A deductive and typed object-oriented language

René Bal and Herman Balsters*

Computer Science Department, University of Twente,
P.O. Box 217, 7500 AE Enschede,
The Netherlands

Abstract. In this paper we introduce a logical query language extended with object-oriented typing facilities. This language, called DTL (from DataTypeLog), can be seen as an extension of Datalog equipped with complex objects, object identities, and multiple inheritance based on Cardelli type theory. The language also incorporates a very general notion of sets as first-class objects. The paper offers a formal description of DTL, as well as a denotational semantics for DTL programs.

Keywords: Query languages, object-oriented databases, inheritance, type theory, resolution, denotational semantics.

1 Introduction and results

In the last decade, the merge of object-oriented programming with object-oriented data structuring principles has lead to a rapid increasement of new developments in the field of databases and logical languages. Object-oriented databases have the advantages of a clean conceptual design as well as the possibility of enforcing better software engineering. Systems equipped with subtyping facilities, such as the Cardelli object-oriented type system (cf. [Card88]), offer a concise and clear way to deal with (multiple) inheritance. Inheritance is a very powerful modelling tool and forms the backbone of many object-oriented data models. Also the availability of complex objects, such as records, lists, variants, and sets offer a wide range of expressiveness. Examples of data models with facilities as mentioned above are O_2 ([LéRi89]), Iris ([LyVi87]) and Machiavelli ([OhBB89]).

Object-orientation has also not left the field of logical languages untouched. Languages like LIFE ([Aït-K91]), F-Logic ([KiLW90]), and [BrLM90,IbCu90, McCa92,MoPo90] are examples of such languages that make extensive use of object-oriented principles to enhance the field of logic programming with the expressiveness and concise modelling possibilities, typical for the object-oriented paradigm. Especially the use of subtyping makes logic programs more structured and easier to understand. In short, the combination of logic programming and object orientation is very promising.

Relational databases and logic programming have been combined resulting in so-called deductive databases. Deductive databases highlight the ability to use a

* Our E-mail addresses are, resp.: rene@cs.utwente.nl, balsters@cs.utwente.nl

logic programming style for expressing deductions concerning the contents of the database. Examples of such languages are Datalog [CeGT90] and LDL [NaTs89]. These languages have gained considerable popularity due to the ease in which it is possible to specify very complex queries.

Recently, research interest has started to arise in the combination of object orientation, databases and logical languages. For example, research initiatives have been started aimed at extending Datalog with object-oriented concepts. In such extensions, the logical component is used to specify the schema of the database, and a distinction is made between base relations and derived relations. Examples of such systems are LOGRES [CCCT90], and Complex Datalog [GrLR92]. Other examples are object-oriented logical languages primarily used for the querying of object-oriented databases, such as [AbKa89,Abit90,AbGr88] and LLO [LoOz91].

The language described in this paper is called DTL, which stands for DataType-Log. DTL is designed as a query language for a database specified in a language called TM. TM ([BaBZ93]) is a high-level specification language for object-oriented database schemas, and has all the facilities that one would expect from a state-of-the-art object-oriented data model. The main novelties of the TM language are the incorporation of predicative sets as first-class objects, and the possibility of defining static constraints of different granularity (i.e. at the object level, class level, and database level), and this in the context of multiple inheritance and full static typecheckability.

In DTL we have taken an approach which sometimes differs considerably from existing object-oriented query languages. For example, answers to DTL queries result in a set of homogeneous elements, in the sense that these elements all have the same so-called minimal type w.r.t. subtyping. This means that if a query asks for `persons`, then the answer should consist of `persons` and not, for example, also specializations of `persons`, say `employees`. This approach differs from the one followed by F-logic and LIFE, where basically there is no distinction made between types and instantiations of types. Our approach is also different from the one followed by IQL, ILOG ([HuYo90]), LOGRES and LLO, where the answer results in a collection of object identifiers. The object identifiers in these cases are related to o-values by means of an o-value assignment (to employ terminology taken from IQL), and in this way the answers actually include specializations of the requested original type.

As mentioned before, DTL incorporates general set constructions, including predicatively defined sets, as first-class objects; i.e. such sets are actual terms in the language. Languages like LOGRES, IQL and LLO also support sets, be it that these sets are restricted to enumerated sets as actual terms in the language. A distinct feature of DTL is its powerful usage of combining predicates and (multiple) inheritance; languages like LLO, LIFE and F-logic also offer a notion of inheritance, but in combination with predicates the version in DTL is less restricted. Yet another feature of DTL is the possibility to navigate freely through the terms by successive projection on attribute components. In other systems like LOGRES, Complex Datalog, IQL and LLO such a navigation is

also possible, be it that the process of navigation in these languages is more complicated than in DTL .

The rest of the paper is organized as follows. We first give an impression of the TM datamodel and offer an example of a TM database specification. After that we will offer an introduction to the DTL language. In section 3 we shall give a more thorough account of DTL including matters concerning typing and the combination of using types and predicates in programs. In section 4 we will give a denotational semantics for DTL programs, and we end with some conclusions and suggestions for future research.

2 The datamodel TM

DTL is meant as a well-founded query language for TM, and within this paper we will only discuss those aspects of TM which are relevant for DTL. For more details on TM the reader is referred to [BaBB92,BaBZ93,BaBV92].

The TM language is a high-level object-oriented datamodel that has been developed at the University of Twente in cooperation with the Politecnico di Milano. The TM language is designed for describing conceptual schemas of object-oriented databases. The TM language contains all of the elements that one would expect from a state-of-the-art object-oriented model, but with important new features, namely the incorporation of

1. predicative descriptions of sets (predicative sets as complex values)
2. static constraints of different granularity (object level, class level, database level)

The strength of TM stems from its richness as a specification language and its formal, type-theoretic background. The TM language is founded in FM, which is based on a typed lambda calculus extended with logic and sets. The subtyping is based on the ideas of the Cardelli type system [Card88], which has been given a set-theoretical semantics in [BaFo91,BaVr91]. TM has complex objects formed from arbitrarily nested records, variant records, sets, and lists. Furthermore, TM is equipped with object identity, multiple inheritance, methods and method inheritance, and this in the context of full static typecheckability. Classes in TM specifications have an extension in the database; the prefix `with extension` in a TM Class declaration is followed by the name of the class extension in the database. For more details on TM and its relation to other object-oriented database languages, we refer to [BaBZ93].

