Semantic Query Optimization through Abduction and Constraint Handling

Gerhard Wetzel
Logic Based Systems Lab, Department of Computer Science
Brooklyn College, NY 11210, USA,
gw@sci.brooklyn.cuny.edu
http://sci.brooklyn.cuny.edu/~gw

Francesca Toni
Department of Computing, Imperial College
London SW7 2BZ, UK,
ft@doc.ic.ac.uk
http://laotzu.doc.ic.ac.uk/UserPages/staff/ft/ft.html

Abstract. The use of integrity constraints to perform Semantic Query Optimization (SQO) in deductive databases can be formalized in a way similar to the use of integrity constraints in Abductive Logic Programming (ALP) and the use of Constraint Handling Rules in Constraint Logic Programming (CLP). Based on this observation and on the similar role played by, respectively, extensional, abducible and constraint predicates in SQO, ALP and CLP, we present a unified framework from which (variants of) SQO, ALP and CLP can be obtained as special instances. The framework relies on a proof procedure which combines backward reasoning with logic programming clauses and forward reasoning with integrity constraints.

1 Introduction

Semantic Query Optimization (SQO) in deductive databases uses implicit knowledge coded in Integrity Constraints (ICs) to transform queries into new queries that are easier to evaluate and ideally contain only atoms of extensional predicates. SQO sometimes allows for unsatisfiable queries to be rejected without accessing the database at all.

ICs in Abductive Logic Programming (ALP) and Constraint Handling Rules (CHRs) in Constraint Logic Programming (CLP) are used in a similar way as ICs in SQO. ALP aims at transforming given goals (observations) into new goals containing only atoms of abducible predicates (hypotheses). ICs are used to reject incompatible hypotheses. CLP aims at transforming given goals into new goals containing only atoms of constraint predicates. CHRs are used to program a user-defined constraint solver which simplifies constraints and checks them for satisfiability. CHRs can be viewed declaratively as ICs.

Based on the similar use of ICs and CHRs as well as of extensional, abducible and constraint predicates, we present a unified framework from which (variants of) SQO, ALP and CLP can be obtained as special instances. The framework relies on a proof procedure, originally defined for ALP, combining backward reasoning with logic programming clauses and forward reasoning with ICs.

The paper is organized as follows. Section 2 gives some background and examples on SQO, ALP and CLP and shows how the three areas are related to each other. Section 3 defines a unified framework in which the three areas are embedded. Section 4 defines the basic proof procedure for the unified framework and some possible extensions. Section 5 shows how SQO can be obtained as a special instance of the framework and its proof procedure. Section 6 gives some application examples.

This paper is based on [31, 32].

2 SQO, ALP, CLP

A logic program consists of clauses of the form

$H \leftarrow B_1 \wedge \ldots \wedge B_n$

where H is an atom, the B_i are literals and all variables are implicitly universally quantified from the outside. Any such clause is said to "define" the predicate of the atom H.

A deductive database is a logic program separated into an *extensional database* (EDB), consisting of unit clauses ($n = 0$), and an *intensional database* (IDB), consisting of any clauses. The predicates solely defined in the EDB are referred to as extensional, whereas the predicates defined in the IDB are referred to as intensional. The EDB is typically very large and therefore looking up information in the EDB during query answering may be computationally explosive. Thus, it is desirable to preprocess and optimize a given query to reduce the necessity of accessing the EDB during the query answering process. The area of research concerned with such query transformation is called **Semantic Query Optimization** (SQO, [3, 4]). SQO uses implicit knowledge coded in *integrity constraints* (ICs) to transform queries into new queries that are easier to answer and ideally contain only atoms of extensional predicates. In some cases, SQO may allow the rejection of unsatisfiable queries without accessing the EDB at all.

Example 1. (SQO)
Query: `employee(X)` $\wedge$ `position(X,manager)` $\wedge$ `bonus(X,B)` $\wedge$ `B=0`
IC: `position(X,manager)` $\wedge$ `bonus(X,B)` $\rightarrow$ `B≠0`
Let the predicates `employee`, `position` and `salary` be extensional. SQO uses the IC to show that the query has no answers without accessing the EDB. If PROLOG were used to process the query, it would have to look at every `employee` fact.

The use of ICs to optimize queries adds to their standard use of checking and maintaining consistency of dynamically changing databases. (For further work on the role and semantics of integrity constraints in deductive databases in general and in SQO in particular see [11] and the references therein.) ICs are used in a similar way in **Abductive Logic Programming** (ALP, [18, 19]), to check and maintain consistency of dynamically generated explanations for observations. In ALP, predicates are either ordinary, processed by backward reasoning with clauses in a given logic program, or *abducible*, undefined in the given program. Explanations consist of atoms of abducible predicates only.

More formally, an abductive logic program consists of a logic program $\mathcal{T}$, a set of ICs and a set of candidate abducible predicates. Δ is an *explanation* for an observation G if

(1) Δ consists of atoms of abducible predicates only
(2) $\mathcal{T} \cup \Delta$ "entails" G
(3) $\mathcal{T} \cup \Delta$ "satisfies" the ICs

The notions of entailment and IC satisfaction can be defined in numerous ways. At the very least IC satisfaction means that $T \cup \Delta$ is consistent with the ICs (*consistency view*). We will adopt a flexible "propertyhood view" in our unified framework (see section 3).

Example 2. (ALP)
Observation: $\texttt{bird} \wedge \texttt{flies}$
IC: $\texttt{penguin} \wedge \texttt{flies} \rightarrow \texttt{false}$.
Let `penguin, albatross` and `flies` be abducible, and `bird` be defined by the logic program
$\texttt{bird} \leftarrow \texttt{albatross}$
$\texttt{bird} \leftarrow \texttt{penguin}$.
Then, $\{\texttt{albatross}, \texttt{flies}\}$ is a consistent explanation for the observation, whereas $\{\texttt{penguin}, \texttt{flies}\}$ is not.

