Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

303

J. W. Schmidt S. Ceri M. Missikoff (Eds.)

Advances in Database Technology – EDBT '88

International Conference on Extending Database Technology
Venice, Italy, March 14–18, 1988
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editors

Joachim W. Schmidt
Fachbereich Informatik, Johann Wolfgang Goethe-Universität
Postfach 11 19 32, D-6000 Frankfurt am Main 11, FRG

Stefano Ceri
Dipartimento di Matematica, Universitá di Modena
Via Campi 213/B, I-41100 Modena, Italy

Michele Missikoff
IASI-CNR
Viale Manzoni 30, I-00185 Rome, Italy

CR Subject Classification (1987): D.3.3, E.2, F.4.1, H.2, I.2.1, I.2.4

ISBN 3-540-19074-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-19074-0 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

Preface

Within the last 20 years the database area has enriched computer science with a set of mutually related results of considerable practical importance and theoretical interest. This "database technology" covers a wide range of achievements, from conceptually rich and semantically well-defined abstract models down to the level of robust and efficient system implementations. Most of the basic issues are remarkably well understood within a framework of adequate and sound formalizations.

During these two decades database technology managed to develop appropriate solutions for a wide variety of commercial applications and its economic success can be taken as a proof of its relevance.

Currently, after a period of conceptual consolidation and implementational improvement, database technology is being challenged again. Pushed by the demands of novel application classes and new types of users, by the expanding complexity of data-based information systems and by the ever increasing amounts of data, as well as being pulled by new developments in hardware and systems architectures, database technology is having to adapt and extend rapidly.

Our intent in creating the EDBT Conference was to provide an international forum appropriate for the exchange of results in research, development, and applications which *extend* the scope of *database technology.* The EDBT Conference is designed to facilitate and stimulate communication among researchers and between academia and industry.

The scientific program of EDBT '88 flows in a single stream of presentations (a format we were close to giving up on when confronted with the overwhelming response of 168 submissions and only 27 slots available!). The papers originated from 28 countries and can be divided roughly into two broad areas. The figures in parentheses indicate the number of submissions which addressed this subject.

Extended Database Semantics (91):

- Complex Database Objects (19),
- Databases and Logic (30),
- Expert Systems and Databases (24),
- Extended Data Semantics and Data Types (18)

Extended Architectures and Systems Support (51):

- Transaction Models and Concurrency (16),
- Data Distribution (11),
- Efficient Data Access (17),
- Data Administration and Control (7).

Most of the remaining submissions concentrated on special database applications (22), in particular in the area of heterogeneous and multimedia databases.

In his invited paper Lucca Cardelli from DEC Systems Research, Palo Alto, presents "Type Systems for Data-Oriented Languages", a concern that is also addressed by the panel on the relationship between "Databases and Programming Languages: A Shotgun Marriage?"

An increasing amount of research and development in our field is done through international co-operation and within joint projects (ESPRIT, Alvey, EUREKA, various national and industrial programs). EDBT '88 reflects this aspect in specific project sessions based on submitted and reviewed short papers. One such session, chaired by Domenico Sacca (University of Calabria, Italy), presents projects that develop "Support for Data- and Knowledge-Based Applications". A second one, organized by Giuseppe Pelagatti (University of Brescia, Italy), concentrates on "Database Applications in Distributed Environments".

Two days of tutorials precede the Scientific Program. Research colleagues working in some of the most rapidly moving areas of database technology present the state-of-the-art in their particular field of interest:

Francois Bancilhon	Object-Oriented Databases
Carlo Batini, Dave Reiner	Database Design: Methodologies and Tools
Phil Bernstein	Transaction Processing Systems
Herve Gallaire	Logic and Databases
John Nestor	Database Technology for Software Engineering
David De Witt	Extensible Database Systems.

An Industrial Exhibition and a Book Fair running in parallel to EDBT '88 attracted the participation of about 25 different companies.