Example 1. An example of a database specification within TM.

```
Class Person with extension PERS
  attributes
      name   : string
      age    : integer
      spouse : Person
```

```
      gender : string
  object constraints
      c1 : gender="Male"  or gender="Female"
      c2 : spouse·spouse=self
  class constraints
  key name end  Person

Class Employee ISA Person with extension EMP
  attributes
      colleagues : ℙEmployee
      salary     : ⟨m_salary:real⟩
  object constraints
      c3 : salary·m_salary ≥ 3000 and salary·m_salary ≤ 10000
end  Employee

Class Manager ISA Employee with extension MAN
  attributes
      salary     : ⟨m_salary:real, r_expenses:real⟩
      department : string
end  Manager

Class Secretary ISA Employee with extension SEC
  attributes
      boss : Manager
end  Secretary
```

In the example above, a generalization hierarchy is defined for `Persons`, `Employees`, `Managers` and `Secretaries`. This generalization hierarchy is defined by means of the statement `C` **ISA** `C'`, occurring in the head of a class definition. It means that for every object e occurring in the extension of `C`, there is an object e' occurring in `C'` such that e' is a generalization of e; hence, the extension of `C` is a specialized subset of the extension of `C'`. In our example this means that generalizations of `Managers` occurring in the extension `MAN` also occur as `Employee` in `EMP`; .i.e., the extension `MAN` contains specializations of a subset of `EMP`.

In TM, objects have an object identifier used for referential integrity, for sharing and for implementation of recursive data structures. They are, however, not directly visible in TM, although there are operations to inspect the value of the object identifiers. As already stated, TM is formally founded in FM. In FM the object identifier is just a label of a record expression. For example, the FM representation of a `Person` is `⟨id:oid, name:string, age:int, spouse:oid, gender:string⟩`, along with certain additional constraints ensuring that the oid-values correctly refer to their corresponding objects. For example, to enforce that the spouse of a `Person` corresponds to some `Person`, we shall add at the database level the following referential integrity constraint

$$\forall y \in PERS\ \exists x \in PERS\ \ y{\cdot}spouse = x{\cdot}\mathtt{id}$$

For more details on such a translation from TM specifications to its FM-counterpart, we refer to [BaBZ93].

DTL is a user language which can be considered as a sugared version the formal language FDTL. Before presenting the formal language we first give a few example queries specified in DTL. The examples throughout this paper are all related to the TM specification in example 1

Example 2. Give all Employees in the database, which have a spouse who is the Secretary of a Manager earning more than 8000, and this Manager is to be a colleague of the Employee in question.

This query could easily be translated to the following DTL specification, where we use the predicate symbol p for the specification of the required Employees

```
p(X) ← EMP(X), Y isa X·spouse, SEC(Y),
       Y·boss·salary·m_salary ≥ 8000, Y·boss in X·colleagues.

? p(X(Employee)).
```

Here the predicates EMP and SEC are used to denote that the variables X and Y reside in the extensions EMP and SEC, respectively. Furthermore, since the types of these extensions are known, the types of the variables occurring within the definition of the predicate can be omitted. In the formal language FDTL, we will use a special predicate, the **db**-predicate, to express that an expression denotes an object residing in the database. By using the **db**-predicate, explicit usage of extension names will not be necessary in FDTL programs.

The dot notation, as usual, denotes record projection; hence, X·spouse denotes a Person object being a spouse of Employee X. Another interesting predicate used in example 2 is the **isa** -predicate; this predicate is used to compare specializations with corresponding generalizations. Hence, the predicate Y(Secretary) **isa** X(Employee)·spouse, informally, evaluates to true if Y is indeed a specialization of the spouse of Employee X.

The query of example 2 could also be defined by making use of more than one predicate. First we introduce a predicate which defines a relation between Secretaries and Managers earning over 8000, after which we could use the predicate in the body of a rule instantiated by arguments of a super- or subtype. In the example below, the predicate is instantiated by a Person and an Employee, both of which have types that are supertypes of the original types Secretary and Manager.

```
secr_wp_man(X(Secretary), Y(Manager)) ← SEC(X), X·boss = Y,
                                        Y·salary·m_salary ≥ 8000.

p(X) ← secr_wp_man(X·spouse, Y(Employee)), Y in X·colleagues,
       EMP(X).

? p(X(Employee)).
```

Analogously, we can use this kind of predicate inheritance to ask for all `Secretaries` which fulfill the requirement `p`. The query then becomes

```
? p(X), SEC(X).
```

If we want all `Persons` satisfying the predicate `p`, the query becomes

```
? p(X(Person)).
```

The instantiation of a predicate employing specialization or generalization is treated in more detail below.

3 The language FDTL

DTL is meant as a typed logical query language for TM which is able to deal with subtyping and inheritance. In the previous section a few example DTL queries were presented. These queries could easily be translated to the formal language FDTL treated below. The object identifiers which are invisible within DTL are used explicitly within FDTL. Furthermore, we have no classes in FDTL, but only types; hence, FDTL is defined on the FM representation of the database. After the definition of the language FDTL is given, we will discuss its semantics informally.

3.1 The definition of FDTL

FDTL supports arbitrarily nested records, variant records, sets and lists. Within this paper, however, we will only deal with records and sets for reasons of a clean exposition. Formal definitions of the full language can be found in [Bal92].

The Types. We assume that the basic types are in a postulated set B. This set contains, among others, the standard types `bool`,`int`,`real`,`string`,`char` and `oid`. The subtype relation defined on $B \times B$ is the identity, i.e. we have no subtype relation between different basic types, since this will lead to problems related to resolution in FDTL programs. We furthermore assume that we have a set of labels L, totally ordered and with lower bound `id` (such an ordering of labels enforces a canonical form for records and record types). We let a vary over L.

Definition 1. The set T (of *types*) is defined as follows

1. $\tau \in T$, whenever $\tau \in B$
2. $\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle \in T$, whenever $a_i \in L$, $\tau_i \in T$ $(1 \leq i \leq m)$, $a_1 < a_2 < \ldots < a_m$ and $m \geq 0$.
3. $\mathbb{P}\tau \in T$, whenever $\tau \in T$

We let ρ, σ and τ over T.

We distinguish two subsets of the set of types. These are the *object types* and the *ordinary types*. The object types T_{obj} are all record types for which the first label is `id : oid`. The ordinary, or non-object, types T_{nor} are all expressions which do not contain any component of type `oid`.

Subtyping. The subtyping relation is defined, conform the well-known Cardelli type theory [CaWe85,Card84,Card88], on the set of object types and on the set of ordinary types, and is extended to the set of all types. The reason for this approach is that we do not want an object type to be a subtype of an ordinary type.

