Combined Inference Databases (Version 1.0.0, 2004)

Marcelo A. T. Aragão Alvaro A. A. Fernandes

Department of Computer Science
Oxford Road, Manchester M13 9PL, UK
{m.aragao|a.fernandes}@cs.man.ac.uk

Abstract

Databases have withstood the test of time and are now indispensable components of the majority of applications. One of the challenges that technological developments throughout the last decade have posed database research is that of eliciting novel and potentially useful information from very many, very large, databases, i.e., knowledge that lies in the data but is, in general, irretrievable via classical query languages. At present, this challenge has been met almost exclusively from an algorithmic viewpoint. Proposals for calculi or algebras that integrate query answering and knowledge discovery have been, comparatively, limited in scope and ambition. In contrast, this paper aims to contribute a new class of logic-based databases that integrate querying data (seen as a deductive task) and discovering knowledge (seen as an inductive one). Most importantly, the paper proposes a database-centred view of the problem that succeeds in providing a unified semantics of both deductive (e.g., query answering) and inductive tasks (e.g., classification, clustering, association, etc.) as expressions in a database calculus that has an equivalent logical algebra. An evaluator for the calculus has been built, as well as a compiler into the algebra, for which an evaluator also exists.

1 Introduction

Mechanisms for knowledge discovery in databases have attracted great attention in recent years, mainly because of the widespread availability of very large amounts of data of great variety. The challenges arising have mostly been met from an algorithmic viewpoint with roots not only in database, but also in machine learning and statistical, research. While results have been impressive as far as algorithmic qualities are concerned, progress on integrating the resulting algorithms into query processors has been comparatively slow. It would be useful to integrate querying data and discovering knowledge, ideally to the extent that, from the viewpoint of end users and developers, the transition from performing one to performing the other would be seamless. Unfortunately, this is far from the case at present. The two kinds of tasks are supported by distinct tool sets: querying data is supported by database management systems (DBMSs), discovering knowledge is supported by data mining tools (DMTs). So loose is the typical coupling between DBMSs and DMTs that, often, significant engineering effort is required for interoperation to be possible.

One broad consequence of this state of affairs is that while data stocks continue to grow in quantity and scope, knowledge stocks are comparatively stagnant. One stark instance is the state of the art in bioinformatics. The Nucleic Acids Research journal publishes every year a special issue with short summaries of the most valuable databases for the biological community. The 2003 issue [Bax03] lists almost 130 data resources. This impressive collection represents a challenge to database research, insofar as [Bax03] expresses no more than hope that the collection will accelerate the pace of biological discovery. Thus, while data stocks in biology grow exponentially, even modest rates of growth in biological knowledge derivable from such stocks remain largely a hope for the future. Some of the issues and challenges in extending database technology with a view to supporting the growth of knowledge stocks can be illustrated as follows. The capability that is missing is that of retrieving information in *intensional* form, e.g., general definitions of gene biological function in terms of molecular sequence and structure, protein homology, gene expression, etc. Parallels between the current state of bioinformatics (e.g., see [BK03]) and other more simple examples can be drawn.

For the purposes of this paper, we observe that query answering (the paradigmatic database task) retrieves *information in extensional form*, i.e., as (potentially large) collections of individual, specific items of information. For example, the answer to the question "Who has participated in which project?" would enumerate (i.e., extensionally present) individual, specific people who have participated in individual, specific projects. In contrast, the goal of knowledge discovery is to retrieve *information in intensional form*, i.e., as a few, general, characteristic statements that are supported by the data. For example, the answer to the question "Who can participate in some project?" would summarize (i.e., intensionally present) in one statement (or, possibly, a few) the constraints and conditions for people (in general) to participate in projects (also in general).

The above remarks make it clear that query answering and knowledge discovery are complementary information management technologies and that it would be beneficial to integrate them for the benefit of users. In spite of this, and of the great interest in this integration task, reconciling query answering and knowledge discovery is nontrivial, since their roots lie somewhat apart (query answering in the intersection of logic and computer science, knowledge discovery in machine learning and the problem-solving-as-search paradigm in artificial intelligence). Thus, any attempt at integration must address this foundational issue before a unified syntax and semantics for the combined task can be proposed.

1.1 Motivating Example

The notion of a *virtual organization* (VO) [Mow97] describes and explains the opportunistic, on-demand formation of, often short-lived, alliances between independent businesses or individuals to pounce on opportunities that would be beyond the reach of any member acting in isolation. In particular, this notion encompasses such concepts as that of a task force, a community of practice, or an advisory body, whose members are independent experts, each in their specific domains. VOs are information businesses in the sense that information is often their primary input and primary product. VOs depend on information management tools whose degree of sophistication is still only barely met by the most advanced research prototypes. Thus, it is our contention in this paper that, by their very nature, VOs are a prime source of usage scenarios in which CID systems can be shown to make a significant difference in practice, as we describe and motivate below.

Assume that a governmental health agency sets up an advisory body that brings together several experts, say, medical doctors and scientists from different hospitals, universities and industrial labs. This advisory body is set up on-demand, for example in response to an epidemic, often as a quickly formed and short-lived [Mow97] alliance between independent experts, each with their specific expertise. Experts are appointed on the strength of their track record, which, in turn, comprises data about affiliation to institutions, publications, professional recognition and past participation in similar endeavours.

The less formal nature of the advisory body and its need to react quickly to events pose particular challenges for information management. The time and effort that can be spent solving a problem are critically scarce, the task force cannot rely on an infrastructure, of the kind that businesses usually can. Thus, for example, the resources may not be provisioned with which to deploy sophisticated information management tools and engineer their inter-operation if these tools are disparate, heterogeneous, and each independently supplied. Although every business learns to cope with this kind of problem, an advisory body simply cannot afford being dragged by and drawn into. An advisory body must focus on its mission-critical goals, deviations from those may mean failure.

Problems of this kind are not easily avoidable, because they may arise even from trivial circumstances. Consider how candidate replacements for an expert (that must decline membership to the advisory body) might be procured. The efficiency in shortlisting candidates is critical, because time is in the essence of responding to an epidemic, hence time for posting announcements and asking the community for suggestions becomes expensive. The efficacy is also critical, because the success of the response depends on an appropriate selection, hence the list of candidates should contain only a few of the best ones.

The solution to the problem requires more than simple querying of track records. There must exist a model of expertise that provides the background for inferring who is competent in what. Likewise, there must exist a model of similarity of expertise that can rank and compare identified experts. Although the knowledge embodied in these two models is tacitly held in the academic communities. Unless there are efficient and efficacious means of eliciting tacit knowledge that solution remains infeasible.

Knowledge discovery in databases aims precisely to generate explicit knowledge from data. The question then arises as to whether the tacit knowledge alluded to might be discoverable in the data held about track records. Since the model of expertise can be discovered using classification techniques and the similarity model can be discovered using clustering techniques, it should be possible to elicit the

required knowledge. Then, it should be possible to retrieve, by querying, from the data on track records a set of answers for the query "Which candidate X can replace expert Y, given the currently-held models of expertise and of expertise similarity?" The solution is graphically depicted in Figure 1.

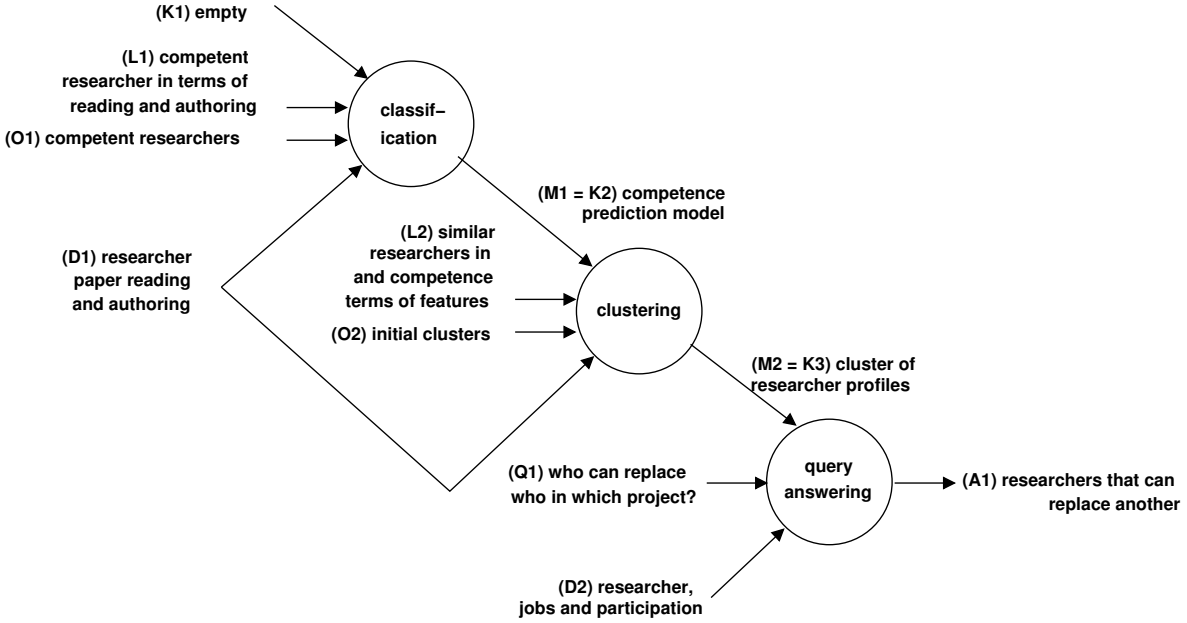


Figure 1: Procuring Candidate Replacements for an Expert

A solution that relies on combinations of techniques is more desirable the least effort it takes to deploy all referred techniques together. The notion of a ideal solution rises therefore from integrating query answering and knowledge discovery steps so as to meet the efficiency and efficacy constraints explained above.

Issues and Challenges Arising Usage scenarios in bioinformatics [BK03], as well as the simple example above, point to issues and challenges for database research that arise across the entire spectrum of database applications. They include:

1. Data retrieval capabilities alone do not suffice. There must be *integration* between query processing engines and knowledge discovery tools.

2. Bolting knowledge discovery tools onto query processing engines (e.g., using connectivity tools such as JDBC) does not suffice because closure is hard to attain. *Closure* and *compositionality* are necessary for computed models to be available for use in query processing through composition mechanisms, without which significant growth in knowledge stocks is unlikely to be attained.

3. *Compositionality* alone does not suffice because if different engines need to interoperate through mediators, the availability of induced models is neither immediate nor seamless, i.e., fluency is hard to attain. A combined engine is necessary so that query answering and knowledge discovery are captured in a single database language with compositionality mechanisms, thereby approaching the ideal of an unnoticeable transition between query answering and knowledge discovery tasks.

4. Seamless composition at *task expression level* does not suffice. The calculus must have an equivalent algebra that, in turn, admits of a physical counterpart for which optimization strategies are possible, otherwise there will not be a corresponding seamlessness at *evaluation level* to that at expression level, i.e., scalability is hard to attain.

Overview of the Proposed Solution This paper aims to contribute a new class of logic-based databases, referred to as CIDs (for *Combined Inference Databases*[1]) that integrate a deductive approach to query answering and an inductive approach to knowledge discovery. The viewpoint adopted

[1] Or for *Combined Induction and Deduction.*

is database-centred in the sense that combining the two kinds of tasks depends crucially on the ability to capture the semantics of both as expressions in a database calculus. Furthermore, since the calculus has an equivalent database algebra, the way is open to pursue efficient and scalable physical algebras for evaluating declarative user-level expressions that fluently and seamlessly combine both query answering and knowledge discovery tasks.

