- the client receives the result and then associates it to its `Remoteval`. To do it, we must enable a listening process that waits for a response packet from any server and then unpacks the value, associating it to the corresponding `Remoteval`.

- we do not worry about packets that arrive to the client until the `val` function is applied. At this moment, the client process must wait for the packet corresponding to the `Remoteval` on which `val` has been applied. While this packet does not arrive,

  the client process unpacks other packets which were previously received and associates them to their own `Remotevals` instead of remaining inactive. This approach needs a process to store the packets coming from the servers until they are processed by `val`.

The latter approach is more suitable than the former to our problem because the subsystem used for the interprocess communication (PVM) offers some facilities to store messages in its internal buffers, until the target process wants to read them. Thus, we can use the PVM daemon to store packed values until they can be processed by the `val` function.

In both cases, we must know which `Remoteval` corresponds to a given response and how to access this `Remoteval`, since it is not necessary passed as an argument to the `val` function. The former problem can be easily solved: it is enough to associate each request with an unique identifier which is put both in the `Remoteval` and in the packet sent to the server, which will send back the response packet with this identifier. In this way, a possible implementation of the ADT may be:

```
type 'a Remoteval = Val of bool * int * 'a;;
                         (* flag * id * result-value *)
```

where `bool` is a flag that points out if the result has already arrived, `int` is the identifier of the request and `'a` is the response value, if it has already arrived. This datatype, implemented at low level, can modify itself (a *mutable* datatype) and holds an inconsistent value in `'a` during the period of time that the flag has a `false` value. The access to the field `'a` is restricted only to the function `val`, guaranteeing the transparency of these changes in the structure.

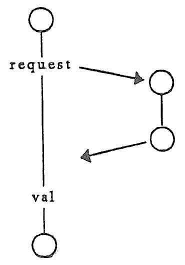

Figure 1: Success

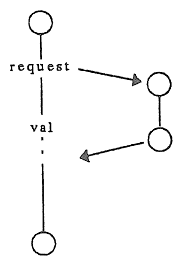

Figure 2: Fault

The latter problem can be solved by using a list which has all the `Remotevals` whose results have not arrived yet to the client. We should include this list into the data involved in the Caml Light garbage collector mechanism, so that we can assure that this list will always have valid references to the `Remotevals` in evaluation all the time.

It will be enough to insert the `Remoteval` into the list when the client makes the request in which it is created. On receiving the response, the `Remoteval` with the same identifier as the response packet must be searched and then removed from the list. At this moment, the response value, received in the packet, is associated to the field `'a` in the ADT, and the lock flag is set to `true`, enabling the access to the data field (a locking mechanism like the one proposed in [10]).

Now, we must think about the different moments in which the result of the remote evaluation can be available in relation to the point in which its value is required. We can identify three different temporal situations:

- *Case 1: Success.* The server sends the result before the client demands it back. (figure 1). In this case, there is no problem because when `val` function is executed, in order to access the data encoded in the `Remoteval`, the desired value, which has been received previously, is obtained, although it may be still packed. If the lock flag is set to `true` then the result is already stored in the data field of the ADT. Otherwise, `val` needs to unpack and decode the result, build the value `'a` and store it in the `'a Remoteval`.

- *Case 2: Fault.* The server has not sent back the result yet when the client requires it. (figure 2). In this case, `val` must stop the evaluation

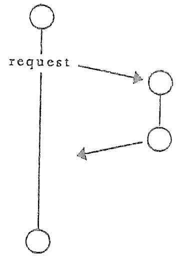

Figure 3: Useless in scope

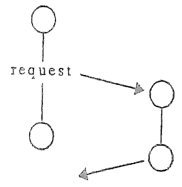

Figure 4: Useless out of scope

until the packet containing the result arrives. The client process waits for any packet coming from any server, performing the unpacking of the results of other ADTs received before the expected one arrives.

- *Cases 3 and 4: Useless.* It is also possible that the result will never be used (an effect derived from the strict evaluation). In this case, we never apply `val` on the obtained `Remoteval`. Moreover, two cases can be distinguished:

  a) the case in which the value is received in the scope in which its `Remoteval` exists (*Useless in scope*, figure 3)

  b) and the case in which the `Remoteval` is out of its scope when the result arrives (*Useless out of scope*, figure 4).

In fact, this last case is not a problem if the result arrives before its `Remoteval` gets out of scope, since the client receives it, unpacks it, associates it with its `Remoteval` and then, when its ADT gets out of scope, the garbage collector removes it from the heap. However, if the result arrives after the `Remoteval` has been removed from the Caml Light heap, a serious problem arises because we must implement a mechanism that should be able to detect the out-of-scope condition of a value and, later, stop the evaluation of the deferred value to avoid the access to a non-existing object. This fact implies a modification in the system of creation/destruction of values, and even in this case, a response may arrive from the server just before the arrival of the signal which stops the server evaluation process, perhaps due to the propagation delays in the communication across the network. The implemented solution is based in transforming the case 4 into the case 3, keeping the `Remoteval`

in scope until the arrival of its result. This can be done using the list of `Remotevals` whose results have not arrived yet, as the presence in the list forces the garbage collector mechanism to avoid the removal of the `Remoteval` from the memory (as there is one reference to the object at least). That is why the `Remoteval` list solves two problems simultaneously.

3 Distributed execution of processes

Our approach makes sense, that is, improves the computational time, only if the server or servers used reside in several machines, the optimal situation being that in which each server resides on a different machine, with the client in another computer.

In order to develop the approach, the PVM software package (*Parallel Virtual Machine*) has been used. The package allows to simulate a multiprocessor virtual machine with distributed memory on a set of machines which can have different architectures and that are connected by nets which can also be of several kinds, the only restriction being the need that all the computers involved run under the Unix operative system. PVM uses the message passing paradigm, as most of the software of this kind [11].

The client must use the function `client_init` to create the servers. This function takes as its argument an integer representing the number of servers the client wishes to use, and returns a `unit` value. In a similar way, the client must use the function `client_exit`, with type `unit -> unit`, when the servers are to be eliminated, and so the distributed evaluation mode exited.

The servers must also use a function called `server_init` to link to the PVM system and communicate with the client.

The main loop of each server receives the requests and orders the pertinent executions using the function `get_next_req`, which waits for the reception of a request and returns the identifier of the service required, which is used to index the vector of service functions. All these functions are of type `unit -> unit` and have the following form:

```
let remote_func () =
  let (x1, x2, ..., xn) = get_args ()
  in response (local_func x1 x2 ... xn);;
```

where the function `get_args` unpacks the arguments of the service. These arguments are used by the function that accomplishes the most important work, called `local_<func>` in our example, whose result is packed and sent to the client by the function `response`.

Since our system has a distributed memory, in the packets that the processes interchange we must not send the pointers to the data to transmit, but the data themselves. That is why we need to know the structure that the Caml Light objects adopt internally, in order to be able to decode them in the messages the client interchanges with the servers, and, later, rebuild accurately those objects in the destination process. The functions that develop these tasks are called, respectively, `encode` and `decode`, and are located both in the client and in the servers, since they all need to encode objects for their sending, and unpack them for their processing (these functions are identical to those used in [8], but using the PVM packing primitives for the basic types).

4 Transformation of a sequential code

In order to show our approach, we will show the modifications needed to transform a program initially executed in a totally sequential fashion in an equivalent code which can be executed in a distributed fashion. The program selected, as simple as useless, is the following:

```
let sqr x = x * x;;
let addsqrs x y = (sqr x) + (sqr y);;
```

The obtention of a distributedly executable program from a sequential one has two steps:

- Creation of the program corresponding to the servers or slaves from the functions we want to execute in remote nodes (function level granularity).
- Creation of the client or master program, that uses the services present in the slave processes through the interface provided by the ADT `Remoteval`, which guarantees the referential transparency and the determinism in the program execution.

In the first step we create the functions to evaluate remotely, which will constitute the service functions, using the template presented in the preceding section. In our example, in order to execute remotely `sqr`, we would use a service function of the form:

```
(* services area *)
let sqr x = x * x;;

(* services interfaces area *)
let remote_sqr () =
  let (x1) = get_args ()
  in response (sqr x1);;
```

In order that the server can identify the service function required by each request, these functions are included in a list or vector, which is possible due to the fact that they all have type `unit -> unit`, encapsulating the functions that do the real work:

```
(* services table *)
let services_table = [| ...; remote_sqr; ... |];;
```

The main body of the server initiates the process (including it in the PVM virtual machine) and manages the reception-evaluation-answering cycle of the requests.

```
server_init();;
let process services =
  while (true) do
     services.(get_next_req()) ()
  done;;
process services_table;;
```

In the second step we are to generate the interfaces for the remote functions in the client, in such a way that if in the original program we have a function with type `'a1 -> 'a2 -> ... an -> 'b`, the corresponding new function will be of type `'a1 -> 'a2 -> ... an -> 'b Remoteval`. In general, the template to use will be of the form:

```
let <func> (x1:<'a1>) (x2:<'a2>) ... (xn:<'an>) =
     ((request <remote_func_id> (x1,x2,...,xn)): <'b> Remoteval);;
```

In our example, the template form is:

```
(* remote interface area *)
let sqr (x1:int) = ((request <remote_sqr_id> (x1)): int Remoteval);;
```

The original function is transformed, requesting the anticipated evaluation of all the remote functions at the beginning of the scope in which they will be used and replacing the references to those values with the access through `val` to the requests previously encapsulated in the `Remotevals`. In our example:

```
let addsqrs x y = let     x_s = sqr x
                      and y_s = sqr y
                       in val x_s + val y_s;;
```

After this, it only remains to define the number of slave processes that will participate in the computations with `client_init` before requesting any remote evaluation. When we desire to finish the calculations, we terminate the slave processes with `client_exit`.

```
client_init 2;;
addsqrs 2 4;;
client_exit ();;
```

5 Results

As an example to show the improvements that can be achieved using the proposed approach, we have constructed a server which provides, among others, an evaluation service of the well-known function of Fibonacci, and a client that requests the application of this function on the list of the integers from 1 to n, where n adopts the values 12, 16, 20, 24, 28 and 32. In each proof the real elapsed times of computation required for a given value of n using a normal Caml Light program have been compared with those obtained using our approach. For the latter case 1, 2, 4, 8 and 16 servers were used, residing each one on a different machine. All the computers used were *SUN SPARCstations ELC*. See table 1.

List	Sequential	Distributed execution				
size	execution	1 serv.	2 servs.	4 servs.	8 servs.	16 servs.
12	0.035	0.120	0.180	0.093	0.137	0.817
16	0.240	0.351	0.248	0.189	0.206	0.191
20	1.641	1.804	1.137	0.841	0.843	0.824
24	11.251	11.433	7.088	5.138	4.616	4.504
28	77.100	77.423	47.894	34.640	30.361	29.807
32	528.492	529.580	327.921	236.903	206.777	202.495

Table 1: Times for the sequential and distributed executions of the Fibonacci function in seconds

References

[1] Leroy, X. *The Caml Light system, release 0.6 Documentation and user's manual*, Project Formel, INRIA Rocquencourt, Sep. 1993.

[2] Mauny, M. *Functional programming using Caml Light*, Project Formel, INRIA Rocquencourt, Sep. 1993.

[3] Leroy, X., Weis, P. *Le Langage Caml*, InterEditions, Paris, 1993.

[4] Cosineau, G., Huet, G. *The CAML primer*, INRIA-ENS V. 2.6.

[5] Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchel, R., Sunderam, V.S. *PVM 3 User's Guide and Reference Manual*, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[6] Gulías, V.M. *Interfaz Gráfica para una implementación del λ-cálculo, construída sobre entornos heterogéneos*, Master thesis, Departamento de Computación, Universidad de La Coruña, Jul. 1994.

[7] Freire, J.L., Gulías, V.M., Molinelli, J.M. *Utilización de la programación funcional para la construcción de servidores en entornos heterogéneos*, In Joint Conference On Declarative Programming GULP-PRODE'94, vol. 2, pp. 351-365, Sep. 1994.

[8] Gulías, V.M., Valderruten, A. *Une interface graphique distribué pour le lambda-calcul supporteé par des Serveurs Fonctionnels*, In JFLA'95, INRIA Jan. 1995.

[9] Hudak, P. *Para-Functional Programming in Haskell*, In Paralell Functional Languages and Compilers, ACM Press, pp. 159-196, 1991.

[10] *Notes On Parallel Computation*, In Structure and Interpretation of Computer Programs, Department of Electrical Engineering and Computer Science, MIT, Dec 1990.

[11] Dongarra, J.J., Geist, G.A., Manchek, R., Sunderam, V.S. *Integrated PVM Framework Supports Heterogeneus Network Computing*, Jan. 1993.

A Logic Language based on GAMMA-like Multiset Rewriting

Paolo Ciancarini, Daniela Fogli*and Mauro Gaspari
Dipartimento di Scienze dell'Informazione
Università di Bologna - Italy
Piazza Porta S. Donato 5 - 40127 Bologna, Italy
E-mail: {cianca,gaspari}@cs.unibo.it

Abstract

This paper describes Gammalög, a logic language based on multiset rewriting. The language combines the ability of describing parallel programs made of multiset transformation rules as in GAMMA with the execution model of logic programming in a strongly typed framework as in Gödel. We describe the design choices, the syntax and the semantics of the language, and a prototype implementation.

Keywords: Parallel Logic Programming, Gamma, Multiset Rewriting.

1 Introduction

Languages like FCP [19] and Parlog [10] deal with parallelism by adding explicit mechanisms for synchronization and communication to the logic programming paradigm. These languages are based mostly on a model of parallel programming called the *stream-based process model*: processes execute logic rules that can spawn new processes communicating via streams; special constraints on streams rule the synchronization among producer and consumer processes. An open issue for this class of languages is to define a satisfactory model-theoretic semantics to provide a declarative characterization of concurrency.

Recently, a paradigm of coordination based on the concepts of generative communication in a shared dataspace is becoming popular [14]. Generative communication means that processes use no channel to communicate: they simply output tuples in a shared dataspace; processes which need input access the tuples associatively. Tuples have a persistence, that is messages survive to processes which originated them.

*Daniela Fogli's current address is Dipartimento di Elettronica per l'Automazione, Università di Brescia, Via Branze 38 - 25123 Brescia, Italy, E-mail: fogli@idea.ing.unibs.it

Some languages have been proposed which follow such a paradigm, for instance: Linda [6], and GAMMA [3]. The shared dataspace model has also been investigated in logic programming: Shared Prolog [5], *μlog* [18] and LO [1] are examples of logic languages based on shared dataspaces. Like the stream-based ones, these logic languages have no standard model-theoretic semantics, even though some of them offer a non standard declarative semantics. For instance the model-theoretic semantics of *μlog* [18] is based on a notion of truth with respect to traces (e.g. truth depends on the sequence of communication events which may occur during the computation).