Definition 2. The relation $\leq$ on $T \times T$ is defined by induction as follows

1. $\beta \leq \beta$, whenever $\beta \in B$
2. $\langle a_1 : \sigma_1, \ldots, a_m : \sigma_m \rangle \leq \langle a_{j_1} : \tau_{j_1}, \ldots, a_{j_n} : \tau_{j_n} \rangle$, whenever $j_1, \ldots, j_n$ is a (not necessarily contiguous) sub-sequence of $1, \ldots, m$, $\sigma_{j_i} \leq \tau_{j_i}$ $(1 \leq i \leq n)$ and $a_1 = \mathtt{id}$ iff $a_{j_1} = \mathtt{id}$
3. $\mathbb{P}\sigma \leq \mathbb{P}\tau$, whenever $\sigma \leq \tau$

Example 3. We have ⟨age:int, name:string, address:string⟩ $\leq$ ⟨age:int, name:string⟩ since the former type has all properties of the latter type, but also an extra property, namely the additional address field. However ⟨id:`oid`, name:string, address:string⟩ $\not\leq$ ⟨age:int, name:string⟩ since the former type is an object type, while the latter is an ordinary type.

The Terms. The terms of FDTL are very similar to the expressions defined in TM and FM. We have constants, variables, records, variants, lists and sets. As explained earlier, we will not deal with lists and variant records within this paper, and furthermore, we will also not deal with aggregate operations defined on sets. The operations which we will discuss in this paper are the projection operation defined on records and the usual operations defined on sets.

For each $\tau \in T$ let C_τ be a (possibly empty) set (of constants), mutually disjoint. We let c_τ vary over C_τ. $C_{\mathtt{bool}}$=`{true,false}`. Furthermore, for each $\tau \in T$ let X_τ be a set (of variables), mutually disjoint, countably infinite and disjoint from the sets C_σ $(\sigma \in T)$. We let $\mathtt{X}(\tau)$ vary over X_τ.

Definition 3. The set E (of *terms*) is defined inductively as follows

1. $c_\tau \in E$, whenever $\tau \in T$, $c_\tau \in C_\tau$
2. $\mathtt{X}(\tau) \in E$, whenever $\tau \in T$, $\mathtt{X}(\tau) \in X_\tau$
3. $\langle a_1 = t_1, \ldots, a_m = t_m \rangle \in E$, whenever $a_i \in L$, $t_i \in E$ $(1 \leq i \leq m)$ and $a_1 < a_2 < \ldots < a_m$ and $m \geq 0$
4. $t{\cdot}a \in E$, whenever $t \in E$, $a \in L$
5. $\{t_1, \ldots, t_m\} \in E$, whenever $t_i \in E$ $(1 \leq i \leq m)$ and $m \geq 0$
6. t_1 *set_op* $t_2 \in E$, whenever $t_1, t_2 \in E$ and *set_op* $\in$ { **union** , **intersect** , **minus** }

We let t vary over E.

The typing rules are often defined by means of minimal typing, since expressions, i.e. terms, can have more than one type, due to the subtyping environment[2].

[2] This kind of type polymorphism is obtained by the following rule

$$t : \sigma, \quad \sigma \leq \tau \Rightarrow t : \tau$$

Fortunately, every term also has a unique minimal type denoted by "::". Our definition of minimal typing [BaFo91] is basically the same as the one given in [Reyn85], and satisfies the following important properties: soundness ($t :: \sigma \Rightarrow t : \sigma$), completeness ($t : \tau \Rightarrow t :: \sigma$, for some $\sigma \in T$), and minimality ($t : \tau\ ,\ t :: \sigma \Rightarrow \sigma \leq \tau$)

Definition 4. The typing rules for terms

1. $\dfrac{c_\tau \in C_\tau}{c_\tau :: \tau}$ 2. $\dfrac{\mathtt{X}(\tau) \in X_\tau}{\mathtt{X}(\tau) :: \tau}$

3. $\dfrac{t_i :: \tau_i\ (1 \leq i \leq m)}{\langle a_1 = t_1, \ldots, a_m = t_m \rangle :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle}$

4. $\dfrac{t :: \langle a_1 : \tau_1, \ldots, t_m : \tau_m \rangle}{(t \cdot a_j) :: \tau_j} (1 \leq j \leq m)$ 5. $\dfrac{t_i :: \tau\ (1 \leq i \leq m)}{\{t_1, \ldots, t_m\} :: \mathbb{P}\tau}$

6. $\dfrac{t_1 :: \mathbb{P}\tau \quad t_2 :: \mathbb{P}\tau}{(t_1\ set\_op\ t_2) :: \mathbb{P}\tau}\ set\_op \in \{$ **union** , **intersect** , **minus** $\}$

Let E^* denote the set of all well-typed terms. $E^* \subseteq E$

If σ is a type then $\mathbb{P}\sigma$ denotes the powertype of σ. Intuitively, a powertype $\mathbb{P}\sigma$ denotes the collection of all sets of terms t of type σ. Note that the semantics of a powertype as well as elements thereof can be infinite, depending on the specific underlying type. The powertype constructor resembles the construction of the powerset $\mathcal{P}(V)$ of a set V in ordinary set theory. A term t in our language is called a *set* if it has a powertype as its type; i.e., $t : \mathbb{P}\sigma$, for some type σ. We stress here that a set in our theory is a *term* and not a type; i.e. we add to the set of types special types called powertypes, and, in addition, we add to the set of terms special terms called sets.

The typing rules 5 and 6 concerning set expressions are a direct consequence of the typing rules for equality, discussed in subsection 3.1. Therefore, we will defer the treatment of these rules until the typing rule for equality is discussed.

A subset of the well-typed terms are the basic terms E_B. This subset consists of all terms not containing operations, i.e. such terms contain only constants, variables and enumerated sets.

The operation Var is defined on all terms and returns the set of variables occurring within a particular term.

The Atoms. For each $n \in N$ let $Pred_n$ be a set (of *n-ary predicate symbols*) mutually disjoint, countably infinite and disjoint from the sets of constants and the sets of variables. We let p and q vary over $Pred_n$.

Definition 5. The set Atm (of *atomic formulas*) is defined as follows

- If p is an n-ary predicate symbol and $t_1 \ldots t_n \in E^*$, then $p(t_1 \ldots t_n)$ is an *atomic formula*, or more simply, an *atom*.

We let A vary over Atm.

We distinguish two kinds of predicates, the *ordinary* predicates and the *built-in* predicates. In addition to the standard built-in predicates which are also used within Datalog [CeGT90], FDTL also has a database predicate, an **isa** -predicate, a membership predicate, and a subset predicate. The database predicate, **db**, is used to denote that objects are taken from the database. Furthermore, since the typing rules for the equality predicate are very severe , an **isa** -predicate is defined to allow for a more liberal comparison of specialized expressions with generalized ones. Similar rules are offered for the membership predicate and subset predicate defined for sets; i.e. we have a strict form, and a more liberal form dealing with specializations and generalizations.

The reasons for adopting strict typing rules for the equality predicate, as well as for typing of object-oriented sets in our theory, are rather technical and are explained in detail in [BaVr91]. Informally, however, one could say that if two terms are to be equal, then they should be equal in *all* aspects; i.e., they should also have exactly the same typing possibilities in the context of subtyping. This leads us to our typing rule for equality of two terms: the predicate $t_1 = t_2$ is correctly typed and of type `bool`, iff t_1, t_2 have exactly the same typing possibilities; i.e., t_1 and t_2 have the same *minimal* type.