The broad contours of the proposed solution can be described as follows:

1. The motivating example above is characteristic of the **targetted functionality**, viz., the ability to derive from databases not only information in *extensional form*, but also information in *intensional form*.

2. The **approach** is logic-based since Datalog languages [AHV95] have been widely studied as representation languages that capture both extensional and intensional forms in both query answering [CGT90] and knowledge discovery [DL01]. In the former, extensional forms include facts and query results, and intensional forms include derivation rules, integrity constraints, views and queries. In the latter, extensional forms include examples and instances used in training and validation, and intensional forms include background knowledge, language biases and the discovered models.

3. The **logic** used is *p-Datalog* [LSV01] which extends the proof and model theories of classical Datalog by allowing certainty annotations that are asserted of (and derived for) clauses. This is essential for closure insofar as induction is not truth-preserving, therefore inductive consequences have to be annotated with some justification measure. p-Datalog is a logic for reasoning with uncertainty, and uncertainty is unavoidable in knowledge discovery from databases.

4. The **database** is viewed as a set of p-Datalog clauses, and this fixes the **definition language**. The **retrieval language** has *inference rules as primitives*. The functionality of a particular CID is determined by the inference rules it implements. For example, a task such as decision rule induction is captured in CIDs as an inference rule, call it ρ_{DR}[2]. Then, if ρ_{DR} is added to a CID Δ with a set of inference rules $I \not\ni \rho_{DR}$ yielding $I' = I \cup \{\rho_{DR}\}$ then, in addition to whatever retrieval possibilities were available, one can retrieve decision rules from Δ using I'. Over these primitives, the classical composition mechanisms for database calculi generate a language that can retrieve information in both extensional and intensional form as desired.

5. The **calculus** is a monoid comprehension calculus [FM00]. This choice is founded on its ability to handle collections uniformly, making it suitable for capturing bulk operations on collections of clauses. Each inference rule (i.e., each retrieval primitive in a CID) is expressed as a monoid comprehension over CIDs, i.e., p-Datalog clause sets. Establishing the structural similarity between an inference rule and a monoid comprehension is a straightforward, but crucial, step in the technical development of the contributions of the paper.

6. The monoid calculus has an equivalent **logical algebra**, thereby allowing the retrieval language to be mapped, after normalization to a canonical form, into a procedural execution plan [FM00].

7. This opens the way for an efficient **physical algebra** to be explored (although this last challenge is not addressed in this paper).

The rest of the paper is structured as follows. Section 2 introduces technical background used to define CIDS. Section 3 reviews the technical development of CIDS. Section 4 describes CID inference rules that express classification and clustering techniques. Section 5 describes the blueprints for a CID engine implementation based on the development of a concrete prototype and describes initial strategies to optimize combinations of query answering and knowledge discovery. Section 6 contrasts our work with other attempts at integrating query and discovery. Finally, Section 7 lists the main contributions and indicates future work that is most pressing.

2 Technical Background

The wider background that the paper takes for granted consists of logic-based databases as Datalog clause sets [CGT90], data mining and knowledge discovery [HMS01] and query processing [GMUW00].

[2]Section 4 describes this and a few other such inference rules.

The specific approaches to the above that the contributions build upon are, respectively, Datalog with certainty annotations [LSV01], inductive logic programming [MdR94] and monoid-based query processing [FM00].

2.1 Datalog with Certainty Annotations

CID systems use a Datalog language, called *parametric Datalog* (or p-Datalog, for short) [LSV01][3]. p-Datalog extends Datalog to contexts where absolute validity is no longer a requirement of the logical system.

A *p-Datalog clause* is a Datalog clause annotated in the neck with a validity assessment C and with a triple of validity combinators $\langle f^{\wedge}, f^{\leftarrow}, f^{\vee} \rangle$ (called the *conjunctive*, *propagator* and *disjunctive* combinators, respectively). The validity assessment can be interpreted (e.g., in [LSV01]) as the degree of certainty associated with the clause, but in CID systems there is no (and there is no need to have any) such precise commitment. The validity combinators determine, respectively, how the validity of the literals in the body are to be combined, how the validity of the body is assigned to the clause as a whole, and how the different validities associated with the different clauses that share the same predicate name are to be combined into a single validity assessment. Thus, if $\mathsf{p(X,Y)} \leftarrow \mathsf{q(X,Z)}$, $\mathsf{p(Z,Y)}$ is a Datalog clause, then one could annotate it with a validity value and validity combinators to yield, e.g., the following p-Datalog clause: $\mathsf{p(X,Y)} \leftarrow\!\!\!\shortmid \top \shortmid\!\!\!- \mathsf{q(X,Z)}, \mathsf{p(Z,Y)}$ $\langle min, min, max \rangle$. If we let $\top$ denote the largest possible validity assessment and min and max denote the functions that return, respectively, the smallest and the largest value in a set, then if we so annotate every clause, then p-Datalog reduces to Datalog.

Formally, a *validity assessment* is any value drawn from a given lattice $\langle \mathbb{P}, \leq \rangle$ in which $\mathbb{P}$ is a set of values and $\leq$ is a partial order over $\mathbb{P}$. Since validity assessments are drawn from a lattice, there exist a top $\top$ and a bottom $\bot$ element, denoting the maximum and the minimum validity assessments in $\mathbb{P}$, respectively. The *validity combinators* $\langle f^{\wedge}, f^{\leftarrow}, f^{\vee} \rangle$ are functions that satisfy certain properties (see [LSV01] for details). They are used to compute the validity value that annotates an inferred p-Datalog clause from the validity assessments that annotate the p-Datalog clauses participating in the inference (again, see [LSV01] for details).

A *p-Datalog program* is a set of p-Datalog clauses. p-Datalog has a model-theoretical, a fixpoint and a proof-theoretical semantics based on a decidable semi-naive procedure for p-Datalog theories [LS01, LSV01]. For simplicity, in the rest of this paper, validity assessments are drawn from the lattice $\langle [0, 1], \leq \rangle$, and, for every clause, the associated validity combinators are $\langle min, min, max \rangle$, which produce conservative propagations of validity assessments (see [LSV01] for details). This allows validity combinators to be omitted thereby removing some clutter from the notation. Prolog conventions are adopted otherwise.

Example

2.2 Inductive Logic Programming

Just as (deductive) logic programming provides the deductive inference rules used in CID systems, *inductive logic programming* (ILP) [MdR94] and data mining techniques provide the inductive ones.

As a machine learning technique, ILP has been successfully applied to tasks that rather than being data-intensive, require more expressive representation languages and more expressive models than propositional ones. A generic statement of the ILP problem is as follows: given some *background knowledge* and a collection of *examples* partitioned into positive and negative ones, find a set of *hypotheses* within some *language bias* that cover the positive examples, but not the negative examples, with respect to the background knowledge. Background knowledge, examples, hypotheses and language biases are clause sets. ILP has been used to tackle many data mining tasks in a relational setting, including classification and clustering.

This has given rise to the notion of relational data mining [DL01]. Relational data mining systematically recasts propositional data mining techniques into clausal-based representations. The main motivation is to overcome a limitation inherent in most mainstream data mining techniques, viz., the single-relation assumption [Wro01] that requires data to be mapped onto a denormalized training relation resulting from a multi-join of the original relations on the attributes of interest to the mining task. Relational data mining is under no such constraint and is, therefore, a more natural approach in database settings.

[3] We adapt, for increased clarity in this context, the original terminology of [LSV01].

2.3 Monoid-Based Query Processing

The Monoid Approach to Query Processing In abstract algebra, a *monoid* is a triple $\langle \mathbb{T}_\oplus, \oplus, \mathcal{Z}_\oplus \rangle$ consisting of a set $\mathbb{T}_\oplus$ together with a binary associative operation $\oplus : \mathbb{T}_\oplus \times \mathbb{T}_\oplus \rightarrow \mathbb{T}_\oplus$, called the *merge* function for the monoid, and an identity element $\mathcal{Z}_\oplus$ for $\oplus$, called the *zero* element of the monoid. Monoids may also be commutative or idempotent. Monoids on scalars are referred to as *primitive monoids*. Examples are shown at left in Table 1, where *int* is the set of natural numbers, $[0, 1]$ is the closed interval of real numbers between 0 and 1 and *boolean* is the set of boolean constants, the respective merge functions are the usual arithmetic and logic operators. A primitive monoid constructs

Primitive Monoids

type $\mathbb{T}_\oplus$	merge $\oplus$	zero $\mathcal{Z}_\oplus$
int	$+$	0
int	$\times$	1
$[0, 1]$	max	0
$[0, 1]$	min	1
$boolean$	$\wedge$	$true$
$boolean$	$\vee$	$false$

Collection Monoids

type $\mathbb{T}_\oplus$	merge $\oplus$	zero $\mathcal{Z}_\oplus$	unity $\mathcal{U}_\oplus$
$set(\alpha)$	$\cup$	$\{\}$	$\lambda x.\{x\}$
$bag(\alpha)$	$\uplus$	$\{\!\{\}\!\}$	$\lambda x.\{\!\{x\}\!\}$
$list(\alpha)$	$+\!\!+$	$[]$	$\lambda x.[x]$

Table 1: Examples

instances of a primitive type. Thus, to construct the *int* 7, one could write $+(0, +(1, +(1, +(2, 3)))) = 7$, or other expressions using instances of *int*, + and possibly 0. Monoids on collections are referred to as *collection monoids*. Examples of collection monoids are shown at right in Table 1. The merge functions $\cup$, $\uplus$ (denoting union without duplicate removal) and $+\!\!+$ (denoting list append) capture well-known the parametric collection types *set*, *bags* and *linear lists*. The zero elements $\{\}$, $\{\!\{\}\!\}$ and $[]$ denote the the empty set, empty bag and the empty list, respectively. Collection monoids require an additional function $\mathcal{U}_\oplus$, referred to as unit function, so that it is possible to construct all possible instances of the collection type. For example, $\mathcal{U}_\oplus = \lambda x.\{x\}$ is the function that given any x returns the singleton $\{x\}$. Thus, to construct the set of integers $\{2, 7, 5\}$ one could write $\mathcal{U}_\oplus(2) \oplus (\mathcal{U}_\oplus(7) \oplus \mathcal{U}_\oplus(5))$ which gives, for $\oplus = \cup$ and $\mathcal{U}_\oplus = \lambda x.\{x\}$, the set $\{2\} \cup (\{7\} \cup \{5\}) = \{2, 7, 5\}$. Likewise, $\lambda x.\{\!\{x\}\!\}$ and $\lambda x.[x]$ return a new bag and a new list, respectively, from any given x. Note that validity combinators should be associative and commutative [LSV01], therefore they give rise to monoids.

