EG, Integration of the Object-Oriented and the Deductive Database Paradigms

Arno Siebes
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In this paper EG, a fragment of a database programming language, is presented. EG is aimed at the integration of object-oriented and deductive datamodels. Different from other proposals in this area, EG consists of a variant of the simply typed λ-calculus with subtyping, over which a Datalog-like language has been defined. The result is a language that can be typed statically, for which both model-theoretic and fixed-point semantics are defined. It is shown that these two kinds of semantics coincide. Moreover, it is shown that an EG-program P has a minimal model, which coincides with its least fixed point.

1 Introduction

The object-oriented and the deductive datamodel paradigms are largely complementary approaches towards the development of database programming languages adequate for new DBMS application areas. One of the main themes in the object-oriented paradigm is the structuring of the data through *complex objects* and *subtyping*. Whereas the focus in the deductive paradigm is on *knowledge*, both in the database and in queries.

There are numerous applications that could profit from the integration of both paradigms. For example, a CAD-tool based on the integrated paradigm allows the engineer to reason about her design [27]. It is therefore not surprising that the integration is an active research area, [19, 9, 22, 12, 2, 18, 17, 4].

In this paper EG, a fragment of a database programming language, is presented. It is aimed at the integration of the two paradigms. Different from other proposals in this area, EG consists of a variant of the simply typed λ-calculus with subtyping, over which a Datalog-like language has been defined. That is, EG is an integrated formalism in the style advocated by Beeri [11]. The result is a (sub) language that can be typed statically, for which both model-theoretic and fixed-point semantics are defined. It is shown that these two kinds of semantics coincide. Moreover, it is shown that an EG-program P has a minimal model, which coincides with its least fixed point.

EG is only a fragment of a language. However, the results of this paper can be applied to extend existing object-oriented database programming languages

that adhere to a logical type model, such as [21, 5]. In such a case, the syntax of EG should be adapted to the syntax of the base language. Hence the name, *the syntax is only an example.*

Comparing EG with the other approaches cited above, we have:

- EG is based on the simply typed λ-calculus with subtyping, that means e.g.:
  1. EG can bc typed statically, unlike [19, 9, 22, 18, 2].
  2. Types are not objects; unlike [17, 12].
  3. Subtyping is an integral part of the typing schema.
- EG has (coinciding) model-theoretic and fixed-point semantics; unlike [4, 12, 9].
- The fragment presented in this paper does not support object invention, negation and grouping.

The last item, which is clearly a drawback, is briefly discussed in the last section.

A problem with the integration of the two paradigms is the reconsiliation of their semantics. The logical type theories mostly used in object-oriented data models originate from typed λ-calculus. Whereas deductive data models are based on (first order) logic. In this paper, category theory is chosen to define the semantics of EG, because of its close connection with both typed λ-calculi and deduction, c.f. [20, 16]. In particular, our own category-theoretical semantics for subtyping [25], are combined with the category-theoretical semantics for logic programming languages given by Asperti and Martini [6].

The organization of this paper is as follows. In Section 2 the syntax of EG is defined. Moreover, it is proved that EG can be typed statically. Due to the integration of two paradigms, the semantics of EG come in two parts. The first part is concerned with the functional aspect of EG, i.e., the interpretation of the terms under (sub)typing; this is the topic of Section 3. The semantics of the deductive database part are defined over this interpretation. In Section 4, model-theoretic semantics are defined, while in Section 5 fixed-point semantics are defined. Moreover, it is shown in Section 5 that these semantics coincide and that each EG program P has a minimal model. In Section 6, the conclusions are formulated, and the addition of object invention, negation and grouping are briefly discussed.

In this article, only proof-sketches are included.

2 The syntax

In this section, the syntax of EG is introduced. Because EG is intended to integrate the object-oriented paradigm with the deductive paradigm, it inherits its syntax from both areas. More precisely, the *terms* of EG in deductive vernacular are its *expressions* in object-oriented terminology[1]. Hence, the definition of the syntax of EG will reflect both kinds of syntax.

[1] To indicate this dual nature, we will call them term-expressions.

First the object-oriented part is defined. That is, *types*, *the subtyping relationship*, *constants* and *terms* are defined, adapting Cardelli's [14] syntax. The important variations are the addition of the powertype construction, denoted by $\mathcal{P}$, and the elimination of function types. Both changes should be obvious from a Datalog point of view. The presentation of this part follows more or less the definitions in [8].

The expressions of the object-oriented part are used as the terms for the deductive part. The definition of this part follows more or less the definition of standard Datalog [28, 15].

Types are built inductively from a set of basic types $\mathcal{B}$. As usual in type theory, we make no further assumptions on the nature of the elements of $\mathcal{B}$. These basic types can be compared to the *abstract* types of IFO [3]. They denote the kind of objects that have no underlying structure, at least relative to the point of view of the database designer.

Besides the set $\mathcal{B}$, we postulate a (totally) ordered set $\mathcal{L}$ of labels. Types are then defined as follows.

DEFINITION 1. The set of types $Type$ is defined inductively as follows:

1. if $\beta \in \mathcal{B}$, then $\beta \in Type$;
2. if $\tau \in Type$, then $\mathcal{P}(\tau) \in Type$;
3. if $\tau_1, \cdots, \tau_n \in Type$ and $l_1, \cdots, l_n \in \mathcal{L}$ with $l_i < l_j$ if $i < j$, then
   (a) $(l_1 : \tau_1, \cdots, l_n : \tau_n) \in Type$ (a record with n fields);
   (b) $[l_1 : \tau_1, \cdots, l_n : \tau_n] \in Type$ (a variant record with n-fields).

The order on the fields in a (variant) record is only enforced to have a normal form. It saves the identification of e.g., records with the same fields but in a different order.

To illustrate, let both *int* and *string* be basic types, then the following examples denote types:

1. $person = (name : string, age : int)$, a *record type* with *labels name* and *age*, refering to the types *string* and *int* respectively;
2. $strorint = [string : string, int : int]$, a *variant record* with labels *string* and *int*, refering to the types *string* and *int* respectively.
3. $group = \mathcal{P}((name : string, age : int))$, a *powertype*, denoting groups of *persons*.