For sets we also make a distinction between severe and more liberal typing rules, similar to the situation with the equality predicate. For example, we have a strict form of set membership (**in**) stating that a term is to be *exactly equal* to some element of a set, and we have a more liberal form (**sin**) stating that a term is to be equal to a *specialization* of some element of a set. Again, the reader is referred to [BaVr91] for more details.

Definition 6. The built-in predicates $Atm_B \subset Atm$ and their typing rules are defined as follows

1. $\dfrac{t :: \sigma \quad \sigma \in T_{obj}}{\mathbf{db(t)}}$
2. $\dfrac{t_1 :: \sigma \quad t_2 :: \sigma \quad \sigma \in \{\mathtt{int}, \mathtt{real}, \mathtt{string}, \mathtt{char}\}}{t_1 \ op \ t_2 \quad op \in \{<, \leq, >, \geq\}}$
3. $\dfrac{t_1 :: \sigma \quad t_2 :: \sigma}{t_1 \ op \ t_2} op \in \{=, \neq\}$
4. $\dfrac{t_1 :: \sigma \quad t_2 :: \tau \quad \sigma \leq \tau}{t_1 \ \mathbf{isa} \ t_2}$
5. $\dfrac{t_1 :: \sigma \quad t_2 :: \mathbb{P}\sigma}{t_1 \ \mathbf{in} \ t_2}$
6. $\dfrac{t_1 :: \sigma \quad t_2 :: \mathbb{P}\tau \quad \sigma \leq \tau}{t_1 \ \mathbf{sin} \ t_2}$
7. $\dfrac{t_1 :: \mathbb{P}\sigma \quad t_2 :: \mathbb{P}\sigma}{t_1 \ \mathbf{subset} \ t_2}$
8. $\dfrac{t_1 :: \mathbb{P}\sigma \quad t_2 :: \mathbb{P}\tau \quad \sigma \leq \tau}{t_1 \ \mathbf{ssubset} \ t_2}$

Let Atm^* denote the set of well-typed atoms. $Atm^* \subseteq Atm$.

Example 4. The **isa** -predicate has the possibility to compare specializations with generalizations. The atom ⟨name="Mary", age=18⟩ **isa** ⟨name="Mary"⟩ is *true*, because the first argument is a specialization of the second argument. Analogously, the **sin** -predicate could be used to check if there exists a generalization of the expression occurring on the left-hand side, which occurs in the

set expression on the right-hand side, for instance ⟨name="Mary", age=18⟩ **sin** {⟨name="Mary"⟩, ⟨name="Jane"⟩}. Analogously, the **ssubset** -predicate could be used for sets; for example, EMP **ssubset** PERS, in example 1.

A FDTL **Program.** A FDTL program is a sequence of Horn clauses.

Definition 7. A Horn clause H is of the form $A_0 \leftarrow A_1, \ldots A_n.$, where $A_i \in Atm^*$ and $A_0 \notin Atm_B (0 \leq i \leq n)$. The variables appearing within a Horn clause are assumed to be universally quantified. There are two notions defined on H: $Head(H) = A_0$ and $Body(H) = A_1, \ldots A_n$.

As usual, the scope of a variable is the Horn clause in which it appears. We will assume that each occurrence of a variable within a Horn clause has the same type; the type of the variable is therefore only stated once. Allowing occurrences of the variables to have different types does not have any effect on the expressiveness of FDTL; it does, however, have a serious effect on the semantics and the resolution of FDTL-programs (and on the readability of programs).

A set of Horn clauses forms a program, albeit not necessarily a correct program. In the next subsection constraints are defined for correct programs, as well as the (albeit informal) meaning of a correct program. For this reason, some additional definitions are presented below.

Definition 8. A *program* P is a set of Horn clauses (sometimes Horn clauses are also called clauses or rules). We let r vary over the rules in a program P.

Definition 9. The set of Horn clauses in a program P with the same predicate symbol q in the head is called the *definition* of q.
$Def_P(q) = \{r \in P \mid \text{predicate symbol of } Head(r) \text{ is } q\}$

Definition 10. Let A be an atom, then the type of A is defined by

- $Type(A) = (\tau_1, \ldots, \tau_n)$, whenever $A = p(t_1, \ldots, t_n)$ and $t_i :: \tau_i$ $(1 \leq i \leq n)$.

The subtyping relation is extended in a straightforward manner to this Cartesian product construct.

Definition 11. Let P be a program and r be a rule occurring in $Def_P(q)$, then we say predicate q is *associated* with $Type(Head(r))$.

3.2 The meaning and correctness of an FDTL program

Due to typing and subtyping, it turns out that not every set of Horn clauses forms a correct program. Within this section we work towards the definition of a correct program, and we shall also provide for an informal semantics of correct programs. We shall start with a simple example and from then on work towards more complex situations.

First consider a situation where the arguments of a predicate all just have a basic type. As already stated there is no subtype relationship between different basic types, which means that for predicates having only arguments of a basic type, there is no possibility to use *predicate inheritance*. By predicate inheritance we mean that a predicate originally associated with a specific type, can also be used in the body of a rule by instantiating that predicate with specialized or generalized expressions w.r.t. the original type. We assume that a predicate is associated with one unique Cartesian product type. Later on we shall also experiment with a more liberal rule; it will turn out, however, that this more liberal rule gives rise to unexpected query results. This means that predicates associated with basic types can only be used by instantiating arguments of the same type as those for which the predicate was originally defined.

A more complex situation occurs when the arguments of predicates are associated with object types. In contrast to the situation sketched above, where only basic types play a rôle, concepts like inheritance are a major issue.

There are several ways to integrate inheritance into FDTL. One way to try to model inheritance, is to also put all specializations of a particular atom in the Herbrand model; i.e. if $p(e)$ is true and e' is a specialization of e, then $p(e')$ is also true. Hence, if $p(e)$ is in the model of an FDTL-program and e' **isa** e, then $p(e')$ occurs also in the model of the FDTL-program. This seems a rather natural rule; if a predicate is valid for a specific relation it is also valid for a specialization of this relation. Such an incorporation of specializations into the model, however, can lead to technical problems as illustrated in the following example

`paid_6000(X(employee))` $\leftarrow$ `X·salary` $=$ `⟨m_salary=6000⟩.`

Here, it is not possible to instantiate the variable `X(employee)` by an expression e of type `manager`, because then the equality predicate becomes incorrectly typed, since `managers` are specialized on the salary attribute. Therefore this way to integrate inheritance is not suitable for FDTL.