A *monoid comprehension* over a monoid $\oplus$ is an expression of the form $\oplus\{e \parallel w_1, w_2, \ldots, w_n\}$, $n > 0$. The merge function $\oplus$ is called the *accumulator* of the comprehension, while the term e and the terms $w_1, w_2, \ldots, w_n$ are called the *head* and the *body* of the comprehension. Each term w_i is called a *qualifier*, and is either a *generator* of the form $v \leftarrow D$, where v is a *range variable* and D is a *generator domain* (i.e., an expression that denotes a collection or a comprehension expression); a *filter* t, where t is a predicate over a range variable in scope; or a *binding* of the form $v \equiv \alpha$, where v is a new range variable and α is an expression that defines possible values for v (i.e., binding is a notational convenience with textual replacement semantics). Monoid comprehensions have a simple procedural interpretation, in which generators give rise to nested loops and multiple filters give rise to a conjunction of predicates. For example, a monoid comprehension of the form $\oplus\{e \parallel v \leftarrow D, p(v)\}$ denotes a result `R` computed as below:

```
R := Z⊕                    //where, e.g., Z⊕ is ∅ if ⊕ is ∪
for each v in D :
  if p(v) then
     R := R ⊕ U⊕(e)  //where, e.g., U⊕(e) is {e} if ⊕ is ∪
return R
```

A monoid comprehension characterizes a set of instances of the type over which it is defined. By applying the reductions given in [FM00] that set is formally derivable. In this sense, the reductions characterize a formal calculus on bulk polymorphic types. Monoid calculus is fully declarative and uniformly expresses collection scans, variable bindings, filtering, nesting and unnesting of collections, and aggregation. Since CID systems are p-Datalog clause sets, the corresponding monoid arises trivially as does the calculus in which retrieval tasks are expressed (as explained in detail in Section 3).

A primitive monoid that is especially relevant here is the *functional composition monoid* whose merge function is functional composition (denoted by $\circ$ and defined by the equation $f \circ g(a) = f(g(a))$) and zero

element is the identity function (denoted by $\lambda x.x$, and defined by the equation $\lambda x.x(a) = a$). The functional composition monoid captures bounded iteration over collections, i.e., given $X = [a_1, a_2, \ldots, a_m]$, the expression $\circ\{\lambda x.f(v,x) \,[\![\, v \leftarrow X\}$ is expanded to $(\lambda x.f(a_1,x)) \circ (\lambda x.f(a_2,x)) \circ \ldots \circ (\lambda x.f(a_n,x))$. Consequently, the expression $\circ\{\lambda x.f(v,x) \,[\![\, v \leftarrow X\}(e)$ computes $f(a_1, f(a_2, f(\ldots, f(a_m, e))))$. For examples, if $X = [7, 6, \ldots, 1]$, $e = 0$ and $f = \lambda x.y + x$, then the expression above accumulates the numbers in X up to 28. Here, the functional composition monoid provides mainly a shorthand bounded applications of an inference rule to itself.

Comprehension expressions can be composed by nesting: a comprehension expression can occur anywhere a variable of the same monoid type can, viz., in the head, in the domain of a generator, or in a filter. Comprehension expressions can be normalized into a canonical form using a terminating, confluent set of rewriting rules [FM00]. Normalization captures and generalizes many optimization techniques, such as pushing selections before joins, and merging selections. Comprehension expressions in canonical form can be translated into an equivalent logical algebra [FM00]. The translation algorithm does further optimization steps (such as flattening nested queries) that the normalization process lets through.

Thus, the monoid comprehension calculus provides CID systems with a common, formal semantics for the application of deductive and inductive inference rules. The availability of a common language (i.e., p-Datalog), combinable inference rules originating from deductive and inductive logic programming, and a common evaluation strategy (via the monoid comprehension calculus), opens the way for the formal characterization of CID systems in Section 3.

3 Combined Inference Database Systems

This section defines CID systems formally. CID systems were introduced informally in Section 1 as an approach towards comprehensive support to information provision. The technical details omitted then are now presented and exemplified. The syntax and declarative semantics of CID systems are a principled attempt to reconcile the foundations of query answering and knowledge discovery, and open the way for their formal study The operational semantics of CID systems is a database-centred attempt to reconcile the evaluation strategies of query answering and knowledge discovery, and opens the way for the development of CID systems, such as described in Section 5. The aim of this section is to provide a formal specification of the syntax and semantics of CID systems.

3.1 CID Systems as Inferential Systems

The logical formalism underlying CID systems is p-Datalog, described in Section 2.1. However, the implicit, fixed, deductive inferential mechanism of p-Datalog is generalised into a parameterised set of inference rules over p-clauses. These *CID inference rules* are either deductive (in which case they derive p-facts, i.e., information in extensional form, as consequences), or inductive (in which case they derive p-rules, i.e., information in intensional form, as consequences). In either case, it is part of the definition of a CID inference rule that it specifies precisely which value and combinators it annotates its consequences with, given the annotations of the premisses involved in the inference. Thus, the choice of interpretation for the parameters of p-Datalog is originally that of uncertain reasoning [LSV01], but in this dissertation, this choice is adapted to encompass inductive inference rules as well. What this adaptation provides is a principled treatment of the validity requirement, since absolute validity is too stringent for knowledge discovery, as explained next.

A classification technique, for example, can be expressed as a CID inference rule, say ρ, as explained in Section 4.2.1. Applications of ρ, induce a set of p-rules that denotes a classification model. The validity of each component of the induced classification model is assessed as part of the application of ρ and explicitly assigned as the annotation of that component. The result is a fully-fledged p-rule. Thus, in CID systems, the intended interpretations of the annotations of a p-clause is that of a *validity assessment* and of *validity combinators*, respectively. With this interpretation, an induced classification rule, e.g., $\mathsf{is\_a(E,s)} \hookleftarrow 0.8 \vdash I_1, I_2, \ldots, I_n \langle max, min, min \rangle$ intuitively denotes that any instantiation of variable E is predicted to be of class s, with validity 0.8, if the conditions $I_1, I_2, \ldots, I_n$ hold for that instantiation of E. Then, elementary production can be applied to verify, deductively, whether a concrete instance e belongs to class s, as illustrated in Section 2.1, taking into account the fact that the validity of the classification rule was deemed to be 0.8. Thus, validity assessments and validity combinators are used to reconcile, in a single logical formalism, the derivation of deductive and inductive consequences in a

principled manner.

A set of CID inference rules defines a new inferential system with deductive and inductive capabilities, in which purposeful compositions of query answering and knowledge discovery steps can be defined. The formalisation of CID inference rules is nevertheless a challenge, insofar as there is not a universal consensus as to what an inference rule [FHV92] is, and even less so for inductive inference rules [MdR94]. However, a simple definition that is sufficient for the aims of this dissertation is given below[4]. In Section 3.2, CID inference rules are defined in abstraction of any operational semantics. CID inference rules are only required to take as *premisses* a set of p-clauses, and infer as *consequence* one p-clause, and to compute validity assessments to annotate consequences using a monotone and continuous function, as defined in Section 2.1. In Section 3.3, CID inference rules are given a concrete operational semantics in terms of well-understood database techniques.

Since their premisses and their consequences are always p-clauses, many different (deductive or inductive) CID inference rules can be applied in the same derivation. Thus, a closure property on the set of available CID inference rules is ensured by the principled treatment of validity assessments. The next section shows how the logical framework that underlies CID systems allows for their formalisation as logic-based databases that preserve the desirable classical metalogical properties of p-Datalog (and hence, of Datalog).

3.2 CID Systems as Deductive Databases

A *combined inference database* (CID) $\mathcal{D}$ is a pair $\langle IDB, EDB\rangle$ where IDB is a set of p-rules, referred to as the *intensional database*, and EDB is a set of p-Datalog ground facts, referred to as the *extensional database*. The clauses in IDB and in EDB are constrained to belong to a given language $\mathcal{L}_\mathcal{D}$, i.e., $(IDB \cup EDB) \subseteq \mathcal{L}_\mathcal{D}$. A *CID schema* defines the lexicon for $\mathcal{L}_\mathcal{D}$ and may include relation names whose extent is not defined either extensionally in EDB or intensionally in IDB. These undefined relation names are, precisely, those whose definition may be computed by inductive inference (thus characterising the discovery of knowledge). In short, the database model of a CID is, therefore, logic-based, with p-Datalog as its underlying logic language.

3.2.1 Syntactical Constraints

The usual constraints imposed on a Datalog-based deductive database [CGT90] are also imposed on a CID, viz., that clauses are range-restricted, and that the sets of relation names occurring in IDB and in EDB are disjoint. The additional constraints imposed on p-Datalog programs [LSV01] are also imposed on a CID, viz., *non-triviality* and *uniqueness* (as described in Section 2.1). In particular, the semantic characterisation of CID system requires only that validity combinators must satisfy the properties listed in Section 2.1. The triple $\langle min, min, max\rangle$ is one example of validity combinators known to satisfy those properties. Finally, to simplify matters, it is assumed that the same validity combinators are associated with every p-clause.

Any Datalog-based deductive database is also a CID in which all of its clauses are annotated with the maximum validity assessment ($\top$) and the $\langle min, min, max\rangle$ validity combinators. Of course, a relational database is a deductive database (and hence, a CID) in which $IDB = \emptyset$.

3.2.2 CID Systems

Unlike a Datalog-based deductive database, in which the inferential apparatus is fixed, the inferential apparatus of CID systems is considered separately. Thus, a *CID system* $\mathcal{S}$ is a pair $\langle \mathcal{D}, \mathcal{I}_\mathcal{D}\rangle$ where $\mathcal{D}$ is a CID and $\mathcal{I}_\mathcal{D}$ is a non-empty set of inference rules. A CID inference rule ρ takes as premisses clauses in $\mathcal{L}_\mathcal{D}$, and infers as consequences p-clauses in the language $\mathcal{L}_\mathcal{D}$ modulo $\mathcal{D}$ (so, derived consequences never conflict with axiomatic data and knowledge in $\mathcal{D}$). The functionality of $\mathcal{S}$ is parameterised by the CID inference rules contained in $\mathcal{I}_\mathcal{D}$. For example, the capability to perform classification is determined by the availability of a corresponding inference rule, say ρ_S, and a CID system $\mathcal{S}$ has this capability if $\rho_S \in \mathcal{I}_\mathcal{D}$. If so, classification rules are inducible in $\mathcal{S}$ by applying ρ_S to clauses in $\mathcal{D}$.

In summary, CID systems have a single underlying logic and a unified inferential apparatus, which taken together, reconcile deductive inference and inductive inference in a principled manner. The next section shows that this uniformity allows for a well-defined semantics[5].

[4] This should not be taken as an attempt to formalise a general notion of inference rules, e.g., in the spirit that is proposed in [FHV92]; such an attempt is beyond the scope of this dissertation.

[5] Proofs can be found in Appendix B.

3.2.3 Model-Theoretical Semantics

The model-theoretic semantics of CID systems extends the model-theoretic semantics of p-Datalog, inasmuch as the notions of valuations and satisfaction generalise their counterparts for p-Datalog [LSV01]. Firstly, the range of valuations is extended from the *Herbrand base* of a p-Datalog program (i.e., containing only ground facts) to a set of clauses $\mathcal{L}_\mathcal{D}$ (i.e., possibly containing rules). Hence, the satisfiability relation is now defined from p-Datalog clauses (referred to as *premisses*) onto a single p-Datalog clause (referred to as a *consequence*). Thus, intensional consequences, i.e., rules, are also admissible and assessed, not only ground facts. Secondly, the satisfiability relation takes a set of CID inference rules $\mathcal{I}_\mathcal{D}$ as an explicit parameter, instead of a fixed inferential apparatus like p-Datalog. Thus, the satisfaction of a clause can be justified in terms of inductive CID inference rules as well. The technical development below follows the steps defined in [LSV01] closely but accommodates the generalisations above. The result is a declarative semantics of CID systems based on validity valuations.

Extended Validity Valuation The first step in developing a declarative semantics for CID systems is to extend validity valuations. An *extended validity valuation* v is a function that annotates all clauses in a set $\mathcal{L}_\mathcal{D}$ with a validity assessment from a domain $\langle \mathbb{C}, \oplus, \otimes, \geq \rangle$, i.e., $v : \mathcal{L}_\mathcal{D} \rightarrow \mathbb{C}$. For example, $v(h \leftarrow l_1, l_2, \ldots, l_n) = C$, for $(h \leftarrow l_1, l_2, \ldots, l_n) \in \mathcal{L}_\mathcal{D}$ and $C \in \mathbb{C}$.

