A Datatype Extension for Simple Conceptual Graphs and Conceptual Graphs Rules

Jean-François Baget

INRIA Rhône-Alpes & LIG / LIRMM
jean-francois.baget@inrialpes.fr

Abstract. We propose in this paper an extension of Conceptual Graphs that allows to use datatypes (strings, numbers, ...) for typing concept nodes. Though the model-theoretic semantics of these datatypes is inspired by the work done for RDF/RDFS, keeping sound and complete projection-based algorithms for deduction has led to strong syntactic restrictions (datatyped concept nodes of a target graph cannot be generic). This restriction, however, allows us to smoothly upgrade our extension to rules, and to introduce functional relations (that compute the value of datatyped concept nodes) while keeping sound and complete reasonings.

1 Introduction

Many extensions of simple conceptual graphs [Sow84] have answered the necessity to use datatypes (strings, integers) to type concept nodes. Among these extensions, actors [LM98] also use procedural relations between nodes. We study such an extension in this paper, the originality of our approach resides in our definition of model theoretic semantics for this datatype extension of simple conceptual graphs. As in [Hay04], values of a datatype will be interpreted by themselves and, as in SPARQL [PS06], a procedural relation between values will be evaluated to true when the result of the application of that procedure returns a result consistant with these values. We present two different semantics for these graphs: though the full semantics (where all interpretations contain all possible values) are more interesting from a knowledge representation point of view, partial semantics will lead to decidable (NP-complete) reasonings akin to the ones used in the simple conceptual graphs formalism. This projection-like mechanism allows to extend the proposed formalism with rules of form "if ... then ... ".

2 D-Graphs: Syntax

Datatype simple conceptual graphs extend simple conceptual graphs by allowing access to types and procedures from a programming language library. Instead of defining exactly this library (an important work for normalization), we decided to characterize the properties it must satisfy for our results to remain valid.

2.1 Programming Language Library

We consider a *library* $\mathcal{L}$, written in some programming language (we have adopted a "Scheme-like" terminology, borrowed from [KCE98]), allowing access to:

Strings and identifiers. Two infinite, countable, disjoint sets $\mathcal{I}$ and $\mathcal{S}$, respectively of *identifiers* and *strings*. The precise rules for forming identifiers and strings depend upon the programming language used. We consider here that an identifier is a sequence of letters, digits and extended alphabetic characters (such as !, ?, but not the double quote "); and that a string is a sequence of characters enclosed in double quotes. By example, `Number`, `Integer`, `Boolean`, `String`, `+`, `not`, `Person`, `John` are identifiers, and `"7"`, `"true"` and `"Bob"` are strings.

Types, values and procedures. Three infinite, countable, pairwise disjoint sets $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{P}$, respectively of *types* (or classes), of *values* (or objects), and of *procedures* (or methods). Though classes are objects in most object-oriented programming languages, we impose disjointness to avoid self-containing sets in models of the library $\mathcal{L}$ (see Sect. 3.1). This restriction can be easily enforced in any library by considering only a subset of the classes and objects available.

Binding identifiers to types and procedures. We consider a bijection $bind : \mathcal{I}' \subseteq \mathcal{I} \rightarrow \mathcal{T} \cup \mathcal{P}$. Identifiers bound to types are called *type identifiers* (we note $bind^{-1}(\mathcal{T}) = \mathcal{I}_T$) and identifiers bound to procedures are called *procedure identifiers* (we note $bind^{-1}(\mathcal{P}) = \mathcal{I}_P$). In the previous example, `Number`, `Integer`, `Boolean` and `String` are type identifiers, and `+` and `not` are procedure identifiers.

Functions `is-a?` and `eqv?`. We consider an equivalence relation over values (encoding that they should normally be regarded as the same) and a relation over $\mathcal{V} \times \mathcal{T}$ (encoding that a value belongs to, is an instance of, a type). They are respectively computed by the functions `is-a?` : $\mathcal{V} \times \mathcal{T} \rightarrow \{\#t, \#f\}$ and `eqv?` : $\mathcal{V} \times \mathcal{V} \rightarrow \{\#t, \#f\}$. Note that these functions are not procedures of $\mathcal{P}$.

Specification 1 (Booleans). *The library $\mathcal{L}$ must satisfy these properties:*

– `Boolean` *is a type identifier of $\mathcal{I}$, and we note $B =$ `bind`(`Boolean`);*
– *$\#t$ and $\#f$ are two values of $\mathcal{V}$ (they correspond to "true" and "false");*
– `is-a?`$(\#t, B) = \#t$ *and* `is-a?`$(\#f, B) = \#t$*;*
– $\forall x \in \mathcal{V}$, `is-a?`$(x, B) = \#t \Rightarrow$ (`eqv?`$(x, \#t) = \#t$ *or* `eqv?`$(x, \#f)) = \#t$.

Specification 2 (About `eqv?`). *Let x and y be two equivalent values of $\mathcal{V}$ (*i.e.* such that* `eqv?`$(x, y) = \#t$*). Then $\forall t \in \mathcal{T}$,* `is-a?`$(x, t) = \#t \Leftrightarrow$ `is-a?`$(y, t) = \#t$.

Signature and application of a procedure. Two mappings `isig` : $\mathcal{P} \rightarrow \mathcal{T}^+$ and `osig` : $\mathcal{P} \rightarrow \mathcal{T}$ respectively define the *input* and *output signatures* of a procedure p. The *arity* of p is the size of `isig`(p). A procedure can be applied to values in order to obtain another value, Thanks to the *partial* function `apply` : $\mathcal{P} \times \mathcal{V}^+ \rightarrow \mathcal{V}$. The functions `isig`, `osig` and `apply` are not procedures of $\mathcal{P}$.