Another attempt at modelling predicate inheritance in FDTL is to have a more elaborate type system for variables. Instead of defining exactly the minimal type of the expressions which could be used to instantiate the variable, only an upperbound is specified. For example, if $p(e)$ is a fact and e `::employee`, then it is possible to query the predicate p by `? p(X(person))`. The informal semantics of such a query is: Give all instances for `X` having type `person` and that satisfy condition p. Hence, the term e is a correct instantiation of the variable `X` having upperbound `person`. This technique is used in, for example Login, F-logic and LLO. In our theory, based on Cardelli subtyping extended with set constructs, such an approach will give rise to answers of queries consisting of a set of terms that are not necessary all equipped with the same minimal type. Such sets of heterogeneously typed elements, however, lead to inconsistencies (cf. [BaVr91]), which makes such a liberal approach using upperbounds unfit for our purposes.

We have therefore chosen for another approach, which is also more powerful than the approaches sketched above. First let us consider how we could specify

predicate inheritance in an explicit manner. Assume that a predicate is defined for a specific record type, then this predicate could be used to define predicates or terms having a sub- or supertype by using the built-in **isa** -predicate, as shown below.

```
well_paid_emp(X(employee)) ← ..., X·salary·m_salary ≥ 6000.

well_paid_man(Y(manager)) ← Y isa X(employee),
                            well_paid_emp(X), ....

well_paid_pers(Z(person)) ← X(employee) isa Z,
                            well_paid_emp(X), ....
```

It is clear that such situations may occur often and therefore we would like to integrate this kind of inheritance within FDTL without having to explicitly specify **isa** -predicate instantiations. We will now explain how this can be achieved.

Suppose that we have the following rule (pertaining to example 1)

```
well_paid(X(employee)) ← db(X(employee)),
                         X·salary·m_salary > 6000.
```

Informally, the predicate `well_paid` defines a set of `employees` occurring in the database which also have a monthly salary greater than 6000. Analogous to a typed logical language without subtyping it is possible to query this simple program by means of a goal where the predicate is equipped with a variable which has the same type as for which the predicate was originally defined. However, by using predicate inheritance we would also like to ask for well_paid `managers`, for example:

```
? well_paid(X(manager)).
```

This query can now be seen as shorthand for the query below, where only substitutions for the variable `X(manager)` are taken into consideration.

```
? well_paid(Y(employee)), X(manager) isa Y(employee).
```

The substitutions for `Y(employee)` could be eliminated by employing some additional dummy predicate. In this case, the query `? well_paid(X(manager))` can be seen as shorthand for

```
dummy(X(manager)) ← well_paid(Y(employee)),
                    X(manager) isa Y(employee).

? dummy(X(manager)).
```

It should be noted that the goal `? dummy(X(manager))` would result in infinitely many substitutions for the variable `X(manager)`, since the only requirement stated is that the `manager` occurs as an `employee` in the database. This will give rise to infinitely many substitutions for the department attribute, since this attribute is not available for `employees`. In order to make this query safe, the variable `X(manager)` should be bound in some manner. This can be done by means of another predicate, for example a database predicate as in

```
? well_paid(X(manager)), db(X(manager)).
```

This query has the same meaning as

```
dummy(X(manager)) ← well_paid(Y(employee)),
                    X(manager) isa Y(employee).

? dummy(X(manager)), db(X(manager)).
```

The answer to this query results in the set of managers occurring in the database which, as an employee, earn more than 6000 (it is assumed that we have, conceptually, a fully replicated database, i.e. all managers also occur as an employee in the database).

The same strategy as described above could also be used to obtain generalizations, as in

```
? well_paid(X(person)).
```

This query is then shorthand for

```
dummy(X(person)) ← well_paid(Y(employee)),
                   Y(employee) isa X(person).

? dummy(X(person)).
```

Hence, analogous to specializations, this technique can be extended to generalizations. The meaning of such a query is defined by considering these queries as shorthand for a query invoking a dummy predicate. Predicates are not only used within queries, they could also be used within bodies of rules defining other predicates. In that case there is no need for an additional dummy predicate; instead the body of the defined predicate can be extended with an extra **isa** -predicate.

Consider the following example using the well_paid predicate (defined for employees)

```
satisfied(X(person)) ← well_paid(X(person)), ....
```

Analogous to queries, this rule can be regarded as shorthand for

```
satisfied(X(person)) ← well_paid(Y(employee)),
                       Y(employee) isa X(person), ....
```

We assumed earlier on that predicates were only allowed to be defined by using one fixed type declaration; i.e. even if a predicate is defined by different rules in some programs, it is defined by using the same type declaration in all of those rules. We will now clarify in more detail why we have chosen for this assumption. Consider the situation where we have more than one rule defining a predicate, and that the predicate is defined for more than one Cartesian product type, as in

```
well_paid(X(employee)) ← db(X(employee)),
                         X·salary·m_salary > 6000.
```

```
well_paid(X(manager)) ← db(X(manager)),
                        X·salary·m_salary > 9000.
```

In the case of the following query

```
? well_paid(X(manager)).
```

we unfortunately not only get managers which are well paid as a manager and earn more than 9000 as answer, but also all managers which are well paid as an employee and earn more than 6000! The reason for this is that the query is actually shorthand for

```
dummy(X(manager)) ← well_paid(X(manager)).

dummy(X(manager)) ← well_paid(Y(employee)),
                    X(manager) isa Y(employee).

? dummy(X(manager)).
```

which will result in an answer being against our intuition. What we would expect are all well paid managers; i.e. well paid as a manager, and not the managers which are well paid as an employee. It is, however, impossible to query the predicate associated with one particular type, i.e. it is not possible to ask for well paid managers, which are solely well paid as a manager. In particular, if a predicate is defined by multiple rules and having different types, then this will often lead to unexpected results. Furthermore, apart from this problem, there exists also another problem of a more formal nature. A query results in a set of answers, and in our object-oriented theory dealing with sets, each set will contain elements all of the same minimal type. If we now allow a predicate to be defined for more then one Cartesian product type, then by querying such a predicate we will obtain a set of answers with different minimal types. This is therefore yet another reason to not allow for predicates defined by multiple rules having different Cartesian product types associated to them.

Let us now consider predicates associated to ordinary types. The interesting thing is that the strategy developed above for the treatment of predicates associated to object types is equally applicable to predicates associated to ordinary (i.e. non-object) types. For example, consider the following program

```
p(⟨name="Paul", age=20⟩).

p(⟨name="Eric", age=20⟩).
```

and the query

```
? p(X(⟨age:int⟩)).
```

This query can be regarded as shorthand for

```
dummy(X(⟨age:int⟩)) ← p(Y(⟨name:string, age:int⟩)), Y isa X.
? dummy(X(⟨age:int⟩)).
```

Hence, we have defined a simple, but powerful mechanism to use predicate inheritance within an object oriented logical query language. The constraints specified for correct programs can now be defined more formally.