An extended validity valuation v is *acceptable* for a CID $\mathcal{D}$ if and only if v ranges over the language $\mathcal{L}_\mathcal{D}$ of $\mathcal{D}$ and assigns to any p-clause in $\mathcal{D}$ a validity assessment that is the same as the initial annotation in $\mathcal{D}$. For example, $v(h \leftarrow l_1, l_2, \ldots, l_n) = C$, for $(h \dashv C \vdash l_1, l_2, \ldots, l_n) \in IDB \cup EDB$ and $C \in \mathbb{C}$. The underlying assumption is that validity combinators assigned to any p-clause in $\mathcal{L}_\mathcal{D}$ comply to the properties listed in Definition 2.1.

CID Satisfiability The second step in developing a declarative semantics for CID systems is to redefine the notion of *satisfiability*. Satisfiability stipulates when an extended validity valuation v satisfies a CID system $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$. Satisfying a CID system $\mathcal{S}$ means characterising the consequences inferrable by applying CID inference rules in $\mathcal{I}_\mathcal{D}$ starting from premisses in $\mathcal{D}$. Therefore, satisfiability is defined in the context of a CID system $\mathcal{S}$, under the assumption that v is acceptable to $\mathcal{D}$; that every CID inference rule $\rho \in \mathcal{I}_\mathcal{D}$ takes as premisses clauses in $\mathcal{L}_\mathcal{D}$ and infers as consequence clauses in $\mathcal{L}_\mathcal{D}$ modulo the clauses in $\mathcal{D}$; and that the validity assessment assigned to a consequence is also computed by ρ through a function, say g_ρ. For example, for elementary production, $g_\rho(\{\!\{C_0, C_1, C_2, \ldots, C_n\}\!\}) = f^{\leftarrow}(C_0, f^{\wedge}(\{\!\{C_1, C_2, \ldots, C_n\}\!\}))$, where $f^{\leftarrow}$ and $f^{\wedge}$ are the propagator and disjunctive combinators respectively. The application of the disjunctive combinator $f^{\vee}$ is not part of the computation of validity assessments in elementary production. Instead, it is applied a posteriori to consolidate into one the many potential assignments resulting from alternative derivations of the same unannotated clause, thereby enforcing the uniqueness constraint on $\mathcal{D}$.

This notion of satisfiability can be intuitively explained as follows. An extended validity valuation v *satisfies* a CID system $\mathcal{S}$ or not depending on which validity assessments it arbitrarily assigns to clauses in the language $\mathcal{L}_\mathcal{D}$ of $\mathcal{D}$. A clause in $\mathcal{L}_\mathcal{D}$ falls into three mutually exclusive cases: either it is in $\mathcal{D}$, or it is inferrable by a CID inference rule in $\mathcal{I}_\mathcal{D}$ from premisses in $\mathcal{L}_\mathcal{D}$, or else it is neither. For clauses in the first case, v satisfies $\mathcal{D}$ if it assigns the validity assessment C that already annotates the clause in $\mathcal{D}$ (since, by assumption, v is acceptable to $\mathcal{D}$). For clauses in the second case, v satisfies $\mathcal{D}$ if it assigns a validity assessment C that is no less than what is computed for the clause by a CID inference rule that derives it. (If such a clause can be derived by alternative CID inference rules in $\mathcal{I}_\mathcal{D}$ or alternative premisses in $\mathcal{L}_\mathcal{D}$, v must not underestimate the aggregation of all alternative assessments using $f^{\vee}$.) For clauses in the third case, v satisfies $\mathcal{D}$ regardless of the validity assessment assigned to them. In short, v, at most, overestimates the validity assessment of possible consequences. In this sense, it adopts a cautionary approach to its own limitations and gives priority to assessments stemming from previous knowledge states. This approach is constructively formalised from ground instances of premisses to the whole language $\mathcal{L}_\mathcal{D}$, as follows.

Definition 3.1. *Let $\mathcal{D} = \langle IDB, EDB \rangle$ be a CID and $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ be a CID system. Let ρ be an inference rule in $\mathcal{I}_\mathcal{D}$ that computes validity assessments using the function g_ρ. Let $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n$ be ground instances of clauses $S_1, S_2, \ldots, S_n$ in $\mathcal{L}_\mathcal{D}$, respectively. Let Q be a clause in $(\mathcal{L}_\mathcal{D} \setminus (IDB \cup EDB))$. Finally, let v be an extended validity valuation that is acceptable for $\mathcal{D}$. Then:*

1. *v satisfies $\{\!\{\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n\}\!\}$ with respect to ρ, denoted $\models_v^{[\rho]} \{\!\{\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n\}\!\}$, if and only if, for*

every clause Q inferrable by ρ from $\{\!\{\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n\}\!\}$,

$$v(Q) \geq g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n))$$

2. *v satisfies $\{\!\{S_1, S_2, \ldots, S_n\}\!\}$ with respect to ρ, denoted $\models_v^{[\rho]} \{\!\{S_1, S_2, \ldots, S_n\}\!\}$, if and only if it satisfies every ground instance of $S_1, S_2, \ldots, S_n$.*

3. *v satisfies $\mathcal{D}$ with respect to $\mathcal{I}_\mathcal{D}$, denoted $\models_v^{[\mathcal{I}_\mathcal{D}]} \mathcal{D}$, if and only if, for every $\rho \in \mathcal{I}_\mathcal{D}$,*

   (a) *v satisfies, with respect to ρ, every finite subbag of p-clauses in $\mathcal{L}_\mathcal{D}$ that can appear as premisses in ρ and*

   (b) *for every $Q \in (\mathcal{L}_\mathcal{D} \setminus (IDB \cup EDB))$, if Q is inferrable by ρ from $S_1, S_2, \ldots, S_n$ then:*

$$v(Q) \geq f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\})$$

   where $f^\vee$ is the disjunctive combinator associated with Q.

4. *v satisfies $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ if v satisfies $\mathcal{D}$ with respect to $\mathcal{I}_\mathcal{D}$.*

$\diamond$

The rationale for this construction becomes clearer once the notion of the least validity valuation is explained, in the next section.

The Least Validity Valuation The third and final step in developing a declarative semantics for CID systems is to show that there exists one *least extended validity valuation* that satisfies each CID system.

The least extended validity valuation depends on an ordering on validity valuations. Intuitively, an extended validity valuation v is smaller than an extended validity valuation u, if v assigns, to clauses in the third case above, a validity assessment less than the one u arbitrarily assigns to the same clauses. Moreover, v is less than u if, likewise, v assigns a smaller validity assessments to clauses in the second case. This is possible if v assigns to premisses of such a clause a validity assessment less than u does (unless these premisses are taken from $\mathcal{D}$, in which case both v and u have to preserve the original validity assessments in order to be acceptable). The least extended validity valuation w minimises the validity assessment assigned. Thus, w assigns the minimal assessment ($\bot$) to clauses in the third case, but preserves the original assessments of clauses in the first case. Therefore, w has limited room for minimising the validity assessment assigned to a direct consequence Q whose premisses all belong to $\mathcal{D}$. If a CID inference rule in $\mathcal{I}_\mathcal{D}$ can compute a validity assessment for Q, say C, then the least w can assign to Q is C, or else, w would satisfy $\mathcal{S}$ as defined above. Hence, w can only minimise up to the point where it coincides with the validity assessment that is assigned to the directly inferred consequences of $\mathcal{S}$. The same argument can be extended to indirect consequences of $\mathcal{S}$. In short, the least extended validity valuation assigns, to all direct or indirect consequences that are derived only from premisses in $\mathcal{D}$, the exact validity assessments computed for them by CID inference rules in $\mathcal{I}_\mathcal{D}$, and aggregated via $f^\vee$. Otherwise, w assigns $\bot$ to all other clauses in $\mathcal{L}_\mathcal{D}$ (i.e., those that require premisses that are not inferrable from clauses in $\mathcal{D}$, or those that are not inferrable at all considering the apparatus in $\mathcal{I}_\mathcal{D}$). This intuitive explanation is formalised as follows.

Let $\langle \mathbb{C}, \oplus, \otimes, \geq \rangle$ be a validity domain, and let $u : \mathcal{L}_\mathcal{D} \to \mathbb{C}$ and $v : \mathcal{L}_\mathcal{D} \to \mathbb{C}$ be any two extended validity valuations for a CID system $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ (such as defined in Definition 3.1). The ordering $\geq$ over $\mathbb{C}$ can be made an ordering over all extended validity valuations acceptable to $\mathcal{D}$. More formally, for any validity valuations u and v, $u \geq v$ if and only if $u(S) \geq v(S)$, for S in the language $\mathcal{L}_\mathcal{D}$, the following equations also hold:

1. $(u \oplus v)(S) = u(S) \oplus v(S)$;
2. $(u \otimes v)(S) = u(S) \otimes v(S)$.

Given this construction, the following lemma holds:

Lemma 3.1. *Let $\mathcal{D}$ be any CID, and $\mathbb{V}_\mathcal{D}$ be the set of all extended validity valuations acceptable to $\mathcal{D}$, then $\langle \mathbb{V}_\mathcal{D}, \oplus, \otimes, \geq \rangle$ is a complete lattice.*

Proof. *See Lemma B.1 in Appendix B.* □

The least element of $\langle \mathbb{V}_{\mathcal{D}}, \oplus, \otimes, \geq \rangle$ is an extended validity valuation $v_{\perp}$ which assigns the minimum assessment $\perp$ to every p-clause in $\mathcal{L}_{\mathcal{D}}$, except those in $\mathcal{D}$ (for which the original validity assessment is retained). Thus, $v_{\perp}$ only satisfies $\mathcal{D}$ if $\mathcal{D}$ has no inferrable consequences in $\mathcal{L}_{\mathcal{D}}$. The greatest element is $v_{\top}$ which assigns the maximum assessment $\top$ to every p-clause in $\mathcal{L}_{\mathcal{D}}$, again, except those in $\mathcal{D}$ (for which the original validity assessment is retained). Thus, $v_{\top}$ trivially satisfies every CID because $v_{\top}$ overestimates the validity assessment of all inferrable consequences.

That the equivalent of the model intersection property in standard logic programming [vEK76, AE82] and deductive databases [CGT90] holds for CID systems can be stated as follows.

Lemma 3.2. *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$ be a CID system, let u and v be any two extended validity valuations, each of which satisfies $\mathcal{S}$, then $u \otimes v$ is also an extended validity valuation that satisfies $\mathcal{S}$.*

Proof. *See Lemma B.2 in Appendix B.* □

The Least Extended Validity Valuation for a CID system corresponds to the notion of a minimal model in standard logic programming [vEK76, AE82] and deductive databases [CGT90].

Theorem 3.1. *Let $\langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$ be a CID system, and $\mathbb{V}_{\mathcal{D}}$ the set of all extended validity valuations that are acceptable to $\mathcal{D}$, then, $\otimes\{v \mid\mid v \leftarrow \mathbb{V}_{\mathcal{D}}, \models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}\}$ is the least extended validity valuation that satisfies $\mathcal{D}$.*

Proof. *See Theorem B.1 in Appendix B.* □

The formal development so far followed closely the development of a semantics for p-Datalog programs in [SV97], but for the rephrasing of lemmas and proofs to the broader context of CID systems. The remainder of this section shows that the declarative semantics of a CID system is a conservative extension of the semantics of p-Datalog programs, and hence, a conservative extension of the semantics of deductive databases.