In this paper we present Gammalög, a parallel logic language which provides a standard model theoretic semantics expressed in terms of multiset rewritings as in GAMMA. We describe the semantics of the language and an implementation based on Gödel [17]. The abstractions provided by GAMMA are made available in a logic programming framework, thus they benefit of all the classical advantages of this paradigm, in particular the ability to have executable specifications which are proved to be correct by the underlying logic formalism. Moreover, the strongly typed framework and the modularity which are inherited from Gödel provide the basis for a rigorous approach to the design and the development of parallel programs.

Another advantage of Gammalög is to provide an executable version of GAMMA: Gammalög programs are compiled by the Gödel compiler. Thus, we provide also a tool for practical experimentation with multiset rewriting as a programming paradigm.

2 GAMMA

GAMMA programs are described in terms of multiset transformations without introducing unnecessary sequentiality [3]. The basic data structure used in GAMMA is the *multiset*. A multiset is just a bag containing items which are stored without any constraint or relationship among one - another. The control structure associated with multisets is the Γ operator; its formal definition can be stated as follows [3]:

$\Gamma((R,A))(M) =$
  if $\forall x_1,...,x_n \in M, \sim R(x_1,...,x_n)$
  then M
  else let $x_1,...,x_n \in M$, be such that $R(x_1,...,x_n)$ in
    $\Gamma((R,A))(M - \{x_1,...,x_n\}) + A(x_1,...,x_n))$.

where (R, A) is a pair of functions specifying the rewriting rule which can be applied on the multiset. R is a *reaction* condition, namely a boolean function which specifies if the rule is applicable. A is an *action*, namely a multiset rewriting, executed when the reaction succeeds. Operationally the Γ operator searches for a subset of M, $\{x_1,...,x_n\}$, such that $R(x_1,...,x_n)$ holds. When the reaction succeeds the elements satisfying it are removed from the multiset and the action $A(x_1,...,x_n)$ generates new elements to be inserted in the multiset. Otherwise, if no elements of M satisfy the reaction condition ($\forall x_1,...,x_n \in M, \sim R(x_1,...,x_n)$) the Γ operator terminates

and the result is M.

What follows is a GAMMA program which calculates the maximum element of a set.

$max\_element(s) = \Gamma((R,A))(s)$ where
  $R(x,y) = x \leq y$
  $A(x,y) = \{y\}$.

GAMMA provides two operators which enable one to combine simple programs and have been introduced in [16]. They are the sequential composition `P o Q` and parallel composition `P + Q` operators. In the rest of this paper we assume that the sequential composition operator is interpreted left to right, i.e. the program `Q` is executed with the multiset returned by `P` as an input only when the program `P` is terminated[1].

The sequential and the parallel operators enable one to build complex programs starting from simple GAMMA programs. For instance, the program *Positive_Integers* which computes the number of positive integers in a multiset can be expressed as the composition of three simple programs, as follows:

Positive_Integers(m)=(Ones + Non_neg) $\circ$ *Add*

where *Add*, is a program which returns the sum of elements of a multiset; it can be written as follows:

$Add = \Gamma(((R,A))(s)$ *where*
  $R(x,y) = true$
  $A(x,y) = \{x+y\}$

Ones is the programs which transforms to 1 all the positive integers in a multiset:

$Ones = \Gamma(((R,A))(s)$ *where*
  $R(x) = x > 1$
  $A(x) = \{1\}$

finally, *Non_neg* is a program which selects all the positive numbers of a multiset of integers. It can be defined in GAMMA as follows:

$Non\_neg = \Gamma(((R,A))(s)$ *where*
  $R(x) = x < 0$
  $A(x) = \{\}$

This means that to compute the number of positive values in a multiset we can execute the first two GAMMA programs in parallel on a shared multiset and when the two programs terminate we compute the number of ones in the multiset (using *"Add"*).

3 Embedding GAMMA in Logic Programming

A GAMMA program consists of a multiset transformation rule; its semantics can be modeled as a relationship on multisets, thus we can represent GAMMA programs as

[1]Instead, in [16] the result of `P o Q` is obtained by executing the programs from right to left.

predicates on multisets provided that we extend logic programming with this new data structure. A GAMMA program then can be translated into a predicate taking two arguments: the first one represents the initial multiset and the second the final one.

In order to add multisets in logic programming we will redefine the approach presented in [11] to add sets. We will use double angle brackets $<<$ and $>>$ to denote multisets; we also represent partially known multisets in this way: $<< x_1, ..., x_n \mid rest >>$ is a multiset containing some known elements $x_1, ..., x_n$ whereas *rest* represents the rest of the multiset. The unification algorithm, which has to take into account the new equality defined on multisets becomes non-deterministic since in general a unification between two multisets has more than one solution. For instance, $<< x, y >>=<< 1, 2 >>$ returns two substitutions: $\{x = 1, y = 2\}$ and $\{x = 2, y = 1\}$ which are both correct. Formal aspects related to this issue are described in an extended version of this paper [8]. In this section we suppose we have defined an extension of logic programming supporting first class multisets. A real implementation of this extension is described in [13].

3.1 Gamma predicates

Given the multiset extension, the translation of the Γ operator in logic programming with multisets is immediate (in the following we denote predicates names with identifiers beginning with upper case letters, while a variable begins with a lower case letter; this is also the choice of Gödel):

```
Program(m_1, m_3)<-
     Step_Program(m_1, m_2) &
     Program(m_2, m_3).
Program (m_1, m_1)<-
     End_Program(m_1).
```

where `Step_Program` is a predicate which expresses one step of multiset transformation. The second clause represents the termination condition for the program.

The predicate `Step_Program` is associated to the pair (R, A) representing the multiset transformation rule in the source GAMMA program.

```
Step_Program(<< x1,...,xn |rest>>, << y1,...,ym |rest>>)<-
     R(x1,...,xn) &
     A(x1,...,xn,y1,...,ym).
```

The reaction condition `R` is translated into a logical predicate which takes $x_1, ..., x_n$ as arguments; the action `A` needs also the additional arguments $y_1, ..., y_m$ representing the elements that must be added to the multiset replacing $x_1, ..., x_n$.

The terminal condition is expressed by the predicate `End_Program` which tests if the reaction condition does not hold (the symbol $\sim$ stands for negation as failure).

```
End_Program(<< x1 >>).
End_Program(<< x1,x2 >>).
```

```
...
End_Program(<< x1,...,xn-1 >>).
End_Program(<< x1,...,xn |rest>>)<- ~ R(x1,....xn).
```

For instance, the translation of the maximum element program presented in Section 2 is as follows:

```
Max_element(m_1, m_3)<-
     Step_Max_element(m_1, m_2) &
     Max_element(m_2, m_3).
Max_element(m_1, m_1)<- End_Max_element(m_1).
Step_Max_element(<<x, y|rest>>, <<z|rest>>)<-
     R(x, y) &
     A(x, y, z).
End_Max_element(<<x>>).
End_Max_element(<<x,y|rest>>)<- ~R(x,y).
R(x, y) <- x =< y.
A(x, y, y).
```

We remark that the predicate `Max_element` never fails and according to the definition of the Γ operator, we assume that if $\mathtt{R}(x_1, ..., x_n)$ succeeds also $\mathtt{A}(x_1, ..., x_n, y_1, ..., y_m)$ succeeds.

3.2 Translating GAMMA operators

The sequential composition operator $\circ$ states that the two GAMMA programs are executed one after the other; it can be translated with a conjunction where the multiset resulting from the execution of the first program is taken as an input of the second program.

To translate the parallel composition operator + we built a new predicate which has three clauses: the first two clauses express the transformation rules of the composing programs, and the last clause is the joint termination condition expressed by the conjunction of the termination conditions of the two programs.

For instance, the program `Positive_Integers` described in Section 2 which computes the number of positive values in a multiset can be translated in logic programming as presented in Figure 1. The different clauses of "`Ones+Non_neg`" can be executed in OR parallel on the same multiset.

3.3 Gammalog

Gammalog is a logic programming language extended with multisets plus the two connectives: Γ and $\equiv$. The former connective allows one to define programs (gamma clauses) following the GAMMA style by specifying the multiset transformation reactions and actions; the latter provides a way to define new programs (definition clauses) exploiting the composition operators $\circ$ and +.

```
Positive_Integers(m_1, m_3) <-
      'Ones+Non_neg'(m_1, m_2) & Add(m_2, m_3).
'Ones+Non_neg'(m_1, m_3) <-
      Step_Ones(m_1, m_2) & 'Ones+Non_neg'(m_2, m_3).
'Ones+Non_neg'(m_1, m_3) <-
      Step_Non_neg(m_1, m_2) & 'Ones+Non_neg'(m_2, m_3).
'Ones+Non_neg'(m_1, m_1) <-
      End_Ones(m_1) & End_Non_neg(m_1).
Step_Ones(<<x|rest>>,<<y|rest>>)<-
      R_Ones(x) & A_Ones(x, y).
End_Ones(<<x|rest>>)<- ~R_Ones(x).
R_Ones(x) <- x > 1.
A_Ones(x, 1).
Step_Non_neg(<<x|rest>>,rest) <-
      R_Non_neg(x) & A_Non_neg.
End_Non_neg(<<x|rest>>) <- ~R_Non_neg(x).
R_Non_neg(x) <- x < 0.
A_Non_neg.
Add(m_1, m_3) <- Step_Add(m_1, m_2) & Add(m_2, m_3).
Add(m_1, m_1) <- End_Add(m_1).
Step_Add(<<x, y|rest>>,<<z|rest>>) <-
      R_Add(x, y) & A_Add(x, y, z).
End_Add(<<x>>).
End_Add(<<x,y|rest>>)<- ~R_Add(x, y).
R_Add(x, y).
A_Add(x, y, x+y).
```

Figure 1: Translating Gamma Operators.

The Γ connective has the following syntax:

```
Max_element(<<x, y|rest>>, <<z|rest>>)Γ
      R(x, y) &
      A(x, y, z).
R(x, y) <- x =< y.
A(x, y, y).
```

where the first clause is a Gamma clause and both `R(x, y)` and `A(x, y, z)` are predicates.

The connective ≡ allows one to define new predicates, starting from predicates defined with the Γ connective. As an example, a definition clause has the form:

```
Positive_Integers ≡ Ones + Pos o Add
```

where + has the higher precedence and parentheses are not allowed. This restriction is necessary to guarantee a correct translation: in fact, Gamma programs like $(P \circ Q) + H$ cannot be translated correctly into clauses exploiting the proposed technique.

Definition 3.1 - Gammalog Program. *A Gammalog program P is composed by a set of Gamma clauses P^Γ, a set of definition clauses $P^\equiv$ and a set of standard clauses P^{LP} defining the reaction P^R and actions P^A predicates.*

Definition 3.2 - Gammalog Query. *A Gammalog query Q is a conjunction of positive literals involving predicates defined in P^Γ or in $P^\equiv$.*

4 Semantics

We present the soundness and completeness results for Gammalog, in particular: the soundness of Gammalog and the completeness of classes of Gammalog programs with respect to the extended SLD resolution. The extension of logic programming with multisets is described in in an extended version of this paper [8]. The related equality axioms are presented in Figure 2.

(Z) $In(0, x, <<>>)$ the empty multiset contains no elements;

(I) $In(n+1, x, Inc\_m(y, z)) \leftrightarrow (x \neq y \wedge In(n, x, z) \vee (x = y \wedge In(n, x, z).$
this rule specifies how the function Inc_m operates;

(W) $In(n, y, x) \rightarrow \exists z(In(0, y, z) \wedge x = \overbrace{Inc\_m(y, Inc\_m(y, ..., Inc\_m(y, z)}^{n}, ...)$
(that is y is inserted n times in x); it is the ''without'' axiom which guarantees the existence of the multiset z without any y;

(L) $In(n+1, y, x) \rightarrow \exists z(In(n, y, z) \wedge x = Inc\_m(y, z))$
''less'' axiom to guarantee the existence of the multiset $x \setminus << y >>$;

(E) $Inc\_m(x, v) = Inc\_m(y, w) \leftrightarrow$
$(x = y \wedge v = w) \vee \exists\, z(v = Inc\_m(y, z) \wedge w = Inc\_m(x, z))$
''equality'' axiom to establish when two multisets are equal;

(R) $\exists z \forall y(In(n, y, x) \rightarrow (In(m, z, x) \wedge In(0, y, z))$
''regularity'' axiom; it guarantees the membership does not generate loops.

(U) $f(x_1, \ldots, x_n) \neq <<>> \wedge In(0, x, f(x_1, \ldots, x_n))$
where $f/n \neq <<>>/0$ and $f/n \neq Inc\_m/2$.

Figure 2: Multiset Axioms.

The axiom (I) corresponds to the axioms (W1) and (W2) in [12], while the axioms (W) and (L) correspond to the axiom (L), in the same paper.

Note that, as with sets, a special equality is required on multisets because the order of elements in a multiset is irrelevant; thus, the *permutativity* property holds. The axiom (E) guarantees this property: it is easy to prove applying it and (I) that the following equality holds:

```
Inc_m(z,Inc_m(y,x)) = Inc_m(y,Inc_m(z,x)).
```

The semantics of Gammalog is given in term of its translation into logic programming with multisets. We define formally the function ψ which given a Gammalog program P returns its translation P^M in logic programming with multisets following the schema described above. The translation function $\psi : P^\Gamma \cup P^\equiv \cup P^{LP} \rightarrow P^M$ is defined in Figure 3:

The clauses of the logic program generated by $\psi(P)$ contain also negated literals, thus we need to consider general programs, SLDNF resolution and program completion in order to prove the soundness and completeness of Gammalog [2].

