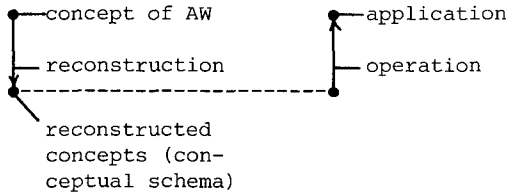
Constructive Abstract Data Types (CAD)

Hartmut H. Wedekind
Universität Erlangen-Nürnberg
Martensstraße 3
8520 Erlangen/West Germany

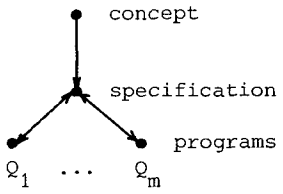
1. Distinction from conventional approaches

The motivation for CAD is to extend the idea of data abstraction to application programming. Conventional approaches confine themselves to operational concepts of system programming like STACK, QUEUE etc. and do not consider concepts like CONTRACT, INVOICE etc. of the application world (AW) with an arbitrary number of operations applicable to them, including those of the ad hoc type.

a) CAD are based upon a different methodology (fig. 1).



a) CAD methodology



b) Conventional methodology (Liskov e.a. [4])

Fig. 1: Different methodologies

Because of its logical defects a concept has to be reconstructed at first before it is operationally used. Reconstruction is a stepwise, acyclic procedure, using only logical means like, abstraction, composition, logical particles etc. Since Frege [2] concepts are gained via propositional forms. Carnap [1] extended the notion of a concept intensionally. Codds normal forms in this sense is an intensional theory of predication, an important refinement in a long history of the theory of concept formation, completely negleted by computer scientists.

b) Theory of abstraction versus "information hiding".

A constructive approach means that all notions and working concepts are introduced explicitly and furthermore successively according to an abstraction process described below. No implicit definitions are allowed as in axiomatization. (Frege has had a passionate dispute with Hilbert around 1900 about the issue "implicit definitions"). The explication of the notions "abstraction" establishes the theory of abstraction and is due to Frege and Lorenzen [5]. For practical computer science purposes the abstraction process must be constructed and formalized systematically. A colloquial statement proposing that abstraction is the art of emphasizing the important features and suppressing the unimportant details is too unspecific to be of any substantial significance. The theory of abstraction is used straight forward in building CAD and replaces the vague "principle of information hiding". Let A(x) and B(x) be two propositional forms like PERSONNEL(x) and ACCOUNT(x).

- Logical equivalence ($\sim$) of A(x) and B(x) as a prestep for abstraction.

$$A(x) \tilde{x} B(x) = {}_{DF}\bigwedge_{x} A(c) \longleftrightarrow B(x) \qquad (1a)$$

- Abstraction, i.e. explicit introduction of a set as an abstract an independent object.

$$C^*[\epsilon_z A(z)] = {}_{DF}\mathbb{A}_X [A(z) \tilde{z} X(z)] \longrightarrow C(X(z)) \qquad (1b)$$

$\mathbb{A}$ is an indefinite, universal quantifier having all A(x) out of every language, whatsoever, as an indefinite range. $\epsilon_z A(z)$ is the new abstract object in the propositional form C*. The abstraction process according (1a) and (1b) must be applied recursively. If $A(z) \longleftrightarrow X(z)$ is interpreted as a synonym statement, then $\epsilon_z A(z)$ is called a concept. Concepts and sets are two kinds of abstract objects, looked upon intensionally or extensionally.

Example:
The example should show that natural languages are not quite suitable to exhibit abstraction.

A(z) = z belongs to the personnel
B(z) = z gehört zum Personal (German)
X(z) = any other language
C(X(z)) = The labor contract of (z belongs to the personnel)
C*(ϵ_zA(z)) = The labor contract of the concept "personnel".

Quotation marks are used in general to notify abstraction.