In SQO, the optimization of a query by means of ICs may introduce new atoms of extensional predicates into the query. Analogously, in ALP, ICs may introduce abducibles into explanations. Thus the use of ICs in SQO and ALP is similar, with extensional, respectively intensional, predicates in SQO taking the role of abducible, respectively ordinary, predicates in ALP. This similarity was first noticed by Kakas [16, 17].

Constraint Logic Programming (CLP, [13, 14]) also considers two kinds of predicates, ordinary predicates, processed by backward reasoning with clauses in a given logic program, and *constraint* predicates, simplified and checked for satisfiability by a built-in constraint solver. A given goal is solved when it is reduced to a set of constraints that cannot be simplified further. Recently, ICs in the form of Constraint Handling Rules (CHRs, [7]) have been used to explicitly user-define the constraint solver.

Example 3. (CLP)
Goal: $\texttt{X > 1} \wedge \texttt{1 > Y} \wedge \texttt{Y > X}$
ICs: $\texttt{X > Y} \wedge \texttt{Y > Z} \rightarrow \texttt{X > Z}$
$\texttt{X > N} \wedge \texttt{N > X} \rightarrow \texttt{false}$.
A CLP system employing a built-in constraint solver working either over a finite integer domain (such as `cc(FD)` [30]) or over the domain of real numbers

(CLP($\mathcal{R}$), [13]) does not need the explicit ICs as the constraint solver recognizes that the constraints in the goal are unsatisfiable over their respective domains.

If no built-in constraint solver is present, the ICs can be used as CHRs to determine the unsatisfiability of the goal: the first IC (transitivity) adds `X > Y` $\wedge$ `1 > X` to the goal and then the second IC becomes applicable and generates `false`.

Note that PROLOG, which cannot make use of the ICs, would either refuse to process the uninstantiated arithmetic expressions or fail to terminate, if it tried to unfold an explicitly given definition of "`>`".

Based on the observation that extensional, abducible and constraint predicates are treated in similar ways in SQO, ALP and CLP and that the use of ICs plays a similar role in the three frameworks, in the next section we define a unified framework in which (variants of) SQO, ALP and CLP can be obtained as special instances. The framework's proof procedure (see section 4) can be used for the generation of optimized queries, explanations, sets of constraints for given queries, observations, goals.

3 Unified Framework and Semantics

Knowledge in the unified framework is represented by definitions and ICs. *(Iff) definitions* are of the form

$H \leftrightarrow D_1 \vee \ldots \vee D_n$

where H is an atom (different from `true` and `false`) and each D_i is a conjunction of literals. Variables in the head H are implicitly universally quantified from the outside, whereas variables in any of the disjuncts D_i but not appearing in the head H are implicitly existentially quantified in the disjunct. A definition is said to *define* the predicate of the atom H.

Definitions are used in place of conventional clauses in order to reduce the amount of non-deterministic search. There may still be several definitions for the same predicate, but we require that the head atoms H of different definitions must not unify. One (iff) definition corresponds to n (if) clauses in a logic program which define the same predicate and whose head atoms unify.

ICs are implications of the form

$A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \vee \ldots \vee B_m$

where the A_i and B_j are atoms. Variables in ICs are implicitly universally quantified from the outside.

Note that an IC could be equivalently written as a *denial* implication

$A_1 \wedge \ldots \wedge A_n \wedge \neg B_1 \wedge \ldots \wedge \neg B_m \rightarrow false$

which, in a deductive database, could have the declarative meaning that if the A_i are entailed by the database and none of the B_j is, then the IC is violated. Procedurally, ICs can be given a forward reasoning interpretation: if the A_i hold, then "propagate" $B_1 \vee \ldots \vee B_m$. Interpreted in this way, ICs are similar to *if-then*, *condition-action* or *production rules* in Artificial Intelligence. Large-scale commercial AI applications like R1/XCON [26] make use of such rules.

While ICs can be transformed into normal logic programs [21] and thus potentially into definitions, operationally useful information may be lost in the transformation and the resulting program may not only be further from the specification (and thus harder to understand and to prove correct), but also less efficient to run.

Definitions and ICs may involve atoms of three kinds of predicates:

- *user-defined* predicates, defined by a set $\mathcal{T}_u$ of domain-specific, user-given definitions,
- *built-in* predicates, including equality as well as arithmetic inequality and arithmetic operators, defined by an implicit set $\mathcal{T}_b$ of definitions,
- *external* predicates, defined by an inaccessible (unknown) set $\mathcal{T}_e$ of definitions.

The union $\mathcal{T}_u \cup \mathcal{T}_b \cup \mathcal{T}_e$ is denoted by $\mathcal{T}$. The union $\mathcal{T}_u \cup \mathcal{T}_b$ of the accessible parts of the theory $\mathcal{T}$ is denoted by $\mathcal{T}'$. The set of all ICs is denoted by $\mathcal{IC}$.

User-defined predicates correspond to ordinary predicates in LP, ALP and CLP as well as intensional predicates in SQO. Built-in predicates correspond to equality in standard LP and ALP and to constraint predicates in CLP. Note that equality is the only constraint predicate in LP from a CLP point of view. The issue of built-in (constraint) predicates in deductive databases is often left open. External predicates correspond to abducibles in ALP and extensional predicates in SQO. The ALP instance of the framework can be thought of as generating those parts of an unknown theory $\mathcal{T}_e$ relevant to the given observation. In the SQO instance the theory $\mathcal{T}_e$ is the known EDB which remains inaccessible until the final optimized query has been formulated.