Strictly speaking, the name of a type does not belong to the formalism. It is introduced for convenience. For example, $\mathcal{P}(person)$ denotes the same type as $\mathcal{P}((name : string, age : int))$.

The collection of types of EG is a partially ordered set. The subtyping relation, which is the straight-forward extension of [14], is defined as follows.

DEFINITION 2. The subtyping relation, denoted by $\leq$, on $Type \times Type$ is defined inductively as follows:

1. $\beta \in \mathcal{B} \Rightarrow \beta \leq \beta$;

2. $\sigma, \tau \in Type \ \wedge \sigma \leq \tau \Rightarrow \mathcal{P}(\sigma) \leq \mathcal{P}(\tau)$;

3. $(\sigma_1 = (l_1 : \tau_1, \cdots, l_n : \tau_n), \sigma_2 = (m_1 : \nu_1, \cdots, m_k : \nu_k) \in Type \wedge \forall i \in \{1 \cdots k\} (\exists j \in \{1 \cdots n\}(m_i = l_j \wedge \tau_j \leq \nu_i))) \Rightarrow \sigma_1 \leq \sigma_2$;

4. $(\sigma_1 = [l_1 : \tau_1, \cdots, l_n : \tau_n], \sigma_2 = [m_1 : \nu_1, \cdots, m_k : \nu_k] \in Type \wedge \forall i \in \{1 \cdots n\} (\exists j \in \{1 \cdots k\}(l_i = m_j \wedge \tau_i \leq \nu_j))) \Rightarrow \sigma_1 \leq \sigma_2$.

With a simple induction proof, the following holds.

LEMMA 1. *The subtyping relation is a partial order.*

For each of the basic types $\beta \in \mathcal{B}$, a set C_β of *constant symbols* is postulated. The constant symbols (constants) of a type τ are then built inductively, mimicking the construction of τ.

DEFINITION 3. Let C_β denote the sets of constants of the $\beta \in \mathcal{B}$. The set of constants of a type $\tau \notin \mathcal{B}$, denoted by C_τ, is defined inductively as follows:

1. $\forall i \in \{1 \cdots n\}(c_i \in C_\tau) \Rightarrow \{c_1, \cdots, c_n\} \in C_{\mathcal{P}(\tau)}$;

2. $\forall i \in \{1 \cdots n\}(c_i \in C_{\tau_i}) \Rightarrow (l_1 = c_1, \cdots, l_n = c_n) \in C_{(l_1:\tau_1, \cdots, l_n:\tau_n)}$;

3. $\exists i \in \{1 \cdots n\}(l = l_i \wedge c \in E_{\tau_i}) \Rightarrow [l = c] \in C_{[l_1:\tau_1, \cdots, l_n:\tau_n]}$.

Note that item 3 indicates that the C_τ are not necessarily disjoint. However, by straightforward induction the following result holds.

LEMMA 2. *If $c \in C_\tau$, there is exactly one proof of this fact.*

Continuing our example from above, we have:

1. $(name = Sara,\ age = 36) \in C_{person}$;

2. $[int = 37] \in C_{strorint}$;

3. $\{(name = Sara,\ age = 36),\ (name = Pete,\ age = 76)\} \in C_{\mathcal{P}(person)}$.

The other ingredient for the definition of term-expressions are the variables. Rather than introducing type-assignment in EG, a set V_τ of variables for each type τ is postulated.

Following [8], the term-expressions are defined first and only then a well-typing relation is defined. The advantage is that non-well-typed expressions do not have to be given semantics.

DEFINITION 4. The set TE of term-expressions is defined inductively by:

1. $\tau \in Type \wedge (t \in C_\tau \vee t \in V_\tau) \Rightarrow t \in TE$;

2. $\forall i \in \{1 \cdots n\}(t_i \in TE) \Rightarrow \{t_1, \cdots, t_n\} \in TE$;

3. $(t_1, t_2 \in TE \wedge \theta \in \{\cup, \cap, \setminus\}) \Rightarrow t_1 \theta t_2 \in TE$;

4. $(\forall i \in \{1 \cdots n\}(t_i \in TE) \wedge \forall i \in \{1 \cdots n\}(l_i \in \mathcal{L}) \wedge$
   $\forall i, j \in \{1 \cdots n\}(i \leq j \Rightarrow l_i \leq l_j)) \Rightarrow (l_1 = t_1, \cdots, l_n = t_n) \in TE$;
5. $t \in TE \wedge l \in \mathcal{L} \Rightarrow t.l \in TE$;
6. $t \in TE \wedge l \in \mathcal{L} \Rightarrow [l = t] \in TE$.

The well-typing relation is now defined by inductive comparison of the structure of the term and the structure of the type.

DEFINITION 5. The relation : on $TE \times Type$ ($t : \tau$ is pronounced as t is a term of type τ) is defined inductively as follows:

1. $\tau \in Type \wedge (t \in C_\tau \vee t \in V_\tau) \Rightarrow t : \tau$;
2. $(\exists i \in \{1 \cdots n\}(t_i : \tau) \wedge \forall i \in \{1 \cdots n\}(t_i : \sigma \Rightarrow (\forall j \in \{1 \cdots n\}(t_j : \sigma))))$
   $\Rightarrow \{t_1, \cdots, t_n\} : \mathcal{P}(\tau)$;
3. $(t_1, t_2 : \mathcal{P}(\tau) \wedge \theta \in \{\cup, \cap, \setminus\}) \Rightarrow t_1 \theta t_2 : \mathcal{P}(\tau)$;
4. $\forall i \in \{1 \cdots n\}(t_i : \tau_i) \Rightarrow (l_1 = t_1, \cdots, l_n = t_n) : (l_1 : \tau_1, \cdots, l_n : \tau_n)$;
5. $(t : (l_1 : \tau_1, \cdots, l_n : \tau_n) \wedge \exists i \in \{1 \cdots n\}(l_i = l \wedge \tau_i = \tau)) \Rightarrow t.l : \tau$;
6. $\exists i \in \{1 \cdots n\}(l = l_i \wedge t : \tau_i) \Rightarrow [l = t] : [l_1 : \tau_1, \cdots, l_n : \tau_n]$;
7. $t : \sigma \wedge \sigma \leq \tau \Rightarrow t : \tau$.