The *dependency graph* D_P for a program P is a directed graph with signed edges. The nodes are the relations occurring in P. There is a positive (resp. negative) edge

```
**Case 1.** c is a Program $\{c_1, \ldots, c_n\}$, $n \geq 2$

  $\psi(c) = \psi(\{c_1\}) \cup \psi(\{c_2, \ldots, c_n\})$

**Case 2.** c is P(<< $x_1, \ldots, x_n$ |r>>, << $y_1, \ldots, y_m$ |r>>) $\Gamma$
              $R(x_1, \ldots, x_n)$ & $A(x_1, \ldots, x_n, y_1, \ldots, y_m)$.

  $\psi(\{c\})$ = {P(m_1, m_3)<- Step_P(m_1, m_2) & P(m_2, m_3),
            P(m_1, m_1)<- End_P(m_1),
            Step_P(<< $x_1, \ldots, x_n$ |r>>, << $y_1, \ldots, y_m$ |r>>)<-
                R($x_1, \ldots, x_n$) & A($x_1, \ldots, x_n, y_1, \ldots, y_m$),
            End_P(<< $x_1$ >>).
            ...
            End_P(<< $x_1, \ldots, x_{n-1}$ >>).
            End_P(<< $x_1, \ldots, x_n$ |r>>)<- ~ $R(x_1, \ldots, x_n)$}.

**Case 3.** $c$ is H ≡ P ∘ Q

  $\psi(\{c\})$ = {H(m_1, m_3) <- P(m_1, m_2) & Q(m_2, m_3)}.

**Case 4.** c is H ≡ P + Q

  $\psi(\{c\})$ = {H(m_1, m_2) <- 'P + Q'(m_1, m_2),
            'P+Q'(m_1, m_3) <- Step_P(m_1, m_2) & 'P+Q'(m_2, m_3),
            'P+Q'(m_1, m_3) <- Step_Q(m_1, m_2) & 'P+Q'(m_2, m_3),
            'P+Q'(m_1, m_1) <- End_P(m_1) & End_Q(m_1)}.

**Case 5.** c ∈ $P^{LP}$

  $\psi(\{c\})$ = {c}
```

Figure 3: The Translation Function.

(r,q) if a clause in P has the relation r in its head and the relation q in a positive (resp. negative) literal in its body. P depends *evenly* (resp. *oddly*) on q if there is a path from p to q with an even (resp. odd) number of negative edges. Given a general program P and a general goal Q, we recall here three basic definitions from [2]:

Definition 4.1 - Strictness. *We say that P is strict w.r.t. Q if no relation occurring in Q depends both evenly or oddly on a relation defined in P.*

Definition 4.2 - Stratified Program. *A program P is called stratified if no cycle with a negative edge exists in its dependency graph.*

Definition 4.3 - Stratification. *Given a program P, a stratification of P is a partition $P_1 \cup P_2 \cup \ldots \cup P_n$ such that each P_i uses positively only relations defined in P_j $j \leq i$, and negatively only relations defined in P_j, $j < i$.*

SLDNF resolution has been proved sound w.r.t, two-valued semantics of program completion [9], while completeness holds for an allowed program P and an allowed goal Q such that P is strict w.r.t. Q and P is stratified [7].

Allowedness is a condition which guarantees that P and Q do not flounder, i.e., there is a safe selection rule for literals which guarantees that only ground negated literals are selected exploiting SLDNF resolution. This condition holds in Gammalog since the multiset we give as an argument to negated literals is always ground.

Lemma 4.4 - Strictness of Gammalog. *A Gammalog program P is strict w.r.t. a Gammalog query Q if P^R is a positive program and P^A is strict w.r.t. actions queries.*

Proof: We have to show that the negation introduced by $\psi(P)$ guarantee strictness. This is true because negation is introduced only for reaction predicates which does not depend on negative goals. Thus, there are no predicates in a Gammalog query which depend both evenly and oddly on a relation defined in the program. ■

Lemma 4.5 - Stratification of Gammalog. *A Gammalog program $P = P^\Gamma \cup P^\equiv \cup P^{LP})$ is stratified if P^{LP} is stratified.*

Proof: If a stratification of P exists then it can be proved stratified [2]. Thus we need to show that such a stratification exists for any Gammalog program which satisfies the hypothesis. Given a Gammalog program P we define a stratification on the program generated by $\psi(P)$ as follows. We proceed analysing the different cases of ψ for each case we provide a stratification of the resulting program: in case 2 we have the following stratification $p_2 = \phi$, $p_2 = \{P, Step\_P, End\_P\}$; in case 3 and 4 we do not introduce negation thus we only define a set $p_2 = \{H\}$ which includes all the clauses generated by the transformation and $p_1 = \phi$; in case 5 we put $p_1 = \{c\}$ and $p_2 = \phi$ for every $c \in P^{LP}$. Finally, considering case 1 we define the stratification of the whole program as follows: $P_1 = \bigcup_{c \in P} p_1$ and $P_2 = \bigcup_{c \in P} p_2$. ■

In the following we summarize the main results of the paper:

Theorem 4.6 - Soundness of Gammalog. *Let P be a Gammalog program and $G = (\leftarrow A_1 \& \ldots \& A_k)$ be a Gammalog goal. If G has a refutation in $\psi(P)$ with computed answer $\theta = \sigma_1 \ldots \sigma_n$ this answer is correct for $P \cup \{G\}$.*

Proof: *The soundness of Gammalog follows immediately from the soundness of logic programming with multisets and of SLDNF. This because a Gammalog derivation always corresponds to a SLDNF derivation.* ■

Theorem 4.7 - Completeness of Gammalog. *Let $P = P^\Gamma \cup P^\equiv \cup P^R \cup P^A$ be a Gammalog program and G be a Gammalog goal. Then, for each correct answer σ for $P \cup \{G\}$ G has a refutation in $\psi(P)$ if P^R is a positive program and P^A is a general program strict (w.r.t. action queries) and stratified.*

Proof: *The proof follows from lemmas 4.4 and 4.5 and from the well known result of [7].* ■

5 Gammalög: an instance of Gammalog

Gammalog is just an abstract computational model, thus if we want to design a real programming language several practical choices need to be done about its execution model. Our choice was to base Gammalog on the language Gödel [17] for a number of reasons: it already supplies a support for sets and multisets as in [12, 13]; it is a well engineerized and relatively efficient tool; it is freely accessible; it is implemented via translation to SICStus Prolog and the Gödel compiler (written in SICStus) is relatively easy to manage and extend. Finally, as the authors say, "Gödel reduces the effort involved in providing a parallel implementation of the language and offers substantial scope for parallelism in such implementations", we hope to simplify our efforts of building a parallel logic language based on GAMMA.

The new language we have designed is named Gammalög; it is an extension of Gödel with the Γ operator plus facilities which allow one to define programs exploiting sequential and parallel composition. For instance, the module in Figure 4 defines the program `Fib` which, given the multiset $<< n >>$, computes the `n-th` Fibonacci number. `Fib` is built as the combination of simple Gammalög programs. The confluence and termination of this program derive directly from the proof of confluence and termination of the same combination of GAMMA programs in [15].

```
 MODULE       Fibonacci.
 IMPORT       Multisets.
 BASE         Elem.
 FUNCTION     F : Integer - > Elem.
 PREDICATE    Zero,Decompose,Add:  Multiset(Elem)*Multiset(Elem);
              R_Zero, R_Decompose:  Integer;
              A_Decompose, A_Add:  Integer * Integer * Integer;
              A_Zero, R_Add:  Integer * Integer.
 GPREDICATE   Fib.
Fib <=> Zero + Decompose o Add.
Zero(<<F(x)|rest>>, <<F(y)|rest>>)<=
            R_Zero(x) | A_Zero(x, y).
R_Zero(0).
A_Zero(_, 1).
Decompose(<<F(x)|rest>>, <<F(y), F(z)|rest>>)<=
            R_Decompose(x) | A_Decompose(x, y, z).
R_Decompose(x)< - x > 1.
A_Decompose(x,x-1,x-2).
...
```

Figure 4: A Gammalög Module.

The connective <=, denotes Gamma clauses and the connective <=> the definition clauses. `Zero` computes the Fibonacci number associated to 0; `Decompose` transforms into 1 all the elements greater than 1 in the multiset applying the Fibonacci low.

Since Gammalög is strongly typed in the same way of Gödel all the predicates which appears in a program must be declared in a given module. For instance, the declarations needed for `Zero,Decompose,Add` are:

`PREDICATE Zero, Decompose, Add:  Multiset(Elem) * Multiset(Elem).`
where `Elem` is a base type for multisets.

Simple Gammalög programs are declared as ordinary Gödel predicates. On the other side the predicate `Fib` being defined with the definition operator need a particular declaration using the keyword `GPREDICATE` (which stands for GAMMA Predicate) this is needed to inform the compiler that the predicate is a GAMMA predicate.

During the translation, this particular declaration allows the compiler to add the appropriate arguments to the GAMMA predicate and to generate an object code corresponding to the rules described in section 3.

5.1 Implementation

The implementation of Gammalög has been realized extending the Bristol version of Gödel [4]. In order to have an adequate runtime support for Gammalög we extended Gödel with first class multisets providing an extended unification algorithm. Then, we extended the Gödel compiler to deal with Gammalög operators. A Gammalög program is first translated into Gödel and then is translated into Prolog by the Gödel compiler.

The extension of Gödel with multisets is based on an extended unification algorithm presented in an extended version of this paper [8]. The implementation is described in [13]. The scanner and the parser have been modified to recognize the new statements. The composition operators o and + have the same precedence of "\/" (OR) and "&" (AND) respectively. The Gödel's code generator has been modified as well in order to provide the translation of the new statements.

To compile clauses containing the definition connective <=> with the operators o and + in their bodies, a particular treatment has been designed: we compile the first operator as a conjunction, while the second operator requires the definition of an auxiliary predicate to handle the intrinsic nondeterminism. A parallel combination of Gammalög simple programs has the form: `Prog <=> P + Q.` and is compiled in Gödel generating a schema similar to the one presented in 1 where the & is replaced by the bar commit ('|'). The Gödel pruning operator [17] which is placed between reaction conditions and actions in `Step_P` clauses. The commit has the declarative meaning of a conjunction, and the following procedural meaning: only one solution is found for a formula in its scope (on the left of the commit), all the other branches arising from the other clauses of the same predicate which contain a commit are pruned. The order in which the statements are tried is not specified, so that the meaning of '|' is close to meaning of the commit of the concurrent logic programming languages.

6 Conclusion

We presented Gammalog: an integration between logic programming and the parallel language GAMMA. We show that to achieve a full integration between the two paradigms, proving Gammalog complete and sound, we need to express reaction conditions as positive logic programs. The implementation we describe is based on Gödel and in our knowledge is the first executable version of Gamma on a general purpose hardware. The language can be used as a specification language to explore the power of multiset rewriting in a logic programming framework.

Aknowledgments. We thank Eugenio Omodeo for his remarks on the contents of this work. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Project 9102 (COORDINATION).

References

[1] J. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[2] K. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–72, May/July 1994.

[3] J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[4] A. Bowers and J. Wang. *Bristol Gödel User Manual.* Department of Computer Science - University of Bristol, University Walk Bristol BS8 1TR UK, 1994.

[5] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[6] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[7] L. Cavedon and J. Lloyd. A completeness Theorem for SLDNF-resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[8] P. Ciancarini, D. Fogli, and M. Gaspari. A Language based on Multiset Rewriting. Technical Report UBLCS, Comp. Science Laboratory, Università di Bologna, Italy, June 1995.

[9] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum-Press, New York, 1978.

[10] K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[11] A. Dovier, E. Omodeo, E. Pontelli, and G. Rossi. {log} : A logic programming language with finite sets. In K. Furukawa, editor, *Proc. 8th Int. Conf. on Logic Programming*, pages 111–124, Paris, France, 1991. MIT Press, Cambridge, MA.

[12] A. Dovier, E. Omodeo, E. Pontelli, and G. Rossi. {log}: A language for programming in logic with finite sets. Technical Report Rap. 04.93, Università degli studi di Roma I, Dip. di Informatica e Sistemistica, May 1993.

[13] D. Fogli. Insiemi e Multi-insiemi nel Linguaggio di Programmazione Logica Gödel. Master's thesis, Universita' di Bologna, Bologna, 1994.

[14] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[15] C. Hankin, D. LeMetayer, and D. Sands. A Calculus of Gamma Programs. Technical Report TR 1758/92, IRISA, INRIA-Rennes, 1992.

[16] C. Hankin, D. LeMetayer, and D. Sands. A Parallel Programming Style and Its Algebra of Programs. In *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 93)*, volume 694 of *Lecture Notes in Computer Science*, pages 367–378. Springer-Verlag, Berlin, 1993.

[17] P. Hill and J. Lloyd. *the gödel programming language. Technical report, Dept. of Computer Science, University of Bristol, 1993.*

[18] J. Jacquet and K. DeBosschere. On the Semantics of μLog. Future Generation Computer Systems, *10(1):93–136, 1994.*

[19] E. Shapiro. The Family of Concurrent Logic Languages. ACM Computer Surveys, *21(3):412–510, September 1989.*

DEDUCTIVE DATABASES

Side effect analysis for logic-based planning

Kave Eshghi, Miranda Mowbray
Hewlett-Packard Laboratories,
Filton Avenue, Stoke Gifford, Bristol BS12 6QZ, England
Phone: +44 117 9228178 Fax: +44 117 9228920
{ke,mjfm}@hplb.hpl.hp.com

Abstract

In this paper we describe the algorithms for planning and, in particular, for side effect analysis used in a software tool for system management, called Dolphin, which is based on declarative programming. In Dolphin the user is given the option of declaring some side effects to be acceptable or unacceptable. This treatment of side effects makes our system different from those described in the logic-based planning literature.

Keywords: Deductive databases, Artificial Intelligence, Logic-based planning, Side effect detection and analysis

1 Introduction

In this paper we describe the planning and side effect analysis algorithms used in a software tool, called Dolphin, which is used for system management. Dolphin is a tool for helping system administrators manage complex, networked computer systems by providing them with a high level, intuitive view of the system in the context of which they can access and manipulate the the system. In Dolphin, a set of Horn clauses and integrity constraints are used to model the managed system.

These clauses and integrity constraints can be considered as the intensional part of a deductive database, while the extensional part of the database is the managed system itself (more about this later). Dolphin models are primarily used for two purposes: to query the state of the managed system, and to change its state. Seen in this light, querying the state of the managed system corresponds to querying the database, and changing the state of the system corresponds to updating the database.

In the querying mode, a top level goal is posed to the Dolphin inference engine, which is reduced to a number of extensional goals using a standard depth-first resolution strategy. The extensional goals are translated into commands which are sent to the underlying system, with the results of these commands translated back into facts which are then used to solve the extensional goals, thus completing the inference process. It is in this sense that the extensional part of the deductive database is the managed system itself.