As $\mathcal{T}_e$ is inaccessible, information about external predicates is only provided by ICs. ICs are also used to provide information about accessible, user-defined or built-in predicates whenever it would be expensive to access $\mathcal{T}$ directly. Therefore, ICs can be thought of as approximations of the definitions in $\mathcal{T}$.

In order to guarantee that ICs be sound approximations of $\mathcal{T}$, we require that the ICs are *properties* of $\mathcal{T}$, i.e. sentences true in all *intended models* of $\mathcal{T}$, $\mathcal{T} \models_{int} \mathcal{IC}$ in short. The user chooses which models are intended.[1] Possible choices include the perfect, stable, and well-founded models of the logic program obtained by rewriting the iff definitions of $\mathcal{T}$ into if clauses. Often there is a more or less canonical choice, e.g. the unique minimal Herbrand model for Horn programs and the unique perfect model for locally stratified logic programs (the latter choice and restriction is made in [22]).

By allowing the user to choose the notion of intended model, we allow a higher degree of flexibility. Note that if a theory $\mathcal{T}$ and a set of ICs are given, then the user is limited in the choice of intended models, whereas if a theory $\mathcal{T}$ is given together with its intended models, then the user is limited in the choice of

[1] Subject to a few restrictions given in [31], e.g. $\mathcal{T}$ must have at least one intended model (which may, however, be three-valued) to rule out the case where everything is a trivial property of a theory without intended models.

ICs — in both cases the user has to ensure that $\mathcal{T} \models_{int} \mathcal{IC}$ holds. For example, if $\mathcal{T}$ contains only

`p` $\leftrightarrow$ `p`

and the intended model of $\mathcal{T}$ is the minimal Herbrand model of `p` $\leftarrow$ `p`, then

`p` $\rightarrow$ `false`

is a property of $\mathcal{T}$ and thus may be used as an IC.

The *initial goal* is a conjunction of literals with all variables being free. The computational task, whose formalization into a proof procedure is given in section 4, is to reduce the given initial goal to a disjunction of answers, which is equivalent to the initial goal in the sense that the goal and the disjunction are satisfied, in the intended models of $\mathcal{T}$, by the same assignments to the free variables. Such answers are obtained by using definitions to "unfold" atomic goals to equivalent disjunctions of goals which can then be "split" using distributivity. Surrogate subgoals are introduced by "propagation" (resolution) with ICs. As in Operations Research, surrogate subgoals are logically redundant, but may be easier to solve.

An atom A is called *suspended*, if there is no accessible definition $H \leftrightarrow D_1 \vee \ldots \vee D_n$ such that A is an instance of H (i.e. $A = H\sigma$ for some substitution σ).[2] Otherwise A is called *reducible*.

Thus atoms of external predicates, whose definitions are inaccessible, are always suspended, whereas atoms of user-defined and built-in predicates are suspended if they are insufficiently instantiated to be an instance of the head of a definition. The way in which definitions are written can be used to control suspension of atoms and the amount of non-determinism in the search for answers.

Example 4. Let the user-defined predicate `p` be defined by the following two definitions:

`p(X,a)` $\leftrightarrow$ `q(X)`

`p(X,b)` $\leftrightarrow$ `r(X)`

Then `p(X,Y)` and `p(a,Y)` are suspended, but `p(X,a)` and `p(a,b)` are reducible. If the definition

`p(X,Y)` $\leftrightarrow$ `( Y=a` $\wedge$ `q(X) )` $\vee$ `( Y=b` $\wedge$ `r(X) )`

were used instead, then atoms of `p` would never be suspended.

$\mathcal{T}_b$ is assumed to be such that atoms like `X<Y`, `X>0` and `plus(2,X,Y)` are suspended, but `1<0`, `1>0` and `plus(2,2,X)` are not suspended (and can be reduced to `false`, `true` and `X=4`, respectively).

ICs are used to process suspended goals. "Propagation" with ICs may add to the goals reducible atoms which can then be "unfolded".

The framework's declarative semantics introduces three different notions of answer so that the proof procedure can be sound with respect to at least the weakest notion (for any permissible choice of intended model). The strongest notion is that of *ideal answer*: A is an ideal answer to the initial goal G_0 if

[2] Note that there is either exactly one or no such definition since heads of different definitions are not allowed to unify.

(A1) A is a conjunction of suspended atoms and implications (whose atoms are also suspended)
(A2) $\mathcal{T} \models_{int} \tilde{\forall}[G_0 \leftarrow A]$
(A3) $\mathcal{T} \not\models_{int} \neg\tilde{\exists}A$

Here $\tilde{\forall}$ and $\tilde{\exists}$ denote the universal and existential closures, respectively, quantifying over all free variables.

Condition (A1) means that goals have to be "unfolded" as far as possible, i.e. until all subgoals are suspended. Condition (A2) says that the initial goal G_0 must follow from the answer A in all intended models of $\mathcal{T}$, for the same assignments to the free variables in G_0. Condition (A3) requires A to be satisfiable in at least one intended model of $\mathcal{T}$. This implies that $\mathcal{IC} \cup \tilde{\exists}A$ is consistent since we assume that $\mathcal{T} \models_{int} \mathcal{IC}$ (and since $\mathcal{T}$ must have at least one intended model). This condition defines the *propertyhood view* of IC satisfaction. The propertyhood view allows a more flexible view of IC satisfaction than those approaches fixing the notion of intended models and requiring the integrity constraints to be true in either some or all models of the theory or database.