The program committee of EDBT '88 was chaired by Joachim W. Schmidt (Frankfurt University, FR Germany); the members were

S. Alagic (Yugoslavia)
A. Albano (Italy)
P. Apers (Netherlands)
M. Atkinson (G. Britain)
F. Bancilhon (France)
R. Bayer (FR Germany)
C. Beeri (Israel)
G. Bracchi (Italy)
M.L. Brodie (USA)
J. Bubenko (Sweden)
S. Ceri (Italy)
Q. Chen (PR China)
P. Dadam (FR Germany)
R. Demolombe (France)
D.J. De Witt (USA)
A. Furtado (Brazil)
G. Gardarin (France)
G. Gottlob (Austria)
L.A. Kalinichenko (USSR)
Y. Kambayashi (Japan)
H. Kangassalo (Finland)
M. Missikoff (Italy)
J. Mylopoulos (Canada)
E. Neuhold (FR Germany)
J.M. Nicolas (FR Germany)
A. Pirotte (Belgium)
A. Reuter (FR Germany)
G. Schlageter (FR Germany)
A. Sernadas (Portugal)
A. Solvberg (Norway)
N. Spyratos (France)
P. Stocker (G. Britain)
K. Subieta (Poland)
B. Thalheim (DR Germany)
D.C. Tsichritzis (Switzerland)
Y. Vassiliou (Greece)
G. Wiederhold (USA)
H. Williams (G. Britain)
C. Zaniolo (USA)
C.A. Zehnder (Switzerland)

We would like to give our sincere thanks to all program committee members as well as to all the other reviewers for their care in evaluating the submitted papers.

Our thanks are also extended to Professor U. Serafini, President of AICCRE, for his active role in promoting EDBT '88 within Europe and to Venetian authorities, and to Professor L. Bianco, Director of IASI-CNR, and the entire secretarial and technical staff of IASI-CNR for their confidence and support given to the conference right from the early stages of the organizational process.

We also acknowledge the continuous help of all the members of the organizational committee: P. Atzeni, A. D'Atri, H. Eckhardt, J. Elmore, G. Gardarin, K.G. Jeffrey, F.L. Ricci, G. Turco and C. Zaniolo.

Finally we would like to thank our secretaries and their assistants, A. Bambey, F. Fühler, U. Kasielke, M. Ritsert, I. Wetzel and A. Ziegler.

Springer-Verlag, in particular I. Mayer, was as always very helpful in preparing the proceedings.

EDBT '88 is being held on San Giorgio Island in Venice from March 14th to 18th, 1988. We would like to thank the Cini Foundation for the opportunity to profit from the stimulating and relaxing atmosphere of this unique setting.

Venice, March 1988

Joachim W. Schmidt (Program Committee Chairman)
Stefano Ceri (Conference Chairman)
Michele Missikoff (Organizing Committee Chairman)

Contents

Invited Paper

L. Cardelli
Types for Data-Oriented Languages 1

Databases and Logic

R. Krishnamurthy and C. Zaniolo
Optimization in a Logic Based Language for Knowledge and Data Intensive Applications 16

P.M.D. Gray, D.S. Moffat and N.W. Paton
A Prolog Interface to a Functional Data Model Database 34

J. Han, G. Qadah and C. Chaou
The Processing and Evaluation of Transitive Closure Queries 49

Expert System Approaches to Databases

A.M. Kotz, K.R. Dittrich and J.A. Mülle
Supporting Semantic Rules by a Generalized Event/Trigger Mechanism 76

M.C. Shan
Optimal Plan Search in a Rule-Based Query Optimizer 92

C. Cauvet, C. Proix and C. Rolland
Information Systems Design: An Expert System Approach 113

Distributed Databases and Transaction Management

C. Beeri, H.-J. Schek and G. Weikum
Multilevel Transaction Management, Theoretical Art or Practical Need? 134

E. Bertino and L.M. Haas
Views and Security in Distributed Database Management Systems 155

E. Bellcastro, A. Dutkowski, W. Kaminski, M. Kowalewski, C.L. Mallamaci, S. Mezyk, T. Mostardi, F.P. Scrocco, W. Staniszkis and G. Turco
An Overview of the Distributed Query System DQS 170