Specification 3 (Signature). *Let $p \in \mathcal{P}$ be a procedure,* `isig`$(p) = (t_1, \ldots, t_k)$ *and* `osig`$(p) = t$*. Then $\forall (v_1, \ldots, v_k) \in \mathcal{V}^k$, with* `is-a?`$(v_i, t_i) = \#t$ *for $1 \leq i \leq k$,* `apply`$(p, v_1, \ldots, v_k)$ *is defined and returns a value v such that* `is-a?`$(v, t) = \#t$*. If $\exists\ 1 \leq i \leq k$ such that* `is-a?`$(v_i, t_i) = \#f$, `apply`$(p, v_1, \ldots, v_k)$ *is not defined.*

Note that `apply` is a (partial) function in a mathematical sense: the value it may return is entirely determined by the parameters p and v_i.

Specification 4 (Equivalent parameters). *Let* $p \in \mathcal{P}$ *be a procedure,* $\texttt{isig}(p) = (t_1, \ldots, t_k)$ *and* $\texttt{osig}(p) = t$. *Then* $\forall (v_1, \ldots, v_k) \in \mathcal{V}^k$, $\forall (v'_1, \ldots, v'_k) \in \mathcal{V}^k$, *with* $\texttt{is-a?}(v_i, t_i) = \#t$ *and* $\texttt{eqv?}(v_i, v'_i) = \#t$ *for* $1 \leq i \leq k$, $\texttt{apply}(p, v_1, \ldots, v_k) = v$ *and* $\texttt{apply}(p, v'_1, \ldots, v'_k) = v' \Rightarrow \texttt{eqv?}(v, v') = \#t$ *(applying a procedure on equivalent parameters must yield an equivalent result).*

External representation, read and write. A partial function $\texttt{read} : \mathcal{I}_T \times \mathcal{S} \rightarrow \mathcal{V}$ builds, when possible, a value v belonging to a type t from the type identifier bound to t and a string s (s is then called an *external representation* of v for the type t). The partial function $\texttt{write} : \mathcal{I}_T \times \mathcal{V} \rightarrow \mathcal{S}$ is used to obtain one of these external representations (called the *canonical representation*) from a value. As before, none of these functions are procedures of $\mathcal{P}$.

Specification 5 (External representations). *The following properties must be satisfied by the functions* `read` *and* `write`*:*

- *Let* n *be a type identifier of* $\mathcal{I}_T$ *and* s *be a string of* $\mathcal{S}$. *Then* $\texttt{read}(n, s)$ *is defined and returns a value* $v \Rightarrow \texttt{is-a?}(v, \texttt{bind}(n)) = \#t$.
- *Let* t *be a type of* $\mathcal{T}$ *and* v *be a value of* $\mathcal{V}$. *Then* $\texttt{write}(t, v)$ *is defined and returns a value* s *if and only if* $\texttt{is-a?}(v, t) = \#t$. *In that case, we have* $\texttt{equiv?}(v, \texttt{read}(\texttt{bind}^{-1}(t), s)) = \#t$.
- *if* v *and* v' *are two values of* $\mathcal{V}$ *such that* $\texttt{eqv?}(v, v') = \#t$ *and* $\texttt{is-a?}(v, t) = \#t$, *then* $\texttt{write}(t, v) = \texttt{write}(t, v')$.

2.2 Vocabulary

The vocabulary (or canon, support) is the structure traditionally used to encode the ontological knowledge in the conceptual graphs formalism.

Definition 1 (Vocabulary). *A* vocabulary *is a tuple* $((T_C, \leq_C), (T_R^1, \leq_1), \ldots, (T_R^k, \leq_k))$ *where* $T_C, T_R^1, \ldots, T_R^k$ *are pairwise disjoint sets of identifiers (respectively of* concept types*,* relation types *of arity 1 to k, and* $\leq_C, \leq_1, \ldots, \leq_k$ *are partial orders on these sets.*

Unlike concept types, type identifiers of $\mathcal{L}$ are not explicitely ordered by a subtype relation. As shown in Sect. 3.1, type identifiers are interpreted by sets that are uniquely determined by $\mathcal{L}$. Inclusion of these sets thus determines the subtype relation. However, expliciting this relation could cause a difference (in case of a bad programming of $\mathcal{L}$) between the interpretations of type identifiers and the interpretation of the (redundant) subtype relation.

Finally, we want to be able to decide if an identifier is a type of the vocabulary or a type or procedure identifier in the library:

Definition 2 (Compatibility). *A vocabulary* V *and a library* $\mathcal{L}$ *are* compatible *if their sets of identifiers are disjoint (*i.e. $(T_C \cup T_R^1 \cup \ldots T_R^k) \cap (\mathcal{I}_T \cup \mathcal{I}_P) = \emptyset$*).*

Note that it is always possible to obtain compatibility between the vocabulary and the library, up to a renaming of identifiers.