Claim. Assume now that a CID system $\mathcal{P} = \langle \langle IDB, EDB \rangle, \{eep\} \rangle$ where *eep* is extended elementary production (See Section 3.3.1). Assume also that v is an extended validity valuation that satisfies $\mathcal{P}$. An extended validity valuation v_{HB} that satisfies the corresponding p-Datalog program *IDB* can be obtained simply by reducing the range of v to the Herbrand base of $\mathcal{L}_{\mathcal{P}}$.

Proof. *See Claim B.1 in Appendix B.* □

The claim states that CID systems conservatively extend p-Datalog programs.

Corollary 3.1. *Let $\mathcal{D} = \langle IDB, EDB \rangle$ and $\mathcal{S} = \langle \mathcal{D}, \{eep\} \rangle$ be, respectively, a CID and a CID system in which every p-clause in $\mathcal{L}_{\mathcal{D}}$ is annotated with maximum validity $\top$, and let $v : \mathcal{L}_{\mathcal{D}} \rightarrow \{\top, \perp\}$ be an extended validity valuation. If v satisfies $\mathcal{D}$ it also gives rise to a Herbrand interpretation for the deductive database that corresponds to $\mathcal{D}$.*

Proof. *See Corollary B.1 in Appendix B.* □

Hence, CID systems conservatively extend classical deductive databases as well.

3.2.4 Fixpoint Semantics

The development of a fixpoint semantics for CID systems is again adapted from the development of a fixpoint semantics for p-Datalog programs [LSV01]. As usual, it is based on an *immediate consequence operator* $T_{\mathcal{D}}$. This section shows that a least fixpoint of $T_{\mathcal{D}}$ exists for every CID system $\langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$. This is used to characterise the fixpoint semantics of CID systems.

Definition 3.2. *Let $\langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$ be a CID system with language $\mathcal{L}_{\mathcal{D}}$, and let $\mathbb{V}_{\mathcal{D}}$ be the set of extended validity valuations that are acceptable to $\mathcal{D}$. The immediate consequence operator $T_{\mathcal{D}}$ is a mapping from $\mathbb{V}_{\mathcal{D}}$ to $\mathbb{V}_{\mathcal{D}}$, such that for every extended validity valuation $v \in \mathbb{V}_{\mathcal{D}}$, for every CID inference rule $\rho \in \mathcal{I}_{\mathcal{D}}$, and for every p-clause $Q \in \mathcal{L}_{\mathcal{D}}$ that can be inferred by ρ from $S_1, S_2, \ldots, S_n$ then:*

$$T_{\mathcal{D}}(v)(Q) = f^{\vee}(\{\!\{ g_{\rho}(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_{\mathcal{D}} \}\!\})$$

where $f^{\vee}$ is the disjunctive combinator associated with Q. ◇

The bottom-up iteration of $T_{\mathcal{D}}$ is defined in the usual manner [vEK76], as follows:

$$T_{\mathcal{D}}^{k} = \begin{cases} v_{\perp} & \text{if } k = 0 \\ T_{\mathcal{D}}(T_{\mathcal{D}}^{k-1}) & \text{if } k \text{ is a successor ordinal} \\ \oplus\{T_{\mathcal{D}}^{\ell} | \ell < k\} & \text{if } k \text{ is a limit ordinal} \end{cases}$$

The computation of the immediate consequence operator corresponds to the iterative traversal of the lattice defined over $\mathbb{V}_{\mathcal{D}}$. The successor ordinal represents the immediate next iteration step on this computation, while the limit ordinal represents the accumulation (according to $\oplus$) of $T_{\mathcal{D}}^{\ell}$ up to the limit ordinal k. Note that for any p-clause Q if Q cannot be inferred by applying a CID inference rule in $\mathcal{I}_{\mathcal{D}}$ from premisses $S_1, S_2, \ldots, S_n$ in $\mathcal{D}$, than $T_{\mathcal{D}}(v)(Q) = \perp$, as assigned by $v_{\perp}$ in $T_{\mathcal{D}}^{0}$. Finally, the least fixpoint of the immediate consequence operator is defined as follows.

Definition 3.3. *The* least fixpoint *of $T_{\mathcal{D}}$, denoted lfp($T_{\mathcal{D}}$), is an extended validity valuation of $\mathcal{D}$ such that it is a fixpoint of $T_{\mathcal{D}}$, and for every other fixpoint u of $\mathcal{D}$, lfp($T_{\mathcal{D}}$) $\leq u$.* ◇

Definition 3.3 and Lemmas B.3 and B.4 in Appendix B are instrumental to demonstrate the equivalence between declarative and fixpoint semantics.

3.2.5 Operational Semantics

The operational semantics for CID systems given in this section is novel, unlike the model-declarative and fixpoint semantics for CID systems, introduced above, that are derived from the corresponding ones for p-Datalog [LSV01]. The operational semantics of p-Datalog is based on either SLD-resolution [Rob65] or a bottom-up semi-naïve evaluation algorithm [CGT90]. In both cases, the inference apparatus is fixed and deductive, while, here, inductive inference must be an integral part of the inferential system. Therefore, a new, reconciled, uniform foundation needs to be laid.

The operational semantics for CID systems is based on a notion of *CID tasks* which capture well-formed compositions of different CID inference rules to derive p-clauses. CID tasks were informally introduced in Section 1.1. In this section, CID tasks are formalised, as follows.

Definition 3.4. *Let $\mathcal{D} = \langle IDB, EDB \rangle$ and $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$ be a CID and CID system, respectively. Let Φ be the function (defined in Section 3.3.2) that applies the disjunctive combinator $f^{\vee}$ so as to consolidate the provisional consequences of ρ and satisfy the constraints in Section 3.2. A CID task admissible by $\mathcal{D}$ is a single-sink, finite directed acyclic graph in which:*

1. *each node denotes a CID inference rule $\rho \in \mathcal{I}_{\mathcal{D}}$ wrapped by Φ;*
2. *each node has one or more incoming arrows that denote the domains the denoted CID inference rule ρ draws premisses from;*
3. *each node has a single outgoing arrow that denotes the consequences the denoted CID inference rule ρ infers;*
4. *an outgoing arrow of a node N may be bound to an incoming arrow of a node N', $N' \neq N$, in which case the binding denotes that the consequences of N are candidate premisses for N';*
5. *incoming arrows that are not be bound to any other node denote parameters of the task (in which case these arrows are assumed bound to collections of p-clauses from $\mathcal{D}$) for the CID task to be evaluated);*
6. *the outgoing arrow of the sink node denotes the outcome of the CID task.*

◇

An example CID task is depicted in Figure 1. A CID task is the specification of one particular inferential task. Despite some structural similarity, a CID task differs from the standard notion of a proof tree in deductive databases in the following respects: (a) nodes in CID tasks denote inference rules rather than premisses; (b) a single CID task denotes the bulk derivation of potentially many consequences, rather than of a single one.

A construct denoting bulk derivations opens the way for an execution semantics that implements an exhaustive, bottom-up, dataflow computation that resembles more closely to the classical mechanisms used in database evaluation engines. One such operational semantics is defined in Section 3.3.

Note that a CID task infers the consequences ejected in the root node, moreover, with the validity assessment annotated to them. Note also that the outcome of a CID task complies with the constraints enforced on CID systems, thus ready to be assimilated, considering the validity assessment are principled propagated through the task. The next section shows that the operational semantics constructed in this section is equivalent to the fixpoint semantics of CID systems, and hence, by transitivity, to the model-theoretic semantics of CID systems.

3.2.6 Equivalence of Different Semantic Characterisations

This section shows that the declarative, fixpoint and operational semantics of CID systems (in Sections 3.2.3, 3.2.4, and 3.2.5, respectively) are equivalent.

Firstly, the declarative and fixpoint semantics are shown to coincide as follows.

Theorem 3.2. *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ be a CID system. Then*

$$lfp(T_\mathcal{D}) = \otimes\{v \parallel v \leftarrow \mathbb{V}_\mathcal{D}, \models_v^{[\mathcal{I}_\mathcal{D}]} \mathcal{D}\}$$

Proof. *See Theorem B.2. in Appendix B.* □

Secondly, the operational semantics is shown be sound and complete, insofar as it is equivalent to the fixpoint semantics.

Theorem 3.3. ***(Soundness)*** *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ be a CID system. Let Q be any p-clause in the outcome of a CID task. Then, if Q is annotated with validity assessment C then $C \leq lfp(T_\mathcal{D})(Q)$.*

Proof. *See Theorem B.3 in Appendix B.* □

The soundness result above guarantees that the evaluation of a CID task posed to a CID system $\mathcal{S}$ contains only consequences inferrable from its CID $\mathcal{D}$ by application of CID inference rules in $\mathcal{I}_\mathcal{D}$.

Theorem 3.4. ***(Completeness)*** *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ be a CID system. Let Q be any p-clause such that $lfp(T_\mathcal{D})(Q) = T_\mathcal{D}^k(Q)$, for some integer k. If $Q \notin \mathcal{D}$, then there exists a CID task that, if evaluated, will contain Q and will annotate Q with a validity value C such that $lfp(T_\mathcal{D})(Q) \leq C$.*

Proof. *See Theorem B.4 in Appendix B.* □

The completeness result above guarantees that if a p-clause Q can be inferred from $\mathcal{D}$, then it is possible to specify at least one CID task that, when mapped to monoid calculus and evaluated, will compute Q. Thus, every data and knowledge item that can be justified in $\mathcal{D}$, with respect to the inference rules $\mathcal{D}$ provides, can also be retrieved by specifying and evaluating one CID task.

In summary, this section has shown that, for any CID system, the defined notions of least extended validity valuation, least fixpoint of its immediate consequence operator and CID tasks all coincide. This implies that CID systems have equivalent model-theoretic, fixpoint, and operational semantics. The next section shows that the operational semantics in Section 3.2.5 naturally gives rise to a concrete operational semantics in terms of the monoid calculus [FM00].

3.3 CID Systems as Classical Database Systems

A database-centred operational semantics is developed for CID tasks as follows. Firstly, CID inference rules introduced in Section 3.1 are shown to be expressible in a monoid calculus. Secondly, the enforcement of the constraints in Section 3.2.1 and thirdly the composition of CID inference rules are also shown to be expressible a monoid calculus. It follows that every CID task admissible by a CID system can be compiled into a single monoid calculus expression, which, in turn, can be translated into an equivalent monoid-algebraic expression to which, as is explained in Section 5, one can make correspond a set of algorithms that give rise to a concrete algebra for evaluating CID tasks. The benefits of this technical move are to ensure that CID systems admit implementations as mainstream database systems, unlike, e.g., the many approaches to deductive databases surveyed in [Min96].