Database Administration

A.P. Sheth, J.A. Larson and E. Watkins
TAILOR, a Tool for Updating Views 190

S.J. Cammarata
An Intelligent Information Dictionary for Semantic Manipulation of Relational Databases 214

F. Rabitti, D. Woelk and W. Kim
A Model of Authorization for Object-Oriented and Semantic Databases 231

Complex Database Objects

D. Beech
A Foundation for Evolution from Relational to Object Databases 251

S. Abiteboul and S. Grumbach
COL: A Logic-Based Language for Complex Objects 271

M. Levene and G. Loizou
A Universal Relation Model for Nested Relations 294

Efficient Data Access

W. Litwin, D. Zegour and G. Levy
Multilevel Trie Hashing 309

A. Sikeler
VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes 336

A. Hutflesz, H.-W. Six and P. Widmayer
The Twin Grid File: A Nearly Space Optimal Index Structure 352

S.M. Chung and P.B. Berra
A Comparison of Concatenated and Superimposed Code Word Surrogate Files for Very Large Data/Knowledge Bases 364

G.Z. Qadah
Filter-Based Join Algorithms on Uniprocessor and Distributed-Memory Multiprocessor Database Machines 388

Efficiency by Replicated Data

A. Milo and O. Wolfson
Placement of Replicated Items in Distributed Databases 414

A. Kumar and A. Segev
Optimizing Voting-Type Algorithms for Replicated Data 428

R. Alonso, D. Barbara, H. Garcia-Molina and S. Abad
Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems 443

Data Types and Data Semantics

K.L. Chung, D. Rios-Zertuche, B.A. Nixon and J. Mylopoulos
Process Management and Assertion Enforcement for a Semantic Data Model . 469

F. Bry, H. Decker and R. Manthey
A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases 488

Special Data

R.H. Güting
Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems 506

N.A. Lorentzos and R.G. Johnson
An Extension of the Relational Model to Support Generic Intervals 528

Short Project Papers

J. Metthey and J. Cotta
ESPRIT: Trends and Challenges in Database Technology 543

Support for Data- and Knowledge-Based Applications

D. Saccà (Session Chairman)
Introduction 549

S. Ceri, S. Crespi Reghizzi, G. Gottlob, F. Lamperti, L. Lavazza, L. Tanca and R. Zicari
The ALGRES Project 551

C. Lécluse, P. Richard and F. Velez
O_2, an Object-Oriented Data Model 556

A.V. Zamulin
Database Programming Tools in the ATLANT Language 563

A. Albano, L. Alfò, S. Coluccini and R. Orsini
An Overview of Sidereus: A Graphical Database Schema Editor for GALILEO 567

DAIDA Team
Towards KBMS for Software Development: An Overview of the DAIDA Project 572

S. Bergamaschi, F. Bonfatti and C. Sartori
ENTITY-SITUATION: A Model for the Knowledge Representation Module of a KBMS 578

S. Himbaut
TELL-ME: A Natural Language Query System 583

Distributed Database Applications

G. Pelagatti (Session Chairman)
Introduction 588

P.M.G. Apers, M.L. Kersten and H.C.M. Oerlemans
PRISMA Database Machine: A Distributed, Main-Memory Approach 590

M. Driouche, Y. Gicquel, B. Kerherve, G. Le Gac, Y. Lepetit and G. Nicaud
SABRINA-RT, a Distributed DBMS for Telecommunications 594

D. Ellinghaus, M. Hallmann, B. Holtkamp and K.-D. Kreplin
A Multidatabase System for Transnational Accounting 600

E. Bertino, F. Rabitti and C. Thanos
MULTOS: A Document Server for Distributed Office Systems 606

W. Johannsen, W. Lamersdorf, K. Reinhardt and J.W. Schmidt
The DURESS Project: Extending Databases into an Open Systems Architecture 616

Types for Data-Oriented Languages
(Overview)

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

1. Introduction

By the term *data-oriented language*, I mean a language whose *main* concern is in the structuring and handling of data. For contrast, procedure-oriented and process-oriented languages are mostly concerned with expressing algorithms and protocols. Other terms, such as the much abused *object-oriented*, can be used to indicate an integration of the above features. A useful and complete language should certainly integrate all of the above "orientations", but here we will mostly focus on data structuring.