The given answer definition is idealistic since $\mathcal{T}$ contains the inaccessible part $\mathcal{T}_e$ which is not available (to the proof procedure) for checking the satisfiability condition (A3). In general all suspended atoms can only be checked for consistency using the ICs. Or, in SQO terminology, it is not normally possible to optimize away and thus reject a query that has no answers but satisfies the ICs without consulting the EDB.

Thus, in addition to ideal answers, two weaker answer notions are defined. A is a *good answer* to G_0 if (A1) and (A2) hold and

(A3$_g$) $\mathcal{T}' \cup \mathcal{IC} \cup CET \cup \tilde{\exists}A$ is consistent (CET is Clark's Equality Theory [5])

The suspension of user-defined and built-in predicates in $\mathcal{T}'$ may still prevent the proof procedure from identifying violations of condition (A3$_g$). Therefore an even weaker third answer notion is introduced: A is a *weak answer* to G_0 if (A1) and (A2) hold and

(A3$_w$) $\mathcal{IC} \cup CET \cup \tilde{\exists}A$ is consistent

Every answer computed by the proof procedure presented in the next section satisfies conditions (A1), (A2) and at least condition (A3$_w$).

Note the similarity of the above conditions to the ALP answer definition given in section 2. In ALP the only suspended atoms are the abducible atoms, so condition (1) of the ALP answer definition is just a special case of condition (A1). Condition (2) and (A2) are similarly variants of each other. Conditions (A3), (A3$_g$) and (A3$_w$) are all different ways of formalizing the satisfiability condition (3).

In section 5 we will compare the notion of answer definition in the unified framework and the notion of optimized query in SQO.

4 Proof Procedure

This section describes a rather abstract (high-level) proof procedure for the unified framework which can be regarded as a generalization and simplification of two abductive proof procedures, the Iff Proof Procedure [8] and SLDNFA [6]. By mainly ignoring efficiency issues, the proposed proof procedure may serve as a scheme for a family of proof procedures which can be obtained as instances by restricting and extending the abstract procedure in different ways. Thus more specialized procedures for ALP, CLP and SQO may be obtained as required.

4.1 Basic operations

A *derivation* of a goal G_n from an initial goal G_0 is a finite sequence of goals $G_0, G_1, \ldots, G_n$ where G_1 is obtained from G_0 by conjoining all ICs to G_0, and, for $0 < i < n$, G_{i+1} is obtained from G_i by applying one of the following operations:

1. *Unfolding* replaces a reducible atom A, which is either a conjunct of the goal or is a condition of an implication in the goal, by its definition. I.e. if there is a substitution σ such that $A = H\sigma$ and $H \leftrightarrow D_1 \vee \ldots \vee D_m$ is an accessible definition in $\mathcal{T}$, then replace A by $(D_1 \vee \ldots \vee D_m)\sigma$.
2. *Propagation* is a form of resolution. If a subgoal contains the conjuncts $p(s_1, \ldots, s_n) \wedge \mathit{Cond} \rightarrow \mathit{Conc}$ and $\mathit{Susp} \rightarrow p(t_1, \ldots, t_n)$, then the "resolvent" $\mathit{Cond} \wedge \mathit{Susp} \wedge (s_1 = t_1 \wedge \ldots \wedge s_n = t_n) \rightarrow \mathit{Conc}$ can be added to the subgoal. Susp is (usually) a conjunction of suspended atoms; the equalities in the resulting implication may allow the instantiation of atoms in Susp such that they can be further unfolded.
3. *Logical equivalence transformations* include splitting (replacing $(A \vee B) \wedge C$ by $(A \wedge C) \vee (B \wedge C)$), normalization of implications (replace $\neg A \wedge B \rightarrow C$ by $B \rightarrow A \vee C$ and replace $(A \vee B) \rightarrow C$ by $(A \rightarrow C) \wedge (B \rightarrow C)$), and logical simplifications such as the replacement of $A \wedge \texttt{false}$ by $\texttt{false}$ and of $\texttt{true} \rightarrow A$ by A.[3]
4. *Equality rewriting* implements the unification algorithm and performs substitutions to deal with equality atoms introduced into the goal. The rewrite rules used are adapted from [8] and are based on [25]. Conceptually, equality rewriting may also be regarded as propagation with the CET axioms of Clark's Equality Theory, followed by deletion of the no longer needed, rewritten terms.

If there is a derivation of $G_n = D \vee \mathit{Rest}$ from an initial goal G_0 and the operations of the proof procedure have been exhaustively applied to the disjunct D of the goal G_n, then a *computed answer* to G_0 is extracted from D after omitting all implications in D which are ICs or could have been derived from ICs without taking the initial goal into account (see [31] for details and justifications) and transforming all remaining implications in D into denials. The latter ensures that no further steps of propagation (with atoms not present in D) could generate new conclusions.

[3] Except that the implication $\texttt{true} \rightarrow A$ is not rewritten to A, if A contains a universally quantified variable. Related proof procedures impose range-restrictedness on definitions and ICs [8] or more complicated safety requirements on the operations.[6]

Example 5. Given the goal disjunct D

`p(a)` $\wedge$ `q(a)` $\wedge$ `[ p(X)` $\rightarrow$ `q(X) ]`

the computed answer is

`p(a)` $\wedge$ `q(a)` $\wedge$ `[ p(X)` $\wedge$ `X`$\neq$`a` $\rightarrow$ `false ]`

The denial expresses that `p(X)` must not hold for any `X` not equal to `a`. If further instances of `p` besides `p(a)` were true, then further instances of `q` would have to be true as well (but they might be defined as `false` in $\mathcal{T}$).

The operations of the proof procedure are all sound in the sense that they derive goals which are equivalent in $\mathcal{T}$, i.e.