Item 2 requires that all terms in a set-type are constructed in the same way. This may seem a severe restriction, however, if we loosen the restriction, anomalies occur; see also [7]. Continuing our running example, we have the following well-typed terms[2]:

1. $(name = Sara,\ age = X_{int}) : person$;
2. $[int = 37] : strorint$;
3. $\{(name = X_{string},\ age = 36),\ Y_{person}\} : \mathcal{P}(person)$.

Definition 5 indicates that a term can have more than one type. Moreover, what is worse, a fact $t : \sigma$ may have more than one derivation. This endangers an unambiguous interpretation of a term. However, it turns out that a term t has a *unique minimal type*, denoted by $t :: \sigma$. Moreover, there is at most one proof of $t :: \sigma$. Rather than proving this directly, the 'minimal' typing is defined first and then it is shown that this definition has these properties.

DEFINITION 6. The relation :: on $TE \times Type$ ($t :: \tau$ is pronounced as t is a term of minimal type τ) is defined inductively as follows:

1. $t :: \tau$, whenever $t \in C_\tau$ or $t \in V_\tau$ and $\tau \in Type$;
2. $\{t_1, \cdots, t_n\} :: \mathcal{P}(\tau)$, if $\forall i \in \{1 \cdots n\} : t_i :: \tau$;

[2]Variables are sometimes annotated with their type for clarity.

3. $t_1 \theta t_2 :: \mathcal{P}(\tau)$, whenever $t_1, t_2 :: \mathcal{P}(\tau)$;
4. $(l_1 = t_1, \cdots, l_n = t_n) :: (l_1 : \tau_1, \cdots, l_n : \tau_n)$, iff $\forall i \in \{1 \cdots n\} : t_i :: \tau_i$;
5. $t.l :: \tau$ if $t :: (l_1 : \tau_1, \cdots, l_n : \tau_n)$ and $\exists i \in \{1 \cdots n\}, l_i = l \wedge \tau_i = \tau$;
6. $[l = t] :: [l : \tau]$, iff $t :: \tau$.

Using induction, it is easy to prove that :: really denotes *the minimal* type of an expression.

LEMMA 3. *If $t : \sigma$, there is exactly one $\tau \leq \sigma$, such that $t :: \tau$. Moreover, there is at most one proof of $t :: \tau$.*

This result concludes the introduction of the object-oriented part, save one last remark. Until now, object identities have not been mentioned and as changes to the database are not considered they do not play a very large role. It is asumed that $Oid \in \mathcal{B}$, and that each type in the database has Oid as an attribute.

As mentioned before, the term-expressions fullfill the role of terms in e.g., Datalog. Usually, a set P of predicate symbols is postulated together with a function $arity : P \rightarrow \mathcal{N}^+$. If $arity(p) = n$, p denotes an n-ary predicate symbol.

In EG, the definition of predicates is subtly different, rather than postulating a function *arity*, we associate a type with each predicate. The role of the type is similar to the role of the *arity*-function, it defines the *atoms* that can be built with the predicate symbol. That is, the type does not denote the type of the predicate symbol (which is in fact impossible, as predicate symbols are not included in the set of expressions), but the type of the *argument* of the predicate symbol.

This leads to the following definition.

DEFINITION 7. Let $Pred$ be the collection of predicate symbols. A predicate declaration consists of a pair (p, τ), such that $p \in Pred$ and $\tau \in Type$.

Examples of predicate declarations are:

1. $(rich, person)$;
2. $(small, strorint)$;
3. $(soccer - team, \mathcal{P}(person))$.

The definition of atoms and literals is now straightforward.

DEFINITION 8. The sets of atoms and literals are defined as follows

- The set $Atom$ of atoms is defined by:
  1. $(p, t) \in Atom$, if (p, τ) is a predicate declaration and $t : \tau$.
  2. $t_1 =_\tau t_2$, $t_1 \neq_\tau t_2$ iff $t_1, t_2 : \tau$;
  3. $t_1 \in_\tau t_2$, $t_1 \notin_\tau t_2$ if $t_1 : \tau$ and $t_2 : \mathcal{P}(\tau)$;
  4. $t_1 \subset_{\mathcal{P}(\tau)} t_2$, $t_1 \not\subset_{\mathcal{P}(\tau)} t_2$ if $t_1, t_2 : \mathcal{P}(\tau)$.

An atom is called *ground*, if it contains no variables.

- A literal is an atom or a negated atom. A literal is called *ground* if the underlying atom is ground.

The literals in item 2, 3 and 4 above are defined over the *built-in* predicates $=$, $\in$ and $\subset$, which have the obvious semantics. Their negations are also added as we can safely allow e.g., $t_1 \neq_\tau t_2$ in the conjunct of a clause. Examples of atoms are:

1. $(rich, (name = Sara, age = X))$;
2. $(small, [string = John])$;
3. $(soccer - team, \{(name = Sara, age = 17), (name = John, age = 23)\})$.

Examples of literals are:

1. $\neg(rich, (name = Sara, age = X))$;
2. $(small, [string = X])$;
3. $\neg(soccer - team, \{(name = X, age = 17), (name = John, age = Y)\})$.

In the remainder of this paper, we will write $rich(name = Sara, age = X)$, rather than $(rich, (name = Sara, age = X))$.

The remainder of the definition of EG is a simple copy of the definition of Datalog. That is, a clause is a finite list of literals and a Horn clause is a clause containing at most one positive literal. As usual, we distinguish three kinds of Horn clauses:

- A fact is a positive Horn clause, i.e., a list containing one atom. A fact is denoted in EG, as an atom followed by a period, e.g.

$$rich(name = Sara, age = 27).$$

- A rule is a Horn clause with one positive literal and at least one negative literal. Rules are denoted as follows:

$$\begin{aligned} rich(X^{person}) \quad \leftarrow \quad & rich(Y^{person}), \\ & parent(par = Y^{person}, child = X^{person}). \end{aligned}$$

  The lhs-atom of the rule (the *head*) is the positive literal of the clause, the rhs is the conjunction of the atoms that underly the negative literals.