Changing the state of the managed system is analogous to the intensional update problem for deductive databases [1], where a high level, intensional goal is posed which is translated into a number of lower level, extensional updates. In fact, the first part of the algorithm we use for this purpose is similar to the abductive algorithms developed for intensional updates. There is another dimension to our problem, however. In deductive databases, the updates are independent, and don't have preconditions or side effects. In our case, we don't have updates. We have actions which, when performed on the underlying system, will bring about the desired update. But actions have preconditions and side effects associated with them, which makes it necessary to take into account the interaction between the actions and their side-effects, and the consequences of the side effects on the rest of the system. This paper discusses some of these issues, with emphasis on the side-effect detection and amelioration aspects.

2 Side Effects

Suppose you want to move from the living room to the kitchen next door. The door is closed. You could open the door and walk through; or you could take a battering ram and break through the wall. Both sequences of actions will get you to the kitchen, but one will have more serious side effects than the other. Some consequences of your actions will be unavoidable if you are to attain your goal - for example, if you are in the kitchen, you will no longer be in the living room. Some consequences will be not inevitable, but tolerable - for instance, you might not mind that the door handle squeaks when you turn it. Some consequences can be eliminated - for example, you may not like the door being open, but if so, you can shut it again once you're in the kitchen. Finally, some consequences will be undesirable and difficult to eliminate - such as the destruction of the wall.

In the classic literature on logic-based planning, (eg. [3], [4]) there are two attitudes to side effects. The first is that a plan should produce the goal with NO side effects, in which case there is no plan which will take you from the living room to the kitchen. The second is that as long as the goal is reached, side effects do not matter, in which case you will have to extend your original simple goal of moving from one room to another, or else you may end up minus a wall. We argue that for some applications of computer-assisted planning it is more natural and flexible to allow some side effects, but not others, and to determine via a dialogue with the user which of the side effects are tolerable. In this paper we describe a way to achieve this.

3 Theoretical Framework

In this section we present the theoretical framework of the planning side effect analysis problem. The logical language used is first order predicate calculus in Clausal Normal Form, of which Horn Clauses are a subset.

1. There is a theory T, which is a set of Horn clauses, and a set of integrity constraints I, where each integrity constraint is a clause. (Although we do not have negation as failure in the language, it can be simulated by an abductive style transformation. We omit the details here, since this is not the point of the paper).

   An integrity constraint is allowed to be broken temporarily in the middle of a plan, but must be satisfied by the state reached at the end of a plan.

2. There is a set B of *extensional predicates*; this is just the set of predicates in the language of the Horn clause theory which do not occur in the head of any clause in T. The set of positive ground atoms whose predicate is in B is denoted by B^+.

3. There is a set A of possible actions, each action $a \in A$ has a set $Pre(a)$ of preconditions and a set $Post(a)$ of postconditions. All the conditions in $Post(a)$ are in $B^+ \cup \{\neg b^+ : b^+ \in B^+\}$.

4. There is an initial state s_0, which is a subset of B^+ such that s_0+T+I is consistent. The interpretation of s_0 is that it is the state of the system in which a condition is true iff it is true in the minimal model of T+s_0.

5. There is a goal G.

A sequence of actions $< a_1, a_2, \ldots, a_k >$ *induces*
a sequence of states $< s_0, s_1, \ldots, s_k >$ iff for each $1 \leq i \leq k$,
$s_i = (s_{i-1} \cup$ (positive atoms in $Post(a_i)$) $\setminus$ (positive atoms whose negation is in $Post(a_i)$))
A sequence of actions $< a_1, a_2, \ldots, a_k >$ is *possible* iff it induces $< s_0, s_1, \ldots, s_k >$ and for each i, each condition in $Pre(a)$ is true in the minimal model of T+s_{i-1}.
A sequence of actions $< a_1, a_2, \ldots, a_k >$ *satisfies* G iff it induces $< s_0, s_1, \ldots, s_k >$, it is possible, and G and I are true in the minimal model of T+s_k. In general there may be more than one plan which satisfies G. (Or there may be none.)

4 Example

This section gives an example of the kind of planning problem that we will address in this paper. It is a simplified version of a planning problem that the authors encountered when using the techniques described in this paper to do remote administration of a UNIX workgroup. Variables are written in upper case.

There are two main kinds of objects considered in this example, users and files. The files are uniquely identified by their pathnames, which will be denoted by the variable PN. Users are uniquely identified by their user IDs. Users also have names, denoted by the variable N, which can change. A file can be read by a user if it is owned by that user, if it is world-readable, or if the user is super-user.

Theory T is the theory
canRead(N,PN) ← ownsFile(N,PN)
canRead(N,PN) ← isUserName(N), isFile(PN), readability(PN,*world*)
canRead(N,PN) ← isFile(PN), isSuperUser(N)
ownsFile(N,PN) ← userID(N,ID), IDofOwner(PN,ID)
isUserName(N) ← userID(N,ID)
isFile(PN) ← IDofOwner(PN,ID)

A file can be owner-readable or world-readable. It is possible to put a status lock on a file, for safety reasons; a file can change between the states owner-readable and world-readable only if its status is unlocked. For security reasons, there is an integrity constraint expressing the condition that at the end of a sequence of actions the only user who can be super-user is the user with user ID 0.

Integrity Constraints The set of integrity constraints I is
readability(PN,*owner*) ∨ readability(PN,*world*) ← isFile(PN)
(ID1=ID2) ← readability(PN,ID1), readability(PN,ID2)
(ID1=ID2) ← userID(N,ID1), userID(N,ID2)
(N1=N2) ← userID(N1,ID), userID(N2,ID)
(ID1=ID2) ← IDofOwner(PN,ID1), IDofOwner(PN,ID2)
userID(N,0) ← isSuperUser(N)

Extensional predicates The set of extensional predicates B is
{readability(PN,ID), userID(N,ID), IDofOwner(PN,ID),
isSuperUser(N), statusLocked(PN)}.

Actions
The actions that are available include the following.
(Strictly speaking, the descriptions below are not of actions but of action schemata; they are turned into actions by substituting constants for each variable.)

statusLock(PN)
Pre: ¬statusLocked(PN). Post: statusLocked(PN)

statusUnlock(PN)
Pre: statusLocked(PN). Post: ¬statusLocked(PN)

makeWorldReadable(PN,READ)
Pre: readability(PN,READ), (READ ≠*world*), ¬statusLocked(PN).
Post: ¬readability(PN,READ), readability(PN,*world*)

makeSuperUser(N)
Pre: ¬isSuperUser(N), isUserName(N). Post: isSuperUser(N)

changeFileOwner(PN,ID1,N,ID2)
Pre: IDofOwner(PN,ID1), userID(N,ID2), (ID1 ≠ ID2).
Post: IDofOwner(PN,ID2), ¬IDofOwner(PN,ID1)

In the initial state, there is a file with pathname *pn*; the goal is to make this file readable by the user with name *miranda*.

Initial state The initial state s_0 includes the conditions
IDofOwner(*pn*,15), userID(*kave*,15), userID(*miranda*,10),
statusLocked(*pn*), readability(*pn*,*owner*).

Goal The goal G is just the condition canRead(*miranda*,*pn*).

5 Generating a Sequence of Actions

Generating a sequence of actions to satisfy the goal is a two step process: first, through back chaining, the top level goal is reduced to a number of extensional goals which, if satisfied, would imply the top level goal. Then a planner is used to find a sequence of actions at the end of which the set of extensional goals will be satisfied.

Our action planning system is different from planning systems described in the literature [2] [3] [4] due to the following requirements, which add complexity to the planning process:
1) Actions only have extensional postconditions, but the goal and the preconditions of actions can be in terms of intensional predicates.
2) We allow integrity constraints which are in general expressed in terms of intensional predicates. The sequence of actions generated must be such that at the end of it, none of the integrity constraints are violated.

In the example, we start from the goal ← canRead(*miranda*,*pn*)
and we resolve it with the clause canRead(N,PN) ← ownsFile(N,PN)
we are left with the goal ← ownsFile(*miranda*,*pn*)
We then resolve this goal with the clause
ownsFile(N,PN) ← userID(X,ID), IDofOwner(PN,ID)
which gives us the goals ← userID(*miranda*,ID), IDofOwner(*pn*,ID)
which, when resolved with the assertion userID(*miranda*,10)
in the initial state, will give the goal ← IDofOwner(*pn*,10)
as the residue. Then we invoke the planner to generate a sequence of actions to satisfy this residual goal. The planner would generate the action sequence
<changeFileOwner(*pn*,15,*miranda*,10)> to satisfy this goal.

Notice that there are two levels of non-determinism in the planning process as described above. Firstly, there is non-determinism in the reduction of the top level goal to extensional goals. For example, if we had chosen the clause
canRead(N,PN) ← isUserName(N), isFile(PN), readability(PN,*world*)
to resolve with the top level goal, we would have ended up with a different set of extensional goals to be satisfied by the action generator. Secondly, there is the traditional non-determinism associated with choosing actions to satisfy the extensional goals. Although in this example the possible actions are unique, in general there can be more than one possible action or sequence of actions to satisfy the given set

of extensional goals.

It is not our purpose to give the details of the planning algorithm in this paper, and the description above is included to provide a context for the side-effect detection algorithm.

6 Detecting side effects

In our notation we list the postconditions of each action. These postconditions will be the consequences that we consider. We don't attempt to describe *all* the consequences of an action, just ones whose effects are necessary for planning purposes and/or may be considered undesirable by the user. For example, changing the owner of a file will involve changing some data base entry, and may increase or decrease the number of bytes of data in the data base, but this effect is not recorded as a postcondition of the action changeFileOwner(PN,ID1,N,ID2).

We consider the *reportable side effects* of a sequence of actions to be those logical statements which are in the union of the sets of postconditions of the actions, which are true in the final state, which were not true in the initial state, and which are not direct consequences of the goal (ie. they are not logical consequences of T+G+I). When we consult the user, it is these statements that we will present.

The choice of this set of statements rather than another to be the side effects that we report to the user is to some extent a matter of taste. It could be argued, for instance, that if a statement is originally true, becomes false during the course of the sequence of actions, and then is made true again, then it should be reported as a side effect; we do not do this, because we do not report anything which was true in the original state. Moreover, it is possible that some of the statements we report will be true in any plan which achieves the goal, although they are not logical consequences of T+G+I. We choose to report such statements, because they may be undesirable to the user. It is safer to give the user the chance of rejecting all plans to move into the kitchen if there really is no way to do it from the given initial state without knocking down the wall, than to go ahead with the battering ram.

How to calculate the reportable side effects

Given a sequence of actions $< a_1, \ldots, a_k >$ satisfying G, it is straightforward to calculate the reportable side effects. This section gives a not particulary efficient, but simple, algorithm which does this calculation.

The algorithm takes as input not only the sequence of actions and G, but also a set *Allowed side effects.* If there has been no communication yet with the user, this set is empty. Conditions are added to it by the algorithm. After each iteration of the algorithm the user has the option of designating some of the reported side effects as OK and not worth reporting, and others as unacceptable. (The remaining side effects may be generated by future plans, but if they are they will be reported to the user.) The side effects that the user indicates are OK and not worth reporting

are added to the set *Allowed side effects.* A new goal is derived, which is just the old goal plus the negations of all the unacceptable side effects. The new goal is fed into the plan generator, which comes up with a sequence of actions to satisfy the new goal (if it can find one); this new sequence of actions is used as input for another iteration of the algorithm to find the reportable side effects, these effects are reported to the user, and so on until the user is satisfied or no sequence satisfying the goal is found.

1. Initialize the set R to the empty set, and *counter* to k.
2. Set $P = Post(a_{counter}) \setminus R$.
3. For each $post \in P \setminus Allowed\ side\ effects$ such T+G+I proves *post*, add *post* to *Allowed side effects* and to R, and remove it from P.
4. Pick $post \in P$. If neither *post* nor $\neg post$ are in R, then add *post* to R.
5. Remove *post* from P. If P is nonempty, return to step 4. Otherwise go on to step 6.
6. If *counter*> 1 then decrease *counter* by one and return to step 2; else go on to step 7.
7. For each *post* in $R \cap Allowed\ side\ effects$, remove *post* from R.
8. For each *post* in $R \cap s_0$, remove *post* from R. For each negative $post \in R$ such that $\neg post \notin s_0$, remove *post* from R.
9. R is now the set of the reportable side effects. Report it to the user.

It is straightforward to check that at the beginning of step 7 the set R contains exactly the conditions *post* which are not direct effects of the goal and are a post-condition of some action a_j where $\neg post$ is not a postcondition of any of the actions $a_{j+1}, a_{j+2}, \ldots, a_k$. It follows from the definition of s_k that the side effects reported to the user are exactly those conditions that are in a postcondition of one of the actions, that are true in the state represented by s_k and false in the state represented by s_0, and that are not direct effects of the goal G.

The user has the option of adding conditions to the set *Allowed side effects*, or changing the goal. There are occasions when changing the goal may be particularly useful. For example, suppose that a plan is generated with reported side effect IDofOwner(*pn*,1). The user says that this is unacceptable, because he or she doesn't want the ownership of the file with pathname *pn* to move from its original owner, who has user ID 15. A new plan is generated with reported side effect IDofOwner(*pn*,2). The user doesn't like this either and a new plan is generated with reported side effect IDofOwner(*pn*,3). The user now spots a pattern and adds the condition IDofOwner(*pn*,15) to the goal. The effect of adding conditions to the set *Allowed side effects* is that these will not be reported if they arise as side effects. This doesn't change the plans that are generated, but can make life simpler for the user by ensuring that irrelevant information is suppressed.

An optional way of further simplifying the data reported to the user is to remove side effects which are redundant because they are implied by other reported side effects together with the integrity constraints. To do this, pick *post* in R and check whether $G+R \setminus \{post\}+\neg post$ violates any of the integrity constraints I. If it does,

remove *post* from R. Pick a new *post* in R which has not yet been checked, and repeat, until all members of R have been checked.
This can be computationally complex, and suppresses the reporting of some side effects which the user may actually want to know about, so we do not make it an integral part of the side detection algorithm.

Using the planner and side effect analyser together

One advantage of the algorithm given above is that is possible to use it incrementally, in conjunction with the plan generator. Suppose the plan generator produces a sequence of actions which satisfies the goal starting from an initial state which is not s_0 but some other state s'_0. Steps 1-5 of the algorithm for side effect detection can be used to find an interim set R' of side effects, equal to the value of R after these steps. R' is stored for later reference. The plan generator can then produce two different sequences of actions which satisfy s'_0 starting at initial state s_0. Now the side effects of the two different plans to satisfy G (which have the same ending but different beginnings) can be calculated, by setting R equal to R' and applying steps 2-9 of the algorithm to the two sequences of actions to satisfy s'_0. More complicated backtracking manoevres during the planning can similarly be followed without too much recalculation by the side effect detector, by appropriate storage of partial results.