$\mathcal{T} \models_{int} \tilde{\forall}[G_i \leftrightarrow G_{i+1}]$

Every computed answer to G_0 is at least a weak answer to G_0. Under certain further assumptions (see [31]) soundness results with respect to the stronger answer notions are also obtainable. Refutation soundness (good and ideal answers are never rejected) can be proven. Completeness results (for every ideal answer there is a corresponding computed answer) are only obtainable for special cases and for extensions of the proof procedure.

4.2 Extensions

Constraint Propagation with Disjunctions (CPD) [31, 22] generalizes propagation so that it can be used within and across disjunct boundaries in goals. CPD can be viewed as a natural implementation, and in some sense generalization, of several methods of handling disjunctive constraints in CLP,[4] such as CHIP's forward-checking and look-ahead techniques, Generalized Propagation and Constructive Disjunction (see [31] for more details and references and [15] for an overview of handling disjunctive constraints in CLP).

More technical improvements of the proof procedure are needed with respect to the efficiency of propagation and its termination. Since propagation is defined as resolution, the problem of restricting propagation can be approached by looking at resolution restrictions proposed in the theorem-proving literature. Possible restrictions include P_1-resolution, which corresponds to the requirement that the expression *Susp* in the definition of propagation is empty, and hyper-resolution, which corresponds more closely to the forward reasoning interpretation of ICs (only apply an IC if all of its condition atoms can be shown to hold).

Another important extension of the proof procedure is the *deletion* operation. It interprets ICs as identifying logically redundant and operationally useless subgoals which can be removed to simplify the goal. Ideally, deletion should be applied automatically by the proof procedure, e.g. the atom `X>0` should be deleted as soon as the atom `X>1` is added to the goal, if the ICs contain transitivity

`X>Y` $\wedge$ `Y>Z` $\rightarrow$ `X>Z`

which is a property of the built-in definition of the ">" predicate. However, the same rule can be applied by propagation if the conclusion `X>Z` is operationally useful. Criteria have to be worked out when ICs are to be used for propagation and when for deletion. See [33] for more on the use of ICs as deletion rules.

[4] Note that, in our framework, any disjunction can function as a disjunctive constraint.

4.3 Implementation

PROCALOG [34], short for *PROgramming with Constraints and Abducibles in LOGic*, is a realization of the unified framework and proof procedure as a programming language. It includes several of the extensions proposed in section 4.2.

Two prototype implementations have been used in [31] to provide some computational results for solutions to sample CLP applications, e.g. the n-queens problem and a warehouse location problem. A more efficient implementation (based on one of the two prototypes) is currently being developed.

5 SQO as an instance of the unified framework

We refer to the logic-based approach to SQO in deductive databases proposed in [3, 4]. The aim of SQO is to transform a given query, by using the intensional database (IDB) and a given set of ICs, but without accessing the extensional database (EDB), into an equivalent query which consists only of EDB (and possibly built-in) atoms and satisfies the ICs.

In the unified framework, the IDB corresponds to the user-defined part $\mathcal{T}_u$ of the given theory $\mathcal{T}$ and the EDB corresponds to the inaccessible theory $\mathcal{T}_e$. Whereas in conventional (deductive) databases the EDB is assumed to be an enumeration of ground facts, in the SQO instance of the unified framework the EDB (i.e. $\mathcal{T}_e$) is not explicitly restricted in this way. Whereas in conventional (deductive) databases the IDB is assumed to be a logic program, in the SQO instance of the unified framework the IDB ($\mathcal{T}_u$) is a set of (iff) definitions. We obtain an exact correspondence, if $\mathcal{T}_u$ is assumed to be in homogenized form, i.e. if $\mathcal{T}_u$ is the set of all definitions for intensional predicates in the completion of a logic program. The approach in [3, 4] restricts the IDB to a set of non-recursive Horn clauses without function symbols and assumes that the ICs are not recursive either. The unified framework does not impose any such restriction. These restrictions have also been partly lifted in further work on SQO such as [23] and [10]. In particular, the latter extends the original approach of [3, 4] to deal with negative literals in the IDB and in ICs as well as IDB clauses with disjunctive heads.

In the unified framework, built-in as well as extensional atoms can be part of the optimized queries. Some recent work, e.g. [29], attempts to apply constraint technology to databases in general and query optimization in particular, usually employing a conventional CLP approach with a built-in constraint solver rather than explicit rules for constraint handling. In our approach built-in (constraint) predicates are naturally accommodated via propagation with ICs and unfolding with $\mathcal{T}_b$.

5.1 SQO semantics

The aim of SQO, to transform a given query into an equivalent query which consists only of EDB (and possibly built-in) atoms and satisfies the ICs, is a special case of the computational task in the unified framework, to reduce a given initial goal to an equivalent disjunction of computed answers, each consisting only of suspended atoms and satisfying the ICs. In fact, if we assume that $\mathcal{T}_u$ is in homogenized form, atoms of user-defined predicates are never suspended and only EDB atoms (and possibly built-ins) are suspended and belong to computed answers.

In order to illustrate how SQO fits in the unified framework's semantics, we now give a declarative semantics for SQO. Other work on SQO usually defines SQO only in operational terms, although the following relationship between the initial and the optimized query is implied. Given an initial query Q_0, the query Q is an *optimized query* of Q_0 if

(S1) Q consists of atoms of EDB predicates only (and possibly built-ins, if any)
(S2) $IDB \cup \mathcal{IC}\ (\cup \mathcal{T}_b\ ) \models \tilde{\forall}[Q_0 \leftrightarrow Q]$
(S3) $IDB \cup \mathcal{IC}\ (\cup \mathcal{T}_b\ ) \cup \tilde{\exists} Q$ is consistent

Condition (S1) is the SQO instance of condition (A1) in the unified framework's semantics. Approaches to incorporate into databases built-in predicates to deal with (in-)equality and arithmetics usually suspend atoms of those predicates in a way similar to the unified framework, thus allowing built-ins to become part of the final optimized query. Note that if $\mathcal{T}_u$ is not in homogenized form and if thus atoms of user-defined predicates may be suspended and become part of an answer, then we may obtain what has been called *intensional answers* in the database literature (e.g. see [28]). Our concept of suspension may be viewed as a generalization of the idea of intensional answers.