- A goal clause is a negative clause, i.e., a list containing negative literals. In EG, a goal clause is denoted as:

$$\leftarrow rich(X^{person}).$$

To define EG programs, we distinguish (as usual) EDB and IDB predicates. In other words, we assume that the collection of predicates consists of two disjoint subsets, viz., EDB and IDB. The EDB predicates denote the extensional database, while the IDB predicates denote the intensional database. So, the special predicates, $=$, $\neq$, $\in$, $\notin$, $\subset$, and $\not\subset$, all belong to the EDB-predicates.

EG-programs are then defined as follows.

DEFINITION 9. Let P be a finite set of Horn clauses, P is an EG program if for each clause C either of the following two holds:

1. C is a fact, whose predicate belongs to EDB excluding the special predicates;
2. The positive literal predicate of C belongs to IDB, and all variables occuring in the positive literal also occur in at least one negative literal.

The exclusion of the special predicates in the first item of the definition is made to ensure that a program does not try to redefine these built-in predicates by adding new facts.

This concludes the definition of the syntax of EG. A straightforward adaption of Cardelli's [14] typechecking algorithm results in a typechecking algorithm for EG. Hence, we have the first important result of this paper.

THEOREM 1. *Type checking EG programs can be done at compile-time.*

3 THE INTERPRETATION OF TERM-EXPRESSIONS

For a standard logical programming language, such as Datalog, the semantics of a program are given with respect to the *Herbrand Universe.* All constants are interpreted as themselves, and the universe consists of all ground facts. For EG this is not a viable option, as, different from e.g. Datalog, the constants of EG are unrelated.

For example, consider the types $person = (name : string, age : int)$ and $employee = (name : string, age : int, sal : real)$ and the constants $(name = pete, age = 35)$ and $(name = pete, age = 35, sal = 3452)$. Let the predicate $rich$ be declared as $(rich, person)$. Because of the subtyping relationship, the atoms $rich(name = pete, age = 35)$ and $rich(name = pete, age = 35, sal = 3452)$ should be considered the same in a very precise sense of the word.

Such identifications are not only necessary for the constants, but for term-expressions in general. In this section, we define how the term-expressions are to be interpreted in such a way that the correct identifications are made.

As explained in the introduction, this interpretation is through category theory. Because of space constraints, this paper can not be self-contained with regard to this theory. The interested reader is refered to [10, 24]. Some non-standard constructions are defined in the appendix.

It is asumed that the basic type are objects in some topos (in general this will be Set). The semantic domain of EG, $\mathcal{T}$, is then formed by the full sup-topos freely generated by $\mathcal{B}$.

The interpretation of the other types is straightforward, records are mapped to cartesian products, variants to sums and powertypes to powerobjects.

DEFINITION 10. The interpretation $I : Type \rightarrow \mathcal{T}$ is defined inductively by:

1. $I(\beta) = \beta$ for $\beta \in \mathcal{B}$;
2. $I((l_1 : \tau_1, \cdots, l_n : \tau_n)) = I(\tau_1) \times \cdots \times I(\tau_n)$;
3. $I([l_1 : \tau_1, \cdots, l_n : \tau_n]) = I(\tau_1) + \cdots + I(\tau_n)$;
4. $I(\mathcal{P}(\tau)) = \mathcal{P}(I(\tau))$.

The interpretation of a constant $c \in C$ is an arrow $c : 1 \rightarrow I(C)$. It is postulated that the interpretation of the elements of the basic types is known.

DEFINITION 11. The interpretation I of constants in $\mathcal{T}$ is defined inductively by:

1. for $c = b \in \beta \in \mathcal{B}$,

$$I(c) = b : 1 \rightarrow \beta;$$

2. for $c = \{c_1, \cdots, c_n\} \in C_{\mathcal{P}(\tau)}$,

$$I(c) = \{\}_{(I(\tau),n)} \circ < I(c_1), \cdots, I(c_n) >: 1 \rightarrow \mathcal{P}(I(\tau));$$

3. for $c = (l_1 = c_1, \cdots, l_n = c_n) \in C_{(l_1:\tau_1,\cdots,l_n:\tau_n)}$,

$$I(c) =< I(c_1), \cdots, I(c_n) >: 1 \rightarrow I(\tau_1) \times \cdots \times I(\tau_n);$$

4. for $c = [l_j = d] \in C_{[l_1:\tau_1,\cdots,l_n:\tau_n]}$,

$$I(c) = \iota_j \circ I(d) : 1 \rightarrow I(\tau_1) + \cdots + I(\tau_n).$$

We will sometimes omit the I, if the context unambiguously implies that we use the interpretation rather than EG itself. More specifically, we will sometimes write c, rather than $I(c)$ and $\tau_1 \times \cdots \times \tau_n$ rather than $I(\tau_1) \times \cdots \times I(\tau_n)$.

The interpretation of term-expressions in general is less straightforward than the interpretation of the constants. The reason is, of course, that there is at most one proof of $c \in C_\tau$, but there may be more proofs of $t : \tau$. To simplify the definition of the interpretation, we first define a *pre-interpretation*. Then we define *conversion functions*, which form the semantic interpretation of subtyping. Then we combine these two interpretations in the definition of the interpretation of a term-expression.

Rather than interpreting variables, and thus terms, as arrows $1 \rightarrow I(\tau)$, variables, and thus terms, are interpreted relative to an *environment*. This interpretation turns out to be most usefull in the next sections; see also [6].

DEFINITION 12. An environment η is a set of (typed) variables. Given an environment η and a type τ, $TE(\eta, \tau)$ denotes the set of term-expressions $t : \tau$, such that $t \in TE(\eta, \tau)$ iff for each variable X occuring in t, $X \in \eta$.

When EG programs are given semantics, the environment will consist of the variables in the clause in which the term-expression occurs. To illustrate $TE(\eta, \tau)$, if $\eta = \{X^{\tau_1}, Y^{\tau_2}\}$ and $\tau = (l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3)$ and $c \in C_{\tau_3}$ then

$$(l_1 = X^{\tau_1}, l_2 = Y^{\tau_2}, l_3 = c) \in TE(\eta, \tau).$$

The pre-interpretation of a term-expression is defined relative to an environment, and its minimal type. Hence, the difference between pre-interpretations and interpretations is that the former are only defined for the minimal type of an expression.