2.3 D-Graphs

Definition 3 (D-graph). *A* datatyped simple conceptual graph *(*D-graph*), defined on a vocabulary $\mathcal{V}$ and a compatible library $\mathcal{L}$, is a tuple $G = (E, R, \gamma, \tau, \mu)$ where E and R are two disjoint sets (respectively of* entities *and* relations*), $\gamma : R \rightarrow E^+$ maps each relation to a tuple of entities (its* arguments*), if $\gamma(r) = (e_1, \ldots, e_k)$ we say that* degree$(r) = k$*, and τ and μ are mappings s.t.:*

- $\forall e \in E, \tau(e)$ *(the* type *of e)is either a concept type of T_C (as usually done for simple CGs, e is then called a* concept node*) or a type identifier of $\mathcal{I}_T$ (e is then called a datatype node);*
- $\forall r \in R, \tau(r)$ *(the* type *of r) is either a relation type of arity* degree(r) *(as usually done for simple CGs, r is then called a* standard relation*) or a procedure identifier n of $\mathcal{I}_P$ such that* $|\texttt{isig}(n)| =$ degree$(r) - 1$ *(in that case, r is called a* computed relation*). If r is a computed relation, then all its arguments must be datatype nodes.*
- $\forall e \in E, \mu(e)$ *(the* marker *of e) can be the generic symbol $*$, which is not an identifier (e is characterized as* generic*) or, if e is a datatype node, a string of $\mathcal{S}$, or, if e is a concept node, an identifier of $\mathcal{I}$ that is not bound in $\mathcal{L}$ and is not a type of V (in both cases, the node e is characterized as* individual*).*

A D-graph $G = (E, R, \gamma, \tau, \mu)$ is graphically represented as follows: as for simple CGs, concept nodes are represented by rectangles and standard relations by ovals; we represent datatype nodes by parallelograms and computed relations by diamonds (as a reminder of actors). If e is an individual entity, then we write $\tau(e) : \mu(e)$ inside the shape representing e, otherwise we only write $\tau(e)$. If r is a relation, we write $\tau(r)$ inside the shape representing r. If $\gamma(r) = (e_1, \ldots, e_k)$, we draw a line between the shape representing r and each of the shapes representing the e_j. If r is a standard relation, these lines are numbered from 1 to k, according to the place of the entity in $\gamma(r)$, if r is a computed relation, then the lines are oriented from "input entities" to relation and relation to "output entity", and only the first $k - 1$ arrows are numbered. Fig. 1 represents such a D-graph. It asserts that it exists a person whose age is greater than 27, the age of Bob. It can be simplified by omitting the output entity with marker `"true"` of a computed relation whose type is a procedure identifier bound to a procedure with output signature bound by `Boolean`.

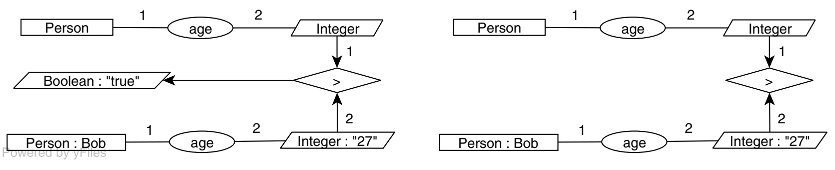

Fig. 1. Two graphical representations of a D-graph G

3 D-Graphs: Semantics

3.1 Interpretations and Models

We extend the usual model-theoretic semantics of simple conceptual graphs (see [ABC06]) to take into account the programming language library. We first define interpretations, then the conditions under which an interpretation is a model of a vocabulary, a library, or a D-graph.

Definition 4 (Interpretation). *Let $S \subseteq \mathcal{S}$ be a set of strings and $\mathcal{I}$ be the set of identifiers. An* interpretation *of $(\mathcal{I}, S)$ is a triple $I = (\Delta, \iota, \sigma)$ where Δ is a non empty set called the* interpretation domain*, and ι and σ are the* interpretation functions*; $\iota : \mathcal{I} \to \Delta \cup 2^{\Delta} \cup 2^{\Delta \times \Delta} \cup \ldots \cup 2^{\Delta^k}$ maps each string to an element of the domain or a set of tuples of elements of the domain; and $\sigma : \mathcal{I} \times S \to \Delta$ is a partial mapping. I is said* full *if $S = \mathcal{S}$.*

Definition 5 (Model of a vocabulary). *Let $S \subseteq \mathcal{S}$ be a set of strings, and $\mathcal{I}$ be the set of identifiers. An interpretation $I = (\Delta, \iota, \sigma)$ of $(\mathcal{I}, S)$ is a model of the vocabulary $V = ((T_C, \leq_C), (T_R^1, \leq_1), \ldots, (T_R^k, \leq_k))$ (we note $(\mathcal{I}, S) \models V$) iff:*

- *if c is a concept type of T_C, then $\iota(c) \in 2^{\Delta}$;*
- *if r is a relation type of T_R^i, then $\iota(r) \in 2^{\Delta^i}$;*
- *if t and t' are two concept or relation types such that $t \leq t'$ (where $\leq$ can be $\leq_C$ or one of the $\leq_i$), then $\iota(t) \subseteq \iota(t')$;*

Note that this definition is a generalization of the usual model-theoretic semantics of the vocabulary of CGs: $I = (\Delta, \iota, \sigma)$ is a model of V if and only if (Δ, ι) is a model of V (in the usual sense of [ABC06]).