Condition (S2) corresponds to condition (A2) in the unified framework's semantics, with two differences: $\mathcal{IC}$ in (S2) replaces $\mathcal{T}_e$ (EDB) in (A2) and and the stronger equivalence $Q_0 \leftrightarrow Q$ in (S2) replaces the implication $G_0 \leftarrow A$ in (A2). The first difference reflects the fact that the EDB plays no role during query optimization, if not through the ICs. If the ICs are properties of IDB $\cup$ EDB, as required by the unified framework, this difference is not substantial. The second difference reflects the fact that the optimized query Q is actually a disjunction of sub-queries $Q_1 \vee \ldots \vee Q_m$ with each sub-query Q_i corresponding more closely to an answer A in the unified framework. In fact, if there are only finitely many answers $A_1, \ldots, A_n$, then their disjunction is equivalent, in the intended models of $\mathcal{T}$, to the initial goal G_0:

$\mathcal{T} \models_{int} \tilde{\forall}[G_0 \leftrightarrow A_1 \vee \ldots \vee A_n]$

However, in the general case, there might be infinitely many answers so that only an implication can be used. Note though that the formulation

$\mathcal{T} \models_{int} \tilde{\forall}[G_0 \leftrightarrow A \vee Rest]$

where *Rest* is some formula, is equivalent to (A2) and possibly closer to (S2).

Condition (S3) corresponds to the good answer condition (A3$_g$). If user-defined predicates are never suspended and equality is the only built-in predicate, then the proof procedure for the unified framework will only compute good answers. The analogue of the ideal answer condition (A3$_i$), which can be expressed in the SQO case as

$IDB \cup EDB \models_{int} \tilde{\exists} Q$

is not suitable, however, because in SQO the optimized query Q may not have any answers in the EDB.

5.2 Proof procedures for SQO

[3, 4] propose a *two-phased compiled approach* consisting of a *compilation phase* and a *transformation phase*.

The compilation phase generates *semantically constrained axioms* which combine the information given in the IDB and the integrity constraints. For example, given the IDB

```
p(X) ← a(X)
p(X) ← b(X)
```

and the IC

```
a(1) → false
```

the compilation phase generates the semantically constrained axiom

```
p(X) ← a(X) ∧ X≠1
```

([3, 4] uses a slightly different syntax). This is achieved using *partial subsumption*, an operation which is based on resolution between the bodies of IDB clauses and integrity constraints. The additional condition `X≠1` is called a *residue* of the IC.

In the unified framework's proof procedure, everything happens at run-time, and there is no pre-compilation. However, the effects of partial subsumption are achieved by propagation (combined with unfolding). For example, if the given query is `p(X)`, the unified framework's proof procedure generates the goal

```
( a(X) ∧ X=1 → false ) ∨ b(X)
```

by first unfolding the goal using the (homogenized) definition

```
p(X) ↔ a(X) ∨ b(X)
```

and then applying propagation.

In practice the use of propagation instead of pre-compilation can have both advantages (no overhead as propagation can be restricted to atoms and implications actually "related" to the initial goal) and disadvantages (the same steps of propagation may have to be executed several times). More importantly, using propagation eliminates the restriction to non-recursive databases in an arguably simpler and more natural way than in the aforementioned extensions of SQO to recursive databases [23, 10].

In the second phase of SQO, the transformation phase, the query is modified by using the semantically constrained axioms obtained in the compilation phase as new clauses to unfold atoms in the given query and introducing the body of the axioms (after appropriate unification) into the new *semantically constrained query*. If more than one semantically constrained axiom unifies with a given query atom, then a disjunction of two semantically constrained subqueries is introduced.

In the unified framework, these query modifications correspond to unfolding (plus splitting, if needed) and equality rewriting. As [4] notice, query modification may lead to built-in expressions (equalities, inequalities, arithmetic operators etc.) becoming fully instantiated and thus evaluable. The unified framework's definitions of built-in predicates and suspension constitute a formalization and generalization of these concepts.

Evaluation of built-in atoms may lead to parts of the query becoming immediately unsatisfiable. Continuing the example given earlier in this section, if the initial query is `p(1)`, then query modification using the semantically constrained axiom `p(X)` $\leftarrow$ `a(X)` $\wedge$ `X`$\neq$`1` yields the new semantically constrained query

`( a(1)` $\wedge$ `1`$\neq$`1 )` $\vee$ `b(1)`.

The first disjunct contains an evaluable atom, $1 \neq 1$, which can be reduced to `false`. Thus the query can be reduced to the second disjunct and the final optimized query is `b(1)`.

In the unified framework, given the initial goal `p(1)`, unfolding yields the disjunction `a(1)` $\vee$ `b(1)`. Propagation with the integrity constraint can then be applied to introduce `false` into the first disjunct, which is afterwards eliminated by logical equivalence transformation. The computed answer is again `b(1)`.