DEFINITION 13. Let $\eta = \{{X_1}^{\tau_1}, \cdots, {X_n}^{\tau_n}\}$, $\tau \in Type$, and $t \in TE(\eta, \tau)$, moreover, assume that $t :: \tau$. The pre-interpretation $PI(\eta, \tau)(t) : \tau_1 \times \cdots \times \tau_n \to \tau$ of t is defined recursively by:

1. for $t = c \in C_\tau$,

$$PI(\eta, \tau)(t) = c \circ !_{\tau_1 \times \cdots \times \tau_n};$$

2. for $t = {X_i}^{\tau_i} \in \eta$, (note this implies that $\tau_i = \tau$)

$$PI(\eta, \tau)(t) = \pi_i;$$

3. for $t = \{t_1, \cdots, t_m\}$, and $\tau = \mathcal{P}(\sigma)$,

$$PI(\eta, \tau)(t) = \{\}_{m,\sigma} \circ (PI(\eta, \sigma)(t_1) \times \cdots \times PI(\eta, \sigma)(t_m));$$

4. for $t = t_1 \theta t_2$,

$$PI(\eta, \tau)(t) = \theta \circ (PI(\eta, \tau)(t_1) \times PI(\eta, \tau)(t_2));$$

5. for $t = (l_1 = t_1, \cdots, l_m = t_m) :: (l_1 : \sigma_1, \cdots, l_m : \sigma_m)(= \tau)$,

$$PI(\eta, \tau)(t) = PI(\eta, \sigma_1)(t_1) \times \cdots \times PI(\eta, \sigma_n)(t_m);$$

6. for $t = t'.l_i$, where $t' :: \sigma = (l_1 : \sigma_1, \cdots, l_m : \sigma_m)$,

$$PI(\eta, \tau)(t) = \pi_i \circ PI(\eta, \sigma)(t');$$

7. for $t = [l_i = t']$, where $t' :: \sigma$,

$$PI(\eta, \tau)(t) = \iota_i \circ PI(\eta, \sigma)(t').$$

In this definition, the construction of the pre-interpretation of a term-expression t of minimal type τ follows the proof that $t :: \tau$. The definition is sound because there is at most one proof that $t :: \tau$.

The definition of these conversion functions follows directly from the categorical interpretation of types.

DEFINITION 14. For $\sigma, \tau \in Type$, with $\sigma \leq \tau$, the conversion function $cv_{\sigma \leq \tau}$: $I(\sigma) \rightarrow I(\tau)$ is defined inductively as follows:

1. $cv_{\sigma \leq \sigma} = id_{I(\sigma)}$;
2. if $\sigma = \mathcal{P}(\sigma')$ and $\tau = \mathcal{P}(\tau')$, with $\sigma' \leq \tau'$, then

$$cv_{\sigma \leq \tau} = (cv_{\sigma' \leq \tau'})^*;$$

3. if $\sigma = (l_1 : \sigma_1, \cdots, l_n : \sigma_n)$, $\tau = (k_1 : \tau_1, \cdots, k_m : \tau_m)$, and $f : \{1 \cdots m\} \rightarrow \{1 \cdots n\}$ such that $k_i = l_{f(i)}$, then

$$cv_{\sigma \leq \tau} = (cv_{\sigma_{f(1)} \leq \tau_1} \times \cdots \times cv_{\sigma_{f(m)} \leq \tau_m}) \circ < \pi_{f(1)}, \cdots, \pi_{f(m)} >;$$

4. if $\sigma = [l_1 : \sigma_1, \cdots, l_n : \sigma_n]$, $\tau = [k_1 : \tau_1, \cdots, k_m : \tau_m]$, and $f : \{1 \cdots n\} \rightarrow \{1 \cdots m\}$ such that $l_i = k_{f(i)}$, then

$$cv_{\sigma \leq \tau} = [\iota_{f(1)}, \cdots, \iota_{f(n)}] \circ (cv_{\sigma_1 \leq \tau_{f(1)}} + \cdots + cv_{\sigma_n \leq \tau_{f(n)}}).$$

There may be more than one way to prove $t : \tau$ from $t :: \sigma$. This could ambiguate the combination of conversion functions and pre-interpretations. However, the following fact holds, by induction using the standard categorical properties:

LEMMA 4. *For* $\sigma \leq \tau \leq \upsilon$: $cv_{\sigma \leq \tau} \circ cv_{\tau \leq \upsilon} = cv_{\sigma \leq \upsilon}$.

Now the interpretation of term-expressions can be defined as follows:

DEFINITION 15. Let $\eta = \{{X_1}^{\tau_1}, \cdots, {X_n}^{\tau_n}\}$ and $\tau \in Type$, the interpretation $I(\eta, \tau)(t) : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ of a term-expression $t \in TE(\eta, \tau)$, with $t :: \sigma$ is defined by:

$$I(\eta, \tau)(t) = cv_{\sigma \leq \tau} \circ PI(\eta, \tau)(t).$$

The interpretation of a term-expression is well-defined, for there is exactly one proof that $t :: \tau$ by Lemma 3, and Lemma 4 ensures that there is at most one conversion function between two types.

4 MODEL-THEORETIC SEMANTICS

In the previous section, we have given an interpretation to term expressions relative to an environment. In this section, we use this interpretation to give model-theoretic semantics to EG-programs.

In the, usual, semantics of logic-programming languages, predicate symbols are interpreted as subsets of the Herbrand Base. For EG, a minor variation of this scheme is needed. The Herbrand base is replaced by the interpretation of a type, and *subobjects* of this interpretation are used rather than subsets.

DEFINITION 16. An interpretation H of an EG-program P is a map that associates a subobject $H(p, \tau)$ of $I(\tau)$ with each predicate declaration (p, τ). Moreover, H should map built-in predicates to their standard interpretation [3].

[3] See the appendix for the construction of the standard interpretations.

Atoms are interpreted by the composition of the interpretation of its predicate and the interpretation of its term. So, the interpretation of an atom depends on an environment.