Definition 6 (Model of a library). *Let $S \subseteq \mathcal{S}$ be a set of strings, and $\mathcal{I}$ be the set of identifiers. An interpretation $I = (\Delta, \iota, \sigma)$ of $(\mathcal{I}, S)$ is a model of the library $\mathcal{L}$ (and we note $(\mathcal{I}, S) \models \mathcal{L}$) if and only if:*

- *$\forall s \in S$, $\forall i \in \mathcal{I}$, $\sigma(i, s)$ is defined if and only if i is a type identifier of $\mathcal{I}_T$ and $\mathtt{read}(i, s) = v$ is defined. In that case, $\sigma(i, s) = \mathtt{read}(i, \mathtt{write}(\mathtt{bind}(i), v))$ (the value associated with the canonical representation of all equivalent ones).*
- *$\forall t \in \mathcal{I}_T, \iota(t) = \{\delta \in \Delta \mid \mathtt{is\text{-}a?}(\delta, \mathtt{bind}(t)) = \#t\}$.*
- *$\forall p \in \mathcal{I}_P$, with $\mathtt{isig}(\mathtt{bind}(p)) = (t_1, \ldots, t_k)$, $\iota(p) = \{(\delta_1, \ldots, \delta_k, \delta) \in \Delta^{k+1}$ s.t. $\mathtt{apply}(p, \delta_1, \ldots, \delta_k) = \delta\}$.*

Definition 7 (Models of a D-graph). *Let $S \subseteq \mathcal{S}$ be a set of strings, and $\mathcal{I}$ be the set of identifiers. An interpretation $I = (\Delta, \iota, \sigma)$ of $(\mathcal{I}, S)$ is a model of the D-graph $G = (E, R, \gamma, \tau, \mu)$ (and we note $(\mathcal{I}, S) \models G$) if and only if there exists a mapping $\alpha : E \to \Delta$ (alpha is called a* proof *of G in I) such that:*

- *for each individual concept node $e \in E$, $\alpha(e) = \iota(\mu(e)) \in \iota(\tau(e))$;*
- *for each individual datatype node $e \in E$, $\alpha(e) = \sigma(\tau(e), \mu(e))$;*
- *for each generic node $e \in E$, $\alpha(e) \in \iota(\tau(e))$;*
- *for each relation $r \in R$, $\gamma(r) = (e_1, \ldots, e_k) \Rightarrow (\alpha(e_1), \ldots, \alpha(e_k)) \in \iota(\tau(r))$.*

Note that if a D-graph is a simple conceptual graph (it has only concept nodes and standard relations), these constraints correspond to the usual constraints on models of a simple conceptual graph.

3.2 Full vs. Partial Semantics

Definition 8 (Satisfiability, validity, consequence). *Let G and G' be two D-graphs defined over a vocabulary V and a compatible library $\mathcal{L}$. An interpretation I of $(\mathcal{I}, S)$* covers *G if every string labelling nodes of G belong to S. Then:*

– *G is* partially *(resp.* fully*)* satisfiable *if there exists an interpretation (resp. a full interpretation) I covering G such that $I \models V$, $I \models \mathcal{L}$ and $I \models G$. Otherwise it is said* partially *(resp.* fully*)* unsatisfiable.

– *G is* partially *(resp.* fully*)* valid *if for any interpretation (resp. full interpretation) I covering G, $I \models V$ and $I \models \mathcal{L} \Rightarrow I \models G$. Otherwise it is said* partially *(resp.* fully*)* invalid.

– *G is a* partial *(resp.* full*)* consequence *of G' if for any interpretation (resp. full interpretation) I covering G and G', $I \models V$, $I \models \mathcal{L}$ and $I \models G' \Rightarrow I \models G$. We note $G' \vdash G$ (resp $G' \Vdash G$).*

With simple conceptual graphs, there is no distinction between full and partial semantics (since there is no string). Every graph is satisfiable, and the only valid graph is the empty graph. Things are more complicated with D-graphs.

Property 1. Let G and G' be two D-graphs defined over a vocabulary V and a compatible library $\mathcal{L}$. Then (since full interpretations are interpretations):

1. G is fully satisfiable $\Rightarrow$ G is partially satisfiable;
2. G is partially valid $\Rightarrow$ G is fully valid;
3. $G' \vdash G \Rightarrow G' \Vdash G$.

There are more consequences with full semantics, and they are relevant (according to the library). To explain the differences between these semantics, let us point out that full interpretations interpret all strings of $\mathcal{S}$, and since all values of $\mathcal{V}$ have an external representation, full interpretations impose $\mathcal{V}$ (more precisely equivalence classes for `eqv?`) to be a subset of the domain of interpretation Δ. With partial semantics, some values may not belong to Δ.

Let us consider the D-graphs of Fig. 2. The empty D-graph G_3 is both partially valid and fully valid. The D-graph G_2 (there exists an integer) is fully valid because Δ contains, by example, the value whose external representation is "7" and α can map the node of G_2 to that value. However, G_2 is not partially valid, since Δ may not contain any value v such that `is-a?`$(v,$ `bind`$($`Integer`$)) = \#t$. The D-graph G_1 ("7 is an integer") is both partially valid and fully valid: even if

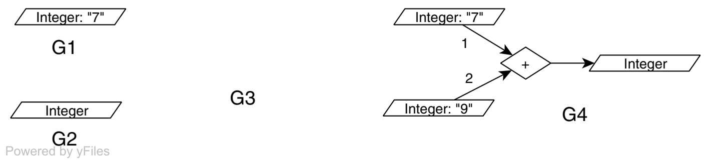

Fig. 2. Examples of D-graphs

an interpretation is not full, it must cover G_1, and thus the value whose external representation is "7" (a string of G_1) belongs to Δ. Finally, the D-graph G_4 ("there exists an integer that is the sum of 7 and 9") is fully valid, but is not partially valid (the value represented by "16" may not belong to Δ).

Property 2. Let G and G' be two D-graphs defined over a vocabulary V and a compatible library $\mathcal{L}$. Then G is fully satisfiable $\Leftrightarrow$ G is satisfiable.