The query transformation phase treats specially the residues that are added to IDB clauses when generating semantically constrained axioms in the compilation phase. Such special treatment is not required in the unified framework as propagation and logical equivalence transformation achieve the desired effects. There is one interesting special case in which a residue in the form of a "unit clause" is obtained. This corresponds to an implication `true` $\rightarrow A$ being derived by the unified framework's proof procedure. Then A could be added to the query as auxiliary information. However, if A already occurs as an atom in the query, then it could also be deleted from the query since the unit clause residue implies that the query atom will simply succeed. This relates to the issue of using integrity constraints sometimes for propagation and sometimes for deletion as mentioned in section 4.2.

6 Applications

The merits of the unified approach lie in its flexibility and its generality, reflected by the relatively few restrictions compared with existing versions of its instances (e.g. the restriction to non-recursive clauses in SQO).

As an example of the flexibility of the framework, consider the problem of configuring a computer system. The possible choices for the components (processor, monitor, memory, operating system, etc.) can be specified by definitions:

`processor(X)` $\leftrightarrow$ `( X = pentium )` $\vee$ `( X = sparc )` $\vee \ldots$

`operating_system(X)` $\leftrightarrow$ `( X = os2 )` $\vee$ `( X = unix )` $\vee \ldots$

Dependencies, whether system constraints or personal preferences, between the different choices can be represented by ICs, in the form of both positive requirements and denials, e.g.

`processor(sparc)` $\rightarrow$ `operating_system(unix)`

`processor(sparc)` $\wedge$ `operating_system(os2)` $\rightarrow$ `false`.

In SQO, `processor` and `operating_system` may be extensional and be regarded as inaccessible during query optimization. In the SQO instance of the framework, the query

`processor(sparc)` $\wedge$ `operating_system(os2)`
is rejected as inconsistent with the ICs without consulting the EDB.

In the CLP instance of the framework, the goal

`processor(X)` $\wedge$ `operating_system(os2)`
results in the constraint `X=pentium` being derived as an answer (but not `X=sparc`).

In the ALP instance of the framework, the definitions of `processor` and `operating_system` might be regarded as unknown and the predicates treated as abducible. Given the observation `processor(sparc)`, the answer

`processor(sparc)` $\wedge$ `operating_system(unix)`
is an abductive explanation which satisfies the ICs (whereas an answer containing `operating_system(os2)` does not).

Further examples of applications of the unified framework which are given in [31] include constraint satisfaction problems (such as n-queens and map coloring), Operations Research applications (such as job-shop scheduling and warehouse location problems) and an approach to configuration which bears some similarity to the CALOG framework are discussed in [31].

The use of ICs to improve the efficiency of logic (PROLOG) programs without having to abandon the original, natural specification is another interesting application of the framework. Again, see [31] for some examples.

7 Conclusion and Related Work

A unified framework for ALP, CLP and SQO has been presented. Variants of and procedures for ALP, CLP and SQO can be obtained as instances of the framework and its proof procedure. Further developments of the ALP and CLP aspects of this work can be found in [22]. The proof procedure presented in section 4 is a simplification and generalization of the abductive proof procedure presented in [8] and [9]. It is also closely related to SLDNFA, an abductive proof procedure defined in [6].

Previous proposals to relate ALP to deductive databases (and thus potentially to SQO) [16, 17] and to integrate ALP and CLP [20, 27] focused either on particular aspects of a combined framework or developed hybrid systems. Our unified framework can be instantiated to obtain (versions of) SQO, ALP and CLP as well as combinations of these frameworks. For example, the combination of SQO and the manipulation of built-in, constraint-like predicates is obtained as a by-product.

Recent activities in parallel with this work point to a convergence of related research. CHR$^\vee$ [1] introduces disjunctions into the bodies of CHRs in a similar way as disjunctions are permitted in the bodies of definitions in the unified framework. Moreover, CHR$^\vee$ is presented in this same volume as a “flexible query language” confirming the potential of CHRs for use in deductive database. Also in this volume, Bressan and Goh [2] apply abductive mechanisms to query optimization and implement them using CHRs. Although their formalism is different, the unified framework we have presented could be thought of as the general theory behind their work.

The framework and its implementations could lend themselves to many well-known problems and applications of Artificial Intelligence, especially problems with a large search space (where even a limited use of the CPD extensions can drastically reduce the search tree) and applications relying on knowledge representation in the form of condition-action (if-then) rules (where integrity constraints can often be used to model the rules in a logical way). The latter include expert systems for configuration problems or medical diagnosis.

Acknowledgements

We are grateful to Robert Kowalski for his collaboration in parts of this research. The first author is supported by ONR grant N00014-96-1-1057. The second author is supported by the UK EPSRC project "Logic-based multi-agent systems".