3.3.1 Representing CID Inference Rules

Recall the definition of elementary production [CGT90]:

$$\frac{h \leftarrow l_1, l_2, \ldots, l_n \quad f_1 \leftarrow \quad f_2 \leftarrow \ \ldots \ f_n \leftarrow}{substitute(h \leftarrow, \theta)} \text{ if } \exists \theta \ \theta = mgu([l_1, l_2, \ldots, l_n], [f_1, f_2, \ldots, f_n]) \quad (1)$$

One reading of (1) is that it requires one to choose the premisses, then ensure that the literals in the body of the leftmost premiss in (1) match the remaining premisses pairwise in order to yield a substitution θ, then eject as consequence the outcome of applying the substitution θ to the head of the leftmost premiss in (1). This reading is consistent with the procedural interpretation of (1) embedded in most classical bottom-up query evaluation algorithms, e.g., the algorithm in Figure 2. The functions *mgu* and *substitute*

```
ep(IDB,EDB) ≝
begin
    Answers := ∅
    for each (h ← l1, l2, ..., ln) in IDB
        for each [f1, ..., fn] of EDB
            θ := mgu([f1, f2, ..., fn], [f1, f2, ..., fn])
            if θ ≠ ▽ then
               Answers := Answers ∪ {substitute((h ←), θ)}
    return Answers
end
```

Figure 2: Procedural Interpretation of Elementary Production [CGT90]

are the standard functions from logic programming, as in [CGT90]. This procedural interpretation of (1) is sufficiently close in structure to the procedural interpretation of monoid comprehensions to suggest that (1) can be captured as a monoid comprehension, as follows. Generators can capture the scans of *IDB* and *EDB* that yield candidate premisses. These are filtered to ensure that they have a most general unifier (i.e., with $\triangledown$ denoting that function *mgu* has failed to find one). Those premisses that satisfy the filter justify the ejection of the corresponding consequence, which is constructed by applying the θ substitution to the head of the first premiss. The monoid comprehension is a CID inference rule, if, in addition, it computes a validity assessment for the consequence from those asserted to the premisses (in the last line in *eep* below). Note that only the conjunctive and the propagator combinators are used to annotate a validity assessment, the disjunctive combinator is applied in outer comprehension to consolidate possibly alternative assessments for the same consequence. The resulting CID inference rule, in this case, is as follows:

$$\begin{array}{l} eep(IDB,EDB) \stackrel{def}{=} \\ \uplus\{\ substitute((h \mapsto C \vdash), \theta) \ \ [\![\\ \quad (h \mapsto C_0 \vdash l_1, l_2, \ldots, l_n) \quad \leftarrow IDB, \\ \quad (f_1 \mapsto C_1 \vdash) \qquad\quad \leftarrow EDB, \\ \quad \ldots, \\ \quad (f_n \mapsto C_n \vdash) \qquad\quad \leftarrow EDB, \\ \quad \theta \equiv mgu([l_1, l_2, \ldots, l_n], [f_1, f_2, \ldots, f_n]), \\ \quad \theta \neq \ \triangledown, \\ \quad C \ \equiv \ f^{\leftarrow}(C_0, f^{\wedge}(C_1, C_2, \ldots, C_n)) \\ \}\end{array}$$

The use of the bag monoid ($\uplus$) is necessary for the sound application of disjunctive combinators [LSV01] later, because they are defined over bags of validity assessments.

Inductive CID inference rules must be elicited as knowledge that appears embedded in data mining algorithms. Since these are often search-based, the structural differences to monoid comprehensions are significant. Therefore, examples of inductive CID inference rules are postponed until Section 4 where a more detailed discussion is provided.

3.3.2 Enforcing Constraints on CID Inference Rules

Deductive inference rules, including *eep*, can derive the same ground fact with different validity assessments if different premisses are taken at different applications, thereby requiring special treatment to

enforce the uniqueness constraint imposed on CID systems[6]. This makes consequences provisional until their possibly different validity assessments are aggregated into a single value and clauses annotated with the minimum value $\bot$ are discarded. The disjunctive combinators are introduced exactly to solve this issue [LSV01]. Their application however is only sound after all alternative validity assessments are computed.

The application of the disjunctive combinator is also expressible in a monoid calculus, as follows. Given a monoid comprehension expressing an inference rule, an outer comprehension wraps it to eject only definitive consequences. More precisely, let the following simple syntactic mappings be defined:

$$\begin{array}{l} \mathit{get\_validity}(\mathsf{h}\!\leftarrow\!\!\dashv C \vdash l_1, l_2, \ldots, l_n)\!=\!C, \\ \mathit{get\_literals}(\mathsf{h}\!\leftarrow\!\!\dashv C \vdash l_1, l_2, \ldots, l_n)\!=\!\langle \mathsf{h}, [l_1, l_2, \ldots, l_n]\rangle, \\ \mathit{set\_validity}(\langle \mathsf{h}, [l_1, l_2, \ldots, l_n]\rangle, C)\!=\!\mathsf{h}\!\leftarrow\!\!\dashv C \vdash l_1, l_2, \ldots, l_n. \end{array}$$

Firstly, *get_literals* is applied to clauses in S stripping them of validity assessments. Let $\Pi(S)$ abbreviate this corresponding monoid comprehension:

$$\Pi(S) \stackrel{def}{=} \cup\{\mathit{get\_literals}(T) \,[\![\, T \leftarrow S\}$$

For example, $\Pi(\{\!\{ \mathsf{h}\!\leftarrow\!\!\dashv 0.4\vdash, \mathsf{h}\!\leftarrow\!\!\dashv 0.7\vdash, \mathsf{h'}\!\leftarrow\!\!\dashv 0.6\vdash \}\!\}) = \{\langle \mathsf{h}, [\,]\rangle, \langle \mathsf{h'}, [\,]\rangle\}$.

Secondly, $\Pi(S)$ is made a generator in a monoid comprehension Ψ that uses *get_validity* to obtain validity assessments must be combined (or discarded if equal to the minimum $\bot$) and ejects an aggregated validity assessment computed using $f^\vee$ over each group of non-trivial validity assessments, as follows:

$$\begin{array}{l} \Psi(S) \stackrel{def}{=} \cup\{\langle T, f^\vee\{\, C \,[\![\, P \leftarrow S, \\ \qquad\qquad\qquad C \equiv \mathit{get\_validity}(P), \\ \qquad\qquad\qquad C \neq \bot, \\ \qquad\qquad\qquad T = \mathit{get\_literals}(P)\ \}\ \rangle\ [\![\\ \qquad T \leftarrow \Pi(S)\} \end{array}$$

For example, for $S = \{\!\{ \mathsf{h}\!\leftarrow\!\!\dashv 0.4\vdash, \mathsf{h}\!\leftarrow\!\!\dashv 0.7\vdash, \mathsf{h'}\!\leftarrow\!\!\dashv 0.6\vdash \}\!\}$ and $f^\vee = max$:

$$\begin{array}{rcl} \Psi(S) & = & \{\langle\langle \mathsf{h}, [\,]\rangle, f^\vee(\{\!\{0.4, 0.7\}\!\})\rangle, \langle\langle \mathsf{h'}, [\,]\rangle, f^\vee(\{\!\{0.6\}\!\})\rangle\} \\ \Psi(S) & = & \{\langle\langle \mathsf{h}, [\,]\rangle, max(\{\!\{0.4, 0.7\}\!\})\rangle, \langle\langle \mathsf{h'}, [\,]\rangle, max(\{\!\{0.6\}\!\})\rangle\} \\ \Psi(S) & = & \{\langle\langle \mathsf{h}, [\,]\rangle, 0.7\rangle, \langle\langle \mathsf{h'}, [\,]\rangle, 0.6\rangle\} \end{array}$$

Thirdly, $\Phi(S)$ uses *set_validity* to construct p-clauses with the aggregated validity assessment in Ψ, for example, constructing $\mathsf{h}\!\leftarrow\!\!\dashv 0.7\vdash$ from $\langle\langle \mathsf{h}, [\,]\rangle, 0.7\rangle$, as follows:

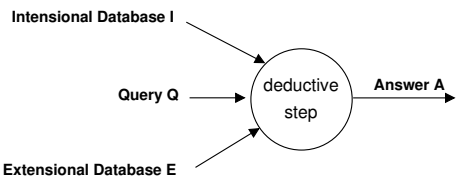
$$\Phi(S) \stackrel{def}{=} \cup\{\mathit{set\_validity}(P, C)\ [\![\ \langle P, C\rangle \leftarrow \Psi(S)\}$$

Thus, $\Phi(\{\!\{ \mathsf{h}\!\leftarrow\!\!\dashv 0.4\vdash, \mathsf{h}\!\leftarrow\!\!\dashv 0.7\vdash, \mathsf{h'}\!\leftarrow\!\!\dashv 0.6\vdash \}\!\}) = \{\mathsf{h}\!\leftarrow\!\!\dashv 0.7\vdash, \mathsf{h'}\!\leftarrow\!\!\dashv 0.6\vdash\}$. In short, $\Phi(P)$ enforces the non-triviality and uniqueness constraints on the outcomes of deductive inference rules.

3.3.3 Composing CID Inference Rules

Given that CID inference rules and the enforcement of constraints on their outcomes have been shown to be expressible in a monoid calculus, this subsection shows how every CID task can be interpreted as denoting a monoid comprehension. More precisely, this mapping is constructed as follows:

1. a node is mapped onto two nested monoid comprehensions, the inner is the denoted CID inference rule (e.g., that in Section 3.3.1), the outer one is Φ (as defined in Section 3.3.2);
2. the incoming arrows of a node are mapped to a generator domain, either through comprehension nesting if the arrow is bound or to term bound by variable instantiation otherwise;
3. the single outgoing arrow of a node is mapped to a term denoting the p-clauses ejected in the head of the comprehension expression associated with the source node;

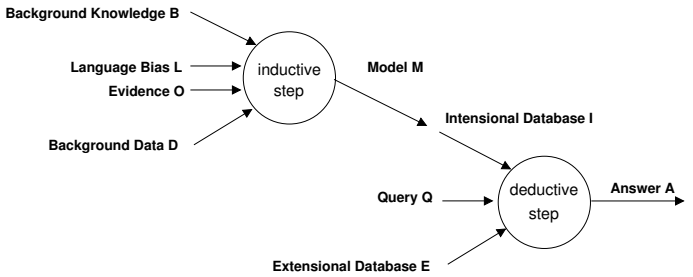


$$\begin{array}{l} \Phi(\oplus\{\ \mathsf{A}\ [\![\\ \qquad R \leftarrow \mathsf{I}, \\ \qquad F \leftarrow \mathsf{E}, \\ \qquad \ldots \\ \}) \end{array}$$

[6] This can be avoided in inductive inference rules by means of an appropriate language bias.

4. an arrow binding is mapped to a generator in the parent node that scans the p-clauses ejected by the corresponding child node;

5. unbound arrows are instantiated to a concrete collection of p-clauses before evaluation;



$$\Phi(\oplus\{\ \mathsf{A}\ \|\ R \leftarrow \Phi(\otimes\{\mathsf{M}\ \|\ N \leftarrow \mathsf{L}, \ldots\}), F \leftarrow \mathsf{E}, \ldots\})$$

6. the whole graph gives rise to a single well-formed, nested monoid comprehension expression;

7. the p-clauses contained in the collection resulting from evaluation are the task outcome.

In summary, every CID task is expressible as a single monoid calculus expression that composes monoid comprehensions denoting CID inference rules.

There are cases when the same CID inference rule is composed with itself several times in order to denote iteration, for example, to compute all deductive consequences of a CID system (as discussed in Section 4). The monoid calculus also provides a convenient, abbreviated notation for bounded iteration, viz., a function composition monoid. For example, the following expression

$$\circ\{\lambda(R,X).\Phi(\rho(R,X))\ \|\ R\ \leftarrow\ N\}(E)$$

denotes $\Phi(\rho(R_n,\ldots(\Phi(\rho(R_2,(\Phi(\rho(R_1,E))))))\ldots))$ for $N=\{R_1,R_2,\ldots,R_n\}$. This short-hand notation preserves the metalogical properties presented in Section 3.2, once, for a known n, it is statically unfolded into a chain of nested monoid comprehensions.

