure 6: Evaluation of a constant function (Explaination: this commutative diagram reflects a natural idea of the function constant, i.e. the evaluated map does not depend on an environment. A diagram to evaluate its argument variable is similar to diagram in Figure 3. The diagram (a) is a commutative-style description of evaluating the argument of the function $f$. The diagram (b) describes the intuitive reasons to observe $f$ as a function constant which results in $f(d)$, the value of $f$ in a point $d$. The diagram (c) determines $f$ as the valid map. An assembling diagrams (a), (b), and (c) gives the commutative diagram (abc). All the diagrams contain the parameters $g_f, g_d$ which are to satisfy the commutative law. Thus, the ‘solution’ of diagram (abc), if exists, relatively $g_f, g_d$ generates the pointers to access an environment.)

Figure 7: Pointers to access the environment with a constant function (Explaination: this commutative diagram gives one of the possible solutions of the diagrams in Figure 6 relatively the parameters $g_f$ and $g_d$. Thus, the parameter $g_d$ is replaced by $1_{D_x} \circ \text{Snd}, g_d$ by $(1_{D_x} \circ \text{Snd}), g_f$ by $f \circ \text{Snd}$, and $g_f$ by $(f \circ \text{Snd})$.)
Historical remarks

The known fact is that an idea to apply the categorical universe of discourse to the field of data models erased in the late 70th – early 80th.

In Russia Professor Lev Kuzin (1928–1997) payed special attention to discuss applications of categories to the field of data modeling1. His early idea was to use some kind of untyped calculi and impose some weak restrictions to capture the intuitive reasons concerning the data models. This approach tends to taking in mind a family of computation models as the restrictions of the untyped combinatory logic or untyped λ-calculus.

During more than a decade he was a project leading researcher (project Data Bank promoted by the Russian Academy of Sciences during 70th – early 80th). Having launched in Russia some research projects in a field he discovered that the vital point of a general data model is to build an access to the computation environment. His proposal was to separate the function symbol and an argument symbol (when the functional expression is included within a query), then to generate access to the domain for argument and to the domain for function separately. The most of the difficulties at this way were observed and some of them were resolved.

Later the more neutral approach was established, roughly speaking, to deal with all the both-syntax-and-semantic entities as with the homogeneous objects. The objects are interlinked by the relations which correspond to the scripts (script-driven data model). This model was extended to capture more dynamics. Studying the transitions between the scripts he verified some pure categorical models (with the explicit notion of a state in a computation model).

Nevertheless, the semantics of the computation in a category was not yet covered. During the last period of his research he was interested in discovering more and more facts which fit in the mathematical ideas beyond a category theory. In his draft research schedule he had done some improvements concerning the feasibility of a categorical abstract machine to evaluate the queries.

References


---

Building the Access Pointers to a Computation Environment