Proof. Thanks to Prop. 1, we only have to prove that G satisfiable $\Rightarrow$ G fully satisfiable. Since $G = (E, R, \gamma, \tau, \mu)$ is satisfiable, there exists a partial interpretation $I = (\Delta, \iota, \sigma)$ and a mapping $\alpha : E \to \Delta$ that satisfies the constraints of Def. 7. The partial interpretation I can be "completed" to a full interpretation $I' = (\Delta', \iota', \sigma')$ (by adding all values of $\mathcal{V}$ to Δ, while respecting the constraints of Def. 6), and $\alpha : E \to \Delta \subseteq \Delta'$ still satisfies all constraints of Def. 7. $\square$

As seen in these examples, full semantics are more interesting than partial semantics. However, they are a tremendous source of computational complexity.

4 Inference Mechanisms for D-Graphs

4.1 Full Validity of D-Graphs

Though we are mainly interested in computating consequence, we discuss the problems linked to computing full validity (a particular case of full consequence, since G is fully valid iff G is the full consequence of the empty D-graph).

Some examples of full validity computation. Let us first consider an "easy example", the D-graph G_4 of Fig. 2. To prove that G_4 is partially valid, we have to prove that there is an integer that is the sum of 7 and 9: we only have to use the library to compute $\texttt{apply}(\texttt{bind}(+), \texttt{read}(\texttt{Integer}, "7"), \texttt{read}(\texttt{Integer}, "9"))$. If $\texttt{bind}(+)$ is decidable, then this method is decidable.

Let us now consider the D-graph G_1 of Fig. 3 (there exists an integer that, added to 7, has result 9). To prove its full validity, we cannot use `apply` as in the previous example. And since the library is a "black box", we cannot guess that this integer can be obtained by substracting 7 to 9. So we have to try to find a value v in $\mathcal{V} \subseteq \Delta$ such that $\texttt{apply}(\texttt{bind}(+), \texttt{read}(\texttt{Integer}, "7"), v) = \texttt{read}(\texttt{Integer}, "9")$. Though we do not have a direct access to values of $\mathcal{V}$ ("black box", again), we can enumerate this countable, infinite set by enumerating all strings s of $\mathcal{S}$ and computing the value, if it exists $\texttt{read}(\texttt{Integer}, s)$. As soon as we enumerate the string "2", we can assert that G_1 is fully valid.

Finally, let us now try to prove that the D-graph G_2 (there exists an integer that is the result of itself added to 7) of Fig. 3 is fully valid. As in the previous example, we enumerate strings until a satisfying value is found, but, this time, no such value exists: the algorithm will run forever...

Characterization of full validity

Definition 9 (Pure D-graph). *A* pure D-graph *is a D-graph whose only entities are datatype nodes and whose only relations are computed relations.*

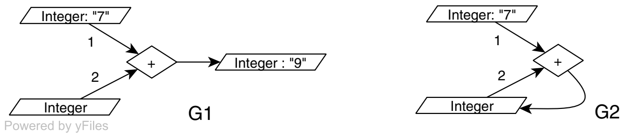

Fig. 3. Examples of D-graphs

Property 3. Only pure D-graphs can be valid or fully valid.

Proof. If a D-graph G is not pure, we can build a full interpretation that is a model of V and $\mathcal{L}$, but not a model of G: build a full interpretation that is a model of $\mathcal{L}$, then for each concept or relation type t of V, define $\iota(t) = \emptyset$. $\square$

Theorem 1 (Characterization of full validity). *Let G be a pure D-graph defined over a vocabulary V and a compatible library $\mathcal{L}$. Then G is fully valid if and only if there exists a (pure) D-graph G' obtained from G by replacing all generic markers in G by strings of $\mathcal{S}$ such that:*

- *for each entity e of G', $f(e) = \texttt{read}(\tau(e), \mu(e))$ is defined;*
- *for each relation r of G', with arguments $\gamma(r) = (e_1, \ldots, e_k, e)$, we have $\texttt{eq?}(\texttt{apply}(\texttt{bind}(\tau(r)), f(e_1), \ldots, f(e_k)), f(e)) = \#t$.*

This is a generalization to any pure D-graph of the algorithm explained for the D-graphs of Fig. 3. However, this algorithm requires the enumeration of all tuples of strings in $\mathcal{S}^k$, where k is the number of generic entities in G. As suggested by the first example (D-graph G_4 of Fig. 2), there is a way to reduce the number of strings to be tested.

Definition 10 (Resolving a D-graph). *Let G be a D-graph, and r be a* determined relation *of G,* i.e. *a computed relation of G whose input arguments $e_1, \ldots, e_k$ are individual datatype nodes and whose output argument is a generic datatype node e. Then resolve(G, r) is the graph obtained from G by replacing the generic marker of e by the string $\texttt{write}(\texttt{bind}(\tau(r)), \texttt{apply}(\texttt{bind}(\tau(r)), \texttt{read}(\tau(e_1), \mu(e_1)), \ldots, \texttt{read}(\tau(e_k), \mu(e_k))))$.*

Property 4. G and $G' = \textit{resolve}(G, r)$ are fully equivalent ($G \Vdash G'$ and $G' \Vdash G$), G is a consequence of G', but G' is not necessarily a consequence of G.

As an immediate consequence, we can compute full validity of a D-graph G by applying, as long as there exists a determined relation, a succession of resolve, then compute full validity of the obtained D-graph G' (where G' is called a resolve of G). Finally, if G can be resolved into a D-graph where all datatype nodes are individual, and if all computed relations are decidable, full validity is decidable.