Data orientation has traditionally been the main characteristic of information systems, but has also played an increasingly important role in general-purpose programming languages. While the first programming languages were mostly algorithmic, with little emphasis on data structures, more recent languages have seen a relative standardization of control-flow features and a large emphasis and experimentation in data structuring, including the packaging of procedures into abstract data types, objects, and modules (see [Cardelli Wegner 85] for a tutorial and bibliography).

At the same time, information systems have evolved towards more expressive ways of modelling reality, which has meant more complex data models and more flexible and integrated query languages. Many people have noted that powerful query languages tightly coupled with complex data models have all the characteristics of programming languages, and have suggested there should be a systematic integration. This has resulted in the development of systems such as Pascal/R [Schmidt 77], Adaplex [Smith Fox Landers 83], PS-algol [PPRG 85], Taxis [Mylopoulos Bernstein Wong 80] and Galileo [Albano Cardelli Orsini 85].

We now have a sufficient number of examples to justify looking for a broad framework. In order to integrate information systems and programming languages, it is first necessary to unify the information-system concept of *data model* with the programming-language concept of *type system*. We argue that *type theory* (the formal study of type, or classification, systems) is the correct framework within which to study and carry out such unification. In this framework we can analyze existing integrated data-oriented systems and to design better ones.

A nice step in unifying data models and type systems was the introduction of *orthogonal persistence* of data [Atkinson Bailey Chisholm Cockshott Morrison 83], which bridges the gap

between ephemeral (programming-language) and persistent (information-system) data. But this is not sufficient; it has become increasingly evident that data models aim to be more expressive than ordinary type systems, and that the nature of information systems impose additional constraints such as the ability to evolve data schemas over time. Hence new concepts have to be developed, and a good general framework is required.

The prominent feature of data-oriented languages is the richness of their type structure. This may include various flavors of structured data (arrays, trees, sets, relations), abstract types, polymorphism, inheritance, computations over types, etc. In fact, the type structure may be so rich that the traditional distinction between values and types is insufficient to characterize it completely.

We shall talk about various uses of *kinds* [McCracken 79] which are the "types" of types, to organize such rich type structures. The main contention of this paper is that kind structures should be of benefit in understanding and developing data-oriented languages.

Richness of type structure seems necessary for world modeling, which is process of formalizing (pieces of) reality or abstract concepts. World modeling has conflicting goals: expressiveness is the most desirable feature, but it has to coexist with reliability, by which we mean the ability to check and maintain world models mechanically (through static typechecking) so that programs can be written and maintained reliably, and also with efficiency, which means that type structures must be easily typecheckable and should facilitate efficient computation.

If we wanted to emphasize expressiveness only, we would probably choose something like set theory as our description system; but this is not efficiently typecheckable and has no obvious relation to efficient computation. Efficiency, on the other hand, has been amply emphasized in the past, generally to the detriment of expressiveness. Here we mainly focus on reliability and, while keeping it fixed, strive for expressiveness with an eye on efficiency.

2. Polymorphism

Our aim is to design statically typed languages with much of the flexibility of untyped languages. We want the rigor of static typing for reliability and efficiency. We want the flexibility of untyped languages for expressiveness. The compromise is a difficult one; some dynamic typechecking may be ultimately required, but we aim to make it as rare as possible. If we can make typing largely static, then we will have largely reached our goals of reliability and efficiency.

A good combination of expressiveness and reliability is reached through various forms of *polymorphism* (sometimes called *genericity*) which is the ability of typed programs to operate on data of different, but related, types. (The ability to operate on data of unrelated types involves *overloading* and *coercions*, and will not be discussed here.)