7 Discussion of the example

The goal of the example can be made true, for example, by
- transferring ownership of the file from *kave* to *miranda*;
- turning off the status lock and then changing the file to world-readable;
- making *miranda* super-user - but this violates one of the integrity constraints.

Suppose that the user had in mind changing the readability, rather than changing the file's owner. Here is an example of the steps that could be gone through by the user, planner and side effect analyser;
1. Planner generates the simple plan <changeFileOwner(*pn*,15,*miranda*,10)>
2. Side effect analyser reports side effects IDofOwner(*pn*,10), ¬IDofOwner(*pn*,15) to the user
3. User says that the effect ¬IDofOwner(*pn*,15) is unwanted. The new goal canRead(*miranda*,*pn*), IDofOwner(*pn*,15) is sent back to the planner.
4. Planner generates <statusUnlock(*pn*), makeWorldReadable(*pn*,*owner*)>.
5. Side effect analyser reports side effects
¬statusLocked(*pn*), ¬readability(*pn*,*owner*), readability(*pn*,*world*).
(If the optional step removing redundant effects from the set of reported effects were used, then ¬readability(*pn*,*owner*) would not be reported.)
6. User says that the effect ¬statusLocked(*pn*) is unwanted, but that the other two are OK. The conditions ¬readability(*pn*,*owner*), readability(*pn*,*world*) are added to

the set *Acceptable side effects*, and the goal canRead(*miranda*,*pn*), IDofOwner(*pn*,15), statusLocked(*pn*)
is sent back to the planner.
7. Planner generates
<statusUnlock(*pn*), makeWorldReadable(*pn*), statusLock(*pn*)>
8. Side effect analyser calculates that there are no reportable side effects. So the plan is fine. A note of the plan is sent to the user; the plan is scheduled.

This procedure for generating a sequence of actions which will satisfy the goal may not in theory terminate. In practice, there is a parameter (which the user can change, default is 10) which is used to bound the number of actions in a sequence. If the procedure fails to find any sequences with length less than the parameter satisfying the goal, the search is terminated. Sequences with too many actions are undesirable because uncertaincies about the exact effects of actions are cumulative when actions are performed in sequence.

References

[1] Bry, F. *Intensional updates: abduction via deduction* Proc. ICLP 90

[2] Denecker, M., Missiaen, L., Bruynooghe, M., *Temporal reasoning with Abductive Event Calculus* Proc. ECAI 92

[3] Chapman, D., *Planning for conjunctive goals* Artificial Intelligence Vol. 32, 1987

[4] Fikes, R. E. & Nilsson, N. J *STRIPS: a new approach to the application of theorem proving to problem solving*, Artificial Intelligence Vol. 2, 1971

[5] Gelfond, M. & Lifschitz, V. *Representing actions in extended logic programming* Proc. JICSLP 92

[6] Kowalski, R. A. *Database updates in Event Calculus* The Journal of Logic Programming 12

[7] Phan Minh Dung, *Representing actions in logic programming and its application in database updates* Proc. ICLP 93

Downward Refinement of Hierarchical Datalog Theories

F. Esposito, N. Fanizzi, D. Malerba and G. Semeraro

Dipartimento di Informatica, Università degli Studi di Bari

Via E. Orabona 4, 70126 Bari, ITALY

Phone: +39 (80) 5443264 Fax: +39 (80) 5443196

E-mail: {esposito, malerbad, semeraro}@vm.csata.it nico@lacam.uniba.it

Abstract. In theory revision, a fundamental operation is the specialization of incorrect theories. In the paper, we propose a novel downward refinement operator ρ_{OI} for hierarchical Datalog theories, which is able to compute a set of most general specializations of an overly general clause. We prove that the operator meets the fundamental properties of properness, local finiteness, and completeness in the space of the Datalog program clauses ordered by the relation of θ-subsumption under object identity. Experimental results show that ρ_{OI} is able to cope effectively and efficiently with the task of revising logical theories for document classification.

Keywords. Theory revision, specialization, θ-subsumption, deductive databases, Datalog theories.

1. Introduction

A theory *T* may be incorrect because, given a new observation *E*, one of the following two cases occurs: 1) *E* is erroneously explained by the theory. Thus, the theory is too general and needs to be specialized. 2) *E* is erroneously not explained by the theory. Thus, the theory is too specific and needs to be generalized. In this paper, we limit to address the first problem. Moreover, this problem is dealt with in a logical framework, since the theory is represented as a set of hypotheses and in turn each hypothesis is a set of clauses expressed in a first-order logic language. The solution to such a problem requires to perform a search for a specialization (or downward refinement) *T'* of the theory *T* such that *E* is not explained by *T'*. This search aims at finding a *minimal* downward refinement of a theory [18]. The same goal is pursued in the field of belief revision [8] and theory contraction [7]. Indeed, in a logic-constrained belief revision approach, a contraction of a belief set with respect to a new fact consists of a peculiar belief change, which requires the retraction of the information causing the violation of consistency, when this property is regarded as an integrity constraint.

In the following section, we briefly present the logic language used to represent observations (*examples*) and theories. In addition to the basic definitions, the model of generalization of θ-subsumption under *object identity*, called θ_{OI}*-subsumption*, is formally defined and compared to θ-subsumption. A novel downward refinement operator for θ_{OI}-

subsumption is defined in section 3. It satisfies relevant properties for refinement operators, such as *local finiteness*, *properness*, and *completeness* [16, 2]. An example of application of such operator to the real-world task of document classification is outlined in section 4.

2. The Representation Language

Henceforth, we refer to [11] for what concerns the basic definitions of a *substitution*, *positive* and *negative literal*, *clause*, *definite* and *program clause*, and *normal program*. Given a first-order expression ϕ, *vars*(ϕ) and *consts*(ϕ) denote respectively the set of the variables and the set of the constants occurring in ϕ. By *logical theory* we mean a set of *hypotheses*, by *hypothesis* we mean a set of program clauses with the same head. In the paper, we are concerned exclusively with logical theories expressed as *hierarchical normal programs*, that is, as normal programs for which it is possible to find a *level mapping* [11] such that, in every program clause $P(t_1, t_2,\ldots,t_n) \leftarrow L_1, L_2,\ldots,L_m$, the level of every predicate symbol occurring in the body is less than the level of *P*. Another constraint on the language is that, whenever we write about clauses, we mean *Datalog* (i.e., function-free) *linked* clauses [10].

Definition 1 (Linked clause) A clause is *linked* if all of its literals are. A positive literal is linked if at least one of its arguments is. An argument of a positive literal is linked if either the literal is in the head of the clause or another argument in the same literal is linked. A negative literal is linked if at least one of its arguments occurs in a linked positive literal.

An instance of a linked clause is the following: $C = P(x) \leftarrow Q(x, y), Q(y, z), \neg R(x, v)$ *C* is linked since all its literals are linked. Conversely, both the clauses $D = C - \{Q(x, y)\}$ and $F = C \cup \{\neg R(v, w)\}$ are not linked. Indeed, the literal $Q(y, z)$ is not linked in *D*, whereas $\neg R(v, w)$ is not linked in *F*. Henceforth, we will indifferently use the set notation and the Prolog notation for clauses and with $|C|$ we will denote the number of literals of a clause *C*.

Semantically, we adopt *negation-as-failure* rule [1] to define the meaning of a negated literal in the body of a program clause.

The differences existing between examples and hypotheses are as follows.

- Each example is represented by one *ground* clause with a unique literal in the head.
- Each hypothesis is a set of constant-free program clauses with the same *head*.

An *example* is *positive* for a hypothesis if its head has the same predicate letter and sign as the head of the hypothesis. An example is *negative* for a hypothesis if its head has the same predicate as the head of the hypothesis, but opposite sign. Thus, more precisely, a negative example is a *generally Horn clause* [9]. Subsequently, we define a quasi-ordering $\leq_{OI}$ on the set LP of the Datalog linked program clauses . This ordering is inspired from the notion of *θ-subsumption* [12] and makes the assumption that terms denoted with different symbols must be distinct (*object identity*). For instance, $P(x) \leftarrow Q(x, a), R(a, z)$ denotes the clause $P(x) \leftarrow Q(x, a), R(a, z), [x \neq a], [x \neq z], [a \neq z]$, under the object identity assumption.

Definition 2 (θ_{OI}-subsumption ordering) Let C, D be two elements of **LP**. We say that *D θ-subsumes C under object identity* (*D θ_{OI}-subsumes C*) if and only if (iff) there exists a substitution σ such that $D.\sigma \subseteq C$ and *σ is a one-to-one mapping*. In such a case, we say that *D is more general than or equal to C* (*D is a generalization* of *C* and *C is a specialization of D*) *under object identity* and we write $C \leq_{OI} D$. We write $C <_{OI} D$ when $C \leq_{OI} D$ and $not(D \leq_{OI} C)$ and we say that *D is more general than C* (*D is a proper generalization of C*) or *C is more specific than D* (*C is a proper specialization of D*). We write $C \sim_{OI} D$, and we say that *C* and *D* are *equivalent clauses*, when $C \leq_{OI} D$ and $D \leq_{OI} C$.

θ_{OI}-subsumption is a strictly weaker order relation than θ-subsumption. A thorough analysis of θ_{OI}-subsumption as a generalization model can be found in [14].

A logical theory is *incorrect* if it is either inconsistent or incomplete. More formally, we introduce the following definitions.

Definition 3 (Inconsistency) A *theory T* is *inconsistent* iff at least one of its hypotheses is inconsistent with respect to (wrt) some negative example. A *hypothesis* is *inconsistent* wrt a negative example *N* iff at least one of its clauses is inconsistent wrt *N*. A *clause C* is *inconsistent* wrt *N* iff there exists a one-to-one substitution σ such that the following conditions are satisfied: 1) $\text{body}(C).\sigma \subseteq \text{body}(N)$, 2) $\neg\ \text{head}(C).\sigma = \text{head}(N)$

where *body(φ)* and *head(φ)* denote the *body* and the *head* of a clause φ, respectively. If at least one of the two conditions above is not met, we say that *C* is *consistent* wrt *N*.

Definition 4 (Incompleteness) A *theory T* is *incomplete* iff at least one of its hypotheses is incomplete wrt some positive example. A *hypothesis* is *incomplete* wrt a positive example *P* iff each of its clauses does not θ_{OI}-subsume *P*.

When a theory turns out to be inconsistent, the inconsistent clauses should be removed and specializations of removed clauses should be added until the consistency property of the theory is restored. These specializations are computed by a *downward refinement operator*.

Definition 5 (Basic definitions) Given a quasi-ordered set $(\mathbf{T}, \leq)$ and a clause *C* in **T**:

1) a *downward refinement operator* ρ is a mapping from **T** to $2^{\mathbf{T}}$, $\rho : \mathbf{T} \rightarrow 2^{\mathbf{T}}$ such that for every *C* in **T**, $\rho(C)$ is a subset of the set $\{ D \in \mathbf{T} \mid D \leq C \}$
2) let ρ be a downward refinement operator, then

   $\rho^0(C) = \{C\}$ $\qquad$ $\rho^n(C) = \{ D \mid \exists E \in \rho^{n-1}(C) \text{ and } D \in \rho(E) \}$

   $\rho^*(C) = \cup_{n\geq 0}\ \rho^n(C) = \rho^0(C) \cup \rho^1(C) \cup \ldots \cup \rho^n(C) \cup \ldots$
3) ρ is called *locally finite* iff $\forall C \in \mathbf{T} : \rho(C)$ is finite and computable

   ρ is called *proper* iff $\forall C \in \mathbf{T} : \rho(C) \subseteq \{ D \in \mathbf{T} \mid D < C \}$

   ρ is called *complete* iff $\forall C, D \in \mathbf{T}$, if $D < C$ then $\exists E$ s.t. $E \in \rho^*(C)$ and $E \sim D$

   ρ is called *ideal* iff it is locally finite, proper and complete.

3. A Downward Refinement Operator

The downward refinement operator proposed in this section relies on the addition of a non-redundant literal to a clause that turns out to be inconsistent wrt a negative example. The space in which such a literal should be searched for is potentially infinite, thus an exhaustive search is infeasible. We can formally define the search space as the partially ordered set (or *poset*) ($\mathbf{LP}/\sim_{OI}, \leq_{OI}$), where $\mathbf{LP}/\sim_{OI}$ is the quotient set of the Datalog linked program clauses and $\leq_{OI}$ is the quasi ordering relation defined in section 2, which can be straightforwardly extended to equivalence classes under $\sim_{OI}$ [14]. Henceforth, we will always work on the quotient set $\mathbf{LP}/\sim_{OI}$ and, when convenient, we will denote with the name of a clause the equivalence class it belongs to.

The search strategy used to solve the problem of downward refinement takes advantage of the structure of the search space. The search is firstly performed in the space of positive literals, containing information coming from the positive examples. When the search in this space fails, it is extended to the space of negative literals, built by taking into account the negative example wrt which the hypothesis turned out to be inconsistent.

Firstly, given a hypothesis *H* which is inconsistent wrt a negative example *N*, all the clauses of *H* that caused the inconsistency are detected. For each inconsistent clause *C*, let us suppose that the subset of the positive examples θ_{OI}-subsumed by *C* were $\{P_1, P_2, \ldots, P_n\}$. The search aims at finding one of the *most general downward refinements under object identity* of *C* against *N* given $P_1, P_2, \ldots, P_n$, denoted with $mgdr_{OI}(C, N \mid P_1, P_2, \ldots, P_n)$. The set of the *most general downward refinements under object identity* of *C* against a negative example *N*, denoted with $mgdr_{OI}(C, N)$, is defined as follows.

$mgdr_{OI}(C, N) = \{ M \mid M <_{OI} C, M$ consistent wrt N, $\forall D \;\; D \leq_{OI} C, D$ consistent wrt N :

$not(M <_{OI} D) \}$

while the set $mgdr_{OI}(C, N \,/\, P_1, P_2,\ldots, P_n)$ is defined as the subset of $mgdr_{OI}(C, N)$ made up of all the clauses that θ_{OI}-subsume the positive examples $P_1, P_2,\ldots, P_n$. Formally:

$$mgdr_{OI}(C, N \,/\, P_1, P_2,\ldots, P_n) = \{M \in mgdr_{OI}(C, N) \mid P_j \leq_{OI} M, j=1,2,\ldots, n\}$$

Throughout this section, we shall denote with C a clause that needs to be specialized, since it is inconsistent with respect to an example N. More precisely, the body of C needs to be subjected to a suitable process of downward refinement. Let us consider the problem of finding one of the clauses in the set $mgdr_{OI}(C, N \,/\, P_1, P_2,\ldots, P_n)$.

