LiTH-IDA-R-88-43

ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Types in Space

Towards Flexible High-speed Object Oriented Data Management Tools of the Future

by

Göran Rydqvist

Mikael R.K. Patel

Leif Larsson

RESEARCH REPORT
CADLAB, November 1988

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

The Department of Computer and Information Science
Linköping University

PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

(*Linköping Studies in Science and Technology. Dissertations.*)

*No 77	**Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982.
No 94	**Hans Lunell:** Code Generator Writing Systems, 1983.
No 97	**Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983.
*No 109	**Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984.
*No 111	**Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984.
*No 155	**Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987.
No 165	**James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987.
No 170	**Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987.
No 174	**Johan Fagerström:** A Paradigm and System for Design of Distributed Systems. 1988.
No 192	**Dimiter Driankov:** Towards a Many-Valued Logic of Quantified Belief. 1988.

Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

No 48	**Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985.
No 52	**Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985.
No 60	**Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985.
No 71	**Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987.
No 72	**Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986.
No 73	**Ola Strömfors:** A Structure Editor for Documents and Programs. 1986.
No 74	**Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986.
No 104	**Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987.
No 108	**Rober Bilos:** Incremental Scanning and Token-based Editing. 1987
No 111	**Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987.
No 113	**Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987.
No 118	**Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987
No 126	**Dan Strömberg:** Transfer and Distribution of Application Programs. 1987
No 127	**Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems. 1987
No 139	**Christer Bäckström:** Reasoning about Interdependent Actions. 1988.
No 140	**Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing. 1988.
No 146	**Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System. 1988.
No 150	**Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behaviour. 1988.

LiTH-IDA-R-88-43 ISSN-0281-4250

Types in Space

Towards Flexible High-speed Object Oriented Data Management Tools of the Future

by

Göran Rydqvist
E-mail: gorry@majestix.liu.se

Mikael R.K. Patel
E-mail: mikpa@majestix.liu.se

Leif Larsson
E-mail: leila@majestix.liu.se

Abstract: Algorithm selection and data modeling are key issues in programming activity. Concepts in the minds of human beings underlie these issues. Algorithm and data together, are models of the behavior of these concepts. Most traditional modeling support systems are based on the architecture of the executing hardware machines, and little attention has been given to the requirements of the underlying concepts.

In this report we extract results from psychology as a starting point and consider the programming situation from that point of view. We use the resulting perspective to search for answers to a number of basic questions: What is the relation between concepts and their executable realizations? How do we achieve the necessary model power implied by this relation? Why are not traditional systems adequate?

This research is supported by the Swedish National Board of Technical Development.

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Contents

1. Introduction

Data modeling and algorithm design are recognized as major problems when realizing computer programs. These two topics are inextricably intertwined which is not always well understood. Throughout the last three decades a number of different theories, methodologies, and tools have been proposed and presented to solve the basic problems involved in the selection of data representation and/or algorithm. Database researchers have focused on the problem of data modeling in relation to minimization of representation, while the algorithm theorists have analyzed algorithms without deep concern for the data representation aspect. As the problem is to model behavior of real or imagined entities, it is obvious that there are important gains to be made from considering the two aspects together.

The previous argument is easily demonstrated by an example. Consider the problem of representing a *set* abstract datatype. Choosing a list structure results in the complexity property of $O(n)$ for locating an element in the set, while using a hash-table representation reduces the same property to a constant factor.

Although the basic problems of algorithm and data structure selection are intertwined thus, there exists an important border between the two. This border may, if identified, be formed into an interface. Such an interface can exist only if a high-level data model provides sufficient abstraction and modeling power, so that we are not forced to mix our representation with the code of the algorithm. High-level specification systems, like Refine [Smith85, REFINE87], use set theory as such a high-level model. We envision that the concept of an association is a powerful, free form that allows the modeling of both type and data structures at a high level, which is very natural to human beings.

The referred types of high-level models are normally associated with a very low runtime efficiency. However, this cost may be reduced by the application of "intelligent" compiling techniques, to a level approaching the most efficient paradigms. Refine is a good example of research heading in this direction.

The report is structured in the following manner. In the first section we investigate the notion of an association, a labeled relation between objects or values, and its graphical equivalent, the semantic network. The second section presents some important results from psychology research. Semantic networks are used as a medium to describe the structure of human memory. We focus on concepts and how they are formed. An important border is identified between formal world con-

cepts, which are well defined, and natural concepts, which do not have precise boundaries and vary with the experience of different people.

The next section starts out by examining the most basic concepts of an executing machine, the primitive data objects (e.g. integers). These are first described on the concept level using semantic networks, and thereafter compared to the typical fix-size realizations that fit the hardware of the executing machines. These fix-sized entities are demonstrated to be restricted realizations of the general association model they are based on.

The resulting fix-size primitive data objects are generalized into arrays of such. Arrays are thus described as restricted semantic nets. The next step is the natural generalization of arrays to aggregates of data, which allow different sized elements and arbitrary attributes. However, the common machine-oriented realization of an aggregate shares its schema with every other aggregate of the same type. This restriction is given special attention in the fourth section. The circle that started with associations, as the basis of concepts, has now been closed. Any further generalization of an aggregate will return to the semantic net.

The section continue by discussing the importance of type systems and the dangers associated with the construction of application type systems that have no connection to the original. The discussion is summarized by number of conclusions concerning the type concept. Four major aspects of a data model are identified: Object type, representation type, position schema, and type schema. The object type is the name of the concept being modeled, the representation type is the name of the data structure used to implement the concept in the target abstract machine, the position schema is the implementation of the representation type, and the type schema is a map that returns the primitive object types associated with each attribute.

The last part of the section reflects on these data model aspects in relation to the three trade-off dimensions speed, space, and model power. For instance, static resolution of one or more of the aspects result in a smaller and faster running system with lower model power.

In the fourth section we critically examine three important data modeling paradigms: records, object oriented programming, and databases. The purpose of the text is to show that the paradigms only solve small parts of the complete data

modeling problem.

The last section uses the four major aspects of a data model to classify the most important programming languages, e.g. Pascal which represents the first generation of Algol derivatives, C++ as the newest contribution to the family of statically compiled object oriented languages initiated by Simula, Smalltalk-80 as the well known, high power interpreted object oriented language, and finally Lisp. None of the examined systems come very close to the desired data model power, and we conclude the discussion by recognizing that all the mentioned systems represent a particular point in the trade-off space (speed, space, and model power), and that a more powerful system should be able to cope with objects having different trade-offs.

The major results presented in this report are:

1) An analysis of the important relations between human beings, specification languages and the executing machines.
2) Natural concepts are not suited to be realized with type hierarchies, records, or any other data model that force fix structure conformation of the individuals in a category.
3) The identification of four major aspects of a data model: Object type, representation type, position schema, and type schema.

The objects represented in programming activity are implementations of concepts that reside in our minds. This relation between concepts and objects indicates that there is much of interest in the structure of concepts; a data model is complete if it is able to represent all of these concepts. Such a complete model is probably not possible. However, we should strive to find models that are able to capture as much as possible.

The boundaries of a concept is not always well defined and concepts are *not* the kind of *classes* found in object oriented systems. The former point can be understood by regarding the structure of a natural concept as a result of cultural influences and individual experience. Cases exist when objects can not be unambiguously classified to a single category. Further, concepts are not necessarily formed from properties present in all of the individuals, but rather from the most common properties.

A data model which provides all of the referred aspects to the application at run-

time, is at the high end of the model-power spectrum, while a system which removes all by compilation to gain execution speed, is at the low end.

The representation type and its associated position schema are mandatory to the existence of an interface between a high-level data model, and actual representation. The interface allow trade-offs to be made by choosing different representations. This cost may further be reduced by the application of "intelligent" compiling techniques, towards a level approaching the most efficient paradigms.

We believe that the material presented in this report sums up to a conceptual framework for the implementation of very powerful data management systems.

2. Semantic Networks

During the past decades, semantic networks have become an important representational media for several research domains including computer science and psychology. The primary application is knowledge representation in some context; the AI community use it to represent the semantics of sentences, expert knowledge in expert systems etc., psychologists use it for models of the structure of human memory. In this report we will use networks to schematically model human memory, and as a universal high level data structure in our quest for clarification of certain important concepts of the computer science domain.

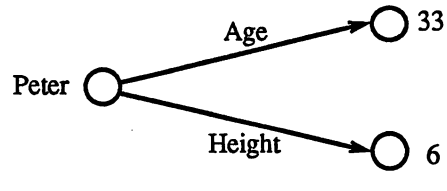


Fig 1: Semantic net example.

Historically the concept of an *association*, and a net of such, trace back as far as the ancient Greeks. An overview and a history of these pre-computer era applications can be found in [Anderson73]. An early and important application to computer science was done by Quillian 1966 in his PhD thesis [Quillian66]. He used the term semantic networks in his attempt to represent the *semantics* of English sentences. Since then, practically every network structure have been labeled semantic although they have been used to represent all sorts of *non-semantic* things. A more correct term would be associative networks [Brachman79]. We will however stick to the term semantic throughout this report, as the distinction often is

fuzzy.