To make the discussion more concrete, we introduce a simple untyped language, which we later extend and use as the basis of typed languages. Here we use x for variables, k for built-in

constants (e.g. numbers and operations), t for tags (e.g. record field names), and a, b, and c for terms.

	introduction	*elimination*
variables	x	
constants	k	
functions	fun(x) b	b(a)
pairs	⟪a,b⟫	lft(c) rht(c)
records	$(t_1=a_1, \ldots, t_n=a_n)$	c.t
variants	$[\![t=a]\!]$	case c of $[\![t_1=x_1]\!]\, b_1 \ldots [\![t_n=x_n]\!]\, b_n$
recursion	rec(x) a	

Under the *introduction* heading we list ways of building (introducing) entities: we have constructors for functions, pairs, records (which are unordered tagged tuples of values) and variants (tagged values). The *elimination* column lists ways of using and decomposing (eliminating) entities: application uses functions, left and right projections decompose pairs, field selection decomposes records, and the case statement analyzes a variant according to its tag (binding the appropriate variable x_i to the contents of the variant in the body b_i). Recursion can be eliminated by expanding a recursive definition.

Characteristic of untyped languages is that all the constructs described in the elimination column may *fail* during computation. Application fails if a non-function is applied; projections fail if a non-pair is decomposed, etc. These possible failures make untyped or dynamically typed programs unreliable: even if programs are carefully coded the first time, later changes may require manual inspection of the entire program to check against failure points; this is an unreliable process and the quality of software degrades.

In order to statically trap failure points, we want to impose a type system on our untyped language. We can do this in many different ways, but any static type system restricts the class of programs that can be written, and hence reduces the expressiveness of the language. The goal is then to preserve as much flexibility as possible through polymorphism. There are two main situations:

- In the untyped language, some functions can be applied to objects of different types without ever failing, since they do not examine objects too closely: this phenomenon is called *parametric polymorphism* (since it can be modeled in typed languages by type parameters):

  (fun(x) x) (3) (fun(x) x) (true)

  Here we have the identity function (which does not examine its argument, but simply returns it) applied to an integer and a boolean. Other parametric polymorphic functions are the length-of-a-list function, which does not examine the list elements but just counts them, and sorting functions, which do not examine the list elements except to compare them.

• In the untyped language, a function that can be applied to a record without failing can also be applied to any record having additional fields, because the additional fields will be ignored. This is called *subtype polymorphism* (since we can define a subtype relation on record types, based on the number, tags, and types of record fields):

(fun(x) x.t) (⦇t=3⦈) (fun(x) x.t) (⦇t=3, u=true⦈)

Here we have a function extracting the t component of a record, which works on any record having at least a t component. A similar kind of subtype polymorphism can be detected in variants: a function correctly handling three classes of differently tagged variant objects, will not fail if only two classes of differently tagged variant objects ever occur.

Many type systems have been devised to deal with polymorphism. In the next section we examine some of the options.

3. Levels

We now examine a number of type systems that can be imposed on our untyped language, from the most restrictive to the most flexible ones. To get a glimpse of their features, we show how to define the identity function in each one of them.

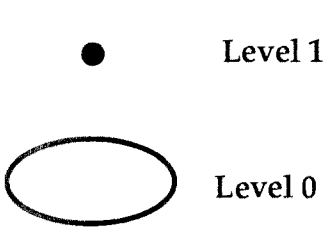
These type systems differ in the number of *levels* they require to be described. Intuitively, values (i.e. the entities expressible in the untyped language) are all lumped together in a set of all values at *Level 0*. At *Level 1* we have types, intended as sets of values, and type operators, intended as functions mapping types to types (e.g. the List type operator is a function mapping the type Int to the type of lists of integer). These two levels are sufficient for most ordinary languages, but we will add *Level 2*, the level of *kinds*. Kinds are the "types" of types and operators; intuitively they are either sets of types or sets of operators.

Types classify data and computations, and kinds classify types and operators. Just as we have values and value computations at Level 0, that are classified by Level 1 entities (types), we also have types and type computations at Level 1 that are classified by Level 2 entities (kinds). We use the notation a:A to indicate that value a has type A, and the notation A::K to indicate that type A has kind K. We use lower case identifiers for Level 0, capitalized identifiers for Level 1, and all caps for Level 2.