Since the downward refinements we are looking for must satisfy the property of maximal generality, it may happen that the specializations of C are overly general, even after some refinement steps. This suggests the possibility of further exploiting the positive examples in order to specialize C. Specifically, if there exists a literal that, when added to the body of C, is able to discriminate from the negative example N that caused the inconsistency of C, then the downward refinement operator should be able to find it.

The process of refining a clause by means of positive literals can be described as follows. For each P_i (i=1, 2,..., n), let us suppose that there exist n_i distinct substitutions such that C θ_{OI}-subsumes P_i. Then, let us consider all the possible n-tuples of substitutions obtained by selecting one of such substitutions for every positive example. Each of these substitutions is used to produce a distinct *residual*, consisting of all the literals in the positive example that are not involved in the θ_{OI}-subsumption test, after having properly turned their constants into variables. Formally, a residual can be defined as follows.

Definition 6 (Residual) Let C be a clause, E an example, and σ_j a one-to-one substitution such that $body(C).\sigma_j \subseteq body(E)$. A *residual* of E wrt C under the mapping σ_j, denoted by $\Delta_j(E, C)$, is the following set of literals. $\quad \Delta_j(\mathrm{E}, \mathrm{C}) = \mathrm{body}(\mathrm{E}).\underline{\sigma}_j^{-1} - \mathrm{body}(\mathrm{C})$

where $\underline{\sigma}_j^{-1}$ is the *extended antisubstitution* (or *inductive substitution*) obtained by inverting the corresponding substitution σ_j. Indeed, an antisubstitution is a mapping from terms into variables [15]. When a clause C θ_{OI}-subsumes an example E through a substitution σ, then it is possible to define a corresponding antisubstitution, σ^{-1}, which is exactly the inverse function of σ. Indeed, σ is a one-to-one function, due to object identity assumption. Then, σ^{-1} maps some constants in E to variables in C, that is: σ^{-1}: $\mathrm{vars}(\mathrm{C}).\sigma \rightarrow \mathrm{vars}(\mathrm{C})$.

It should be observed that not all constants in E have a corresponding variable according to σ^{-1}. Therefore, for our purposes, we introduce the extension of σ^{-1}, denoted with $\underline{\sigma}^{-1}$, that is defined on the whole set of constants occurring into E, *consts(E)*, and takes values in the set of the variables of the language.

$$\underline{\sigma}^{-1}(c_n) = \begin{cases} \sigma^{-1}(c_n) & \text{if } c_n \in \mathrm{vars}(C).\sigma \\ \_ & \text{otherwise} \end{cases}$$

Henceforth, variables denoted by _ will be called *new* variables and managed as in Prolog.

The residuals obtained from the positive examples P_i, i = 1, 2,..., n, can be exploited to build a *space of complete positive downward refinements*, denoted with $\mathbb{P}$, and formally defined as follows.

$$\mathbb{P} = \bigcup_{\substack{i=1,2,\ldots n \\ j_i=1,2,\ldots n_i}} \; \bigcap_{k=1,2,\ldots n} \Delta_{j_k}(P_k, C)$$

Moreover, let us denote with θ_j, j = 1, 2,..., m, all the substitutions which make C inconsistent wrt N. Let us define a new set of literals.

$$\mathbb{S} = \bigcup_{j=1,2,\ldots,m} \Delta_j(N, C)$$

Then, the following theorem holds :

Theorem 1. *Given a clause C that θ_{OI}-subsumes the positive examples $P_1, P_2,\ldots, P_n$ and is inconsistent wrt the negative example N, then any linked clause $C' = C \cup \{ l\}$, with $l \in \mathbb{P} - \mathbb{S}$, is in $mgdr_{OI}(C, N \,/\, P_1, P_2,\ldots, P_n)$.*

Formally: $\qquad \{ C' \mid C' = C \cup \{ l\}, l \in \mathbb{P} - \mathbb{S} \} \subseteq mgdr_{OI}(C, N \mid P_1, P_2,\ldots, P_n)$

Note that l is an element of *body(C')*, since it is negated.

Proof. See Appendix A.

Theorem 1 states that every downward refinement built by adding a literal in $\mathbb{P}$ - $\mathbb{S}$ to the inconsistent clause C restores the properties of consistency and completeness of the original hypothesis.

Let us suppose now that the operator did not succeed in refining C wrt N in a complete and consistent way. In such a case, a change of representation must be performed in order to search for literals in another space (the operator that performs this process is mainly a transposition of a similar operator for VL_{21} clauses to clausal logic [3]). Therefore, it is necessary to define a new target space, called the *space of negative downward refinements*. Given a clause C, an example N and the set of all substitutions θ_j, j = 1, 2,...,m, such that

C is inconsistent wrt N, the *space of negative downward refinements*, denoted with S_n, is the following set of literals. $S_n = neg(S) = neg(\cup_{j=1,2,\ldots,m} \Delta_j(N, C))$

where, given a set of literals $\varphi = \{ l_1, l_2, \ldots, l_n \}$, $n \geq 1$, $neg(\varphi)$ denotes the set of literals $\{\neg l_1, \neg l_2, \ldots, \neg l_n \}$. As for the process of downward refinement by positive literals, we are interested into a specific subset of S_n, because of the properties satisfied by its elements. Such a subset, called *space of consistent negative downward refinements*, is denoted with S_c and is defined as follows. $S_c = neg(\cap_{j=1,2,\ldots,m} \Delta_j(N, C))$

Indeed, it is possible to prove the following result:

Theorem 2. *Given a clause C, an example N and the set of all substitutions θ_j, $j=1,2,\ldots,m$, such that C is inconsistent wrt N, then any linked program clause $C' = C \cup \{ l\}$, with $l \in S_c$, is in $mgdr_{OI}(C, N)$.*

Formally: $\{ C' \mid C' = C \cup \{ l\}, l \in S_c \} \subseteq mgdr_{OI}(C, N)$

Note that l is a negated literal occurring in the body of C'.

Proof. See Appendix B.

Theorem 2 easily extends to any linked literal l which introduces new variables, due to negation-as-failure rule. Generally speaking, we can say that, given a clause C and an example N such that C is inconsistent wrt N due to some substitutions θ_j, $j = 1,2,\ldots, k$, the search for a complete and consistent hypothesis can be viewed as a two-stage process: the former stage searches into the space $P - S$, the latter into S_c. By means of Theorems 1 and 2, we are now able to formally define our novel downward refinement operator, denoted with ρ_{OI}

Definition 7 (The downward refinement operator ρ_{OI})

$$\rho_{OI}: LP \rightarrow 2^{LP}, \forall C \in LP: \rho_{OI}(C) = \{ C' \mid C' = C \cup \{ l\}, l \in (P - S) \cup S_c \}$$

Theorem 3. *The downward refinement operator ρ_{OI} is ideal.*

Proof. See Appendix C.

The ideality of the refinement operator ρ_{OI} is owed to the peculiar structure of the search space when ordered by the relation $\leq_{OI}$. In the same search space ordered by θ-subsumption, an ideal refinement operator does not exist, as stated by the following result [16, page 315]: "*Theorem 10. A locally finite, complete and proper downward refinement operator for unrestricted search spaces ordered by θ-subsumption does not exist.*"

This property is a consequence of the existence of uncovered infinite strictly ascending chains of clauses in an unrestricted search space.

4. Application to Document Classification

The downward refinement operator has been applied in the area of *document classification* [4], which is a crucial step in the task of electronic document processing. Some experiments have been performed in order to empirically verify that the operator ρ_{OI} is effective and efficient to revise inconsistent theories. To this purpose, we implemented ρ_{OI} into INCR/H [5], an incremental system for theory revision, and compared the theories produced by INCR/H to those inferred *from scratch* by INDUBI/H [4], along two dimensions - predictive accuracy and computational time.

The results obtained show that the operator ρ_{OI} is able to produce theories whose predictive accuracy is statistically comparable to that of theories inferred *from scratch*. Nevertheless, as expected, the overall efficiency of the process of theory inference results largely increased. These promising results lead us to take into consideration the idea of integrating the downward refinement operator into PLRS [13], the learning module of IBIsys, a software environment for office automation distributed by Olivetti.

A thorough description of the application to document classification can be found in [6].

5. Conclusions

The problem of clause specialization is central to theory revision. It can be cast as a search through a space of clauses ordered by a generality relation. We have proposed a downward refinement operator that takes advantage of the structure of the search space in order to restore the correctness property of a theory by finding its minimal specialization. An extensive experimentation in the domain of office document classification has shown that this operator is able to refine effectively and efficiently logical theories for document classification.

References

1. Clark, K. L., Negation as failure, in *Logic and Databases*, H. Gallaire and J. Minker (Eds.), Plenum Press, New York, 293-321, 1978.
2. De Raedt, L., *Interactive Theory Revision*, Academic Press, San Diego, CA, 1992.
3. Esposito, F., Malerba, D., and Semeraro, G., Negation as a Specializing Operator, in *Advances in Artificial Intelligence*, Lecture Notes in Artificial Intelligence 728, P. Torasso (Ed.), Springer-Verlag, 166-177, 1993.
4. Esposito, F., Malerba, D., and Semeraro, G., Multistrategy Learning for Document Recognition, *Applied Artificial Intelligence*, Vol.8, No.1, 33-84, 1994.

5. Esposito, F., Malerba, D., and Semeraro, G., INCR/H: A System for Revising Logical Theories, in *Proceed. of the MLnet Workshop on Theory Revision and Restructuring in Machine Learning*, ECML-94, Arbeitspapiere der GMD N.842, S. Wrobel (Ed.), 13-15, 1994.
6. Esposito, F., Fanizzi, N., Malerba, D., and Semeraro, G., Revision of Logical Theories, in *Proceed. of the 4th Congress of the Italian Association for Artificial Intelligence*, Lecture Notes in Artificial Intelligence, G. Soda (Ed.), Springer-Verlag, 1995 (forthcoming).
7. Fuhrmann, A., Theory Contraction through Base Contraction, *Journal of Philosophical Logic* 20, 175-203, 1991.
8. Gärdenfors, P., and Rott, H., Belief Revision, in *Handbook of Logic in AI and Logic Programming, Vol. IV: Epistemic and Temporal Reasoning*, Chapter 4.2, 1992.
9. Grant, J., & Subrahmanian, V.S., Reasoning in Inconsistent Knowledge Bases, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, N.1, 177-189, 1995.
10. Helft, N., Inductive Generalization: A Logical Framework, in *Progress in Machine Learning - Proceedings of EWSL 87*, I. Bratko & N. Lavrac (Eds.), Sigma Press, Wilmslow, 149-157, 1987.
11. Lloyd, J.W., *Foundations of Logic Programming*, Second Edition, Springer-Verlag, New York, 1987.
12. Plotkin, G. D., A Note on Inductive Generalization, in *Machine Intelligence 5*, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, 153 - 163, 1970.
13. Semeraro, G., Esposito, F., and Malerba D., Learning Contextual Rules for Document Understanding, *Proceed. of the 10th Conference on Artificial Intelligence for Applications (CAIA'94)*, IEEE Computer Society Press, Los Alamitos, CA, 108-115, 1994.
14. Semeraro, G., Esposito, F., Malerba, D., Brunk, C., and Pazzani, M., Avoiding Non-Termination when Learning Logic Programs: A Case Study with FOIL and FOCL, in *Logic Program Synthesis and Transformation - Meta-Programming in Logic*, Lecture Notes in Computer Science 883, L. Fribourg and F. Turini (Eds.), Springer-Verlag, 183-198, 1994.
15. Siekmann, J. H., An Introduction to Unification Theory, in *Formal Techniques in Artificial Intelligence - A Sourcebook*, R. B. Banerji (Ed.), Elsevier Science Publishers B.V. (North Holland), 1990.
16. van der Laag, P. R. J., and Nienhuys-Cheng, S.-H., Existence and Nonexistence of Complete Refinement Operators, in *Machine Learning: ECML-94 - Proceedings of the European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 784, F. Bergadano and L. De Raedt (Eds.), Springer-Verlag, 307-322, 1994.
17. VanLehn, K., Efficient Specialization of Relational Concepts, *Machine Learning 4*, 1, 99-106, 1989.
18. Wrobel, S., On the proper definition of minimality in specialization and theory revision, in *Machine Learning: ECML-93 - Proceedings of the European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 667, Pavel B. Brazdil (Ed.), Springer-Verlag, 65-82, 1993.

Appendix A.

In order to prove $C' <_{OI} C$, let us observe that in the space $(\mathrm{LP}/\sim_{\mathrm{OI}}, \leq_{\mathrm{OI}})$ the set of all constant-free generalizations of a clause C' corresponds to the set $2^{C'}$, thus each proper generalization of C' has a number of literals less than the number of literals of C' [17]. Therefore $C \subset C' \Rightarrow C' <_{OI} C$, i.e. C' is a proper downward refinement of C under θ_{OI}-subsumption.

Let us show now that C' is consistent wrt N. First of all, observe that

$$\forall j=1, 2,\ldots, m: \neg\mathrm{head}(C').\theta_j = \neg\mathrm{head}(C).\theta_j = \mathrm{head}(N).$$

Moreover,

$\forall j=1, 2, \ldots, m : \mathrm{body}(C').\theta_j = \mathrm{body}(C').\underline{\theta}_j = (\mathrm{body}(C) \cup \{l\}).\underline{\theta}_j =$

$\mathrm{body}(C).\underline{\theta}_j \cup \{l\}.\underline{\theta}_j$ (1)

$l \in \mathbb{P} - \mathbb{S} \Rightarrow l \notin \mathbb{S} = \cup_{j=1,2,\ldots,m} \Delta_j(\mathrm{N}, \mathrm{C}) \Rightarrow \forall j=1, 2,\ldots, m : 1 \notin \Delta_j(\mathrm{N}, \mathrm{C})$.

By definition of $\mathbb{P}$, $l \in \mathbb{P} \Rightarrow l \notin \mathrm{C}$, then $\forall j=1, 2,\ldots, m$: $l \notin \mathrm{body}(\mathrm{N}).\underline{\theta}_j^{-1} \Rightarrow$

$l.\underline{\theta}_j \notin \mathrm{body}(\mathrm{N}) \Rightarrow \{l\}.\underline{\theta}_j \not\subset \mathrm{body}(\mathrm{N})$.