DEFINITION 17. Let H be an interpretation of an EG-program P, (p, σ) be a type declaration, $\eta = \{{X_1}^{\tau_1}, \cdots, {X_n}^{\tau_n}\}$ an environment, and $t \in TE(\eta, \sigma)$ a term-expression, the interpretation of $p(t)$ with respect to H and η, denoted by $I(H, \eta)(p(t))$, is the arrow

$$I(H, \eta)(p(t)) : I \to I(\tau_1) \times \cdots \times I(\tau_n)$$

which is defined as the pullback of $H(p, \sigma)$ along $I(\eta, \sigma)(t)$.

Next, conjunctions of atoms are interpreted as the composition of the interpretations of all of its constituting atoms.

DEFINITION 18. Let H be an interpretation of an EG-program P, let $\{(p_i, \sigma_i)\}_{i \in \{1 \cdots n\}}$ be a set of type declarations, $\eta = \{{X_1}^{\tau_1}, \cdots, {X_n}^{\tau_n}\}$ an environment, and $\{t_i \in TE(\eta, \sigma_i)\}_{i \in \{1 \cdots n\}}$ a set of term-expressions, the interpretation of $p_1(t_1), \cdots, p_n(t_n)$ with respect to H and η, denoted by $I(H, \eta)(p_1(t_1), \cdots, p_n(t_n))$ is the arrow

$$I(H, \eta)(p_1(t_1), \cdots, p_n(t_n)) : J \to I(\tau_1) \times \cdots \times I(\tau_n)$$

which is defined as the limit of $\{I(H, \eta)(p_i(t_i))\}_{i \in \{1 \cdots n\}}$.

In definitions 17 and 18, the environments are left unspecified. Both atoms and conjuncts are interpreted as arrows on $I(\tau_1) \times \cdots \times I(\tau_n)$, which is more or less the type of the environment. This observation suggests a natural association between clauses and environments, viz., associate with a clause the collection of all the variables it contains. Or, equivalently for EG clauses, the collection of all variables in the body of a clause.

This natural environment will be the standard choice in the remainder of this paper. Moreover, $I(H)(\cdots)$ is used as a shorthand for $I(H, \eta)(\cdots)$, where η denotes the natural environment of the clause under discussion.

Validity of clauses is defined in the standard categorical way.

DEFINITION 19. Let H be an interpretation of an EG-program, a clause ϕ is valid in I, denoted by $H \models \phi$, if:

1. let $\phi = \leftarrow (p_1, \sigma_1), \cdots, (p_n, \sigma_n)$ then:

$$H \models \phi \Leftrightarrow (I(H, \eta)(p_1, \sigma_1), \cdots, (p_n, \sigma_n) = id_\eta)$$

2. if $\phi = (q, \sigma) \leftarrow (p_1, \sigma_1), \cdots, (p_n, \sigma_n)$ then:

$$H \models \phi \Leftrightarrow (\exists \,\text{monic}\, m \,(I(H, \eta) \circ m = I(H, \eta)(p_1, \sigma_1), \cdots, (p_n, \sigma_n))).$$

The definition of a model is now simply the standard definition.

DEFINITION 20. Let P be an EG-program, and H an interpretation of P, H is a model of P, denoted by $H \models P$, is defined as follows:

$$H \models P \Leftrightarrow (\forall \phi \in P(H \models \phi)).$$

In the next section, we will prove that each program P has a minimal model, and, moreover, we show that this minimal model can be computed by a fixed-point construction in the usual way.

5 Least fixed-point semantics

In the previous section we defined the model theoretic semantics of an EG program P. In this section, we define the *fixed-point* semantics of P. Moreover, we show that each model of P is also a fixed point of P and vice versa. Finally, we show that P has a minimal fixed point, which is also a minimal model.

The definition of the fixed-point semantics closely follows standard practice in this area. That is, with each program P a (monotonic) function f_P on a complete lattice HA_P is associated. HA_P is defined as follows.

Definition 21. Let P be an EG-program, with predicate definitions: (p_1, τ_1), $\cdots, (p_n, \tau_n)$. HA_P, is defined by:

$$HA_P = Sub(I(\tau_1)) \times \cdots \times Sub(I(\tau_n)).$$

HA_P is a Heyting Algebra [16], so it is certainly a complete lattice.

Lemma 5. *HA_P is a complete lattice.*

The function f_P that is associated with a program P is defined as follows.

Definition 22. Let P be an EG-program, with predicate definitions $(p_1, \tau_1), \cdots, (p_n, \tau_n)$ and $X \in HA_P$. The function $f_P : HA_P \rightarrow HA_P$ is defined pointwise by:

1. X induces uniquely an interpretation $H(X)$ of P by:
$$H(X)(q) = \begin{cases} \pi_i(X) & \text{if } q = p_i \\ \text{the standard interpretation} & \text{if } q \text{ is a built-in predicate} \end{cases}$$

2. If p_i is an IDB-predicate, with facts $p_i(c_1), \cdots p_i(c_n)$, X_i is the subobject of $I(\tau_i)$ constructed by the epic/monic factorization of:
$$H(X)(p_i) \cup I(H(X))(p_i(c_1)) \cup \cdots \cup I(H(X))(p_i(c_n)).$$

3. If p_j is an IDB-predicate, defined by the rules:
$$\begin{aligned} p_j(t_1) &\leftarrow q_{1,1}(s_{1,1}), \cdots, q_{1,n_1}(s_{1,n_1}). \\ &\vdots \\ p_j(t_m) &\leftarrow q_{m,1}(s_{m,1}), \cdots, q_{m,n_m}(s_{m,n_m}). \end{aligned}$$
Define a set $\{\alpha_i\}_{i \in \{1 \cdots n\}}$ of subobjects as follows: α_k is the subobject of τ_j contructed by the epic/monic factorization of:
$$I(H(X))(p_j(t_1)) \cup I(H(X))(q_{1,1}(s_{1,1}), \cdots, q_{1,n_1}(s_{1,n_1})).$$

Define $X_i = \alpha_1 \cup \cdots \cup \alpha_m$.

4. $f_P(X) = (X_1, \cdots, X_n)$.

By construction, $f_P(X) = X \cup C$, hence we have the following lemma.

LEMMA 6. *If P is an EG-program, f_P is monotonic on HA_P.*

Since HA_P is a complete lattice, we have by Knaster-Tarski [26] the following lemma.