A simple, informal drawing of a semantic net may look like figure 1. If we choose a node, say *Peter*, we may regard Peter as an *object* - i.e. we temporarily fix our perspective for a specific purpose. Outgoing arcs are *attributes* and target nodes are *values*. This perspective yields an alternative representation as a set of *(o,a,v) 3-tuples*.

```
get(Peter, Age) -> 33
get(Peter, Height) -> 6
```

Fig 2: Access function get().

We introduce a function *get(o,a)* which takes two parameters, an *object* and an *attribute*, and returns the *value* pointed to. In figure 1 we would get the results in figure 2.

```
put(Peter, Age, 33)
put(Peter, Height, 6)
```

Fig 3: Access function put().

The companion function to get is *put(o,a,v)* which does one of two possible things: If *a* exists, then the *current v* is replaced by *parameter v*, else *a* and *v* is added. *O* is created if it does not exits. For instance, figure 1 could be constructed by the use of put, as in figure 3.

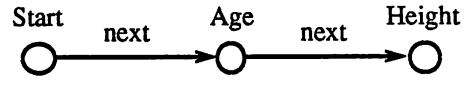


Fig 4: List Returned by all().

A third function *del(o,a)* removes an existing *outgoing arc* from *o*. A fourth and last function *all(o)* returns a set of all attributes of *o*. *All(Peter)* applied in the context of figure 1 would return the result in figure 4 or some other conventional implementation of a set. The idea behind the *all* function is to allow an agent that does not know the structure of an object, to access it.

A strong benefit of semantic networks is that indices and other redundant structures may explicitly be represented in the notation itself. For instance, to add an inverse index to figure 1. The loop construct, in figure 5, iterates through all

attributes of Peter and binds each in turn to A. Each attribute, in the second line, is *regarded* as an object, and Peter is added as an attribute. The idea is to repeat this process with several Persons and get a result like in figure 6.

```
forall(Peter, A)
  put(A, Peter, get(Peter, A))
```

Fig 5: Inverse index construction.

The objects Age and Height may now be accessed to answer questions like, *all objects where age > 30*, etc.

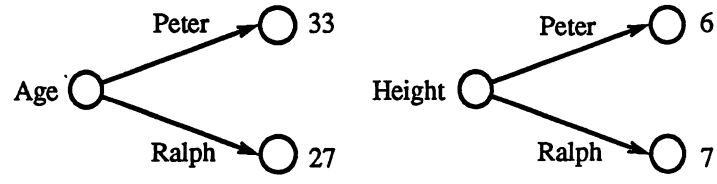
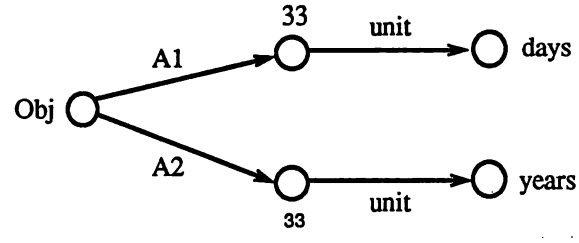


Fig 6: Simple inverse index.

What has been presented so far is the basic, intuitive idea of the network representation. We proceed by showing what kind of trouble we might encounter. Consider figure 7.




```
get(Obj, A1)   -> 33
get(33, unit)  -> AMBIGUOUS
```

Fig 7: The intuitive model is ambiguous.

The analysis of the situation show that our intuitive identification of a node by its *symbol* need to be improved to *position and symbol*, where position is a *unique coordinate* in an arbitrary space. We summarize with the following definitions.

```
Node:  (Position, Symbol)
Arc:   (Source Node, Attribute Position, Destination Position)
```

The Arc definition is somewhat cryptic. The second element is a *node* instead of a

simple *symbol*, because we need to be able to switch perspective on the attribute, and regard it as an object which may have its own attributes. This was exemplified above by objects Age and Height. The change of perspective is called *view inversion.*

We have reached definitions that result in a representation which is believed to have an *expressive power* "sufficient to encode any fact or concept that is encodeable in any other formal, symbolic system" [Hendrix79]. Semantic networks may thereby serve as a unifying medium of representation between diverse specialized representations appropriate to their respective domains [Shubert79]. The networks will in this report be used to explain important properties of human memory, data structures, and computer hardware.

3. Human Perception and Memory

In this section we describe a model of human perception and memory. The presentation is based on material from "Human Information Processing" [Lindsay77] and focuses on the parts which are relevant to our discussion of some difficult concepts of the computer science domain.

Human memory consists at least of *sensory*, *short term* and *long term* memory subsystems. Sensory memory is closely coupled to the external world sensors. It holds sensory information long enough for the visual analysis parts of the brain to work correctly. Short term memory holds information for a few seconds to a couple of minutes. The classical description [Miller76] describes short term memory as a passive store. In more recent theoretical descriptions [Baddeley83] it is described as a temporary workspace (working memory), which is closely associated with *consciousness* and *thought.* By concentrating, for instance, we can recall information from the long term memory, or focus on small, selected partitions of the sensory input.

The long term memory makes up a persons total knowledge of the world. Sensory information is partially fed to it without conscious intervention. We choose to picture this memory as å large semantic network. Learning is divided into two separate processes: *Adding relations* to the existing knowledge or *extending* the number of nodes in the network. The former increase the understanding of what we already know, and the latter occurs if we are exposed to sensory patterns that we have no previous knowledge of or when we excite our brain with new imagined

things or worlds.

We imagine the extension of the network to be separate, in the sense that it is unrelated to the previously known. By experimenting, mentally or physically, inferring or recognizing properties the interconnectivity of the network increases. Any relation may be removed if not recalled frequently. The phenomenon is a kind of cashing, a trade-off between *computation* (speed) and *memory* (space).

The network is scanned by a hypothetical mind process that tries to find similarities among subnets, guided by teachers, active analytical thinking, or maybe predefined rules with *evolutionary* advantages. The process results in new subnets linked to the individual entities, from which they were formed, with *Is-A* relationships [Brachman83]. The process is called *abstraction* and the result is called a *concept.*

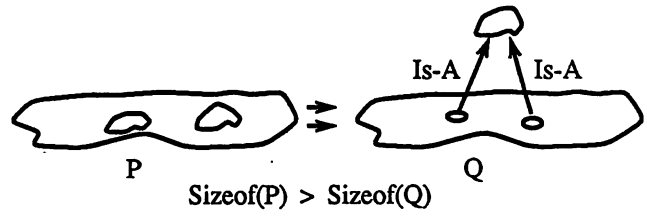


Fig 9: Abstraction.

Axiomatic systems are an important special case among the variety of imagined/conceived worlds. The concepts formed by applying the rules of an axiomatic system differ from those, whose structure is based on real world experience or imagined world exploration. We call the former type *formal* concepts and the latter *natural* concepts. The facts in a formal world is either axioms or theorems constructed from the axioms. Thereby, the formal concepts are defined precisely and unambiguously which enable the powerful symbolic manipulations that are provided by existing formal systems, e.g. mathematics. Natural concepts are not as "nice" however. They do not always have precise boundaries and they show cultural as well as individual differences.

The description below of a natural concept is formulated in the spirit of empirical research in natural categories and their conceptual structure [Rosch73, 74, 75, 77]; summarized in "Psychology and Language" [Clark77].

A natural concept is a kind of average of all the individual objects from which it was formed, where an attribute is part of the concept only if it is common enough to exceed a certain threshold.

Note that this does not say that the concept is the most typical of the individuals, the prototype, but rather a new part of the net with no corresponding percept. Some individuals are more "typical" to the category, though, and the prototype is what people generally have in mind when they use the concept. Further, "average" should not be interpreted as union of all the attributes of all the individuals. The emphasis is instead on "threshold", which ensures that the attributes of the concept are very common among the individuals.

The concept model in the previous paragraph require some examples.

Imagine a dog.

This phrase triggers a mind picture or some similar feeling of the concept dog. This dog is different in different peoples minds, because the concept was formed from different *individuals*, which depend on personal experience. It is similar enough, though, to enable communication.

The dog is the size of a cat.

Most peoples' pictures will at this point switch to a particular dog or subcategory. Most dogs of these peoples experience were larger. This indicate the *average* and *threshold* properties of the model.

An *iconic picture* capture the most distinguishing features of the concept it represents. The artist creating the icon uses his talent to draw only the most *characteristic* parts, which somehow define the essence of the concept. The referred object or concept is easily recognized. This does not require the observer to have seen the icon before, as is the purpose of icons.

The structure of natural concepts, as described above, implies that there exist no simple method of deciding the category of an unknown object. We believe that classification is done by matching the unknown object to every concept in long term memory with a matching procedure that returns a *similarity estimate*. This estimate is interesting only if it exceeds a threshold, and is in that case fed to short term memory. In most cases only one concept match, which means that the object unambiguously belong to a certain category; the higher the estimated value

is, the more typical member is the object. In other cases where two or more estimates are high, the process indicates that the object is difficult to classify, for instance a small tree would result in a high value for both "tree" and "shrubbery".

