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.

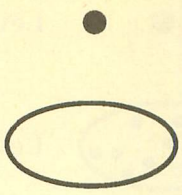
These type systems differ in the number of *levels* they require to be described. Intuitively, values (i.e. the entities expressible in the untyped language) are all lumped together in a set of all values at *Level 0*. At *Level 1* we have types, intended as sets of values, and type operators, intended 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.

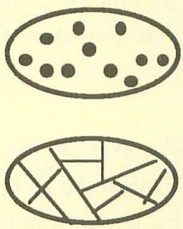



```
def id: Value =
    fun(x:Value) x
id(3)
```

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.

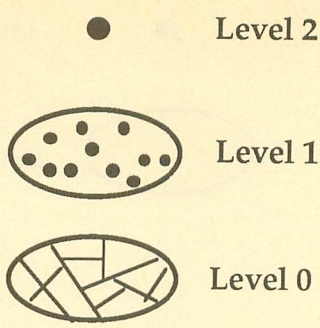



```
def id: Int→Int =
    fun(x:Int) x
id(3)
```

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.



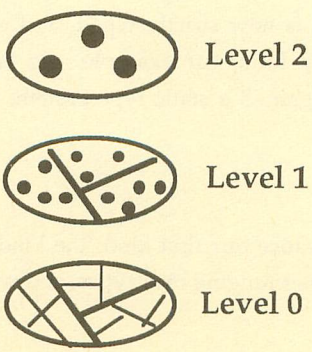

```
def id: All[A::TYPE] A→A =
    fun[A::TYPE] fun(x:A) x
id[Int](3)
id[Bool](true)
```

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).



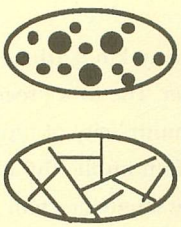
```
Def Endo:: TYPE⇒TYPE =
    Fun[A::TYPE] A→A
def id: All[A::TYPE] Endo[A] =
    fun[A::TYPE] fun(x:A) x
id[Int](3)
```

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.).




```
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
```

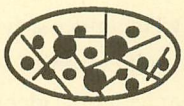
is A a Level 2 or a Level 1 entity? And is x a Level 1 or a Level 0 entity? This cannot be determined because both the following applications are legal (the latter uses Type:Type):

```
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 $(\!|t_1=a_1, \ldots, t_n=a_n|\!)$ 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

version of the Bool type, together with the Level 1 entities True::BOOL, False::BOOL, which are lifted versions of Level 0 booleans, and with the relevant operators such as conditional. Similarly for the INT kind, etc. This can provide a rich computational languages for defining complex type structures at compile-time.

ALL[A::K]L is the kind of operators from kind K to kind L, normally written K⇒L if A does not occur in L. For example, TYPE⇒TYPE is the kind of unary type operators, such as List, and TYPE⇒(TYPE⇒TYPE) is the kind of binary type operators, such as the function space operator "→". These kinds provide the mechanism for performing *static kindchecking* of type computations involving type operators. The kindchecking rules for operators are analogous to the typechecking rules for functions.

The idea of subtyping was introduced case by case in the previous section for each type construction, but it can also be captured uniformly by a new class of kinds. We can define POWER[A] to be the kind of all the subtypes of type A [Cardelli 88]. Hence B::POWER[A] means that B is a subtype of A. By introducing the binding fun[A<:B] as an abbreviation of fun[A::POWER[B]] (and similarly for All[A<:B]), we can express subtype polymorphism very precisely, including dependencies between input and output subtypes:

```
def id: All[A<:B] A→A =
    fun[A<:B] fun(x:A) x
```

Here id is the polymorphic identity restricted to the subtypes of some type B. Given a subtype A of B and an object of that subtype, it returns an object of that same subtype.

A very speculative but interesting possibility is to introduce the equivalent of Ref types at the kind level. Suppose we indicate by MODIF the kind of modifiable types, i.e. types which may change in time, and by B::MODIF=Modif[Int] a modifiable type which is currently bound to Int (with the rule that Modif[B] has kind MODIF for any type B). Then the value b:B=modif(3) would be correctly typed at the present moment (with the rule that modif(b) has type Modif[B] if b has type B), but we risk breaking the type system by later assigning B := Bool (a Level 1 assignment) and extracting an integer as a boolean out of b.

This problem explains why type computations should generally be side-effect free, but the above situation is actually quite common in module systems, where version stamps are used to keep track of interfaces (types) which have changed, and of modules (values) which have to be reexamined. Hence we may imagine that modif(3) is a version-stamped integer, whose version stamp is invalidated when the assignment B := Bool is performed. Any following attempt to access the integer inside modif(3) will result in checking its version stamp against the current version stamp, and failing.

If we have subtypes, we must also have subkinds, because the subtype relation A<:B induces a natural reflexive and transitive subkind relation (indicated by "<::"), given by POWER[A]<::POWER[B]. More interestingly, POWER[A]<::TYPE must hold, since any collection of types (here, the subtypes of A) must be included in the collection of all types.

Some applications

Since kinds are collections of types, we can imagine introducing kinds such as RECORD, the collection of all record types, SET, the collection of all set types, RELATION, the collection of all set-of-record types, etc. In fact, we already have enough machinery to define them without taking them as primitives. For example, all record types are subtypes of the empty record type ⦇⦈, hence we can define RECORD = POWER[⦇⦈]:

```
Def Record::TYPE = ⦇⦈
DEF RECORD = POWER[Record]
```

We can now exhibit a kind for an "And" operator which takes two record types and returns a record type which contains the fields of both (we can assume that this operation fails or returns a trivial type if corresponding field types do not match):

```
And :: RECORD ⇒ RECORD ⇒ RECORD
⦇t=3, u=true⦈ : And[⦇t:Int⦈][⦇u:Bool⦈]
```

Although this operator (and others to follow) cannot be defined within the language, the fact that we can express its kind means that we can combine it freely with other operators.

Similarly, we can define relation kinds, together with natural join operations for relations (as defined in [Buneman Ohori 87]):

```
Def Relation::TYPE = {Record}
DEF RELATION = POWER[Relation]

Join :: RELATION ⇒ RELATION ⇒ RELATION
join : ALL[A::RELATION] ALL[B::RELATION] (A×B) →. Join[A][B]
```

The "Join" type operator produces the result type of natural join operations; it is similar to "And", but works on sets of records (relations). The "join" operation is parameterized by two relation types A and B, and for each pair of relations it produces a relation (the natural join of the two) whose type is Join[A][B]. For example:

```
def r : Join[{⦇t:Int, u:Bool⦈}][{⦇u:Bool, v:String⦈}] =
    join[{⦇t:Int, u:Bool⦈}][{⦇u:Bool, v:String⦈}]
        (<{⦇t=3, u=true⦈}, {⦇u=true, v="xx"⦈}>)
    ≡ {⦇t=3, u=true, v="xx"⦈} : {⦇t:Int, u:Bool, v:String⦈}
```

Finally, we can view a database as a set of relations, and we can exhibit the kinds and

types of database merge operators which perform simple set-theoretical unions of relations (database merge is actually more complex; this covers only one of the necessary steps):

```
Def DataBase::TYPE = {Relation}
DEF DATABASE = POWER[DataBase]

Merge :: DATABASE ⇒ DATABASE ⇒ DATABASE
merge : ALL[A::DATABASE] ALL[B::DATABASE] (A×B) → Merge[A][B]