2. Reconstruction phase

We take the concept STACK as an example to get a reference to the conventional discussion. The concept is fairly uninteresting from application programming point of view.

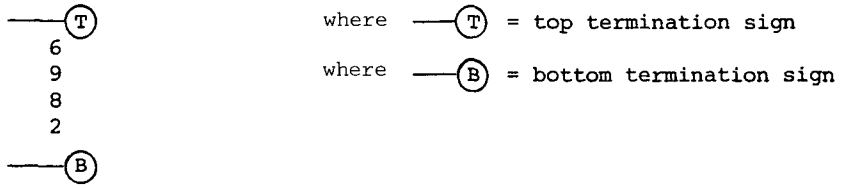


Fig. 2: Intuition of a STACK. (Members of the stack are supposed to be unique).

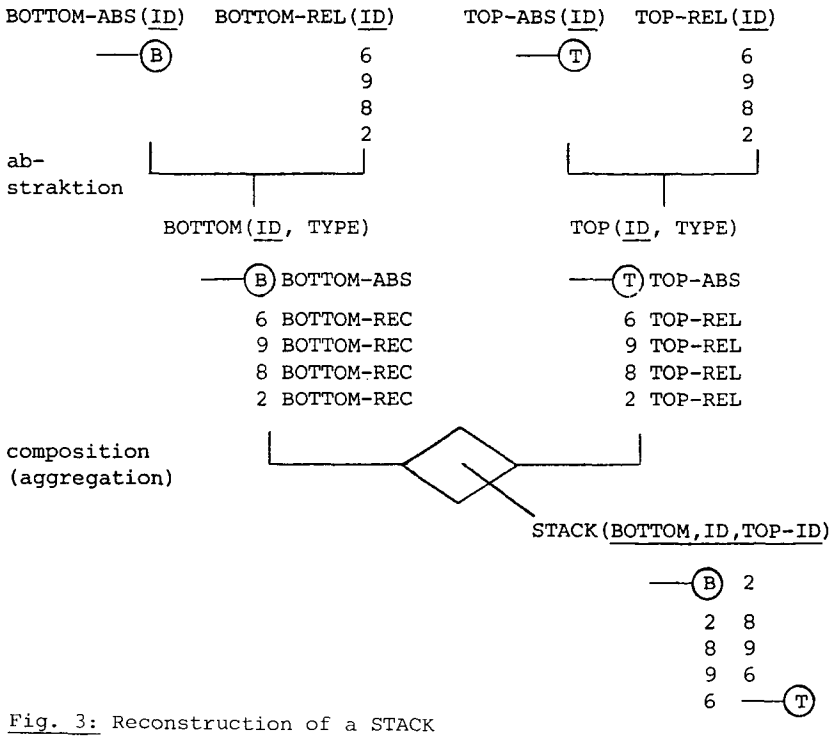


Fig. 3: Reconstruction of a STACK

For a complete reconstruction integrity constrains in the sense of DB Systems must be imposed.

In case of push:

$$STACK_{After} = STACK_{Before} - \{<v, —(T)> | <v, —(T)> \in STACK_{Before}\}$$

$$\cup \{<v,x> | <v, —(T)> \in STACK_{Before}\}$$

$$\cup \{<x, —(T)>\}$$

In case of a pop: similar

From the stack example we learn that reconstruction by means of abstraction and composition, and integrity constraints (invariance conditions in case of a change) constitutes a concept completely.

3. Operation phase

In this phase one can start with concepts already reconstructed, e.g. PERSONNEL (PNR, ...) and ACCOUNT (PNR ...). We want to show that the specification of application programs has to start with concept formation using abstraction theory. For example, the concept DISMISSAL (DM) is gained in three abstraction steps (fig. 4).