We display the level structure of the various type systems by simple *level diagrams*, based on the intuition of types as sets of values and kinds as sets of types and operators.

One type

Our initial untyped language can be described by the following level diagram. There is a uniform set of values, and since there are no type distinctions we can imagine that there is a single type, the type of all values, called Value. Hence the type level is collapsed to a single point.

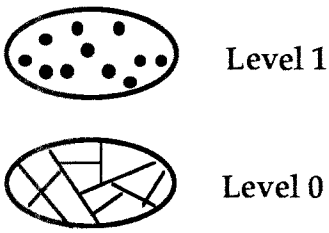



```
def id: Value =
    fun(x:Value) x
id(3)
```

In this system, the identity function is defined and used as shown above; definitions have the general form "def x:A = a". (We could have simply written fun(x)x, as in the untyped language, since all variables have the same type.)

Many types

We now introduce type distinctions: our value set is partitioned into distinct regions of booleans, integers, boolean functions, integer functions, and so on, each identified by a distinct type (we drop the universal Value type). All program variables must now be annotated with their types. We obtain a *monomorphic* type system (each value having a single type) not much different from Pascal's.

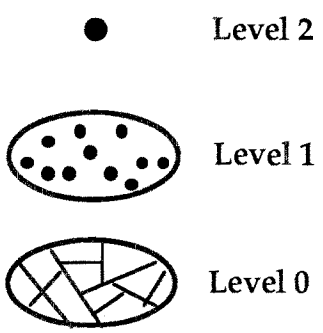



```
def id: Int→Int =
    fun(x:Int) x
id(3)
```

The (integer) identity function is now strictly typed, and we have lost the ability to express polymorphic functions of any kind: for example, the boolean identity must be written separately. We have however gained a static type system, in which run-time failures can be detected at compile-time.

One kind

We are now ready to introduce our first kind: the kind of all types, called TYPE. In this system we can introduce variables ranging over types, and we can classify them by giving them the kind TYPE.



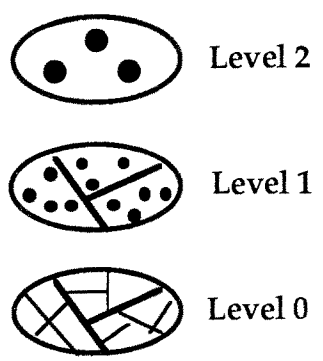

```
def id: All[A::TYPE] A→A =
    fun[A::TYPE] fun(x:A) x
id[Int](3)
id[Bool](true)
```

The above is the polymorphic identity function and two different uses of it (we use round brackets for entities at Level 0 and square brackets for entities at Level 1). We have a function taking a type as an argument, and a corresponding application of a type to such function. The type of a polymorphic function has the form All[A::TYPE]B; this is the type of a function taking a type A and returning a value in B. (We could have simply written fun[A] fun(x:A) x, since all type variables have the same kind.)

We now have a language supporting parametric polymorphism, through the notion of abstraction over type variables which can be instantiated at different types.

Many kinds

Finally, we can introduce more kinds. For example we can introduce all the kinds of the form K⇒L, for kinds K and L; TYPE⇒TYPE is then the kind of all one-argument type operators (such as List). Level 1 acquires type operators, in addition to types, and it is partitioned by the various kinds existing at Level 2 (note that type operators are functions, not types: there is no value whose type is a type operator).



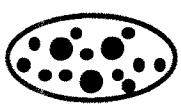
```
Def Endo:: TYPE⇒TYPE =
    Fun[A::TYPE] A→A
def id: All[A::TYPE] Endo[A] =
    fun[A::TYPE] fun(x:A) x
id[Int](3)
```

Here Endo(-morphism) is a type operator which given a type A returns the type of functions from A to A; its kind is TYPE⇒TYPE. The polymorphic identity is a function which given a type A returns an endomorphism on A, in particular the identity over A.

Several other classes of kinds can be introduced, and we shall see later on.

Kinds are types (Type:Type)

We should mention some "degenerate" type systems, which are obtained by collapsing the level structure we have just set up. We can collapse kinds and types by the simple assumption the there is a type of all types, including itself. We then lose the distinction between kinds and types (and also our syntactic distinction between "::" and ":", between square brackets and round brackets, etc.).