By selecting two or more concepts a person can abstract new concepts, analogously with the process described above. The abstraction process may thus be repeated on the concepts to form higher level concepts. The interpretation of this is *not* that all concepts have a fixed position in a single hierarchy - the reality is tremendously more complex. As stated in the first sentence of this paragraph, any concepts may be the *temporary atoms* when abstracting new concepts. As virtually everything we learn somehow is explained in terms of something else [Hofstadter79], these concepts probably form a huge, cyclic "spaghetti" type graph.

Concepts are created and named in natural language practice as a convenience to the speakers and in programming practice as a means to handle the *complexity* of the underlying implementations. The names themselves can be constructed from combinations of phonemes, from combinations and/or differences of other similar/dissimilar object or concept names.

We summarize this section by emphasizing the important conclusions reached by the reasoning path we have followed.

1) There are two kinds of concepts: *Natural* and *formal* concepts.

We understand the former to have fuzzy borders and an internal structure which vary with cultural differences and individual experience. This implies that the objects we have in computers that represent these concepts, can not without *losses* be forced into the kind of uniformity supported by the currently popular data modeling paradigms. We will elaborate this point in the critique sections. The latter category are *precise* and *well defined* to suit the purpose to which they are constructed, which make them easier to handle with fixed structures in computers.

2) As the individuals belonging to a category can be very different, attributes of natural concepts can *not* in the general case [Putnam75] be factored out and inherited without unnatural constructs which allow selective inheritance.

Inheritance is thereby reduced to an implementation strategy in such cases. We will further explain this point in the "Critique of Object Oriented Programming" section.

3) Classification of an unknown object requires a match to every concept in memory.

Each match result in a *similarity estimate*. The highest estimate is the natural category. This process is clearly non trivial.

4) Names are "keys" to lookup concepts inside our brains.

There is a strong relation between concepts and types, which make names very important to programming practice. We will continue to discuss this point in the following section.

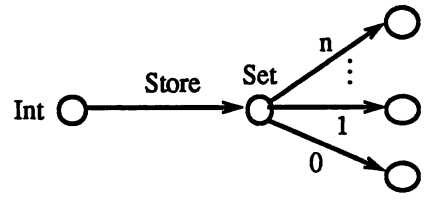
4. Types

In this section we will start by looking at the most basic concepts in programming languages, the *primitive types*, and analyse them by applying the high level view provided by the semantic networks. The second part of the section investigates the relation between the networks and the higher level data structures we use in everyday programming. The view we advocate is that the entities of both levels can be viewed as *compilations* or *reduction* of the general semantic network (association) model.

The bottom-up view of data structures of the initial section and the top-down view of the following are combined with the conceptual level in the third part, where we argue for a wide perspective of what a type is. Finally we conclude the section with a discussion of trade-offs in relation to problem complexity.

We begin this section by taking a close look at the most basic entities in programming languages, the primitive datatypes of the particular *abstract machine*. Such an abstract machine "know" among other things how to do arithmetic on different representations, like integer, char, real. These representations are provided externally as a set of *primitive*, atomic *types* of a programming language that is built upon this abstract machine.

The precise definition of an *integer* as stated in mathematics literature is unnecessarily complicated for our purpose. We provide an informal, but sufficient one for our argumentation: *An integer is unbounded and store whole numbers.* The naive interpretation of the definition is a *set,* where each element in the set represents one unit of the number - a style known as Peano arithmetic.




```
| all( get(Int, Store) ) |-> n
```

Fig 10: Semantic net implementation of an integer.

An implementation of such a set, using the semantic network notation, is shown in figure 10. We compute the integer number by counting the number of elements in the set. This set realization is an unrestricted realization of the definition above, and storage space is proportional to the size of the number stored. It is however cumbersome in real life computation. A second interpretation of the integer definition is shown in figure 11.

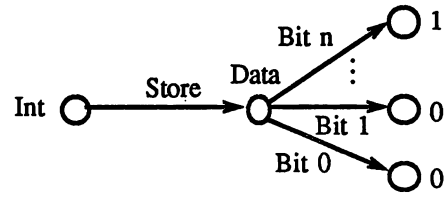


Fig 11: Semantic net implementation of an integer.

The central idea is to give the symbols, that make up the number, a weight according to their respective positions. We choose a minimal set of symbols {0, 1}. A sequence of these form a binary number, for instance 101, which is interpreted as $1*2^2 + 0*2^1 + 1*2^0 = 5$, translated to decimal base.

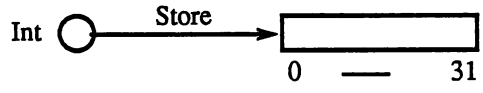


Fig 12: Restricted implementation of an integer.

The implementation in figure 11 has a number of advantages compared to traditional fixed size implementations. Storage space is proportional to $log_2 n$ of the

stored number, and there is *no bound* on the maximum size number we can store. In practice, though, it is not a good solution, because it does not match well with the most efficient implementations of integers in hardware. Good utilization of hardware resources and fast physical paths require fix widths. The solution supported by today's systems is a fix chunk of memory that should be large enough, typically *16* or *32* bits. Note that this solution, exemplified in figure 12, is a *restricted* implementations of the full concept.

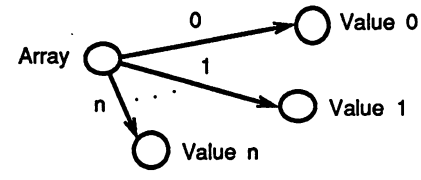


Fig 13: Array as a regular structure in a semantic net.

We continue by investigating the relation between the composite datastructures/types we use in everyday programming and semantic nets. The fixed representation shown in figure 12 is naturally generalized to a common and useful regular sequence of values, conventionally named *Array*. A high level view of an array, as an association, is provided by the semantic nets, figure 13.

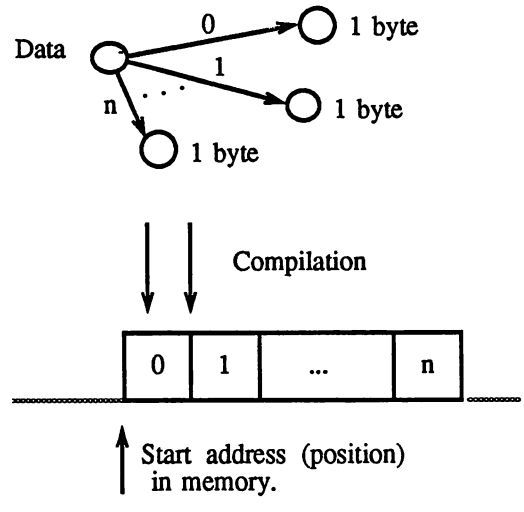


Fig 14: Hardware memory.

In figure 13 the array is shown as a close relative to the *aggregate*, commonly referred to as *record*, with the restriction that the attributes must be positive *integer symbols*. It is important to note that there is no restriction on differing value sizes, as in the further restricted version, where the arcs and target nodes are removed by compilation. These are replaced by *sequence* and a *starting address*, forcing each element to a constant size, which enable indexing to be performed as

a simple *address calculation*.

Semantic nets and the conceptual model of an *record* are close: The center node is the record object, the labels of the outgoing arcs are attributes. The important differences are that records are *fix* size entities and the values may not themselves always have relations connected to them. These values are further *private* to the record, i.e. they may not be referred to from the outside, see figure 15. The implications of these limitations will be further investigated in the "Critique of Records" section. A Lisp type *property-list* is closer still. Its size is dynamic, although it suffers from the same reference problem as records.

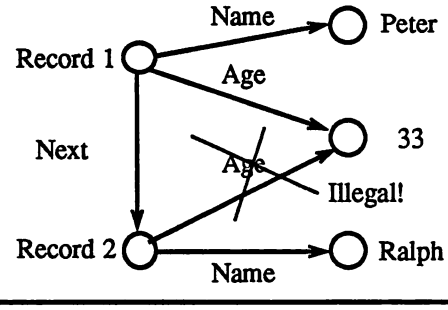


Fig 15: Illegal value references.

At the application level, the remedy to the reference problem is to make the node with symbol *33* in figure 15 a new record or property-list, and change the declarations of the age attribute to a *pointer*. The change is not transparent, though, and the value in the new record must be accessed via a special field name, which is clearly undesirable.

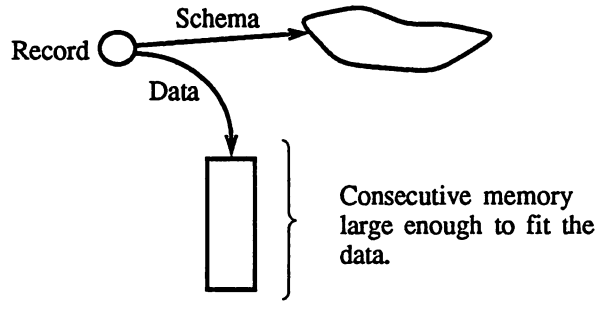


Fig 16: Record model.

The implementation model of a record is shown in figure 16. The *schema* part of the record is a map between attribute symbols and positions, in the figure, offsets from the start of record's memory address.