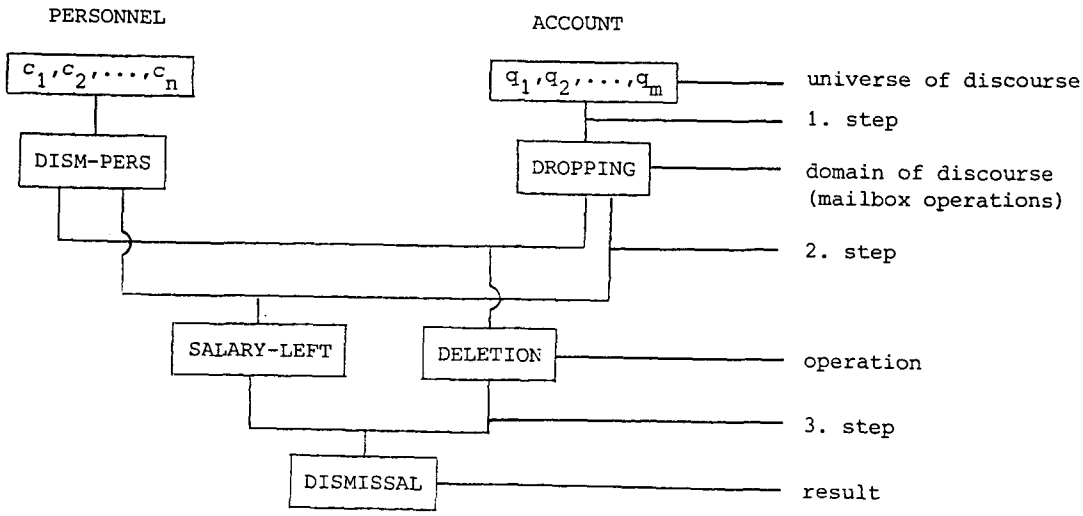


Fig. 4: Formation of the CAD DISMISSAL

Starting from PERSONNEL(x) and ACCOUNT(x) the concepts "Dismissed Personnel" (DISM-PERS), "Dropping" (DROPPING) etc. are gained stepwise and acyclic using the abstraction process (1a) and (1b). Thus program modules are systematically generated and not produced by an ingenious intuition using metaphorical principles like the one of information hiding. A CAD yields the construction of integrity preserving transactions in the sense of DB-systems. Likewise, operational integrity constraints must be imposed. It is within the concept of DISMISSAL that SALARY-LEFT has to be execute before DELETION. Taking Habermann's notion of a path expression, the operational integrity of our example is represented by

PATH SALARY-LEFT; DELETION END.

";" denotes strict sequential execution. As a DB-transaction one can write

```
TRANSACTION
   BEGIN
        :
        READ PERSONNEL(c_i),
        READ ACCOUNT (q_i),
        [SALARY-LEFT(c_i,q_i);DELETION(c_i,q_i)],
        :
   END
```

[ ] denotes an atomic action from the user's point of view, which must be executed completely or not at all. The compution of SALARY-LEFT without DELETION is not admitted, because it would violate the concept "DISMISSAL".
The notion of a type in programming languages has to be translated into the notion of a transaction in DB-systems. The notions correspond to each other, but they are not equivalent.

References:

[1] Carnap, R.: Meaning and necessity, The University of Chicago Press, 1947 (reprint from 1936).

[2] Frege, G.: Funktion, Begriff, Bedeutung, Vandenhoeck und Ruprecht Publ. Comp., Göttingen 1971 (reprint from 1879).

[3] Habermann, A.N.: On the concurrency of parallel processes, in: Jones, A.K. (ed.): Perspectives in Computer science, Academic Press, New York, p. 77-90.

[4] Liskov, B.H. e.a.: Specification techniques for data abstraction, in: IEEE Transaction on Software Engineering, SE-1, (1975), p. 7-19.

[5] Lorenzen, P.: Normative Logic and Ethics, Bibliographisches Institut, Mannheim, 1969.

[6] Smith, J.M. and Smith, D.C.: Database Abstraction - Aggregation and Generalization, in: ACM TODS, Vol. 2 (1977), No. 2, p. 105-133.