Unbounded iteration requires a kind of fixpoint mechanism that is not part of the monoid calculus [FM00]. Nevertheless, it is comparatively straightforward to enrich the calculus above with a fixpoint construct. A CID inference rule ρ that must be iterated over the domain X gives rise to a function $\rho : \mathbb{T}_\oplus \rightarrow \mathbb{T}_\oplus$ over a collection monoid X of type $\mathbb{T}_\oplus$. If the fixpoint of ρ exists, it is defined as follows.

$$fixpoint(\lambda X.\rho(X), Current) \stackrel{\text{def}}{=}$$
begin
  New := $\Phi(\rho(Current))$
  if $New \subseteq Current$
    then return $Current$
    else return $fixpoint(\lambda X.\rho(X), \Phi(New \oplus Current))$
end

where $\oplus$ is the merge operation, and $\subseteq$ the containment predicate over collections of p-clauses. If the CID inference rule ρ converges to a fixpoint (e.g., as does extended elementary production) then the fixpoint construct above also preserves the metalogical properties presented in Section 3.2.

Finally, the monoid comprehension denoting a CID task can be evaluated either by mapping it to the nested-loop procedural semantics of such expressions, or, taking advantage of optimisation opportunities at each step, by translating it to an abstract algebra and then mapping the result to a concrete one. Moreover, the evaluation strategy is uniform for any CID task, irrespective of whether deductive or inductive inference rules are involved. This characteristic is essential in matching, at the evaluation stage, the seamlessness achieved at the specification stage.

4 CID Inference Rules

The purpose of this section is to show how the capabilities of CID systems can be exploited to provide information in extensional and intensional forms, through query answering and knowledge discovery, respectively. Querying, in CID systems as elsewhere, retrieves data items. The novelty lies in their ability to combine querying with knowledge discovery, a process that facilitates uncovering *interesting* [FPS96a] knowledge items. In this section, deductive CID inference rules are shown to capture query answering,

while inductive CID inference rules are shown to capture usual knowledge discovery techniques, such as classification, clustering and association. Either way, CID inference rules are specified in the same monoid calculus [FM00]. Therefore, the operational semantics of CID systems, as described in Section 3, provides the foundations upon which query answering and knowledge are reconciled at the task expression level.

4.1 Expressing Deductive Query Answering

Elementary production [CGT90] characterises deductive consequences of a deductive database. After embedding in some proof procedure, elementary production become the essential core of a process by means of which all the answers for a particular query are inferred. In this section, the CID inference rule referred to as extended elementary production (as defined in Section 3.3.1) is shown to be likewise sufficient to express the computation of all the deductive consequences of a CID system and all the answers to a query posed to this CID system.

4.1.1 Bottom-up Computation of Deductive Consequences

There exists a direct relationship between the syntactical form of p-rules and the number of times that extended elementary production needs to be applied in order to derive all their deductive consequences. This mirrors the relationship that between the syntactical form of Datalog rules and the number of iterations in the classical bottom-up naïve evaluation algorithm [CGT90]. The composition of deductive CID tasks should therefore take this relationship into account in order to guarantee that results are exhaustive.

Conjunctive CIDs Firstly, a CID $\langle IDB, EDB\rangle$ in which every p-rule denotes a conjunctive query[7] requires a single application of extended elementary production for its consequences to be computed. For every p-rule r in IDB, every relation name occurring in the body of r denotes an EDB relation. If so, then EDB must already contain all the ground facts that might possibly match literals in the body of r, since no inferred consequence from r can do so. Thus, the following CID task is sufficient to characterise all deductively inferrable consequences of a conjunctive CID:

$$\begin{array}{l} \mathit{conjunctive\_consequences}(\langle IDB, EDB\rangle) \stackrel{def}{=} \\ \quad \Phi(\mathit{eep}(IDB, EDB)) \end{array}$$

where *eep* is as defined in Section 3.3.1 and Φ enforces the uniqueness constraint, as defined in Section 3.3.2. The CID in Appendix A falls in this case, as illustrated in the following example.

Example 4.1. Let $\mathcal{D}_0$ be the CID in Appendix A. Then, the expression

$$\mathit{conjunctive\_consequences}(\mathcal{D}_0)$$

denotes:
{relevant_paper(diabetes_control, biochemistry_explained, lewis, bioinformatics, 2003)$\leftarrow\!\!\shortmid 0.3 \shortmid\!\!\rightarrow$
relevant_paper(diabetes_control, epidemiology_explained, fiona, bioinformatics, 2003)$\leftarrow\!\!\shortmid 0.4 \shortmid\!\!\rightarrow$
relevant_paper(diabetes_control, immunology_in_24hours, fiona, bioinformatics, 2003)$\leftarrow\!\!\shortmid 0.3 \shortmid\!\!\rightarrow$
relevant_paper(sars_epidemic, bioinformatics_in_24hours, sally, ieee_csb, 2003)$\leftarrow\!\!\shortmid 0.4 \shortmid\!\!\rightarrow$
relevant_paper(sars_epidemic, genetics_for_dummies, zoe, ismb, 2003)$\leftarrow\!\!\shortmid 0.5 \shortmid\!\!\rightarrow$
relevant_paper(sars_epidemic, immunology_in_24hours, fiona, bioinformatics, 2003)$\leftarrow\!\!\shortmid 0.3 \shortmid\!\!\rightarrow$
relevant_paper(sars_epidemic, molecular_biology_explained, james, ieee_csb, 2003)$\leftarrow\!\!\shortmid 0.5 \shortmid\!\!\rightarrow$
relevant_paper(sars_epidemic, molecular_biology_unleashed, lynda, ieee_csb, 2003)$\leftarrow\!\!\shortmid 0.5 \shortmid\!\!\rightarrow$}

Each p-fact lists the attributes of a research paper that might be of interest in the context of a project, insofar as the subject matter of the former may be relevant for the work carried out in the latter. $\triangleleft$

[7] Conjunctive queries are already more expressive than the propositional knowledge representation used by mainstream data mining techniques [AHV95].

Hierarchical CIDs A CID $\langle IDB, EDB\rangle$ in which IDB relation names can non-recursively occur in the body of p-rules (i.e., in which the IDB dependency graph [CGT90] contains no cycle) requires multiple applications of extended elementary production for all its deductive consequences to be computed. The first application derives consequences from rules which depend on no other (i.e., the conjunctive ones), then, a second application derives consequences from rules that depend at most on the conjunctive rules, and so on. This indicates the need for stepwise application of extended elementary production. In this sense, the use of *conjunctive_consequences* alone is not sufficient. Since the dependency graph of IDB is finite and contains no cycle, all paths in it are finite. Thus, the number of deductive steps needed is bounded, and can statically computed in the form of the length of the longest path in the dependency graph, or alternatively, by the number of strata in a stratification of IDB [CGT90].

In this case, p-rules are still as expressive as conjunctive queries in the sense that they can be totally unfolded [TS84], however they explicitly represent hierarchical relationships between different knowledge items, which conjunctive queries do not. Thus, their potential for clearer knowledge representation justifies the additional complexity of computing their deductive consequences.

The monoid calculus offers a shorthand notation to express statically bounded iteration in the definition of CID tasks, as described in Section 3.3.3, viz., the functional-composition monoid. A stratification[8] IDB^s of IDB is denoted by a list of its strata, each of which is a set of p-rules that can be used to guide the unfolding of the functional-composition monoid denoting the iterative application of elementary production. Thus, the set of the all deductive consequences of a stratified hierarchical CID $\langle IDB^s, EDB\rangle$ coincides with the set p-clauses denoted by the following expression:

$$\begin{array}{l} \mathit{hierarchical\_consequences}(\langle IDB^s, EDB\rangle) \stackrel{def}{=} \\ \quad \circ\{\lambda X.\Phi(\mathit{eep}(R, X) \cup X) \,[\![\; R \leftarrow IDB^s \;]\!]\}(EDB) \end{array}$$

where R ranges over the strata of IDB^s and X is initialised with EDB and then successively accumulates (in the style of a *fold* functional) the conjunctive consequences of each stratum R. Note that Φ is used to enforce the uniqueness constraint at each step. The comprehension above expands to a CID task that nests as many applications of extended elementary production as there are strata in IDB. This can be illustrated as follows.

Example 4.2. Let the following p-rules r_1 and r_2 define two relations:

```
researcher_contact_details(Researcher,Address,Telephone,URL)↤1.0⊢
  researcher(Researcher,_,_,_,_,Affiliation,_),
  research_unit(Affiliation,_,_,_,_,Address,Telephone,URL)
project_contact_details(Project,Manager,Address,Telephone,URL)↤1.0⊢
  project(_,Project,Manager,_),
  researcher_contact_details(Manager,Address,Telephone,URL)
```

r_1 defines the contact details of a researcher as being the address, telephone and URL of the unit he/she is affiliated to. r_2 defines the contact details of a project in terms of those defined by r_1 for its manager. Let a CID $\mathcal{D}_1$ have the same extensional database EDB_0 as the CID in Appendix A and the intensional database $IDB_1 = [\{r_1\}, \{r_2\}]$. IDB_1 has two strata because project_contact_details/5 *depends on* researcher_contact_details/4 *and* project/3*, and hence the set of all deductive consequences of $\mathcal{D}_1$ can only be computed by two applications of extended elementary production. The expression*

$$\mathit{hierarchical\_consequences}(\langle[\{r_1\}, \{r_2\}], EDB_0\rangle)$$

unfolds into the expression

$$\Phi(\mathit{eep}(\{r_2\}, EDB_1) \cup EDB_1)$$

where EDB_1 is

$$\Phi(\mathit{eep}(\{r_1\}, EDB_0) \cup EDB_0)$$

Then, $\Phi(\mathit{eep}(\{r_1\}, EDB_0)$ denotes:

[8] Stratification is mentioned here only for the purposes of clarity. It does not affect the results in Section 3, which hold for the union of all strata.

{researcher_contact_details(fiona, manchester, 270, www_cmmc_nhs_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(james, manchester, 275, www_cs_man_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(lewis, london, 213, www_ic_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(lynda, cambridge, 221, www_ebi_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(peter, edinburgh, 982, www_ed_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(sally, cambridge, 221, www_ebi_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
researcher_contact_details(zoe, manchester, 278, www_biomed_man_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$}

which are the consequences inferred in the first application of eep, just as if conjunctive_consequences were used. Then, in turn,

$$\Phi(eep(\{r_2\}, \Phi(eep(\{r_1\}, EDB_0) \cup EDB_0)$$

denotes:
{project_contact_details(diabetes_control, fiona, manchester, 275, www_cmmc_nhs_uk)$\leftarrow\!\!\mid 1.0 \vdash$,
project_contact_details(sars_epidemic, lynda, cambridge, 221, www_ebi_ac_uk)$\leftarrow\!\!\mid 1.0 \vdash$}

Hence, the CID inference rule denotes the union of both sets of p-facts with those originally in EDB_0. ◁

Recursive CIDs Like hierarchical CIDs, a CID $\langle IDB, EDB\rangle$ in which the dependency graph of *IDB* may contain cycles also requires multiple iterations of extended elementary production applications for its deductive consequences to be computed but, unlike hierarchical CIDs, the bound on the number of iterations needed cannot be determined statically.

This kind of iteration can only be captured by a fixpoint construct, e.g., as described in Section 3.3.3. Thus, the set of all deductively inferrable consequences of a recursive CID $\langle IDB, EDB\rangle$ coincides with the set of p-clauses denoted by the following expression:

recursive_consequences(EDB,IDB) $\stackrel{def}{=}$
fixpoint(λX*.(eep(X,IDB)),EDB)*

where the current consequences are initialised with the extensional database *EDB*, and, at each step, *eep* is applied and the results accumulated until the fixpoint is reached.