An access to a record pointed to by *p* first access the schema to get an offset, add

it to *get(p,data)*, and complete the access. One of the large gains from compiled languages is that the attribute is *known* at compile time, which permits the compiler to do the schema computation at compile time, throw away everything except the data pointer and generate a special instruction as in figure 36. This kind of instruction is supported directly by many hardware processors.

We mentioned above that the schema is a map between attribute symbol and position. More precisely we define a *position schema* to be a function

```
F(o, a) -> position
```

The schema function *F* takes an object *o* and an attribute *a* as parameters and returns a position associated with *a*.

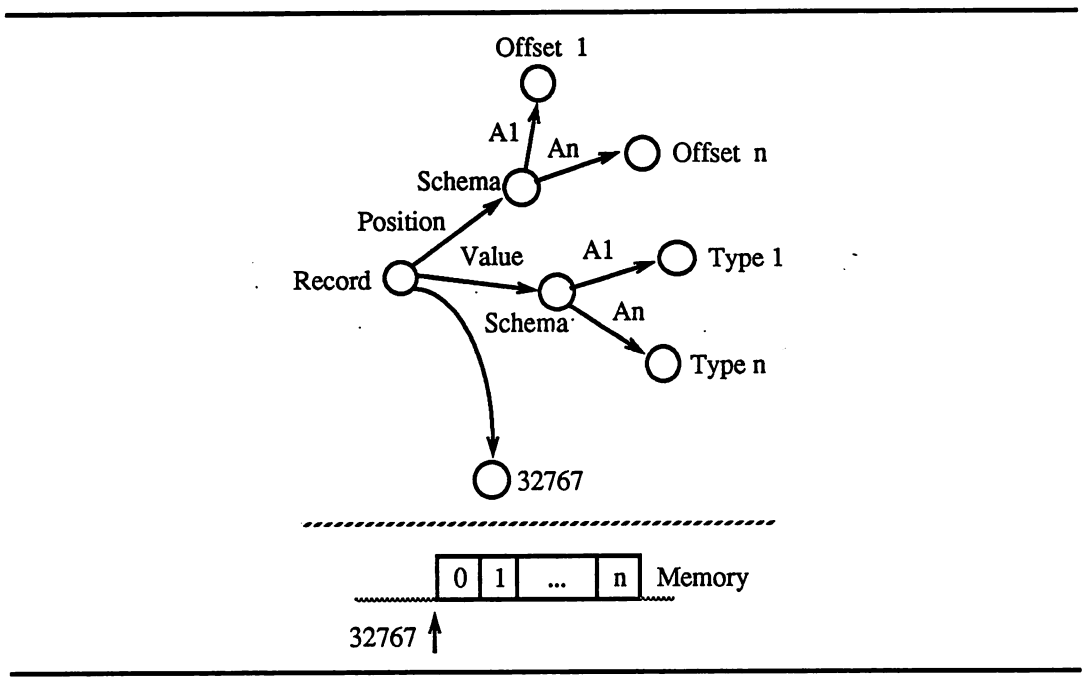


Fig 17: Record implementation model.

This view holds for any implementation of an *association* datatype. As a record is composed from primitive types it is possible to put the types of these into a schema too. We call this a *type schema* and define it as a function.

```
G(o, a) -> Type
```

The schema function *G* takes an object *o* and an attribute *a* as parameters and returns the type associated with *a*.

We conclude that semantic networks, as argued above, is a *superset* of arrays and

records though both, especially in their non restricted versions, provide nice working abstractions. Any higher level data structures, like heaps, stacks, queues, hash indexes, b-trees etc. can be composed from arrays, records and code. Thereby we have demonstrated that semantic networks can be used to fully describe these too.

We will now consider the problem of what a type is, what an attribute is and how they relate. We start out with a simple *window system* example. This window system contains only two kinds of windows: those with *white background* and those with *black background.* Note that we carefully avoid the word type.

The example in figure 18 is simple but adequate to convey the modeling options available at this level. We have in the second part removed the *type attribute*, and concatenated each of its values to the *name.* What is really the difference between the two? Is it just syntactic or does it reach further? What is confusing about the example is that the difference between type and attribute values seem to disappear - somehow they are interchangeable. We have deliberately made the example confusing to strengthen our points.

```
Implementation 1
  Class window
     int type* {Black, White}
  end;

Implementation 2
  Class black_window end;
  Class white_window end;
```

Fig 18: Simple window system.

Figure 19 show a type system with a simple implementation of a symbol table. Each entry in the symbol table has a name, which is the type from the view of this type system, here called *system_type* to distinguish it from the type attribute. Note that the type attribute is nothing but an attribute to the "system" type system, as is shown in the figure by the lone arc labeled type ending in "nowhere". What we actually try to do is to build a *second* type system upon the initial one.

Dynamic typing in the figure is to have the *system_type* attribute available at runtime. If the language provided an *iswindow()* predicate, it would be trivial to build

*The word "type" is not a reserved word. It is simply a field identifier.

your own predicates *isblackwindow()* and *iswhitewindow()* that operate on the window type. This approach would result in more freedom from the "system" type system than if you use the *isblack_window()* and *iswhite_window()* that are naturally provided by the second implementation, at the cost that you must build that part yourself - an application type system. This type system is however free from the arbitrary restrictions imposed by language constructors.

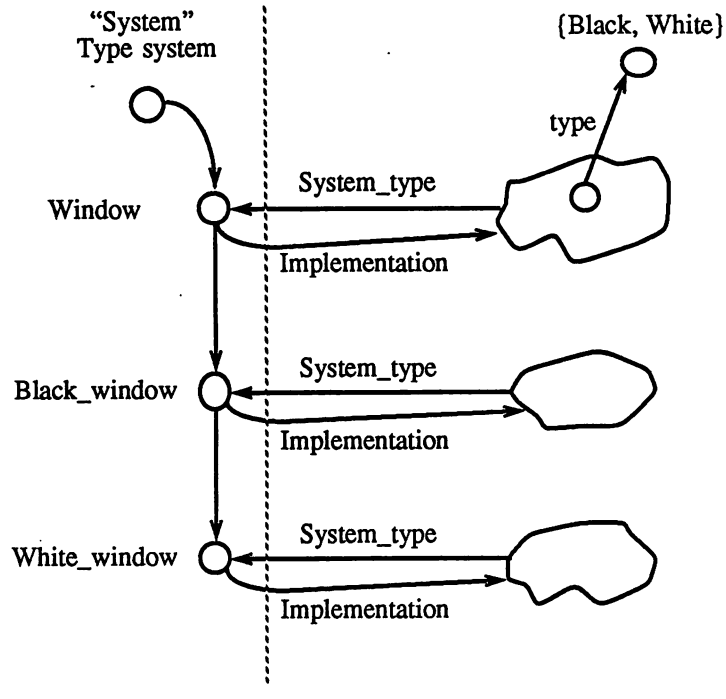


Fig 19: Types and Attributes.

Some of the confusion in the initial example might have been avoided by changing implementation 1 in figure 18 to the version shown in figure 20.

```
Class window
   int color {Black, White}
 end;
```

Fig 20: Attribute naming variation.

The important conclusions of this section is that you cannot extend the system types, i.e. the types that are built in, by adding your own attribute *type* to the objects you create, and that *how* the split between types and attributes is done, is not simple. We notice the potential of an attribute free system, although the number of types would be very high (cartesian products).

Now we are prepared to discuss the wide perspective of the type concept. In our earlier presentation of concepts we mentioned that a name or a word triggers a concept in our minds. This was emphasized, even though it is obvious, because it plays such a central role.

Figure 21 show that a name refers to both concept and implementation. The relation to the concept is, of course, implicit, while the relation to the implementation is explicitly part of the used notation or its semantics. If there was not a strong relation between name and concept, *naming would not matter*. Anyone familiar with the programming level of computer systems know that this is not the case. On the contrary, naming plays a crucial role in any kind of problem solving situation. If the name choices are good it serves to strongly *reduce* the problem of handling complexity.

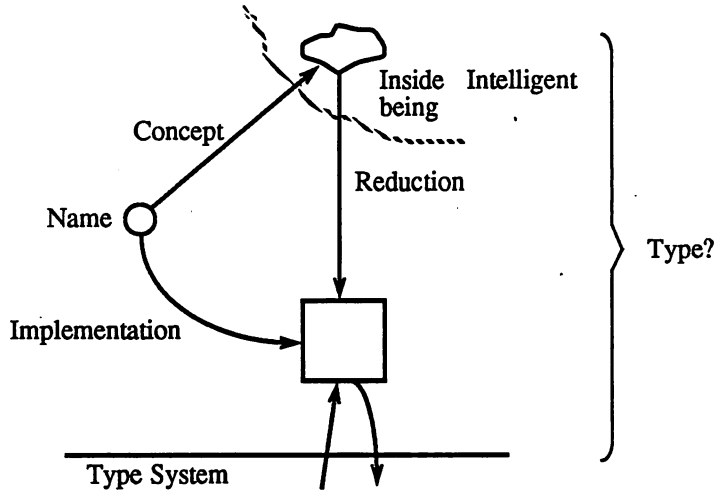


Fig 21: What is a type?