Levels 1,2

Level 0

```
def Endo: Type→Type =
    fun(A:Type) A→A
def id: All(A:Type) Endo(A) =
    fun(A:Type) fun(x:A) x
id(Int)(3)
```

The problem with this collapsed system is that, although we still have static typechecking, we lose the ability to perform what we might call *static levelchecking*. That is, it is not possible to determine the proper level of certain expressions. For example, in the expression:

```
fun(A:Type) fun(x:A) x
```

is A a Level 2 or a Level 1 entity? And is x a Level 1 or a Level 0 entity? This cannot be determined because both the following applications are legal (the latter uses Type:Type):

```
id (Int) (3)          id (Type) (Int)
```

Lack of level distinctions causes problems in compilation, since compilation strategies are generally based on the idea the Level 0 entities are evaluated at *run-time*, and all the other levels are processed at *compile-time*.

Types are values

Finally, we can collapse all three levels, obtaining something similar to an untyped language. The difference is that there are now types and kinds in the value domain, which can be manipulated. Smalltalk, with its notions of classes and metaclasses, has some of these features.

Levels 0, 1, 2

```
def Endo: Type→Type =
    fun(A:Type) A→A
def id : All(A:Type) Endo(A) =
    fun(A:Type) fun(x:A) x
id(Int)(3)
```

In spite of all the type information we lose static typing; dubious programs such as id(3)(4), and failing programs such as 3(4), are now statically legal.

4. A three-level language

In the rest of the paper, we adopt the many-kinds (three levels) option, which seems to be a natural extrapolation of language evolution. The first procedural languages only had a fixed number of types. Recent languages have a multiplicity of types, and user-definable types. More advanced, polymorphic, languages have (often implicitly) a kind of all types, type operators, and other non-trivial level structures. Future languages will have a richer kind structure, with user-definable kinds.

Level 0: values

In this subsection we sketch the Level 0 (values) of a three-level data-oriented language. Basically, we have typed versions of the values in our initial untyped language, and we also add a few more.

We have basic data values such as "ok" (a trivial value), booleans, characters, strings, integers and reals.

We have records $\langle t_1=a_1, \ldots, t_n=a_n \rangle$ with field selection. Record fields are unordered, must have disjoint tags, and can store arbitrary values, including functions.

We have variants, $[\![t=a]\!]$ with a case statements. Enumerations are a special case of variants, when the data contents are trivial (e.g. "ok") and only the tags matter.

We have typed higher-order functions. These include ordinary functions taking values as arguments and returning values as results, such as

```
def succ = fun(x:Int) x+1                succ(3)
```

and polymorphic functions, taking types as arguments, and returning values as results:

```
def id = fun[A::TYPE] fun(x:A) x                id[Int](3)
```

Note that polymorphic functions inhabit Level 0, although they take types as arguments.

We have tuples, including both tuples of values, such as ⟪3,4⟫ and mixed tuples of types and values, such as ⟪Int, 0, succ⟫. In the latter case the type components of a tuple can only be handled in a limited way, so that these tuple represents elements of abstract types [Mitchell Plotkin 85].

We have sets {a, b, c} and set operations. Sets of records are relations, which admit generalized relational algebra operations [Buneman Ohori 87].

A value b=ref(c) is a modifiable reference to a value c; it can be dereferenced by deref(b) and assigned by b := c'.

Recursion is used to define recursive functions and other recursive values, such as data structures containing loops.

Exceptions can also be seen as special values which can be "raised" and "trapped" to interrupt and resume the normal flow of control.

Level 1: types and operators

At Level 1 we have the types of Level 0 entities, and operators among Level 1 entities. We also introduce a reflexive and transitive relation of *subtyping*, denoted by A<:B (A is a subtype of B) for types A and B. Subtyping is intuitively understood as set inclusion between types: if A<:B then any value of type A also has type B.