Definition 12 Type correctness of predicate definitions. Let q be an ordinary predicate defined in an FDTL program P. Program P should then satisfy the following requirement

$$\forall r, r' \in Def_P(q) \;\; Type(Head(r)) = Type(Head(r'))$$

Definition 13 Type correctness for usage of predicates. Let q be an ordinary predicate defined in an FDTL program P, and let A be a literal with predicate symbol q. Program P should then satisfy the following requirement

$$\begin{aligned}&\forall r \in P \;\; (A \in Body(r) \Rightarrow \exists r' \in Def_P(q)\\ &(Type(A) \leq Type(Head(r')) \vee Type(Head(r')) \leq Type(A)))\end{aligned}$$

Definition 14. Let P be a correct program, and let q be an ordinary predicate defined in P. The type of q in P, conform definition 3.15, is then defined by

$$Typ_P(q) = Type(Head(r)), \quad \text{where } r \in Def_P(q)$$

4 A semantics of FDTL

In the previous section it has been demonstrated how inheritance of predicates can be integrated within FDTL by employing the **isa** -predicate. Furthermore it has been discussed how, by using shorthand notation, the **isa** -predicate could be omitted in some cases. We shall define a transformation Tf_P which translates a correct program P into a program P' where all abbreviations are removed and inheritance is made explicit by means of **isa** -predicates. For such an explicit FDTL-program we shall provide proper semantics.

Definition 15. The transformation Tf_P, which transforms a correct program P into a program P', is defined as follows

Tf_P =**while** $\exists A_i \in Body(r)$, $r \in P$, $A_i = q(t_1, \ldots, t_n)$, q is an ordinary predicate symbol, $Typ_P(q) = (\tau_1, \ldots, \tau_n)$ **and** $\exists t_j \;\; (1 \leq j \leq n) \;\; t_j :: \sigma$, $\sigma \neq \tau_j$

do

replace A_i **by:** $q(t_1, \ldots, t_{j-1}, \mathtt{X}(\tau_j), t_{j+1}, \ldots, t_n)$ **where** $\mathtt{X}(\tau_j) \notin Var(r)$;

if $\tau_j \leq \sigma$

then add: t_j **isa** $\mathtt{X}(\tau_j)$ to the body of r

else add: $\mathtt{X}(\tau_j)$ **isa** t_j to the body of r

The semantics of an FDTL program should provide a link between FM and FDTL. In this way, in contrast to other logical languages, we define an actual denotational semantics for terms in FDTL. This denotational semantics is used to obtain a typed Herbrand model of an FDTL program. The definitions of the denotational semantics are derived from [BaFo91]

Postulation 16. For $\beta \in B$, let $[\![\beta]\!]$ be a non-empty set. Let $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$

Definition 17. For each $\tau \in T$, a set $[\![\tau]\!]$ is defined by induction on the structure of τ as follows

1. $[\![\beta]\!]$ is postulated
2. $[\![\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle]\!] = \{(a_i, d_i) \mid 1 \leq i \leq m \ \wedge \ d_i \in [\![\tau_i]\!]\}$
3. $[\![\mathbb{P}\tau]\!] = \mathcal{P}([\![\tau]\!])$, where $\mathcal{P}$ denotes the power set operator

We let d vary over any $[\![\tau]\!]$

Definition 18. $U = \bigcup_{\tau \in T} [\![\tau]\!]$, the universe in which the semantics of both types and expressions will find their place.

The subtyping relation and a conversion function on the semantics is now defined formally as follows.

Definition 19. For each pair $\sigma, \tau \in T$ with $\sigma \leq \tau$ we define a function $cv_{\sigma \leq \tau} \in [\![\sigma]\!] \rightarrow [\![\tau]\!]$ by induction as follows

1. For each $\beta \in B$:
   - $cv_{\beta \leq \beta} = identity_\beta \in [\![\beta]\!] \rightarrow [\![\beta]\!]$
2. Let $\sigma = \langle a_1 : \sigma_1, \ldots, a_m : \sigma_m \rangle$ and $\tau = \langle a_{j_1} : \tau_{j_1}, \ldots, a_{j_m} : \tau_{j_m} \rangle$; if $j_1, \ldots, j_n$ is a (not necessarily contiguous) sub-sequence of $1, \ldots, m$ and $\sigma_{j_i} \leq \tau_{j_i}$ $(1 \leq i \leq n)$ then:
   - $cv_{\sigma \leq \tau}(\{(a_i, d_i) \mid (1 \leq i \leq m\}) = \{(a_{j_i}, cv_{\sigma_{j_i} \leq \tau_{j_i}}(d_{j_i})) \mid 1 \leq i \leq n\}$
3. Let $\sigma = \mathbb{P}\sigma'$ and $\tau = \mathbb{P}\tau'$; if $\sigma' \leq \tau'$ then:
   - $cv_{\sigma \leq \tau}(S) = \{cv_{\sigma' \leq \tau'}(d) \mid d \in S\}$ for $S \in [\![\sigma]\!]$

Definition 20. A database state db is a record $\langle a_1 = \{t_{1_1}, \ldots, t_{1_{n_1}}\}, \ldots, a_m = \{t_{m_1}, \ldots, t_{m_{n_m}}\}\rangle$, where each label a_i $(1 \leq i \leq m)$ corresponds to the extension name of a table, and each t_{i_j} $(1 \leq j \leq n_i)$ $(1 \leq i \leq m)$ denotes an object occurring in the extension a_i.

Example 5. The database of example 1 has minimal type $\langle$`PERS:`$\mathbb{P}$`person, EMP:`$\mathbb{P}$`employee, MAN:`$\mathbb{P}$`manager, SEC:`$\mathbb{P}$`secretary`$\rangle$. For more details we refer to [BaBZ93].

The *typed Herbrand interpretation* I for an FDTL program P belonging to database state db is given by the following:

1. The domain of I is the universe U
2. An assignment of an element d of U, to each ground term g in P.
3. An assignment of an n-ary predicate p_I, to each n-ary relation $< t_1, \ldots, t_n >$ in P.