There is an interesting relation between concept and implementation. Often the concept exits before the implementation, developed in some other discipline like mathematics. For instance, consider the addition operation applied on integers. Addition of integers is a general concept of something that is able to perform the mathematical concept of addition, which is not limited in any sense. An implementation in hardware is always a reduction of this generality into something manageable, like a 32 bits wide position system. The reduction property implies that the relationship is potentially *one* (concept) to *many* (implementations).

Note that no specific part of figure 21, has been labeled *type*. Our goal is not to arrive at yet another type definition, but to find a unifying view on all of them. Depending on temporary perspective choices, type mean different things. The type

concept is obviously complex and can not be simply defined without loosing something.

An implementation of a concept may from one view be just that, while from another, it is a type. Consider an implementation of a window. From the view of the inside of the object, it is an implementation of a concept, a display window, while from the outside we regard it as a type that may be instantiated when needed. We note that the relation between type system and implementation is important. The "type view" of the window only exists in our minds, until it is entered into the type system.

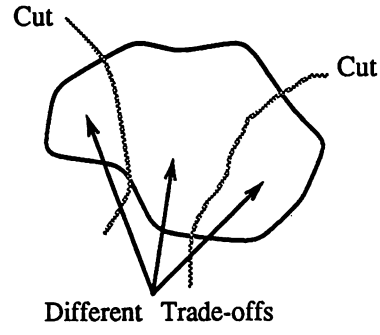


Fig 22: Different cuts in the set of types belonging to an application.

The following few paragraphs consist of a discussion of *trade-offs*. It is our strong belief that almost all difficult problems are difficult largely because trade-offs have to be made so that the resulting system is able to present answers in reasonable time.

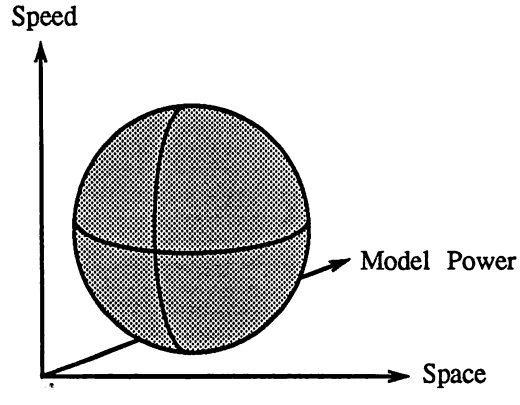


Fig 23: Trade-off dimensions.

Figure 22 is represents all the types of a particular application. A programmer of this application largely "knows" what different types have different trade-off requirements in each set. The programmer thereby have to cut the set of types

into subsets with similar trade-off requirements. This is an absolute necessity for most difficult problems.

The three *trade-off dimensions* seen in figure 23 are *speed*, *space* and *model power*. Ideally you would like to choose any point in the positive three dimensional space they form. Unfortunately, there are dependencies among the three variables, strongly *limiting* the number of possible choices. We choose*, as an analogy, to picture this reachable space of choices, as the surface of a sphere. If we choose a point suitable to our application, it will most certainly not land on the surface. We are thereby forced to choose a possible trade-off *on* the surface of the sphere.

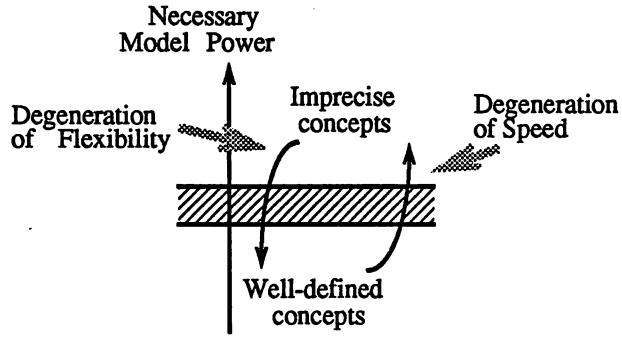


Fig 24: Model power axis.

Figure 24 zoom in on the model power axis. In the space to the right of the "Necessary model power" axis we imagine a large number of concepts that we want to implement. The concept range from well-defined to imprecise.

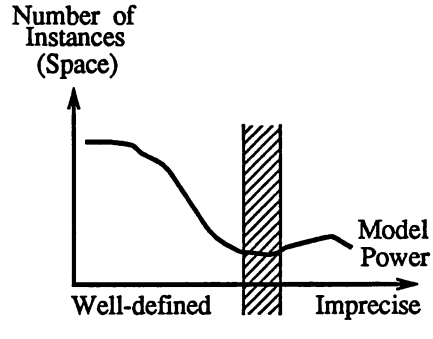


Fig 25: Space axis.

The arrow down show that if we implement an imprecise concept directly with a lower modeling power than is strictly necessary, we loose flexibility. This does not mean that we can not run our program unless the processor is at the "high"

*The sphere was chosen to illustrate a hypothetical space. It is not based on reality.

level required. Instead we need to build a more powerful abstract machine before we start the modeling process, and this machine must be built with great care.

There is an interesting relation between the number of instances and model power. Figure 25 show the first quadrant where the "number of instances" control the y axis and model power range from well-defined (low) to imprecise (high) on the x axis. The graph is not based on actual measure, but has been drawn to show the difference that exist because we use well-defined concepts to implement imprecise ones.

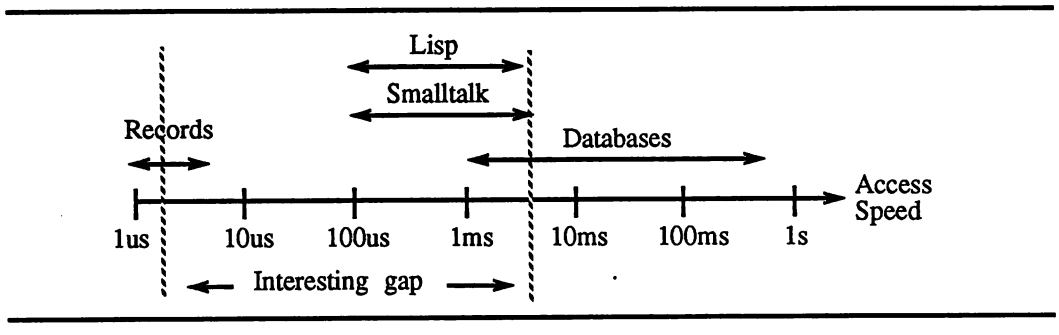


Fig 26: Speed axis.

Finally the speed axis, shown if figure 26, approximates the speed it takes a number of languages or systems* to perform a single primitive value access. The exact values of each system varies in reality from implementation to implementation. However, differences of a magnitude remain independently of the skill of the programmers in each case. The large gap between records and the rest of the systems is an open invitation to every self-respecting programmer!

5. Current Approaches

In this section we critically view three major data modeling paradigms. Records are the first subject to be scrutinized, and an important one, because records are the primary implementation choice of the two other paradigms, namely object oriented programming and database systems. Our attack of object oriented programming focuses on the use of inheritance as the sole relation. This part of the discussion is independent of Records, as is most of the database systems discussion. The purpose of the text is to show the inadequacy of each paradigm and we will argue strongly against some *solve-it-all* cultures.

5.1 Criticism of Records

Many problems surface when trying to use records as a data modeling primitive.

*References are given in the Programming Languages section.

We will take a look at some examples concerning illusory space savings, forced uniformity in the set of attributes, a single type for each attribute, and separate data description. More on the subject can be found in [Kent79].

The major idea of the record is that the schema is separate from the data instance, we call this *shared schema*. The motivation is that we save space by avoiding the repetition of the same information. We call the reversed view *distributed schema*; the information to interpret a data instance is stored explicitly in that instance in such a way that it may be accessed to get a description of the data. Casually glancing at the record data model, the schema appears to be distributed.. The extraction of the schema is a computational, machine-oriented way of handling large amounts of data. The space saved by sharing the schema easily becomes *illusory*. The resulting rigid system does not handle variation well, and the user is confronted with the unnatural requirement of predicting the worst case. This estimate is then allocated in every instance, resulting in much waste.

Person-1		Person-2	
name	"Stan Smith"	name	"Ann Smith"
address	"Park Avenue"	address	"Riverside Dr."
age.	24	age	22
		maiden-name	"Jones"

Fig 27: Variation among instances of a single type.

A *person* is an example of a natural concept we would like to represent with a record. The record model restrict us in two dimensions: All persons *must* have the same fields, and a given field is forced to contain information of equal size in all records. What attributes would be needed to model a person entity? Consider name, address, social security number, height, age, sex, maiden name etc. All of these attributes are not needed in *every* person instance. Some people have not got a social security number only girls have maiden names etc.

To accommodate the variations, we have three options:

1) Define the record format to include the *union* of all relevant fields, where not all the fields are expected to have values in every record.

These null values naturally leads to storage overhead. A user or application programmer is forced to predict every possible field that may appear in a person

record, and there is no restriction on *what* fields have values *when*.

2) Allow the same field to have *different meanings* in different records.