LEMMA 7. *f_P has a least-fixed point on HA_P.*

It is easy to see that each model H of P induces (uniquely) a fixed-point of f_P. H induces an element X_H of HA_P by 'forgetting' the built-in predicates. The fact that H is a model implies for the IDB-predicates that

$$I(H)(p_i) \supseteq I(H)(p_i(c_j))$$

for each fact $p_i(c_j)$. Moreover, it implies that for each EDB-predicate:

$$I(H)(p_j(t_1)) \supseteq I(H)(q_{1,1}(s_{1,1}), \cdots, q_{1,n_1}(s_{1,n_1}))$$

for each rule $p_j(t_1) \leftarrow q_{1,1}(s_{1,1}), \cdots, q_{1,n_1}(s_{1,n_1})$. Hence, X_H is a fixed-point. Similarly, each fixed-point induces (uniquely) a model; simply add the standard interpretation of the built-in predicates. This leads to the following result.

LEMMA 8. *Let H be a model of an EG-program P. H induces (uniquely) a fixed point of f_P. Conversely, let X be a fixed-point of f_P. X induces (uniquely) a model of P.*

Combining the Lemmas 7 and 8, we have one of our main results on EG, viz.

THEOREM 2. *An EG-program P has a minimal model M_P.*

For the (terminating) computation of the minimal model M_P of a program P, it is assumed that there is a natural isomorphism between the functors $ID : \mathcal{T} \rightarrow \mathcal{T}$ and $Hom(1, -) : \mathcal{T} \rightarrow Set$. This assumption yields that for each object $O \in \mathcal{T}$, $O \cong Hom(1, O)$, i.e., objects are completely determined by their elements [4].

Under this assumption, the function f_P on $HA_P = Sub(I(\tau_1)) \times \cdots \times Sub(I(\tau_n))$ induces a function g_P on $Hom(1, I(\tau_1)) \times \cdots \times Hom(1, I(\tau_n))$. Application of g_P is the addition of the new elements computed by f_P. Similar to f_P, g_P is monotone on a complete lattice and thus g_P has fixed-points. Moreover, the fixed-points of g_P correspond naturely with the fixed-points of f_P.

Following [1], it is straightforward to prove that the set of elements that can be computed inductively by g starting with the empty set is finite. Hence, the minimal fixed-point of g_P is finite. Given the natural correspondence with the fixed-points of f this leads to the final result of this paper.

THEOREM 3. *Under the assumption made above, $\bigsqcup_{i=1}^{\infty} {f_P}^i(\bot)$ computes the minimal fixpoint of f_P. M_P is the model induced by this fixed-point.*

[4] The assumption holds, e.g. if $\mathcal{T}$ is a subcategory of Set.

6 Conclusions and future research

EG is a (fragment) of a database programming language that consists of a variant of the simply typed λ-calculus with subtyping, over which a Datalog-like language has been defined. The result is a language that can be typed statically, for which both model-theoretic and fixed-point semantics of EG are defined. It is shown that these two kinds of semantics coincide. Moreover, it is shown that an EG-program P has a minimal model, which coincides with its least fixed point.

From an OODB perspective, the most obvious omission in EG is the lack of *methods*. Given the difficulties of the integration of the functional and the logical programming paradigm, it seems troublesome to allow expressions of a functional type in EG. However, it is well-known that a DBPL with object creation is strictly stronger than the same language without object creation. This means that supporting the application of methods increases the expressiveness of the language.

From a deductive databases point of view, the most obvious ommision in EG is the lack of negation and grouping. Similar to the short-coming mentioned above, it is well known that the addition of these two constructs increases the expressiveness of the language. In the remainder of this section, we briefly address both issues.

6.1 Method-application

In the previous sections term-expressions are interpreted as arrows in $\mathcal{T}$. Clearly, term-expressions built from the term-expressions as defined in this paper and method-applications can still be interpreted as arrows if method application can be interpreted as *function application.*

Given that all new term-expressions can still be interpreted as arrows in $\mathcal{T}$, both the model theoretic and the fixed-point semantics of EG as developed in this paper are still valid. Moreover, all theorems simply carry over to this new situation. Except, of course, that minimal models may be infinite.

So, if we can interpret method-application as function application, the extension is straight-forward. It is well known that it is problematic to interpret a method as a function in a language with subtyping, as subtyping of functional expressions is contra-variant in its first argument and covariant in its second. However, the inherent functionality of methods makes it plausible that each application of a method m can still be interpreted as the application of a associated function f, provided we do not require that we allways associate the same function with m.

The other problem with this extension, especially in the case of the method *new*, is that *Oid* will need some more structure than that of a simple basic type. It seems that Ohori's [23] representation of an object identity in a pure functional language represents a good starting point.

6.2 Negation and grouping

The various stratification approaches, used for grouping and/or negation, are all built on top of standard Datalog and its semantics. In this paper, it is shown that all important properties of the semantics of standard Datalog also hold for

the semantics of EG. Hence, adding negation and grouping to the language is a rather straightforward exercise.

More in particular, note that category theory is a *constructive* theory, the *inner logic* of a topos is in general non-classical. Hence, the obvious way to extend EG is to adopt the constructive approach to these problems, as initiated by Bry [13].