Definition 21. An *assignment* A is a family of functions $A_\tau \in X_\tau \rightarrow [\![\tau]\!]$, $(\tau \in T)$. For assignment A, $\tau \in T$, $\mathtt{X}(\tau) \in X_\tau$, $d \in [\![\tau]\!]$ we define the assignment $A[x \mapsto d]$ for all $\sigma \in T$, $\mathtt{Y}(\sigma) \in X_\sigma$ by

$$(A[x \mapsto d])_\sigma(\mathtt{Y}(\sigma)) = A_\sigma(\mathtt{Y}(\sigma)) \text{ , if } \sigma \neq \tau \text{ or } \mathtt{Y}(\sigma) \neq \mathtt{X}(\tau)$$
$$= \quad d \quad \text{ , if } \sigma = \tau \text{ and } \mathtt{Y}(\sigma) = \mathtt{X}(\tau)$$

Definition 22 Minimal semantics. Let A be an assignment. A partial function $[\![]\!]^*_A \in E \hookrightarrow U$ is defined as follows by induction on the derivation of the minimal type of its argument

1. $[\![c]\!]^*_A = [\![c]\!]$, whenever $c \in C_\tau$
2. $[\![\mathtt{X}(\tau)]\!]^*_A = A_\tau(\mathtt{X}(\tau))$, whenever $\mathtt{X}(\tau) \in X_\tau$
3. $[\![\langle a_1 = t_1, \ldots, a_m = t_m \rangle]\!]^*_A = \{(a_i, [\![t_i]\!]^*_A) \mid 1 \leq i \leq m\}$, whenever $t_i :: \tau_i$
4. $[\![t{\cdot}a]\!]^*_A = f(a)$, where $f = [\![t]\!]^*_A$, whenever $t :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$, $a = a_j$ for some j, $1 \leq j \leq m$
5. $[\![\{t_1, \ldots, t_m\}]\!]^*_A = \{[\![t_1]\!]^*_A, \ldots, [\![t_m]\!]^*_A\}$, whenever $t_i :: \tau$ $(1 \leq i \leq m)$
6. $[\![t_1 \textbf{ union } t_2]\!]^*_A = [\![t_1]\!]^*_A \cup [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \mathbb{P}\tau$
7. $[\![t_1 \textbf{ intersect } t_2]\!]^*_A = [\![t_1]\!]^*_A \cap [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \mathbb{P}\tau$
8. $[\![t_1 \textbf{ minus } t_2]\!]^*_A = [\![t_1]\!]^*_A \setminus [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \mathbb{P}\tau$

Definition 23. The semantics of a Cartesian product type (which is used for the typing of predicates) is defined as follows

$$[\![(\tau_1 \ldots, \tau_n)]\!]^*_A = \{(d_1, \ldots, d_n) \mid d_i \in [\![\tau_i]\!]^*_A \ (1 \leq i \leq n)\}$$

For a predicate $q \in P$, this yields (cf. definition 14):

$$[\![q]\!]^*_A \subseteq [\![Typ_P(q)]\!]^*_A$$

Definition 24. For a given formula F, its truth under assignment A with database state db for I, written as $I \models_{A,db} F$ is inductively defined by

- if $p(t_1, \ldots, t_n)$ is an atomic formula and p is an ordinary predicate symbol then

$$I \models_{A,db} p(t_1, \ldots, t_n) \Leftrightarrow ([\![t_1]\!]^*_A, \ldots, [\![t_n]\!]^*_A) \in [\![p]\!]^*_A \wedge t_i \in E_B \ (1 \leq i \leq n)$$

- $I \models_{A,db} \mathbf{db}(t)$ iff there exists a table a in the database db, such that $[\![t]\!]^*_A \in [\![db{\cdot}a]\!]^*_A$, whenever $t :: \tau$, $t \in E_B$ and $\tau \in T_{obj}$
- $I \models_{A,db} t_1 \otimes t_2$ iff $[\![t_1]\!]^*_A \otimes [\![t_2]\!]^*_A$, whenever $t_1, t_2 \in E_B$ and $\otimes \in \{\leq, <, \geq, >\}$ and $t_1, t_2 :: \mathtt{int}$ or $t_1, t_2 :: \mathtt{real}$ or $t_1, t_2 :: \mathtt{string}$, or $t_1, t_2 :: \mathtt{char}$
- $I \models_{A,db} t_1 = t_2$ iff $[\![t_1]\!]^*_A = [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \neq t_2$ iff $[\![t_1]\!]^*_A \neq [\![t_2)]\!]^*_A$, whenever $t_1, t_2 :: \tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \textbf{ isa } t_2$ iff $cv_{\sigma \leq \tau}([\![t_1]\!]^*_A) = [\![t_2]\!]^*_A$, whenever $t_1 :: \sigma$, $t_2 :: \tau$ and $\sigma \leq \tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \textbf{ in } t_2$ iff $[\![t_1]\!]^*_A \in [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \mathbb{P}\tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \textbf{ sin } t_2$ iff $cv_{\sigma \leq \tau}([\![t_1]\!]^*_A) \in [\![t_2]\!]^*_A$, whenever $t_1 :: \sigma$, $t_2 :: \mathbb{P}\tau$ and $\sigma \leq \tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \textbf{ subset } t_2$ iff $[\![t_1]\!]^*_A \subseteq [\![t_2]\!]^*_A$, whenever $t_1, t_2 :: \mathbb{P}\tau$ and $t_1, t_2 \in E_B$
- $I \models_{A,db} t_1 \textbf{ ssubset } t_2$ iff $cv_{\mathbb{P}\sigma \leq \mathbb{P}\tau}([\![t_1]\!]^*_A) \subseteq [\![t_2]\!]^*_A$, whenever $t_1 :: \mathbb{P}\sigma$, $t_2 :: \mathbb{P}\tau$ and $\sigma \leq \tau$ and $t_1, t_2 \in E_B$

- $I \models_{A,db}$ **true**, whenever **not** $I \models_{A,db}$ **false**,
- if F and G are formulas then
  $I \models_{A,db}$ **not** F iff not $I \models_{A,db} F$
  $I \models_{A,db} F \vee G$ iff $I \models_{A,db} F$ or $I \models_{A,db} G$
  $I \models_{A,db} \forall x F$ iff $I \models_{(A,db)[x/t]} F$ for all $t \in D$ where $x :: \sigma \Leftrightarrow t :: \sigma$.

This definition is straightforward (cf. Lloyd [Lloy87]).

5 Conclusions and future work

We have defined a logical language, called DTL which can be considered as a far reaching extension of Datalog equipped with complex objects, object identities, and multiple inheritance based on an extension of Cardelli type theory. The DTL language also incorporates a very general notion of sets as first-class objects. It has furthermore been explained how this language could be used to query an object-oriented database specified along the lines of a state-of-the-art data model called TM. In DTL we have defined a simple, yet powerful way of combining multiple inheritance with predicates in logic programs. Furthermore, we have offered a simple denotational semantics for DTL programs. Though we have not described our evaluation strategy for DTL programs in this paper, we do mention that this strategy has been proven to be sound and complete. For details concerning the resolution of DTL programs we refer to [Bal92,BaBa93]

As far as future research is concerned, we are presently interested in implementation issues related to DTL. We mention that a TM-based DBMS prototype has been realized in the logical language LIFE, and that we are currently investigating implementation possibilities of DTL along similar lines.