The meaning of the field would be interpreted by adding an extra type field to the record. Unfortunately the interpretation of this record will only be known by the application that conceived it. The database and independent applications treat the two conceptually associated fields as separate chunks of data, with no known restrictions between them. Further, space will be wasted if not all the data *happens* to be of equal size.

3) Define a new record type for every combination of fields.

This approach eliminate the storage overhead, but if the data varies too much, the number of record types will explode. The desired correspondence between concept and record disappears, and no restriction exist that prevent two records to model the same entity at the same time.

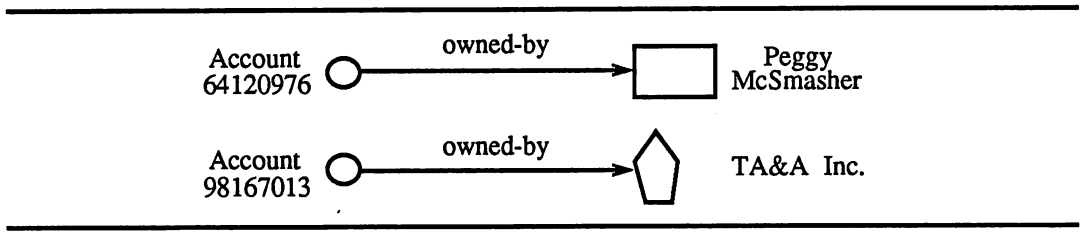


Fig 28: Different types referred by a single attribute.

Our second example show some problems of the restricted "second dimension". Suppose we have a *bank account* record type. An account can belong to either a corporation, or to a person. This relationship is naturally modeled by having an *owned-by* field in the account record. The problems occur because persons are identified by *social security number*, while corporations are identified by *name* (string). The solutions available are the same as the three listed above, and are accompanied with the same list of problems.

We conclude this section by stating that records, although very useful as is evidenced from the vast experience collected of their application to various problems, are not sufficient to handle the full user-level *data modeling requirements*. Further problems exist that we have not pointed to here, like lack of self-description, unrestricted access etc. We do not, however, belittle the usefulness of the record model as a tool to build higher level models. Instead we strive to emphasize a

number of arguments that imply that *the record model is not sufficient.*

5.2 Criticism of Object Oriented Programming

The currently very popular "religion" called *object oriented programming* [Wegner86] strongly advocates that types should be structured into hierarchies, or lattices in the case where multiple inheritance is allowed. It is a common notion that these type structures reflect the structure of our minds, or stronger, that our minds are structured in the same manner.

Our previous discussion illuminates a few things however: The boundaries of a concept is not always well defined and concepts are *not* the kind of *superclasses* found in object oriented systems. The former point can be understood by regarding the structure of a concept as a result of cultural influences and individual experience. Cases exist when objects can not be unambiguously classified to a single category as argued in the "Human Perception and Memory" section. These and the following arguments are of course only relevant for natural concepts, but are *very* relevant in that case. Our discussion on individuals and concepts should be enough to make the latter point clear. Concepts are not necessarily formed from properties present in all of the individuals, but rather from the most common properties. The primary consequence of this is that inheritance among natural concepts describing types is *artificial* and requires dangerous additions like selective inheritance.

Most of the available object oriented systems uses the class mechanism to store the schema in only one place. This implementation choice, while saving memory, leads to several problems. Every instance is forced to implement all instance variables of the class, which is clearly incompatible with natural concepts. To provide dynamic creation of new classes, the meta-class concept is introduced; a class whose instances is classes. The dilemma here is that the chain of meta-classes is potentially infinite [Ungar87].

Our general feeling is that inheritance works well for well defined, formal world concepts, and not as well for objects from the real world. The conclusion is that inheritance is an implementation choice, as is the class/instance separation. We also regard inheritance as a syntactic convenience, which is one of the most common arguments for: *You only have to describe the difference to preexisting classes.*

accessing dates can be written independently of how dates are represented. A change in representation, only effects the access functions. We notice that the representation type and the position schema, both are fixed by the access functions, yielding two S's. The type schema is available through system predicates, like integerp, atomp, stringp and the object type can never be part of the Lisp type system. An alternative view is that the application programmer resolves the object type statically. We conclude that access functions result in a S^3D system.

```
(setq '(88 08 28))
(defun year (date) (car date))
(defun month (date)
  (car (cdr date)))
(defun day (date)
  (car (cdr (cdr date))))
```

Fig 40: Access functions in Lisp.

The second Lisp programming culture we are interested in differs in one very important aspect: The position schema is stored with the data in a list structure called property-list, see figure 41. The access functions provided by the system, *getprop* and *putprop*, performs a search for the requested attribute, resulting in a representation that is position independent. We conclude that Lisp property-list programming is a S^2D^2 system.

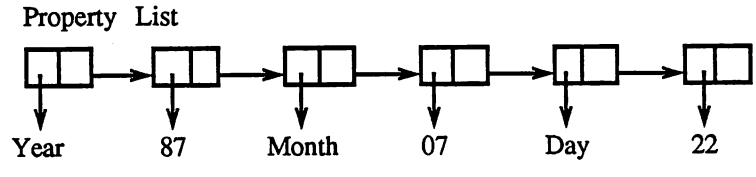


Fig 41: Common implementation of a property-list.

It is important to understand that a D^4 system is *not* what is needed to solve all problems of data modeling, neither is S^4, SDSD or any other combination. What we envision is a system which provide a D^4 data model to the application programmer. The system must be able to do trade-offs, or be directed by the programmer, towards a result containing sets of data with different trade-offs, ranging from D^4 to S^4. Our conclusion is that the systems discussed all are good. They do not, as is recognized by the computer science community, provide good solutions to all application domains, but solve their respective domain well. The kind of system we quest for will be a superset of all these systems and handle as large a domain as possible. It is most unlikely that a single system will be able to han-

dle all of the problems involved, though we are convinced that the current domains can be enlarged substantially by adapting the presented views.

7. Conclusion

We have in this paper discussed data modeling with varying emphasis, based on the human being as the problem solver. The initial idea was that any implementation in a computer is a restricted realization of concepts in our minds. Psychology provides us with models of human memory and the concepts stored therein, thereby determining the necessary expressive power of a data model. We envision that the concept of an association is a powerful, free form that allows the modeling of both type and data structures at the required level.

This view, applied to basic concepts in the computer science domain yield an interesting perspective. The concept of an integer can be modeled using associations. The executable realization, a primitive data object, is severely restricted. Following the thread of reason we generalize the primitives into regular collections of such, namely arrays. Arrays is further generalized into structures allowing different object sizes, namely aggregated data. Any further generalization lead back to the starting point - we have come full circle.

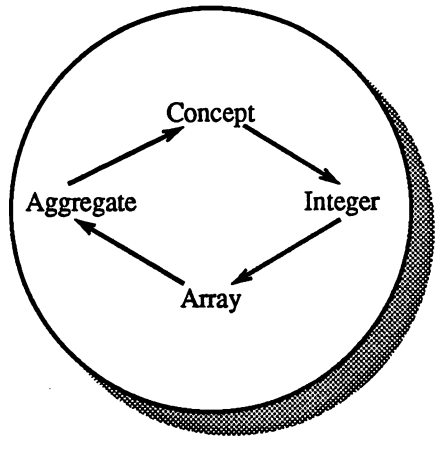


Fig 42: Conceptual representation circle.

The operator selection types of the primitive data objects can be shared between several objects. This is the first major aspect of a data model. The aggregated data can further share attribute to position map (schema). This is the second aspect.

By including the human being into the very difficult discussion of types, we have

been able to find a supposedly unifying view on the type concept, if not a definition. The view allow us to find the last two aspects of a data model, the abstract type and the representation type. The abstract type is the name of the concept we have in our minds, and the representation type is an executable realization. Each realization represents a particular trade-off in storage space, execution time, and model power. Regarding each of these as a dimension, we are able to discuss particular trade-offs as points in this space.

A critical examination of existing data modeling paradigms show that these are inadequate to handle the requirements stated in our analysis. Especially, the record paradigm is too rigid and lack persistency, the object oriented style attacks too many problems at once without thorough understanding of the underlying theories, and databases are too rigid and too slow.

Application of the four major aspects of a data model result in an interesting classification of programming languages. By assigning an S for statically resolved by a compiler, or a D for dynamically available at runtime, to each of the four aspects of each language we get a symbol that is easily interpreted in the model power and execution speed dimensions. Several D's imply a flexible and powerful system, while a domination of S's imply fast but inflexible.

We introduce a notation to express different classifications conveniently. The notation is a position system, where each position corresponds to an attribute in enumerated order: Object type, Representation type, Position schema, Type schema. For instance SDSD means static object type, dynamic representation type, static position schema, and dynamic type schema. The same symbol appearing in sequence is abbreviated with a superscript index, for instance S^2DS.

Pascal is a good representative for the first generation of Algol derivatives. We classify Pascal to S^4. C++ is the newest contribution to the family of statically compiled object oriented languages, initiated by Simula. C++ is classified as S^4. The Smalltalk-80 high power interpreted object oriented system is DS^2D system. Lisp programming based on access functions is S^3D and finally Lisp programming based on property-lists is S^2D^2.