Then, looking back at (1), we can conclude that: $\forall j=1, 2,\ldots, m$: $\mathrm{body}(C').\theta_j \not\subset \mathrm{body}(\mathrm{N})$.

This proves that C' is consistent wrt N. Indeed, any other substitution causing the inconsistency of C' would be a superset of a θ_j ($j=1, 2, \ldots, m$) because of our assumption that they were the only possible one-to-one substitutions s.t. $\mathrm{body}(C).\theta_j \subseteq \mathrm{body}(N)$ and we have just proved that each of them makes C' consistent wrt N.

Now suppose that $\exists\, F$ which is consistent wrt N and s.t.

$C' <_{OI} F \leq_{OI} C \Rightarrow |C'| = |C|+1 > |F| \geq |C| \Rightarrow |F| = |C|$. But F is a specialization of C, then it can be inferred that $F \sim_{OI} C$. Thus F is inconsistent wrt N, just as C.

According to the hypotheses of the theorem:

$\forall\, k=1, 2, \ldots, n$: $\mathrm{head}(C).\sigma_{j_k} = \mathrm{head}(P_k)$ and $\mathrm{body}(C).\sigma_{j_k} \subseteq \mathrm{body}(P_k)$ (σ_{j_k} are the substitutions that appear in the definition of $\mathbb{P}$).

$l \in \mathbb{P} \Rightarrow l \in \cap_{k=1,2,\ldots,n} \Delta_{j_k}(P_k, C) \Rightarrow \forall\, k=1, 2, \ldots, n$: $l \in \mathrm{body}(P_k).\underline{\sigma}_{j_k}^{-1} \Rightarrow$

$\forall\, k=1, 2, \ldots, n$: $l.\underline{\sigma}_{j_k} \in \mathrm{body}(P_k)$.

Then, $\mathrm{body}(C').\underline{\sigma}_{j_k} = \mathrm{body}(C).\underline{\sigma}_{j_k} \cup \{l\}.\underline{\sigma}_{j_k} \subseteq \mathrm{body}(P_k)$, $\forall\, k=1, 2, \ldots, n$. Then, $P_k \leq_{OI} C'$.

□

Appendix B.

As to the proof that C' is a proper downward refinement of C under θ_{OI}-subsumption, refer to Appendix A.

Given a linked program clause $C' = C \cup \{l\}$, with $l \in \mathbb{S}_c$, in order to prove that C' is consistent wrt N, let us suppose (reductio ad absurdum) there exists a substitution σ_1 s.t. C' is inconsistent wrt N. Then, from definition 3, it results that:

1) $\text{body}(C').\sigma_1 \subseteq \text{body}(N)$ 2) $\neg\, \text{head}(C').\sigma_1 = \text{head}(N)$

As a consequence, σ_1 is also one of the k substitutions that make C inconsistent wrt N. We also have from the hypotheses of the theorem:

$\text{body}(C').\sigma_1 = \text{body}(C').\underline{\sigma}_1 = (\text{body}(C) \cup \{l\}).\underline{\sigma}_1 = \text{body}(C).\underline{\sigma}_1 \cup \{l\}.\underline{\sigma}_1$

with $l \in \mathbb{S}_c = \text{neg}(\cap_{j=1,2,\dots,m} \Delta_j(N, C))$. But:

$\{l\}.\underline{\sigma}_1 \subseteq \mathbb{S}_c.\underline{\sigma}_1 \subseteq \text{neg}(\Delta_1(N, C)).\underline{\sigma}_1 = \text{neg}(\text{body}(N).\underline{\sigma}_1^{-1} - \text{body}(C)).\underline{\sigma}_1$

$= \text{neg}(\text{body}(N).\underline{\sigma}_1^{-1}.\underline{\sigma}_1 - \text{body}(C).\underline{\sigma}_1) = \text{neg}(\text{body}(N) - \text{body}(C).\underline{\sigma}_1) \subseteq \text{neg}(\text{body}(N))$ and

$\{l\}.\underline{\sigma}_1 \subseteq \text{body}(C').\underline{\sigma}_1 = \text{body}(C').\sigma_1 \subseteq \text{body}(N)$, according to 1).

But this is impossible since $\text{body}(N) \cap \text{neg}(\text{body}(N)) = \emptyset$.

In order to prove that C' is in $mgdr_{OI}(C, N)$, it remains to demonstrate that:

$\forall D,\ D \leq_{OI} C, D$ consistent wrt N: $not(\, C' <_{OI} D\,)$.

Suppose (ad absurdum) that there exists D s.t. $D \leq_{OI} C$, D consistent wrt N and $C' <_{OI} D$. Then:

$C' <_{OI} D \Rightarrow |D| < |C'| = |C|+1 \Rightarrow |D| \leq |C|$ (2)

But: $D \leq_{OI} C \Rightarrow |D| \geq |C|$. (3)

Therefore, from (2) and (3), it results: $|D| = |C|$.

By hypothesis, C is a generalization of D, but the only constant-free generalization of D having the same number of literals of D is D itself. Thus, $C = D$ and this is a contradiction because C is inconsistent wrt N, whilst D is consistent wrt N by hypothesis.

□

Appendix C.

(properness)

ρ_{OI} is proper as a consequence of the definition of $mgdr_{OI}(C, N)$ and of Theorems 1 and 2. Indeed, $\rho_{OI}(C) \subseteq mgdr_{OI}(C, N)$.

(local finiteness)

The choice of l in ρ_{OI} is related to the construction of the sets $\mathbb{S}_c$, $\mathbb{P}$ and $\mathbb{S}$. Note that the number of one-to-one substitutions such that a clause C θ_{OI}-subsumes a clause D is finite and equal to $|vars(D)| \times (|vars(D)| - 1) \times \dots \times (|vars(D)| - |vars(C)| + 1)$.

It is worthwhile to note that $\mathbb{S}_c$ is an intersection of a finite number of residuals, by definition.

This number depends on the (finite) number of substitutions between the clause C to be refined and the example N which causes the problem of inconsistency. In turn, each residual is a finite difference-set of literals between two clauses. Thus, $\mathbb{S}_c$ is finite and computable. $\mathbb{P}$ is also an intersection of a finite number of difference-sets between two clauses. This number depends on the number of substitutions between C and the positive examples already processed. Finally, the set $\mathbb{S}$ is the union of a finite number of difference-sets between two clauses. This number depends on the number of substitutions between C and N. Since these sets are both finite and computable, ρ_{OI} is locally finite.

(completeness)

Let C, D be two clauses such that $D <_{OI} C$ ($C, D \in \mathbb{LP}$). In this case, there exist some substitutions σ_j, $j=1, 2,\dots, s$ s.t. $|\Delta_j(D, C)| = r$.

For a given $j \in \{1, 2,\dots, s\}$, let us consider the literals in $\Delta_j(D, C)$. Then, we may write D as follows: $D = C.\sigma_j \cup \{l_1, l_2,\dots, l_r\}$, where $l_k.\underline{\sigma}_j^{-1} \in \Delta_j(D, C)$, $k = 1, 2,\dots, r$.

We can build the following set of clauses:

$\{F_h\}_{h=0,1,\dots,r}$, where $F_h = C.\sigma_j \cup \{l_1, l_2,\dots, l_h\}$, for $h = 0, 1,\dots, r$.

Note that: $F_0 = C.\sigma_j$ and $F_r = D$.

In order to demonstrate the completeness property, it is to be proven that:

$\forall k = 0, 1,\dots, r\text{-}1$: $F_{k+1} \in \rho_{OI}(F_k)$.

For a given $k \in \{0, 1,\dots, r\text{-}1\}$, let us consider $F_{k+1} = C.\sigma_j \cup \{l_1, l_2,\dots, l_{k+1}\} = F_k \cup \{l_{k+1}\}$.

Let us suppose now, without loss of generality, that the database of the available positive examples is made up of the set $\{P_1, P_2,\dots, P_n\}$ and that N_k is the negative example which calls for the downward refinement operator ρ_{OI}

If l_{k+1} is a positive literal in the body of F_{k+1}, then, by looking back at the definition of ρ_{OI}, we note that we are able to build the sets $\mathbb{P}$ and $\mathbb{S}$ such that $l_{k+1} \in \mathbb{P} - \mathbb{S}$, and then $F_{k+1} = F_k \cup \{l_{k+1}\} \in \rho_{OI}(F_k)$. In fact, the set $\mathbb{P}$ depends on the positive examples $\{P_1, P_2,\dots, P_n\}$, and the set $\mathbb{S}$ depends on N_k, which can be chosen in such a way that $l_{k+1} \notin \Delta_m(N_k, F_k)$ for each substitution γ_m between N_k and F_k causing F_k to be inconsistent wrt N_k.

If l_{k+1} is a negative literal in the body of F_{k+1}, then by definition of ρ_{OI}, we are able to build the set $\mathbb{S}_c$, which in turn depends on N_k and F_k, in such a way that $l_{k+1} \in neg(\Delta_m(N_k, F_k))$ for each substitution γ_m above.

□

Integrity Constraints Evolution in Deductive Databases

Danilo Montesi*

Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ, UK
d.montesi@doc.ic.ac.uk

Franco Turini

Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56125 Pisa, Italy
turini@di.unipi.it

Abstract

Integrity constraints evolution refers to constraints that can change over time. Integrity constraints form a unit called integrity constraints theory that is expressed through a Horn logic language. Unfortunately, integrity constraints evolution mirrors the problem of updates in logic programming. In this paper we introduce a new approach to constraints evolution for deductive databases extending the traditional Datalog language to accommodate integrity constraints. We consider permanent and temporary constraints. The temporary constraints are defined in a query, that is they hold only for that query, while the permanent constraints are defined in rules database, that is they hold forever. Thus different queries can have different temporary constraints and this allow their evolution. We propose a language for integrity constraints evolution in deductive databases that is an extension of Datalog and we provide its operational semantics. Our approach turns to fit in already developed methods for efficient constraints checking.

*The work of this author has been partially supported by the EU *Human Capital and Mobility* Compulog-Group grant N. ERBCHBGCT930365.

1 Introduction

Deductive databases use a uniform and declarative language to express several database concepts such as data, views, queries and integrity constraints [6]. The traditional approach consider data and views together as the database and the integrity constraints as a separate component to express the meta data or 'knowledge' that a database must satisfy. However, despite the language uniformity, there are several limitations, among which the difficulty to express in logical term the evolution of data and integrity constraints. The problem of evolution/updates of the data even if not satisfactory solved (specially for views) have been approached [1]. Many approaches have been proposed to extend logic languages to accommodate updates [2]. Some of them are based on logics involving time, others just try the smooth integration of updates and declarative query languages [4, 9].

The problem of integrity constraints evolution, instead, remains largely unexplored. The fact that integrity constraints are expressed through Horn logic language lead to consider Horn clauses evolution/update, that as we have said, has not been properly solved. Thus the logical nature of integrity constraints does not allow to consider evolving integrity constraints in a logical framework. Evolving constraints are not fixed once and forever. Instead they can change over time to reflect different changes in data semantics expressed through integrity constraints [14]. Such evolving constraints are useful in the prototyping phase of the development of a database system or for many applications where the domain knowledge changes very often. The fact that integrity constraints evolution have been little attention is also related to the traditional database view where integrity constraints are fixed and the most important research issue is the efficient detection of constraint violations [5, 13, 15].

In this paper we investigate the problem of handling evolving integrity constraints [11]. We do not aim at solving the problem of updating constraints. Indeed, we do not allow the integrity constraints updates, but to define them in a *dynamic context.* Traditionally a deductive database has three components: a query Q, a database DB and an integrity constraints theory IC. The basic idea of our approach is to define integrity constraints in two different components, splitting the integrity constraints IC into *permanent* and *temporary* ones (IC_p and IC_t respectively). The former are accommodated into the rules database giving rise to a new database containing data and meta data, called DB^{IC_p}. The latter are combined into the query giving rise to a new query Q^{IC_p}. The possibility of allowing changes to (part of the) constraints relies on the fact that the query is a dynamic component of the system. The rules database, instead, is assumed to be a time invariant component. Thus the constraints defined in the query are temporary, that is they hold only for such a query. A new query can embody different (temporary) constraints. The resulting integrity constraint theory (IC_r) is built starting from the temporary ones (defined in the query) and enriching them by means of the involved constraints of the rules database. The involved constraints are those which are defined in the rules database and which are used for the evaluation of the query. Permanent and/or temporary constraints may be empty. Putting all the constraints in a query allows one to change all of them in the next query, while storing them in the rule database does not allow any change, but new constraints can be added in a new query. Any trade off between these extreme situations is possible. Flexibility, that is more temporary constraints and less permanent constraints is paid in terms of computational effort for constraint checking.

The contribution of this paper is to extend the traditional Datalog language to allocate temporary and permanent integrity constraints. The resulting $Datalog^{IC}$ language allows to express constraints evolution extending the Datalog language. Then we define the operational semantics for $Datalog^{IC}$ and show that our approach can be used together with already developed methods for efficient constraints checking. Finally we discuss the trade off between flexibility and efficiency of the language altogether with further extensions considering updates to the data.

In the following we assume some knowledge of Datalog language and database concepts [16]. The remainder of this paper is organized as follow. Section 2 provides several examples. Section 3 introduces the $Datalog^{IC}$ language. Section 4 introduces the semantics and Section 5 discusses several issues related to efficient constraints checking and the use of knowledge in query optimization. Finally, section 6 contains some concluding remarks.

2 An overview of our approach

The classic view considers the database as a component storing data (intensionally or extensionally) and consider the integrity constraints as another component (usually separate) storing knowledge. Therefore, even if data, views and integrity constraints can be expressed by means of the same logic language, they are regarded as different components as shown in the next example.

Example 2.1 *Consider the intensional database (IDB) describing high and medium sale departments as those that have the sale volume in a fixed range. Consider the permanent constraint (IC_p)"no departments can be on the floor f2 and f6 and the sale of a department must be greater then 0" and the temporary one (IC_t) "no departments can be on the floor f3 and f4 and the sale of a department must be greater then 0".* [1]

[1]We denote in the examples in bold the formula that denotes knowledge.