4.2 Satisfiability of D-Graphs

Fig. 4 shows two D-graphs that are not satisfiable. The D-graph G_1 expresses that "the sum of 2 and 7 is 11" and the graph G_2 that "there is an integer that is

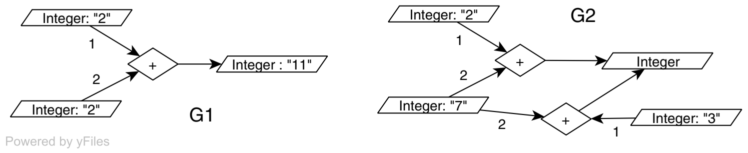

Fig. 4. Examples of D-graphs

both the sum of 2 and 7 and the sum of 3 and 7". This latter example highlights the main difference between our approach and the one used in conceptual graphs actors: though actors consider it as a "convergence problem", our semantics-based approach defines this D-graph as unsatisfiable.

Property 5. A D-graph G is satisfiable iff its pure D-graph restriction G' (obtained by removing all concept nodes and all standard relations) is satisfiable.

Proof. ($\Rightarrow$) is immediate, the restriction of a proof of G in I is a proof of G' in I. For ($\Leftarrow$), let us now consider a proof α of G' in an interpretation (Δ, ι, σ). Then α is a proof of G' in $(\mathcal{V}, \iota, \sigma)$. We build a proof α' of G in $I' = (\Delta', \iota', \sigma')$ where Δ' is the union of $\mathcal{V}$ and of concept nodes of G, and define, for all concept types t, $\iota(t) = \Delta'$, and, for all relation types r of arity k, $\iota(r) = \Delta'^k$. □

Property 6. A pure D-graph G is either fully valid or unsatisfiable.

Proof. The truth value of a pure D-graph is only determined by the library. □

We conclude that, as for full validity, satisfiability is decidable when the D-graph G can be resolved into a D-graph G' where all datatype nodes are individual, and computed relations are decidable.

4.3 Partial and Full Consequences of D-Graphs

D-graphs consequence is the problem we are most interested in: is a D-graph Q (a query) consequence of a D-graph, or a set of D-graphs (the knowledge base), or is there an answer to Q in a given knowledge base ?

Problems with full consequence. A D-graph G is fully valid iff it is the full consequence of the empty D-graph $\emptyset$. So a sound and complete algorithm for full consequence must tackle with full validity of a query. However, we have seen (in Sect. 4.1) that a condition to be able to compute full validity was that G could be resolved into a D-graph without any generic datatype node. To compute full consequence, this condition should be translated to a restriction on queries, which we believe to strong: the only generic datatype nodes would be determined by the query itself, a useless feature in the query language.

An extension of projection. As discussed above, full semantics would lead either to undecidable calculus of consequence, or unacceptable syntactic restrictions for queries. This is why we focus on partial semantics of D-graphs. Though these semantics are less interesting, our motivation is to be able to compute consequence using a mechanism akin to simple conceptual graphs projection; this mechanism will be called D-projection.

Definition 11 (D-projection). *Let us consider two D-graphs defined over a vocabulary V and a compatible library $\mathcal{L}$, $G = (E_G, R_G, \gamma_G, \tau_G, \mu_G)$ and $H = (E_H, R_H, \gamma_H, \tau_H, \mu_H)$. A* D-projection *from H into G is a mapping π from the entities of H onto the entities of G such that (where we note $\rho(H, c) =$ $\texttt{read}(\tau_H(c), \mu_H(c))$):*

- *for each concept node c of H, $\tau_G(c) \leq_C \tau_H(c)$;*
- *for each individual concept node c of H, $\mu_H(c) = \mu_G(\pi(c))$;*
- *for each individual datatype node c of H, $\texttt{eq?}(\rho(H, c), \rho(G, \pi(c))) = \#t$;*
- *for each generic datatype node c of H, $\texttt{eq?}(\texttt{read}(\tau_H(c), \mu_G(\pi(c))), \rho(G, \pi(c))) = \#t$;*
- *for each standard relation r of H, s.t. $\gamma_H(r) = (e_1, \ldots, e_k)$, there is a standard relation r' of G, with $\tau_G(r') \leq_k \tau_H(r)$ and $\gamma_G(r') = (\pi(e_1), \ldots, \pi(e_k))$;*
- *for each computed relation r of H, s.t. $\gamma_H(r) = (e_1, \ldots, e_k, e)$, $\texttt{eq?}(\rho(G, \pi(e)),$ $\texttt{apply}(\texttt{bind}(\tau_H(r)), \rho(G, \pi(e_1)), \ldots, \rho(G, \pi(e_k)))) = \#t$.*

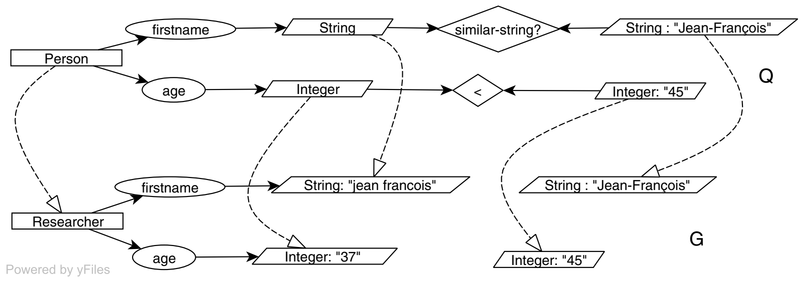

Fig. 5. An example of D-projection

Fig. 5 features an example of D-projection. Dashed arrows represent the mapping π from the query Q (is there a person whose first name is a string similar to "Jean-François" and whose age is lesser than 45), where `similar-string?` is bound to a predicate returning $\#t$ when the distance between two strings is small, to a knowledge base G. The two individual datatype nodes marked `"Jean-François"` and `"45"` have to be present in G for a D-projection to exist.