6 References

[AbKa89] S. Abiteboul & P. C. Kanellakis, "Object identity as a query language primitive," in *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989*, J. Clifford, B. Lindsay & D. Maier, eds., ACM Press, New York, NY, 1989, 159–173, (also appeared as SIGMOD RECORD, 18, 2, June, 1989).

[Abit90] S. Abiteboul, "Towards a deductive object-oriented database language," *Data & Knowledge Engineering* 5 (1990), 263–287.

[AbGr88] S. Abiteboul & S. Grumbach, "COL: A Logic-based Language for Complex Objects," in *Advances in Database Technology—EDBT '88*, J. W. Schmidt, S. Ceri & M. Missikoff, eds., Springer-Verlag, New York–Heidelberg–Berlin, 1988, 271–293, Lecture Notes in Computer Science 303.

[Aït-K91] H. Aït-Kaci, "An overview of LIFE," in *Next Generation Information System Technology*, J. W. Schmidt & A. A. Stogny, eds., Proceedings of the First International East/West Data Base Workshop, Kiev, USSR, October 1990, Springer-Verlag, New York–Heidelberg–Berlin, 1991, 42–58, Lecture Notes in Computer Science # 504.

[Bal92] R. Bal, "DataTypeLog a deductive object-oriented query language," University of Twente, Technical Report INF92-79, Enschede, 1992.

[BaBa93] R. Bal & H. Balsters, "DTL: A deductive and typed object-oriented language," Universiteit Twente, INF93-41, Enschede, The Netherlands, 1993.

[BaBB92] R. Bal, H. Balsters & R. A. de By, "The TM typing rules," University of Twente, Technical Report INF92-80, Enschede, 1992.

[BaBZ93] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *ECOOP 1993 Kaiserslautern*, 1993.

[BaBV92] H. Balsters, R. A. de By & C. C. de Vreeze, "The TM Manual," University of Twente, technical report INF92-81, Enschede, 1992.

[BaFo91] H. Balsters & M. M. Fokkinga, "Subtyping can have a simple semantics," *Theoretical Computer Science* 87 (September, 1991), 81–96.

[BaVr91] H. Balsters & C. C. de Vreeze, "A semantics of object-oriented sets," in *The Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data (DBPL–3), August 27–30, 1991, Nafplion, Greece*, P. Kanellakis & J. W. Schmidt, eds., Morgan Kaufmann Publishers, San Mateo, CA, 1991, 201–217.

[BrLM90] A. Brogi, E. Lamma & P. Mello, "Inheritance and hypothetical reasoning in logic programming," in *Ninth European Conference on Artificial Intelligence*, L. C. Aiello, ed., Stockholm, Sweden, 1990, 105–110.

[CCCT90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca & R. Zicari, "Integrating object-oriented data modeling with a rule-based programming paradigm," in *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, NJ, May 23–25, 1990*, H. Garcia-Molina & H. V. Jagadish, eds., ACM Press, New York, NY, 1990, 225–236, (also appeared as SIGMOD RECORD, 19, 2, June, 1990).

[CaWe85] L. Cardelli & P. Wegner, "On understanding types, data abstraction, and polymorphism," *Computing Surveys* 17 (1985), 471–522.

[Card84] L. Cardelli, "A semantics of multiple inheritance," in *Semantics of Data Types*, G. Kahn, D. B. Macqueen & G. Plotkin, eds., Lecture Notes in Computer Science #173, Springer-Verlag, New York–Heidelberg–Berlin, 1984, 51–67.

[Card88] L. Cardelli, "Types for data-oriented languages," in *Advances in Database Technology—EDBT '88*, J. W. Schmidt, S. Ceri & M. Missikoff, eds., Springer-Verlag, New York–Heidelberg–Berlin, 1988, 1–15, Lecture Notes in Computer Science 303.

[CeGT90] S. Ceri, G. Gottlob & L. Tanca, *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, New York–Heidelberg–Berlin, 1990.

[GrLR92] S. Greco, N. Leone & P. Rullo, "COMPLEX: An Object-Oriented Logical Programming System," *IEEE Transactions on Knowledge and Data Engineering* 4 (august 1992), 344–359.

[HuYo90] R. Hull & M. Yoshikawa, "ILOG: Declaritive Creation and Manipulation of Object Identifiers," in *Proceedings Sixth International Conference on Data Engineering, Los Angeles, CA, February 5–9, 1990*, IEEE Computer Society Press, Washington, DC, 1990, 455–468.

[IbCu90] M. H. Ibrahim & F. A. Cummins, "Objects with logic," in *AMC 18th Annual Computer Science Conference*, Washington DC, 1990, 128–133.

[KiLW90] M. Kifer, G. Lausen & J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," Dept. Comp. Sc. State University of New York at Stony Brook, 90/14, Stony Brook, 1990.

[LéRi89] C. Lécluse & P. Richard, "Modeling Complex Structures in Object-Oriented Databases," in *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1989, 360–368.

[Lloy87] J. W. Lloyd, *Foundations of Logic Programming*, Symbolic Computation, Springer-Verlag, New York–Heidelberg–Berlin, 1987.

[LoOz91] Y. Lou & M. Ozsoyoglu, "LLO: A Deductive Language with Methods and Method Inheritance," in *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, CO, May 29–31, 1991*, J. Clifford & R. King, eds., ACM Press, New York, NY, 1991, 198–207, (also appeared as SIGMOD RECORD, 20, 2, June, 1991).

[LyVi87] P. Lyngbaek & V. Vianu, "Mapping a semantic database model to the relational model," in *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA, May 27–29, 1987*, U. Dayal & I. Traiger, eds., ACM Press, New York, NY, 1987, 132–142, (also appeared as SIGMOD RECORD 16, 3, December, 1987).

[McCa92] F. G. McCabe, *Logic and Objects*, International Series in Computer Science, Prentice-Hall International, London, England, 1992.

[MoPo90] L. Monteiro & A. Porto, "A transformational view of inheritance in logic programming," in *Seventh International Conference on Logic Programming*, D. H. D. Warren & P. Szeredi, eds., MIT Press, Cambridge, MA, 1990, 481–494.

[NaTs89] S. Naqvi & S. Tsur, *A Logical Language for Data and Knowledge Bases*, Principles of Computer Science, Computer Science Press, Rockville, MD, 1989, 288 pp..

[OhBB89] A. Ohori, P. Buneman & V. Breazu-Tannen, "Database programming in Machiavelli-a polymorphic language with static type inference," in *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989*, J. Clifford, B. Lindsay & D. Maier, eds., ACM Press, New York, NY, 1989, 46–57, (also appeared as SIGMOD RECORD, 18, 2, June, 1989).

[Reyn85] J. C. Reynolds, "Three Approaches to Type Structure," in *Mathematical Foundations of Software Development*, H. Ehrig *et al.*, ed., Lecture Notes in Computer Science #185, Springer-Verlag, New York–Heidelberg–Berlin, 1985, 97–138.