References

1. S. Abiteboul and C. Beeri (1988). On the Power of Languages for the Manipulation of Complex Objects. Tech. Rep. RR 989, INRIA.
2. S. Abiteboul and S. Grumbach (1988). COL, a Logic-Based Language for Complex Objects, in *Proc. EDBT*, 271–293.
3. S. Abiteboul and A. R. Hull (1987). IFO, a formal semantic data model. *ACM Transactions on Database Systems* 12, 525–565.
4. S. Abiteboul and P.C. Kanellakis (1989). Object identity as a query language primitive, in *Proc. ACM SIGMOD*, 159–173.
5. Antonio Albano, Giorgio Ghelli, and Renzo Orsini (1989). Types for Databases: The Galileo Experience, in *Proc. of the 2nd Int. Workshop on Database Programming Languages*, 196–206.
6. A. Asperti and S. Martini (1989). Projections Instead of Variables: A Category Theoretic Interpretation of Logic Programs, in *Logic Programming, Proceedings of the sixth international conference*, 337–352.
7. H. Balsters and C.C. de Vreeze (1990). A Formal Theory of Sets in Object Oriented Contexts. Tech. Rep. INF90-74, University of Twente.
8. Herman Balsters and Maarten M. Fokkinga (1991). Subtyping can have a simple semantics. *Theoretical Computer Science (to appear)*.
9. F. Bancilhon and S. Khoshafian (1986). A calculus for complex objects, in *Proc. 5th ACM Symp. on Principles of Database Systems*, 53–59.
10. Michael Barr and Charles Wells (1990). *Category Theory for Computing Science.* Prentice Hall.
11. C. Beeri (1989). Formal Models for Object Oriented Databases, in *Proceedings of the First Int. Conf. on Deductive and Object Oriented Databases*, 405–430.
12. Catriel Beeri, Roger Nasr, and Shalom Tsur (1988). Embedding *psi*-terms in a Horn-clause logic language, in *Proc. 3rd International Conf. on data and knowledge bases*, 347–358.
13. Francois Bry (1989). Logic Programming as Constructivism: A Formalization and its Applications to Databases, in *Proc. 8th ACM Symp. on Principles of Database Systems*, 34–50.
14. L. Cardelli (1988). A semantics of multiple inheritance. *Information and Computation* 76, 138–164.
15. S. Ceri, G. Gottlob, and L. Tanca (1990). *Logic Programming and Databases.* Springer Verlag.
16. R. Goldblatt (1979). *Topoi, The Categorical Analysis of Logic.* North-Holland.
17. M. Kifer and G. Lausen (1989). F-logic: A Higher-Order Language for Reasoning About Objects Inheritance and Scheme, in *Proc. ACM SIGMOD*,

134–146.
18. M. KIFER AND J. WU (1989). A Logic for object-oriented logic programming - Maiers O-logic revisited, in *Proc. 8th ACM Symp. on Principles of Database Systems*, 379–393.
19. G.M. KUPER (September 1985). *The Logical Data Model: A New Approach to Database Logic.* Ph.D. thesis, Stanford University, Department of Computer Science.
20. J. LAMBEK AND P.J. SCOTT (1986). *Introduction to Higher Order Categorical Logic.* Cambridge University Press.
21. C. LECLUSE, P. RICHARD, AND F. VELEZ (1988). O_2, an object-oriented datamodel, in *Proc. ACM SIGMOD*, 337–352.
22. SHANIN NAQVI AND SHALOM TSUR (1989). *A Logical Language for Data and Knowledge Bases.* Computer Science Press.
23. ATSHUSHI OHORI (1990). Representing Object Identity in a Pure Functional Language, in *Proc. third int. conf. on Database Theory*, 41–55.
24. BENJAMIN C. PIERCE (1990). A Taste of Category Theory for Computer Scientists. Tech. Rep. CS-90-113, Carnegie Mellon University, To be published in Computer Surveys.
25. ARNO SIEBES (1990). *On Complex Objects.* Ph.D. thesis, Twente University.
26. A. TARSKI (1955). A lattice-theoretical theorem and its applications. *Pacific Journal of Mathematics* 5, 285–309.
27. T. TOMIYAMA AND P.W.J. TEN HAGEN (1987). The Concept of Intelligent Integrated Interactive CAD Systems. Tech. Rep. CS-R8717, CWI.
28. JEFFREY D. ULLMAN (1988). *Principles of Database and Knowledge-Base Systems 1.* Computer Science Press.

A SOME NON-STANDARD CONSTRUCTIONS

Not all functions and objects used in this paper have a standard name in category theory. In this appendix, we show how they can be constructed:

- $twist_{A,B} =< \pi_B, \pi_A >: A \times B \rightarrow B \times A$;
- $\in_A: \epsilon_A \rightarrow \mathcal{P}(A) \times A$ is the canonical injection from the definition of power-objects.
- $\{\}_{1,A} : A \rightarrow \mathcal{P}(A)$ (the singleton-set builder) is the unique arrow defined by the universal property of powerobjects and the (monic) arrow $< id_A, id_A >: A \rightarrow A \times A$;
- $\cup_{\mathcal{P}(A)} : \mathcal{P}(A) \times \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ (the typed union) is defined by the following construction:
  1. let $f = id_{\mathcal{P}(A)} \times \in_A: \mathcal{P}(A) \times \epsilon_A \rightarrow \mathcal{P}(A) \times \mathcal{P}(A) \times A$;
  2. let $g = twist_{\mathcal{P}(A),\mathcal{P}(A)} \circ f : \mathcal{P}(A) \times \epsilon_A \rightarrow \mathcal{P}(A) \times \mathcal{P}(A) \times A$;
  3. let (k, l) be the epic/monic factorization of
     $$[f, g] : \mathcal{P}(A) \times \epsilon_A + \mathcal{P}(A) \times \epsilon_A \rightarrow \mathcal{P}(A) \times \mathcal{P}(A) \times A;$$
  4. $\cup_{\mathcal{P}(A)}$ is the unique arrow defined by the universal property of power-objects and the (monic) arrow l;

$\cap_{\mathcal{P}(A)}$ and $\setminus_{\mathcal{P}(A)}$ are constructed similarly.

- $\{\}_{n,A} : A^n \to \mathcal{P}(A)$ is the arrow constructed from $\{\}_{1,A}$ and $\cup_{\mathcal{P}(A)}$.
- let $f : A \to B$, $f^* : \mathcal{P}(A) \to \mathcal{P}(B)$ is constructed as follows:
  1. let $g = (id_{\mathcal{P}(A)} \times f) \circ \in_A : \epsilon_A \to \mathcal{P}(A) \times B$;
  2. let (k, l) be the epic/monic factorization of g;
  3. f^* is the unique arrow defined by the universal property of powerobjects and the (monic) arrow l.

The standard interpretation of the *built-in predicates* is a straightforward excercise in Category Theory, see [16]. For example, $=_A$ is constructed as the equalizer of

$$id_{A \times A}, twist : A \times A \to A \times A.$$

The intepretation of $\neq_A$ is defined as the pullback of the arrow $\neg \circ \chi_{=_A}$; where $\neg : \Omega \to \Omega$, and Ω is the classifying object of $\mathcal{T}$. The other standard predicates are interpreted similarly.