Property 7 (Soundness). If there is a D-projection from Q into G, then Q is a partial consequence (and thus full consequence) of G.

However, completeness of D-projection is not achieved in the general case. Let us consider some causes of incompleteness:

1. *Normality:* as for simple conceptual graphs, the knowledge base G must be put into its equivalent normal form;

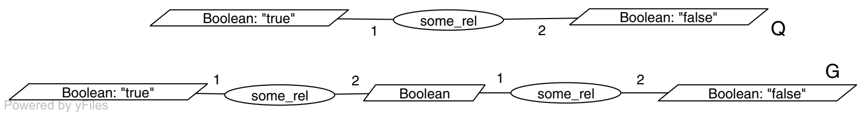

Fig. 6. There is no D-projection from Q into G

2. *Assertion of individual datatype nodes:* as seen in the previous example (Fig. 5), all necessary values must be represented by individual datatype nodes in the knowledge base G;
3. *Satisfiability:* If the knowledge base G is not satisfiable, then anything is consequence of G, and in particular queries that have no projection into G;
4. *Branching:* [LM07] show that adding atomic negation to simple conceptual graphs leads to incompleteness of projection. We adapt their counterexample to D-graphs: there is no D-projection from Q to G in Fig. 6, but there is one whether we consider that the generic node of G represents $\#t$ or $\#f$.

The first cause of incompleteness is handled by updating the normalization operation, the second by adding individual datatype nodes of the query to the knowledge base. However, to answer to the third and fourth causes of incompleteness, we need to adopt a severe restriction.

Definition 12 (Normalization). *A D-graph G is put into its normal form nf(G) by fusioning all concept nodes having the same individual marker, and all datatype nodes e e' such that* $\texttt{eq?}(\texttt{read}(\tau(e), \mu(e)), \texttt{read}(\tau(e'), \mu(e'))) = \#t$.

Theorem 2 (Soundness and completeness). *Let G and Q be two D-graphs over a vocabulary V and a compatible library $\mathcal{L}$. Let us consider that G can be resolved into a D-graph G_r without generic datatype node. Then Q is a partial consequence of G if and only if G is unsatisfiable or there is a D-projection into the D-graph G' that is the normal form of the disjoint union of G_r and of all individual datatype nodes of Q.*

Because of lack of space, we do not include the proof of this theorem in this paper. The proof framework used is the one in [Bag05].

We have characterized partial consequence of D-graphs as a kind of graph homomorphism, akin to projection. To do so, we gave up the more interesting full semantics and restricted allowed target D-graphs (representing the knowledge base) to the ones that can be resolved into D-graphs without generic datatype nodes (a slight improvement with respect to databases that forbid generic nodes). As an added benefit, this characterization as a graph homomorphism allows us to extend the proposed formalism to datatyped conceptual graphs rules.

5 Datatyped Conceptual Graphs Rules (D-Rules)

We present in this section an extension of D-graphs to rules, as done for simple conceptual graphs [Sal98]. We present their syntax, their partial semantics, and briefly show how the derivation mechanism of rules can be updated for D-rules.

5.1 Syntax

Definition 13 (D-rule). *Let V be a vocabulary, and $\mathcal{L}$ be compatible library. A* datatyped conceptual graph rule *over V and $\mathcal{L}$ is a D-graph R where one partial subgraph of R is identified as the* hypothesis*. The other part of R (its not necessarily a D-graph) is called its* conclusion*.*

In the rest of this paper we will represent D-rules in the same way as D-graphs, but the shapes representing entities and relations of the hypothesis will be shaded in gray. The graph R of Fig. 7 represents such a D-rule, asserting that "for every even integer x, there exists an integer obtained by dividing x by two".

5.2 Semantics

To update the partial semantics of D-graphs to D-rules, we have to define under which conditions an interpretation is a model of a D-rule, as well as the (partial) consequence problem with D-rules:

Definition 14 (Models of a D-rule). *Let $S \subseteq \mathcal{S}$ be a set of strings, and $\mathcal{I}$ be the set of identifiers. An interpretation $I = (\Delta, \iota, \sigma)$ of $(\mathcal{I}, S)$ is a model of a D-rule R if every proof α that I is a model of the hypothesis of R can be extended to a proof α' that I is a model of R.*

Definition 15 (D-rules (partial) consequence). *Let G and G' be two D-graphs defined over a vocabulary V and a compatible library $\mathcal{L}$, and $\mathcal{R}$ be a set of D-rules defined over V and $\mathcal{L}$. We say that G' is a partial consequence of G and $\mathcal{R}$ if all (partial) interpretations that are models of $V, \mathcal{L}, G$ and of all D-rules in $\mathcal{R}$ is also a model of G'. We note $G, \mathcal{R} \vdash G'$.*

5.3 D-Derivation

With simple conceptual graphs rules, consequence can be computed with forward chaining: finding a sequence of transformations (rule applications) of the target graph G such that G' can be projected into G. With D-graphs, we must first point out that, to be able to find a D-projection corresponding to the semantics into any of these derived graphs, these D-graphs must be resolved into D-graphs without generic datatype nodes. We will then restrict ourselves to D-rules that ensure that property to be true at each step of the derivation.