$IDB =$ `Hsaledept(Depno,Mgrno,Floorno,Item,Vol)` $\leftarrow$
`Dept(Depno,Mgrno,Floorno),`
`Sale(Depno,Item,Vol),Vol` $>$ `80000.`

`Msaledept(Depno,Mgrno,Floorno,Item,Vol)` $\leftarrow$
`Dept(Depno,Mgrno,Floorno),`
`Sale(Depno,Item,Vol),`
`Vol` $>$ `20000,Vol` $<$ `50000.`

$$IC_p = \leftarrow \mathbf{Dept(X,Y,f2),Dept(X,Y,f6).}$$
$$\leftarrow \mathbf{Sale(X,Z,V),V > 0.}$$

$$IC_t = \leftarrow \mathbf{Dept(X,Y,f3),Dept(X,Y,f4).}$$
$$\leftarrow \mathbf{Sale(X,Z,V),V > 0.}$$

The labels of the constraints reflect the permanent/temporary status that the programmer has in mind. In order to have a formal model for the evolution of such knowledge we should be able to have a logical model for the evolution of Horn clauses used to express the temporary constraints. As we have said this, is an open problem [17]. We approach this problem associating the permanent constraints with the intensional database and the temporary constraints with a query. The former constraints are permanent, since that part of the database is not assumed to change. The later constraints are temporary since queries can change over time. Thus different queries, in general, will have different integrity constraints associated to them.

Example 2.2 *Consider the transformed intensional database where the knowledge is associated to rules bodies inside {...}.*

$IDB^{IC_p} =$ `Hsaledept(Depno,Mgrno,Floorno,Item,Vol)` $\leftarrow$
$\{\leftarrow \mathbf{Dept(X,Y,f2),Dept(X,Y,f6),}$
$\leftarrow \mathbf{Sale(X,Z,V),V > 0}\}$
`Dept(Depno,Mgrno,Floorno),`
`Sale(Depno,Item,Vol),Vol` $>$ `80000.`

`Msaledept(Depno,Mgrno,Floorno,Item,Vol)` $\leftarrow$
$\{\leftarrow \mathbf{Dept(X,Y,f2),Dept(X,Y,f6),}$
$\leftarrow \mathbf{Sale(X,Z,V),V > 0}\}$
`Dept(Depno,Mgrno,Floorno),`
`Sale(Depno,Item,Vol),`
`Vol` $>$ `20000,Vol` $<$ `50000.`

The informal reading of a rule of the form $H \leftarrow \{IC_1, \ldots, IC_n\}B_1, \ldots, B_m$ is that H is true if $B_1, \ldots, B_m$ is true and the database satisfies $IC_1, \ldots, IC_n$. We recall that a database DB satisfies a set of integrity constraints IC if $DB \models IC_i$ for each IC_i in IC, otherwise DB violates IC.

Example 2.3 *Temporary constraints can be defined in a query as follows*

$$Q^{IC_t} = ? \quad \{\leftarrow \mathrm{Dept(X,Y,f3)}, \mathrm{Dept(X,Y,f4)}, \leftarrow \mathrm{Sale(X,Z,V)}, \mathrm{V} > 0\}$$
$$\texttt{Hsaledept(Depno,Mgrno,Floorno,shoe,90000)}.$$

The evaluation of Q^{IC_t} in IDB^{IC_p} in addition to the binding for the query part produces the resulting integrity constraint theory $IC_r = IC_p \cup IC_t$. Obviously, if one or more constraints are defined both in the query and in the database there is no way to 'retract' its effect.

Example 2.4 *Consider the transformed intensional database IDB^{IC_p} of the Example 2.2 and the query $Q^{\emptyset}$, that is the query of the Example 2.3 with empty temporary constraints. This lead to the resulting integrity constraints theory $IC_r = IC_p$ that is different from the previous one due to the empty temporary constraints.*

Let us now consider the evaluation process of a query. It has two phases. The first phase is the query-answering where the bindings (for the variable of the query part) are computed and integrity constraints are collected and their consistency is checked. The second phase is the constraints-checking.

Example 2.5 *Consider the query*

$$Q^{IC'_t} = \; ? \quad \{\leftarrow \mathrm{Dept(X,Y,f3)}, \mathrm{Dept(X,Y,f4)}\}$$
$$\texttt{Hsaledept(Depno,Mgrno,Floorno,shoe,90000)},$$

the database IDB^{IC_p} of Example 2.2 and the state $EDB =$ `Dept(9,2,f2)`, `Dept(6,1,f4)`, `Sale(9,shoe,90000)`, `Sale(8,book,120000)`. *The query-answering phase computes the bindings* `Depno/9`, `Mgrno/2`, `Floornr/f2` *and the resulting integrity constraints theory, that is*

$$\begin{aligned} IC_r = \; & \leftarrow \mathrm{Dept(X,Y,f2)}, \mathrm{Dept(X,Y,f6)}. \\ & \leftarrow \mathrm{Sale(X,Z,V)}, \mathrm{V} > 0. \\ & \leftarrow \mathrm{Dept(X,Y,f3)}, \mathrm{Dept(X,Y,f4)}. \end{aligned}$$

The constraints-checking phase verifies if IC_r satisfies EDB.

In the above example IC_r is not violated in EDB, but is violated in $EDB' = EDB \cup$ `Dept(6,1,f3)`. Instead the query $Q^{\emptyset}$ computes as resulting constraint theory IC_p which is not violated in EDB'.

At this point we should clarify some points about integrity constraints. First the permanent constraints inherit all the classic results such as the *efficient constraints*

checking [5]. This due to the fact that the required hypothesis hold: permanent constraints are assumed to be consistent and they are satisfied in the current database state. The second hypothesis allows to check integrity constraints only on the difference between the current state and the next state. The second point is related to the role of integrity constraints that are seen as global invariant property of the database. Our approach instead, gives to permanent constraints the status of local property, that is local to the rule where they are defined. Thus the database programmer should take the responsibility to locate the integrity constraints in the rules. all rules... This can be seen as a step back with respect to the current situation, but it is not. Indeed, this process to 'attach' integrity constraints to rules can be done automatically and it is not visible to the programmer once that the temporary/permanent status of constraints are specified leaving to already developed techniques the work to decide where to locate integrity constraints [3, 8]. This process often called semantic query optimization allow to exploit the permanent constraints for efficient query evaluation. We do not discuss this process since it is beyond the scope of this work and is an ongoing research. The important point is that for efficient constraints checking we can take advantage of already available methods while achieving knowledge evolution. This is due to the fact that our approach grows on top of those methods. even if they do not allow to accommodate temporary constraints. Thus our approach can take advantage of those techniques for efficient computation in a more dynamic framework that allow us to model knowledge evolution.

Finally, our approach leads to another possible optimization. Since, we already know that the permanent constraints does not violate the current database we have only to check the temporary part of IC_r. It might also happen that the temporary constraints are (partially) included in the permanent one.

3 *Datalog*^{IC}

We introduce a rule-based language corresponding to the language informally introduced by means of examples in Section 2. The extensional database (EDB) is a set of extensional ground atoms, while the intensional one is a set of rules of the form

$$H \leftarrow \{IC_1, \ldots, IC_n\}B_1, \ldots, B_m$$

where $B_1, \ldots, B_m$ (as in Datalog) is the query part which cannot be empty, H is an intensional atom and $IC_1, \ldots, IC_n$ is the integrity constraint part. The two parts do not share variables. We consider only constraints that are denials where all the variables are universally quantified and they are built from extensional literals. A query is a rule with no head of the form ? $\{IC_1, \ldots, IC_n\}B_1, \ldots, B_m$. A *Datalog*^{IC} database DB consists of the extensional database and of the intensional database. Note that the above language is an instance of constraint logic programming [7]

4 Semantics

The operational semantics of $Datalog^{IC}$ is given below in Natural Deduction style. For any database DB and a query Q, we denote by $DB \vdash_\theta Q$ the fact that there is a top-down derivation of Q in DB with answer θ. We reserve the symbol ϵ to denote the empty (identity) answer. The top-down derivation relation is defined by rules of the form

$$\frac{Assumptions}{Conclusion}[Conditions]$$

asserting the *Conclusion* whenever the *Assumptions* and *Conditions* hold. A relation $DB \vdash_\theta Q$ holds if it is the first of a finite sequence of similar relations such that each is a consequence of some of the relations following it in the sequence according to the rules discriminated below

$$\frac{}{DB \vdash_\epsilon \Box}$$

$$\frac{DB, \vdash_\theta \{IC_1\}B_1 \qquad DB \vdash_\sigma \{IC_2\}B_2}{DB \vdash_{\theta\sigma} \{IC_1, IC_2\}B_1, B_2}$$

$$\frac{DB, \vdash_\theta \{IC_1\}B_1}{DB \vdash_{\theta\sigma} \{IC_1, IC_2\}B_2}[H \leftarrow \{IC_2\}B_2 \in DB, \sigma = mgu(B_1, H) \text{ and } EDB \models IC_1, IC_2]$$

The first rule states that an empty goal is derivable in every database with empty answer. The second rule states that to derive a non-empty conjunct you have to derive each conjunct in turn. The third states that to derive a query you have to reduce it to the body of a rule.

The above three rules can be used to implement an interpret of the proposed language to evaluate a query. In the third rule the constraints violation is checked through any of the already developed systems verifying if the condition $EDB \models IC_1, IC_2$ holds. This may lead to inefficiency and therefore an incremental constraints checking is required as discussed in the next section.

5 Discussion

As we have said, efficient constraints checking can be realized for the permanent constraints due to assumption that the integrity constraints are satisfied in the current database state [5]. Unfortunately, none these methods can be applied to the temporary constraints for which we have to check that they are satisfied in the current database state and that they are consistent among them and with the

permanent constraints. This expected (negative) result underlines the trade off between temporary and permanent constraints. Temporary constraints checking is expensive and hard to optimize due to their dynamic nature.

However, it is possible to improve the efficiency of the resulting integrity constraints checking (incrementally) only the temporary one with respect to the current database state. It might also happen that the temporary constraints are (partially) included in the permanent one leading to some further optimization.

There is another interesting point about integrity constraints (and updates). Traditionally, integrity constraints are checked to verify if the update leads to a state which satisfies them. If this is not the case the updates are rejected. Such updates are very often seen as external agents which change the database state. It would be interesting instead to express in addition to integrity constraints also updates in rule bodies. $\mathcal{LDL}$ for instance allow updates in rule body [12]. In this case we can link together the integrity constraints and the potential source of constraints violation, namely the updates. This should allow to be updates driven in the checking process and thus more efficient while keeping the temporary constraints. This is the case of integrity constraints in active databases [10].

6 Conclusion

We have seen a new approach to dynamic constraints definition. The main advantage of this approach is to allow permanent and temporary constraints in a logical and clean framework that allow to apply already known constraints checking methods.

References

[1] S. Abiteboul. Updates, a New Frontier. In M. Gyssens, J.Paredaens, and D. Van Gucht, editors, *Proc. Second Int'l Conf. on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1988.

[2] A. J. Bonner and M. Kifbiber. An overview of transaction logic. *Theoretical Computer Science*, To appear, 1994.

[3] U. S. Chakravathy, J. Grant, and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Transaction on Database Systems*, 15(2):162–207, June 1990.

[4] W. Chen. Declarative Specification and Evaluation of Database Updates. In C. Delobel et al., editor, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, pages 147–166, 1991.

[5] H. Decker. Integrity Enforcement in Deductive Databases. In *Proc. Int'l Conf. on Expert Database Systems*, pages 271–285, 1986.

[6] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and database: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[7] J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19:503–581, 1994.

[8] A. Y. Levy and Y. Sagiv. Semantic Query Optimization in Datalog Programs. In *Proc. of the ACM Symposium on Principles of Database Systems*. ACM, New York, USA, 1995. To appear.

[9] S. Manchanda and D. S. Warren. A Logic-based Language for Database Updates. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, 1987.

[10] D. Montesi and R. Torlone. A Rewriting Technique for the analysis and the Optimization of Active Databases. In G. Gottlob and M. Y. Vardi, editors, *Proc. Fifth Int'l Conf. on Database Theory*, volume 893 of *Lecture Notes in Computer Science*, pages 238–251. Springer-Verlag, 1995.

[11] D. Montesi and F. Turini. Knowledge Evolution in Deductive Databases. In *Int. Symposium on Knowledge Retrieval, Use and Storage for Efficiency*, 1995. To appear.

[12] S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[13] J-M. Nicolas. Logic for Improving Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982. Springer-Verlag.

[14] X. Qian and G. Wiederhold. Knowledge-based Integrity Constraint Validation. In Y. Kamabayashi, editor, *Proc. Twelfth Int'l Conf. on Very Large Data Bases*, pages 3–22, 1986.

[15] F. Sadri and R. Kowalski. Integrity Checking in Deductive Databases. In P. Hammersley, editor, *Proc. Thirteenth Int'l Conf. on Very Large Data Bases*, pages 61–69, 1987.

[16] J. D. Ullman. *Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[17] D. S. Warren. Database Updates in pure Prolog. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 244–253. Institute for New Generation Computer Technology, 1984.

Declarative reconstruction of updates in logic databases: a compilative approach *

M. Carboni[1], V. Foddai[2], F. Giannotti[1], and D. Pedreschi[2]

[1] CNUCE Institute of CNR
Via S. Maria 36, 56125 Pisa, Italy
e-mail: F.Giannotti@cnuce.cnr.it

[2] Dipartimento di Informatica, Univ. Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: pedre@di.unipi.it

Abstract

Deductive database languages exhibit an evident dichotomy in the way they support queries and transactions. Query answering is based on declarative semantics and fixpoint based (bottom-up) evaluation. Transactions are based on procedural semantics and top-down evaluation, as for instance in the logic database language $\mathcal{LDL}$ [NT88]. This paper presents a technique to compile updates on standard logic programs to be evaluated with the usual bottom up evaluation mechanism. The compilation is based on the concept of XY-stratification [AOZ93] which is a syntactic property of non-monotonic recursive programs. XY-stratified programs use stage arguments to integrate control on state transition within the deduction process.

Keywords. Deductive Databases, Logic Databases Languages, Logic Database Updates, XY-stratification.

1 Introduction

Logic database languages use a declarative style both to represent knowledge and operations on database relations. To coherently model the application domain, a deductive database should also express its dynamically changing aspects. As a matter of fact, updates are a primary concern of any database language.

On the one hand, deductive databases naturally support powerful and declarative query languages, and queries can be efficiently executed using a bottom-up, fixpoint-based procedure. Also, sophisticated optimizations such as magic sets are available to capture the advantages of top-down execution, when needed. On the other hand, deductive databases traditionally suffer from limitations in describing the dynamic and transactional aspects of database systems.

*This work has been carried out within the EC-US Cooperative Activity Project ECUS-033 – Deus Ex Machina.