def d : Merge[{{(t:Int, u:Bool)}}][{{(u:Bool, v:String)}}] =
    merge[{{(t:Int, u:Bool)}}][{{(u:Bool, v:String)}}]
        (<{{(t=3, u=true)}}, {{(u=true, v="xx")}}>)
    ≡ {{(t=3,u=true)}}, {(u=true,v="xx")}} : {{(t:Int,u:Bool)}}, {(u:Bool,v:String)}}
```

Here we have merged two simple databases containing one relation each, where each relation contains a single record (see [Ohori 87] for the precise type rules).

5. Formalization

Type systems can be precisely described as formal systems based on *judgements*. A judgement is a relation between environments and expressions, inductively defined by axioms and inference rules; for example: "in environment E, term a has type A" is a judgement.

Typical judgements found in most type systems, and in the system informally described in this paper, include the following. Here E is an *environment* specifying types for variables (of the form "x:A") and kinds for type variables (of the form "X::K"); "E ⊢ ..." reads "Given the environment E we can establish that ...":

⊢ E env	E is a legal environment
E ⊢ K kind	K is a kind
E ⊢ A :: K	A has kind K
E ⊢ A type	A is a type (same as E ⊢ A::TYPE)
E ⊢ a : A	a has type A
E ⊢ K = L	K and L are equivalent kinds
E ⊢ A = B	A and B are equivalent types
E ⊢ K <:: L	K is a subkind of L
E ⊢ A <: B	A is a subtype of B (same as E ⊢ A::POWER(B))

Most type constructions are characterized by rules of Formation, Introduction, Elimination, and Subtyping. For example, the usual typechecking rules for functions can be expressed as follows (where "E, x:A" is the environment E extended with the assumption "x:A", and the

horizontal line reads "implies"):

Formation

$$\frac{E \vdash A \text{ type} \quad E \vdash B \text{ type}}{E \vdash A \to B \text{ type}}$$

Introduction

$$\frac{E, x{:}A \vdash b{:}B}{E \vdash \text{fun}(x{:}A)\, b : A \to B}$$

Elimination

$$\frac{E \vdash b{:}A \to B \quad E \vdash a{:}A}{E \vdash b(a) : B}$$

Subtyping

$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \to B <: A' \to B'}$$

In such a formalized framework, we say that a term b has a type B if it is possible to infer $E \vdash b{:}B$ according to the axioms and inference rules (given some environment E providing declarations for the free variables of b and B). See [Cardelli 88] for an example of a complete inference system.

6. Conclusions

Data-oriented languages may benefit from a rich kind structure. We have shown that kinds can provides a framework for relational and database-wide operations, for subtype relations, for schema computations, and perhaps even for schema evolution.

References

[Albano Cardelli Orsini 85] A.Albano, L.Cardelli, R.Orsini: **Galileo: a strongly typed, interactive conceptual language**, *Transactions on Database Systems*, June 1985, 10(2), pp. 230-260.

[Atkinson Bailey Chisholm Cockshott Morrison 83] M.P.Atkinson, P.J.Bailey, K.J.Chisholm, W.P.Cockshott, R.Morrison: **An approach to persistent programming**, *Computer Journal* 26(4), November 1983.

[Buneman Ohori 87] P.Buneman, A.Ohori: **Using powerdomains to generalize relational databases**, submitted for publication.

[Cardelli 84] L.Cardelli: **A semantics of multiple inheritance**, in *Semantics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Cardelli 88] L.Cardelli: **Structural subtyping and the notion of power type**, *Proc. POPL 1988.*

[Cardelli Wegner 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, *Computing Surveys*, Vol 17 n. 4, pp 471-522, December 1985.

[McCracken 79] N.McCracken: **An investigation of a programming language with a polymorphic type structure**, Ph.D. Thesis, Syracuse University, June 1979.

[Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, *Proc. POPL 1985.*

[Mylopoulos Bernstein Wong 80] J.Mylopoulos, P.A.Bernstein, H.K.T.Wong: **A language facility for designing database intensive applications**, *ACM Transactions on Database Systems* 5(2), June 1980.

[Ohori 87] A. Ohori: **Orderings and types in databases**, *Proc. of the Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[PPRG 85] Persistent Programming Research Group: **The PS-algol reference manual - second edition**, *Technical Report PPR-12-85, University of Glasgow, Dept. of Computing Science*, Glasgow G12 8QQ, Scotland, 1985.

[Smith Fox Landers 83] J.M.Smith, S.Fox, T.Landers: **Adaplex: rationale and reference manual**, second edition, *Computer Corporation of America*, Four Cambridge Center, Cambridge, Mass. 02142, 1983.

[Schmidt 77] J.W.Schmidt: **Some high level language constructs for data of type relation**, *ACM Transaction on Database Systems* 2(3), pp. 247-281, September 1977.

Optimization in a Logic Based Language for Knowledge and Data Intensive Applications

Ravi Krishnamurthy *Carlo Zaniolo*

MCC, 3500 Balcones Center Dr., Austin, TX, 78759

Abstract

This paper describes the optimization approach taken to ensure the safe and efficient execution of applications written in LDL, which is a declarative language based on Horn Clause Logic and intended for data intensive and knowledge based applications. In order to generalize the strategy successfully used in relational database systems we first characterize the optimization problem in terms of its execution space, cost functions and search algorithm. Then we extend this framework to deal with rules, complex terms, recursion and various problems resulting from the richer expressive power of Logic. Among these is the termination problem (safety), whereby an unsafe execution is treated as an extreme case of poor execution.

1. Introduction

The Logic Data Language LDL, combines the expressive power of a high level logic based language (such as Prolog) with the non-navigational style of relational query languages, where the user need only supply a correct query, and the system is expected to devise an efficient execution strategy for it. Consequently, the query optimizer is given the responsibility of choosing an optimal execution --a function similar to that of an optimizer in a relational database system. A relational system uses knowledge of storage structures, information about database statistics and various estimates to predict the cost of execution schemes chosen from a pre-defined search space and to select a minimum cost execution in such a space.

A LDL system offers to a user all the benefits of a database language --including the elimination of the impedance mismatch between the language and the query language-- in addition, its rule based deductive capability and its unification-based pattern matching capability make it very suitable for knowledge based and symbolic applications. The power of LDL is not without a cost, since its implementation poses non trivial compilation and optimization problems. The various compilation techniques used for LDL were described in [BMSU85, SZ86, Za85, ZS87]. This paper concentrates on the optimization problem; i.e., the problem of devising an efficient execution strategy for the given query. The termination problem (safety) is also tackled in this context, since the lack of termination can be viewed as an extreme case of poor termination. Therefore, our optimization problem revisits the well-known problem of control in logic programs as per Kowalski's famous equation Algorithm = Logic + Control [Kw 79]. Several approaches to this arduous problems have been proposed in the past.