References

1. Abdennadher, S.; Schütz, H.: CHR$^{\vee}$: A Flexible Query Language. In this volume.
2. Bressan, S.; Goh, C. H.: Answering Queries in Context. In this volume.
3. Chakravarthy, U. S.; Grant, J.; Minker, J.: Foundations of Semantic Query Optimization for Deductive Databases. In: Minker, J. (ed.): *Foundations of Deductive Databases and Logic Programming*, pp. 243–273, Morgan Kaufmann 1988
4. Chakravarthy, U. S.; Grant, J.; Minker, J.: Logic-Based Approach to Semantic Query Optimization, *ACM Transactions on Database Systems* **15** (2), pp. 162–207, 1990
5. Clark, K. L.: Negation as failure. In: Gallaire, H.; Minker, J. (eds.): *Logic and Data Bases*, pp. 292–322, Plenum Press 1978
6. Denecker, M.; De Schreye, D.: SLDNFA: an abductive procedure for abductive logic programs, *Journal of Logic Programming* **34** (2), pp. 111–167, 1997
7. Frühwirth, T.: Constraint Handling Rules. In: Podelski, A. (ed.): *Constraint Programming: Basic and Trends*, pp. 90–107, LNCS 910, Springer Verlag 1995
8. Fung, T. H.: *Abduction by Deduction*. Ph.D. Thesis, Imperial College 1996
9. Fung, T. H.; Kowalski, R. A.: The Iff Proof Procedure for Abductive Logic Programs, *Journal of Logic Programming* **33** (2), pp. 151–165, 1997
10. Gaasterland, T.; Lobo, J.: Processing Negation and Disjunction in Logic Programs Through Integrity Constraints, *Journal of Intelligent Information Systems* **2**, pp. 225–243, 1993
11. Godfrey, P.; Grant, J.; Gryz, J.; Minker, J.: Integrity Constraints: Semantics and Applications. To appear in: Chomicki, J.; Saake, G.: *Logics for Databases and Information Systems*, Kluwer 1998
12. Janson, S.; Haridi, S.: Programming Paradigms of the Andorra kernel language, Saraswat, V.; Ueda, K. (eds.): *Proc. of the Int. Symp. on Logic Programming*, pp. 167–186, MIT Press 1991
13. Jaffar, J.; Lassez, J.-L.: Constraint Logic Programming, *Proc. of the 14^{th} ACM Symp. on the Principles of Programming Languages*, pp. 111–119, 1987
14. Jaffar, J.; Maher, M.: Constraint Logic Programming: A Survey, *Journal of Logic Programming* **19/20**, pp. 503–581, 1994

15. Jourdan, J.; Sola, T.: The Versatility of Handling Disjunctions as Constraints. In: Bruynooghe, M.; Penjam, J. (eds.): *Proc. of the* 5^{th} *Intern. Symp. on Programming Languages Implementation and Logic Programming*, pp. 60–74, Springer Verlag 1993
16. Kakas, A. C.: Deductive Databases as Theories of Belief, Technical Report, Imperial College, 1991
17. Kakas, A. C.: On the Evolution of Deductive Databases, Technical Report, Imperial College, 1991
18. Kakas, A. C.; Kowalski, R. A.; Toni, F.: Abductive Logic Programming, *Journal of Logic and Computation* **2** (6), pp. 719–770, 1992
19. Kakas, A. C.; Kowalski, R. A.; Toni, F.: The role of abduction in logic programming. To appear in: Gabbay, D. M. et al. (eds.): *Handbook of logic in Artificial Intelligence and Logic Programming*, vol. 5, pp. 235–324, Oxford University Press 1998
20. Kakas, A. C.; Michael, A.: Integrating Abductive and Constraint Logic Programming. In: Sterling, L. (ed.): *Proc. of the* 12^{th} *Int. Conf. on Logic Programming*, pp. 399–413, MIT Press 1995
21. Kowalski, R. A.; Sadri, F.: Logic Programs with Exceptions. In: Warren, D. H. D.; Szeredi, P. (eds.): *Proc. of the* 7^{th} *Int. Conf. on Logic Programming*, pp. 598–613, MIT Press 1990
22. Kowalski, R. A.; Toni, F.; Wetzel, G.: Executing Suspended Logic Programs, to appear in a special issue of *Fundamenta Informaticae* ed. by K. Apt.
23. Lakshmanan, L. V. S.; Missaoui, R.: Pushing Semantics into Recursion: A General Framework for Semantic Optimization of Recursive Queries. In: *Proc. of the Intern. Conf. on Data Engineering*, Taiwan, 1995
24. Manthey, R.; Bry, F.: SATCHMO: A Theorem Prover Implemented in PROLOG. In: Lusk, E.; Overbeek, R. (eds.): *Proc. of the* 9^{th} *Conf. on Automated Deduction*, pp. 415–434, LNCS 310, Springer-Verlag 1988
25. Martelli, A., Montanari, U.: An efficient unification algorithm, *ACM Trans. on Prog. Lang. and Systems* **4** (2), pp. 258–282, 1982
26. McDermott, J.: R1: A Rule-Based Configurer of Computer Systems, *Artificial Intelligence* **19** (1), pp. 39–88, 1982
27. Maïm, E.: Abduction and Constraint Logic Programming, In: Neumann, B. (ed.): *Proc. of the* 10^{th} *European Conf. on Artificial Intelligence*, 1992
28. Pirotte, A.; Roelants, D.; Zimányi, E.: Controled generation of intensional answers, *IEEE Trans. on Knowledge and Data Engineering*, **3** (2), pp. 221–236, 1991
29. Ross, K. A.; Srivastava, D.; Stuckey, P. J.; Sudarshan, S.: Foundations of Aggregation Constraints, *Theoretical Computer Science* **193** (1–2), pp. 149-179, 1998
30. Van Hentenryck, P.; Saraswat, V. A.; Deville, Y.: Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`. In: Podelski, A. (ed.): *Constraint Programming: Basic and Trends*, pp. 293–316, Springer Verlag 1995
31. Wetzel, G.: *Abductive and Constraint Logic Programming*, Ph.D. thesis, Imperial College 1997
32. Wetzel, G.: A Unifying Framework for Abductive and Constraint Logic Programming. In: Bry, F.; Freitag, B.; Seipel, D. (eds.): 12^{th} *Workshop on Logic Programming (WLP'97)*, pp. 58–68, LMU München 1997
33. Wetzel, G.: Using Integrity Constraints as Deletion Rules. In A. Bonner et al. (eds): *Proceedings of the DYNAMICS'97 post-conference (ILPS'97) workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases*
34. Wetzel, G.; Kowalski, R. A.; Toni, F.: PROCALOG — Programming with Constraints and Abducibles in Logic. In: Maher, M. (ed.): *Proc. of the 1996 Joint Int. Conf. and Symp. on Logic Programming*, p. 535