In correspondence with the basic values, we have the basic types Ok (whose only value is "ok"), Bool, Char, String, Int, Real, etc. There are no non-trivial subtyping relations among basic types.

A record type $\{t_1:A_1, \ldots, t_n:A_n\}$ is the type of a record value $\{t_1=a_1, \ldots, t_n=a_n\}$ if A_i is the type of a_i, for all i. Record types are identified up to reordering of their fields. Two record types are in the subtype relation A<:B, if all the tags of B appear in A, and the corresponding types are in the subtype relation. This subtyping relation between record types models some forms of multiple inheritance [Cardelli 84].

A variant type $[\![t_1:A_1, \ldots, t_n:A_n]\!]$ is the type of a variant value $[\![t=a]\!]$ if there is an index i such that $t=t_i$ and $a:A_i$. Variant types are identified up to reordering of their fields. Two variant types are in the subtype relation A<:B if the tags of B include the tags of A, and the corresponding types are in the subtyping relation.

The type of a function fun(x:A)b is A→B, if b has type B. The type of a polymorphic function fun[X::K]b is All[X::K]B, if b has type B (where B may have occurrences of X). Subtyping of function spaces is given by the following rule: A→B <: A'→B' if A' <: A (note the inversion) and B<:B'. Similarly, All[X::K]B <: All[X::K']B' if K'<::K (where <:: denotes a *subkind* relation, discussed later), and B<:B' under the assumption that X::K'.

The type of a pair ⟪a,b⟫ is the cartesian product A×B, for a:A and b:B. The types of mixed tuples of types and values (abstract types) are discussed in [Mitchell Plotkin 85].

Set types $\{A_1, \ldots, A_n\}$, relation types, and relation algebra operators are discussed in detail in [Ohori 87]. For set types, A<:B if for each A_i in A there is a B_j in B with $A_i <: B_j$.

The type of an assignable object ref(a) is Ref(A), if a:A. The subtyping rule for Ref types cannot be simple subtyping of the domain types, because unsafe programs could then be written. Hence we require that Ref(A)<:Ref(A') if A<:A' and A'<:A.

Recursive types and operators are supported by the construct Rec(X::K)A, which we can use to define list types, tree types, etc. Subtyping of recursive types is determined by expanding the recursive definitions.

The most unusual feature of Level 1 is the presence of type operators, which perform computations over types in much the same way functions perform computations over values. The syntax is also very similar:

```
Fun[X::K]B          A[B]
```

The first expression denotes an operator which given an entity of kind K (e.g. a type if K=TYPE) returns the Level 1 entity B (e.g. a type, or another operator). The second expression denotes the application of an operator to a Level 1 object (e.g. a type, or another operator). We can then define, for example, an operator List such that List[Int] is the type of integer lists, and an operator Tree such that Tree[Int][Bool] is the type of binary trees with integer leaves and boolean nodes.

Since operators represent almost-ordinary computations (but one level "up"), we might want to write programs such as:

```
Def F = Fun[X::BOOL] if X then Int else Bool end
F[True]
```

but what are "True" and "BOOL" here? This "True" inhabits Level 1, hence it is different from the "true" value. We can think of "true" as a run-time boolean, and "True" as a compile-time boolean; "True" is a *lifted* version of the value "true". Similarly the conditional in the example is a lifted version of the conditional at the value level. Then the "BOOL" kind is a *lifted type* (from Level 1 to Level 2). The idea of lifted values and types may seem strange, but in a sense it is unavoidable: it can be shown that once operators are allowed, lifted booleans, integers, etc. can be defined purely in terms of higher-order operators. An important restriction is that all computations at Level 1 must be side-effect free, to preserve sound typing.

Level 2: kinds

The main novelty in this presentation is the richness of structure at the kind level. Most languages have no kinds at all, and even polymorphic languages may have only one: the kind of all types. Here we introduce many more classes of kinds.

TYPE is the kind of all types, i.e. it is the "type" of basic types, record types, variant types, function types, tuple types, etc. Its presence means that variables of kind TYPE, ranging over all types, can be introduced.

As we have seen in the previous section, we can introduce a kind BOOL, which is a lifted