It is important to understand that a D^4 system is *not* what is needed to solve all problems of data modeling, neither is S^4, SDSD or any other combination. What

we envision is a system which provide a D^4 data model to the application programmer. The system must be able to do trade-offs, or be directed by the programmer, towards a result containing sets of data with different trade-offs, ranging from D^4 to S^4.

The major results presented in this report are:

1) An analysis of the important relations between human beings, specification languages and the executing machines.
2) Natural concepts are not suited to be realized with type hierarchies, records, or any other data model that force fix structure conformation of the individuals in a category.
3) The identification of four major aspects of a data model: Object type, representation type, position schema, and type schema.

The objects represented in programming activity are implementations of concepts that reside in our minds. This relation between concepts and objects indicates that there is much of interest in the structure of concepts; a data model is complete if it is able to represent all of these concepts. Such a complete model is probably not possible. However, we should strive to find models that are able to capture as much as possible.

The boundaries of a concept is not always well defined and concepts are *not* the kind of *classes* found in object oriented systems. The former point can be understood by regarding the structure of a natural concept as a result of cultural influences and individual experience. Cases exist when objects can not be unambiguously classified to a single category. Further, concepts are not necessarily formed from properties present in all of the individuals, but rather from the most common properties.

A data model which provides all of the referred aspects to the application at runtime, is at the high end of the model-power spectrum, while a system which removes all by compilation to gain execution speed, is at the low end.

The representation type and its associated position schema are mandatory to the existence of an interface between a high-level data model, and actual representation. The interface allow trade-offs to be made by choosing different representations. This cost may further be reduced by the application of "intelligent" compiling techniques, towards a level approaching the most efficient paradigms.

We believe that the material presented in this report sums up to a conceptual framework for the future implementation of very powerful data management systems.

Acknowledgments

We would like to direct special thanks to Mr. Nils Dahlbäck and Mr. Erik Stoy for helpful critique and support, to the rest of the members of the CADLAB research group at the Department of Computer and Information Science, Linköping University, and to Sylvester Stallone for his great movies.

References

Abelson85

H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs,* MIT Press, McGraw-Hill, pp. 152, 1985.

Allen78

J. Allen, *Anatomy of LISP*, McGraw-Hill, 1978.

Anderson73:

J. R. Anderson, and G. H. Bower, *Human associative memory*. Holt, New York, 1973.

ANSI83

American National Standard Institute, *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD-1815A-1983, New York, 1983.

Atkinson87

M. P. Atkinson, O. P. Buneman, *Types and Persistence in Database Programming Languages,* ACM Computing Surveys, vol. 19, no. 2, pp. 105- 190, June 1987.

Baddeley83

A.D. Baddeley, *Working memory*, Philosophical Transactions of the Royal Society, London, pp 302, 311 - 324, 1983.

Bic86

L. Bic, J. P. Gilbert, *Learning form AI: New Trends in Database Technology,* IEEE COMPUTER, pp 44 - 54, March 1986.

Birtwistle73

G. Birtwistle, O-J. Dahl, B. Myhrhaug, K.Nygaard, *Simula Begin,* Studentlitteratur, Lund, Sweden, 1973.

Brachman79

R. J. Brachman, *On the Epistemological Status of Semantic Networks,* Associative Networks, Academic Press, New York, pp. 3 - 50, 1979.

Brachman83

R. J. Brachman, *What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks,* IEEE COMPUTER, October 1983, pp. 30 - 36.

Brogida85

A. Brogida, *Features of Languages for the Development of Information Systems at the Conceptual Level,* IEEE Software, pp. 63 - 72, January 1985.

Cardelli85

L. Cardelli, P. Wegner, *On Understanding Types, Data Abstractin and Polymorphism,* ACM Computing Surveys, vol. 17, no. 4, pp. 471 - 522, December 1985.

Clark77

H. H. Clark, E.V. Clark, *Psychology and Language,* Harcourt Brace Jovanovich Inc., New York, 1977.

Codd70

E. F. Codd, *A relational model for large shared data banks,* Comm. ACM 13:6, pp. 377 - 387, 1970.

CODASYL71

*CODASYL Database Task Group,*Report, ACM, New York, April 1971.

Dahl70

O-J. Dahl, B. Myrhaug, K. Nygaard, *SIMULA Common Base Language,* Norwegian Computing Center S22. Oslo, Norway, 1970.

Danforth88

S. Danforth, C. Tomlinson, *Type Theories and Object-Oriented Programming,* ACM Computing Surveys, vol. 20, no. 1, pp 29 - 72, March 1988.

Date81

C. J. Date, *An Introduction to Database Systems,* Addison-Wesley, Reading Mass, 1981.

Donahue85

J. Donahue, A. Demers, *Data Types Are Values,* ACM Transactions on Programming Languages and Systems, vol. 7, no. 3, pp. 426 - 445, July 1985.

Goldberg85

A. Goldberg, D. Robson, *Smalltalk-80 - The Language and Its Implementation,* Addison-Wesley, Reading, Mass., 1985.

Goodwin

J. W. Goodwin, *ITEM: A simple knowledge representation tool,* Unpublished manuscript, Department of Computer and Information Science, Linköping Uni-

versity, S-581 83 LINKÖPING, SWEDEN.

Hendrix79

G. G. Hendrix, *Associative Networks: Represetntation and Use of Knowledge by Computers,* Ed. N. V. Findler, Academic Press, pp. 51 - 92, 1979.

Hofstadter79

D. R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid,* Section: *Ant Fuge,* Penguin Books, pp. 310 - 365, 1979.

Jensen74

K. Jensen, N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, 1974.

Kellogg86

C. Kellogg, *From Data Management to Knowledge Management,* IEEE COMPUTER, pp. 75 - 84, January 1986.

Kent79

W. Kent, *Limitations of Record-Based Information Models,* ACM Transaction on Database Systems, vol. 4, no. 1, pp. 107 - 131, March 1979.

Larsson88

T. Larsson, Z. Peng, *A Relational Approach to VLSI Design Databases,* Research Report LiTH-IDA-R-88-37, September 1988, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN. Also in Proc. of the 36th ISMM International Conference on Mini and Microcomputers and Their Applications, Sant Feliu de Guixols, Spain, June 27-30, 1988.

Lindsay77

P. H. Lindsay, D. A. Norman, *Human Information Processing,* Academic Press, New York, 1977.

McCalla83

G. McCalla, N. Cercone, *Approaches to Knowledge Representation,* IEEE COMPUTER, pp. 12 - 18, October 1983.

McCarthy60

J. McCarthy et. al., *LISP 1 Programmer's Manual,* Computation Center and Research Laboratory of Electronics, M.I.T., Cambridge, 1960.

Miller83

G. A. Miller, P. N. Johnson-Laird, *Language and Perception,* Cambridge: Cambridge University Press, 1976.

Padgham88

L. Padgham, R. Rönnquist, *LINCKS: An Imperative Object Oriented System,* Research Report LiTH-IDA-R-88-15, June 1988, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN. Also in Proc. of the 20th Annual Hawaii Int. Conf. on System Sciences, Hawaii, 1987.

Pirsig74

R. M. Pirsig, *ZEN and the Art of Motor Cycle Maintenance*, CORGI, pp. 75 - 77, 1974.

Putnam75

H. Putnam, *The Meaning of Meaning: Mind, Language and Reality,* Philosophical Papers, Vol. 2, Cambridge: Cambridge University Press, 1975.

Quillian66

R. M. Quillian, *Semantic Information Processing*, Ed M. Minsky, Mass: The MIT Press, pp. 227 - 170, 1968.

REFINE87

Refine User's Guide, For Refine Version 2.0, Reasoning Systems Inc., 1801 Page Mill Road, Palo Alto, CA 94304, 1987.

Rosch73

E. Rosch, *On the internal structure of perceptual and semantic categories*. Ed. T. E. Moore, Cognitive devleopment and the acquisition of language. New York: Academic Press, pp 111 - 144, 1973.

Rosch74

E. Rosch, *Linguistic relativity,* Ed. A. Silverstein, Human communication: Theoretical perspectives. New York: Halsted Press, pp 95 - 121, 1974.

Rosch75

E. Rosch, *Cognitive representations of semantic categories*, Journal of Experimental Psychology: General, pp 104, 192 - 233, 1975.

Rosch75

E. Rosch, C. B. Mervis, *Family resemblances: Studies* in the internal structure

of categories, Cognitive Psychology, pp 7, 573 - 605, 1975.

Rosch77

E. Rosch, *Human categorization,* Ed. N. Warren, Advances in cross-cultural psychology, Vol. 1, London: Academic Press, 1977.