Definition 16 (Resolvable D-rule). *A D-rule is* resolvable *iff all generic datatype nodes in its conclusion are the last argument of a computed relation.*

Definition 17 (Rule application). *Let G be a D-graph and R be a D-rule. R is said* applicable *to G if there exists a D-projection π from the hypothesis of R into G. In that case, the application of R on G following π is the D-graph obtained by making the disjoint union of G and of the conclusion of R then, for each relation r of the conclusion such that $\gamma_i(r) = e$ is an entity of the hypothesis, replace $\gamma_i(r)$ by $\pi(e)$. Finally, we resolve the obtained graph (see Fig. 7).*

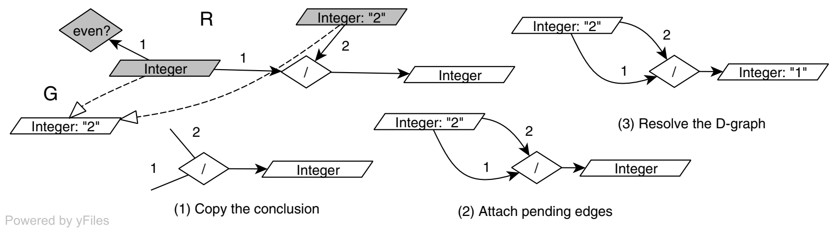

Fig. 7. Applying a D-rule on a D-graph

Property 8. The application of a resolvable D-rule on a D-graph that can be resolved into a D-graph without generic datatype nodes is also a D-graph that can be resolved into a D-graph without generic datatype nodes.

A second problem, that must be handled during derivation, is that the derivation process may generate D-graphs that are not satisfiable.

Theorem 3 (Soundness and completeness). *Let V be a vocabulary, $\mathcal{L}$ be a library, G and Q be two D-graphs and $\mathcal{R}$ be a set of rules over V and $\mathcal{L}$. Then $G, \mathcal{R} \vdash Q$ if and only if there exists a D-graph G' obtained from the disjoint union of G and of all individual datatype nodes in $\mathcal{R}$ and Q by a sequence of rule applications or normalizations such that either G' is unsatisfiable or there exists a D-projection from Q into G'.*

The proof in [BS06] is easily adapted to D-rules. As deduction with simple conceptual graphs rules, partial consequence with D-rules is semi-decidable.

5.4 Computing Factorial with D-Rules

The rule R_1 in Fig. 8 expresses that "if $y = x + 1$ and the *fact* of x is z, then the *fact* of y is $y * z$" (*fact* is a standard relation). With full semantics, this rule is sufficient to compute factorials: if the knowledge base contains the D-graph "the *fact* of 1 is 1", then all queries of form "what is the *fact* of an individual integer n?" can be correctly answered.

However, with partial semantics, nothing ensures that the values required for the recursive computation are in the interpretation domain. One way to overcome this limitation is to force these values to be interpreted, by generating

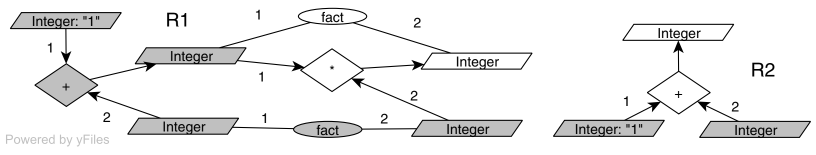

Fig. 8. D-rules that compute the factorial procedure

all integers. This is the role of the rule R_2 of Fig. 8. These two rules can be compared to the ones in [Min98] to compute factorial.

6 Conclusion

We have proposed in this paper an extension of simple conceptual graphs [Sow84] and of rules [Sal98] using datatypes (as in [Hay04]) and procedural relations (similar to the actors of [LM98])). We have proposed two model-theoretic semantics, based upon a library written in some programming language.

While full semantics are more interesting from a KR point of view, we can compute partial consequence with D-graphs and D-rules (provided that the knowledge base does not contain generic datatype nodes). The last example provides a hint on ways to overcome the limitations of partial semantics.

Further work will consist in an implementation of D-graphs and D-rules (on top of CoGiTaNT), and optimizations of reasonings (inspired by [BS06]).

References

[ABC06] Aubert, J.-P., Baget, J.-F., Chein, M.: Simple conceptual graphs and simple concept graphs. In: Schärfe, H., Hitzler, P., Øhrstrøm, P. (eds.) ICCS 2006. LNCS (LNAI), vol. 4068, pp. 87–101. Springer, Heidelberg (2006)

[Bag05] Baget, J.-F.: Rdf entailment as a graph homomorphism. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 82–96. Springer, Heidelberg (2005)

[BS06] Baget, J.-F., Salvat, E.: Rules dependencies in backward chaining of conceptual graphs rules. In: Schärfe, H., Hitzler, P., Øhrstrøm, P. (eds.) ICCS 2006. LNCS (LNAI), vol. 4068, pp. 102–116. Springer, Heidelberg (2006)

[Hay04] Hayes, P.: RDF Semantics. Technical report, W3C Recommendation (2004)

[KCE98] Kelsey, R., Clinger, W., Rees, J. (eds.): Revised^5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices 33(9), 26–76 (1998)

[LM98] Lukose, D., Mineau, G.W.: A comparative study of dynamic conceptual graphs. In: Proc. of KAW'98 (1998)

[LM07] Leclere, M., Mugnier, M.-L.: Some algorithmic improvments for the containment problem of conjunctive queries with negation. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 401–418. Springer, Heidelberg (2006)

[Min98] Mineau, G.W.: From actors to processes: The representation of dynamic knowledge using conceptual graphs. In: Mugnier, M.-L., Chein, M. (eds.) ICCS 1998. LNCS (LNAI), vol. 1453, pp. 65–79. Springer, Heidelberg (1998)

[PS06] Prudhommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, W3C Working Draft (2006)

[Sal98] Salvat, E.: Theorem proving using graph operations in the conceptual graphs formalism. In: Proc. of ECAI'98, pp. 356–360 (1998)

[Sow84] Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, London, UK (1984)