Rönnquist84

R. Rönnquist, E. Sandewall, *The Relationship Between Ordered and Unordered Trees in I.M.S. Theory,* Research Report LiTH-IDA-R-84-04, April 1984, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

Rönnquist86

R. Rönnquist, *The Information Lattice of Networks Used for Knowledge Representation,* Research Report LiTH-IDA-R-86-02, January 1986, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

Sandewall83

E. Sandewall, S. Hägglund, L. Jonesjö, O. Strömfors, *Formal Specification and Implementation of Operations in Information Management Systems,* Research Report LiTH-IDA-R-83-02, February 1983, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

Sandewall83

E. Sandewall, S. Hägglund, L. Jonesjö, O. Strömfors, *Theory of Information Management Systems,* Research Report LiTH-IDA-R-83-03, September 1983, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

Shubert79

L. K. Schubert, R. G. Goebel, N. J. Cercone, *Associative Networks: Represetntation and Use of Knowledge by Computers,* Ed. N. V. Findler, Academic Press, pp.121 - 175, 1979.

Siemienski83

P. Siemienski, *A Specialized Database for VLSI CAD,* Research Report LiTH-IDA-R-83-09, December 1983, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

Siemienski84

P. Siemienski, *A Specialized CAD Database,* Research Report LiTH-IDA-R-84-23, December 1984, Department of Computer and Information Science, Linköping University, S-581 83 LINKÖPING, SWEDEN.

SIGPLAN76

Conference on Data: Abstraction, Definition and Structure, SIGPLAN, 11. special issue, 1976.

Smith85

D. R. Smith, G. B. Kotik, S. J. Westfold, *Research on Knowledge-Based Software Environments at Kestrel Institute,* IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985.

Stroustrup86

B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

Stroustrup86

B. Stroustrup, *An Overview of* C++, SIGPLAN Notices, Vol. 21, No. 10, October 1986.

Tichy87

W. F. Tichy, *What Can Software Engineers Learn form Artificial Intelligence?,* IEEE COMPUTER, pp 43 - 54, November 1987.

Ullman82

J. D. Ullman, *Principles of Database Systems,* Computer Science Press Inc., pp. 12, 1982.

Ungar87

D. Ungar, R. B. Smith, *Self: The Power of Simplicity*, OOPSLA '87 Conference Proceedings, Orlando, FL, 1987, pp. 227 - 241. Available as SIGPLAN Notices, Vol. 22, No. 12, December 1987

Wegner86

P. Wegner, *Classification in Object-Oriented Systems*, SIGPLAN Notices, Vol. 21, No. 10, October 1986.

Wijngaarden75

A. Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, *Revised report on the Algorithmic Language Algol 68,* Acta Informatica 5, pp1 - 236, 1975.

Woodward74

P. M. Woodward, S. G. Bond, *Algol 68-R Users Guide,* Her Majesty's Stationery Office, London, 1974.

Zaniolo86

C. Zaniolo, H. Ait-Kacit, D. Beech, S. Cammarata, L. Kerschberk, and D. Maier, *Object Oriented Database Systems and Knowledge Systems*, Expert Database Systems, ed. Larry Kerschberg, pp. 49 - 65, The Benjamin/Cummings Publishing Company Inc., 1986.

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr:
Report no: LiTH-IDA-R-88-43

Upplagans storlek:
Number of copies:

Datum:
Date: November 1988

Projekt:
Project:

Titel:
Title: Types in Space - Towards Flexible High-speed Object Oriented Data Management Tools of the future

Författare:
Author: Göran Rydqvist, Mikael R.K. Patel, Leif Larsson

Uppdragsgivare:
Commissioned by:

Dnr.:
Call no:

Rapporttyp:
Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk:
Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord):
Abstract (150 words):

Abstract: Algorithm selection and data modeling are key issues in programming activity. Concepts in the minds of human beings underlie these issues. Algorithm and data together, are models of the behavior of these concepts. Most traditional modeling support systems are based on the architecture of the executing hardware machines, and little attention has been given to the requirements of the underlying concepts.
.br
In this report we extract results from psychology as a starting point and consider the programming situation from that point of view. We use the resulting perspective to search for answers to a number of basic questions: What is the relation between concepts and their executable realizations? How do we achieve the necessary model power implied by this relation? Why are not traditional systems adequate?

Nyckelord (högst 8):
Keywords (8):

Keywords: Data modeling, databases, object oriented programming, object oriented databases, semantic networks, programming languages, data types, type inheritance.

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-88-37 **Tony Larsson, Zebo Peng:** A Relational Approach to VLSI Design Databases. Also in *Proc. of the 36th ISMM International Conference on Mini and Microcomputers and Their Applications*, Sant Feliu de Guixols, Spain, June 27-30, 1988.

LiTH-IDA-R-88-36 **Zebo Peng:** Semantics of a Parallel Computation Model and its Applications in Digital Hardware Design. Also in *Proc. of the 1988 International Conference on Parallel Processing,* Pennsylvania State University, August 15-19, 1988.

LiTH-IDA-R-88-35 **Zebo Peng:** A Horizontal Optimization Algorithm for Data Path/Control Synthesis. Also in *Proc. of the 1988_IEEE International Symposium on Circuits and Systems,* Helsinki Univ. of Technology, Finland, June 7-9, 1988.

LiTH-IDA-R-88-34 **Arne Jönsson, Nils Dahlbäck:** Talking to a computer is not like talking to your best friend. Also in *Proc. of the First Scandinavian Conference on Artificial Intelligence*, March 9-10, 1988, Tromsö, Norway

LiTH-IDA-R-88-32 **Erik Sandewall:** An Approach to Non-Monotonic Entailment. To appear in *Methodologies for Intelligent Systems*, vol II, *Proc. of ISMIS 1988 conference*, Torino, Italy, 1988, Z.W. Ras and L. Saitta (eds), *North Holland*, 1988, pp. 391-397.

LiTH-IDA-R-88-31 **Erik Sandewall:** The Semantics of Non-Monotonic Entailment Defined Using Partial Interpretations.

LiTH-IDA-R-88-30 **Erik Sandewall:** Future Developments in Artificial Intelligence. Also in *Proc. of European Conference on Artificial Intelligence (ECAI)*, Munich, August, 1988.

LiTH-IDA-R-88-29 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part III: Decision Procedure.

LiTH-IDA-R-88-28 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part II: Concurrent Actions.

LiTH-IDA-R-88-27 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part I: Sequential Actions.

LiTH-IDA-R-88-26 **Lin Padgham:** NODE: A Database for Use by Intelligen Systems. Also in *Proc. of ISMIS '88,* Torino Italy, October 12-15, 1988.

LiTH-IDA-R-88-23 **Johan Fagerström, Yngve Larsson:** Two Contributions on Debugging Distributed Systems. Also in *Proc. of ACM SIGPLAN/SIGOPS, Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 5-6, 1988.

LiTH-IDA-R-88-22 **Johan Fagerström:** Design and Test of Distributed Applications. Also in *Proc. of the International Conference on Software Engineering, ICSE*, Singapore, April 1988.

LiTH-IDA-R-88-21 **Johan Fagerström, Lars Strömberg:** A Paradigm for Distributed System Design and Test. Also in *Proc. of COMPCON'SPRING 88*, San Fransicso, Febr-March 1988.

LiTH-IDA-R-88-19 **Lin Padgham, Ralph Rönnquist:** From a Technical to a Humane Environment: A Software System Supporting Cooperative Work. Also in *Proc. of the GDI Internationa Conference on USER INTERFACES,* Rüschlikon, Switzerland, October 20-21, 1986.

LiTH-IDA-R-88-18 **Nils Dahlbäck:** Mental Models and Text Understanding - a Commented Review.

LiTH-IDA-R-88-17 **Bengt Lennartsson:** Multi-Level Software Architectures - Industrial Experiences and Research Directions. This report is a revised version of a paper in *Advance Working Papers of The Second International Workshop on Computer-Aided Software Engineering.* Cambridge, Massachusetts. July 12-15, 1988.

LiTH-IDA-R-88-16 **Lin Padgham:** A Model and Representation for Type Information and Its Use in Reasoning with Defaults. Also in *Proc. of AAAI'88, American Association for Artificial Intelligence*, 1988.

The Department of Computer and Information Science
Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *ASLAB - Application Systems Laboratory,* which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science,* which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratory*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

Research Reports 1988

LiTH-IDA-R-88-43 **Göran Rydqvist, Mikael R.K. Patel, Leif Larsson:** Types in Space - Towards Flexible High-speed Object Oriented Data Management Tools of the Future.

LiTH-IDA-R-88-42 **Mikael R.K. Patel:** What should a Programming Environment for Threaded Interpretive Languages provide? Also in *Proc. of the 8th Rochester Forth Conference on Programming Environments*, Rochester, New York, USA, June 1988.

LiTH-IDA-R-88-41 **Tony Larsson:** Hardware Verification based on Algebraic Manipulation and Partial Evaluation. Also in *Proc of IFIP WG 10.2 International Working Conference on "The Fusion of Hardware Design and Verification"*, Glasgow, Scotland, July 3-6, 1988.

LiTH-IDA-R-88-39 **Zebo Peng, Krzysztof Kuchcinski, Bryan Lyles:** CAMAD: A Unified Data Path/Control Synthesis Environment. Also in *Proc. of the IFIP TC-10 Conference on Design Methodologies for VLSI and Computer Architecture*, Pisa, Italy, September 19-23, 1988.

LiTH-IDA-R-88-38 **Z** [illegible] Systems. Also published in [illegible] *Euromicro Journal*, V[illegible]