Such a construct is needed in the presence of recursive definitions. For example, if a new relation were to be defined that structured the terms in the relation *expertise*/1 as a taxonomy, modelling such facts as that genomics is an specialisation of genetics, then recursive rules would be required to in order to define all the relationships between expertise terms. If so, then the fixpoint construct above would be required in order to compute all the deductive consequences of such a taxonomy.

The construct above guarantees that the consequences of recursive rules can be computed at the calculus level. However, the translation of such construct into an algebraic execution plan requires the monoid algebra to extended with a fixpoint operator (as done, e.g., in [BFP$^+$95, SP00]). Moreover, propagating all the implications of such algebraic extension through the translation to monoid algebra requires more work than is described in Section 5. Therefore, in this dissertation, CID systems with recursive p-rules are only described at the calculus level without accounting for them at the algebraic level too.

4.1.2 Query Answering

This section describes how a query over a CID system is expressed and how its answer set is declaratively specified as a CID task.

Given a CID system $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D}\rangle$, an *acceptable query* over $\mathcal{D}$ is specified by one p-rule $q \leftarrow l_1, l_2, \ldots, l_n$ in $\mathcal{L}_\mathcal{D}$ such that the relation names of all literals $l_1, l_2, \ldots, l_n$ occur in $\mathcal{D}$, and q is a relation name not occurring in $\mathcal{D}$.

The set containing all the answers of an acceptable query $q \leftarrow l_1, l_2, \ldots, l_n$ coincides with the set of p-clauses denoted by the following expression.

query_answers(($q \leftarrow l_1, l_2, \ldots, l_n$*),*$\mathcal{D}$*)* $\stackrel{def}{=}$
Φ*(eep({*$q\leftarrow\!\!\mid \top \vdash l_1, l_2, \ldots, l_n$*}, hierarchical_consequences(*$\mathcal{D}$*)))*

where $q \leftarrow l_1, l_2, \ldots, l_n$ is implicitly annotated with the maximum validity assessment $\top$ to give rise to a corresponding p-rule. This annotation has the effect of making any combination of validity assessments neutral to the presence of the relation name q, i.e., there is no interference on the validity assessment of the consequences (i.e., the answers to the query) arising from the fact that q has been expressed as a p-rule, which is for evaluation purposes only. Then, the deductive inference rule *eep* is applied to an intensional database containing only $q \leftarrow\!\dashv \top \vdash\!\!- l_1, l_2, \ldots, l_n$ and an extensional database that contains the set of hierarchical consequences[9] of the CID $\mathcal{D}$.

The following example illustrates a query answering task.

Example 4.3. Recall the CID $\mathcal{D}_0$ in Appendix A. Let the following be a query, say Q, that characterises which researchers are allocated to which projects.

allocation(Project,Description,Researcher,Position)←
  researcher(Researcher,_,_,_,_,_,_),
  project(Project,Description,_,_),
  participation(Researcher,Project,Position).

Assuming that hierarchical_consequences($\mathcal{D}_0$) yields all deductive consequences of $\mathcal{D}_0$, then query_answers(Q,$\mathcal{D}_0$) denotes:

{allocation(p1, sars_epidemic, fiona, consultant)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p1, sars_epidemic, lynda, project_leader)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p1, sars_epidemic, peter, research_assistant)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p1, sars_epidemic, zoe, research_fellow)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p2, diabetes_control, fiona, consultant)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p2, diabetes_control, lewis, medical_doctor)$\leftarrow\!\dashv 1.0 \vdash\!\!-$,
allocation(p2, diabetes_control, sally, research_assistant)$\leftarrow\!\dashv 1.0 \vdash\!\!-$}

i.e., a list of projects and their participating members, resulting from a single application of eep to the deductive consequences of $\mathcal{D}_0$. ◁

Logic-based languages can be seen as underpinning a database calculus in which query answering can be effectively expressed. In this section, in contrast, p-clauses have only been considered as symbolic objects whose manipulation is formally described in another formalism, viz., a monoid comprehension calculus. This move may seem unnecessary insofar as it does what could already be done using traditional deductive database techniques [CGT90, Das92, AHV95]. However, there is added value in this move because it opens the way for unifying the evaluation of query answering and knowledge discovery (which would be difficult, if not impossible, to handle comprehensively and in a principled manner using classical deductive database techniques). This is the underlying reason is for relying on a more abstract database language. Doing so partially removes some peculiarities of deductive query answering and places the approach reported here at a level in which a monoid calculus acts as a common foundation upon which knowledge discovery tasks can also be expressed, as described in the next section. This foundation is also useful in underpinning the development of a database engine that supports both query answering and knowledge discovery, as described in Section 5. So, the significance of the results in this section becomes clearer at the end of the next.

4.2 Expressing Inductive Knowledge Discovery

This section shows that inductive CID inference rules can capture knowledge discovery. Firstly, one inference rule is introduced simply to highlight and discuss the specific issues in expressing inductive inference. Then, other rules are introduced that express classification, clustering and association to illustrate that CID systems are expressive enough to capture uniformly and in a principled manner very popular knowledge discovery techniques that are algorithmically very dissimilar to one another.

In CIDs, inductive inference rules, like deductive ones, specify how to choose premisses, what conditions must be satisfied in order to yield legitimate consequences, how to construct such consequences, and which validity assessment[10] should annotate the ejected consequences. Inductive inference rules, unlike deductive ones, are neither truth-preserving nor deterministic, i.e., it is neither the case that a set of valid premisses necessarily entails valid consequences, in the classical-logic sense, nor is it the case that a set of premisses necessarily justifies a single consequence. The use of p-Datalog already allows for a principled treatment of the validity issues, but non-determinism is, so far, uncatered for.

[9] Query answering can be similarly defined for more expressive CIDs, if necessary.

[10] Potentially, which validity combinators too, although this possibility is not taken any advantage of in this dissertation.

For example, the two facts:

is_competent_in(zoe,genetics)←
is_competent_in(sally,genetics)←

support the inductive inference of many rules, in the sense that any of those rules would entail both facts if the data and knowledge in the CID in Appendix A are assumed to hold. For example, the following rules, among others, are supported:

is_competent_in(R,genetics)←researcher(R,_,good,_,A,assistant),research_unit(A,_,lab,_,cambridge,_,_)
is_competent_in(R,genetics)←researcher(R,_,_,_,A,assistant),research_unit(A,_,lab,_,cambridge,_,_)
is_competent_in(R,genetics)←researcher(R,_,_,_,A,_),research_unit(A,_,lab,_,_,_,_)
is_competent_in(R,genetics)←researcher(R,_,_,_,_,_)

which are ordered from less to more general[11].

This non-determinism is undesirable in practice (and, often, in principle). It is usually avoided by imposing a preference (i.e., a bias towards selecting one clause rather than other, e.g., the most concise, or the least general). Machine learning [Lan96, Mit97] and data mining algorithms [WF99] use combinations of search, language and pruning biases. Since CID systems perform an exhaustive, bottom-up, dataflow computation, a language bias is the only means available to specify such preferences and address non-determinism.

In order to explain the role of a language bias and the general strategy described in this section to express knowledge discovery techniques, consider the following inductive CID inference rule:

$$
\begin{array}{l}
\mathit{inductive\_consequences(L,O,D)} \stackrel{def}{=} \\
\uplus\{\ (h \leftarrowtail C \vdash l_1, l_2, \ldots, l_n)\ \ [\![\\
\quad (h \leftarrow l_1, l_2, \ldots, l_n) \quad \leftarrow L, \\
\quad (o \leftarrowtail C_0 \vdash) \quad \leftarrow O, \\
\quad (f_1 \leftarrowtail C_1 \vdash) \quad \leftarrow D, \\
\quad \ldots, \\
\quad (f_n \leftarrowtail C_n \vdash) \quad \leftarrow D, \\
\quad \theta \equiv mgu([h, l_1, l_2, \ldots, l_n], [o, f_1, f_2, \ldots, f_n]), \\
\quad \theta \neq \ \triangledown, \\
\quad C \ \equiv \ f^{\leftarrow}(C_0, f^{\wedge}(C_1, C_2, \ldots, C_n)) \\
\}
\end{array}
$$

The CID inference rule above is inspired on extended elementary production, and hence is structurally similar to it (cf. Section 3.3.1), but yields p-rules instead of p-facts. Premisses are chosen from a set O of examples and a set D of background data items. The consequence to be ejected is meant to be a rule derived from the generalisation of the premisses. For example, let O, D and L be the following three p-clause sets:

{is_competent_in(zoe,genetics)←1.0⊢,
is_competent_in(sally,genetics)←0.8⊢}

{researcher(zoe,4,good,20,a1,assistant) ←1.0⊢,
researcher(sally,0,good,24,a2,assistant) ←1.0⊢,
research_unit(a1,genomics,lab,ebi,cambridge,221,www_ebi_ac_uk) ←1.0⊢,
research_unit(a2,macromolecule,lab,sanger,cambridge,223,www_sanger_ac_uk) ←1.0⊢}

{is_competent_in(R,genetics)←
researcher(R,_,good,_,A,assistant),research_unit(A,_,lab,_,cambridge,_,_),
is_competent_in(R,genetics)←
researcher(R,_,good,_,A,researcher),research_unit(A,_,lab,_,cambridge,_,_) }

Thus, the language bias L is favouring more specialised rules, among the possibilities enumerated at the start of this section. Using the operational semantics of monoid comprehensions (see Section 2.3), one can verify that the first rule in L will match the requirements of *inductive_consequences* twice, i.e., once for $\theta = \{\mathsf{R/zoe}, \mathsf{A/a1}\}$ and then again for $\theta = \{\mathsf{R/sally}, \mathsf{A/a2}\}$. Therefore, this rule is ejected twice,

[11] A rule r is more general than a rule r' if variables in r can, upon being instantiated, match constant terms in r' (if any) and if the literals in the body of r occur in the body of r' [NCdW96].

once with validity assessment $C = 0.7 = min(0.7, min(\{1.0, 1.0\}))$ and then again with $C = 0.8 = min(0.8, min(\{1.0, 1.0\}))$, assuming, as elsewhere in this dissertation, that $f^{\leftarrow} = f^{\wedge} = min$. The second rule in L finds no matching examples in O, therefore it is not supported as an inductive consequence. The rest of comprehension validates and assesses the legitimacy of each generalisation enumerated from L. The same monoid comprehension Φ described in Section 3.3.2 can be used to enforce uniqueness constraint on the preliminary result above and, assuming $f^{\vee} = max$ consolidate the outcome as:

{is_competent_in(R,genetics)$\dashv\!\!\mid 0.8 \vdash\!\!\mid$
researcher(R,_,good,_,A,assistant),research_unit(A,_,lab,_,cambridge,_,_)}

The CID inference rule above introduced an extra comprehension generator that scans the language bias L for only the preferred generalisation. This strategy replaces (at least, at the calculus level) the need for any of the several generalisation techniques, e.g., anti-unification [IA93], which imply some form of non-determinism, and usually lead to an approach based on search algorithms. The extra generator enumerates the candidate consequences to be checked against the premisses, and hence controls non-determinism and constrains the form of the induced consequences. In the example above, other rules that are more generic than the two in L were not even considered. Thus, the language bias denotes an explicit preference, viz., only clauses in L are candidate consequence, and the possible existence of supporting premisses for the clause not generated is, therefore, implicitly deemed to be irrelevant.

The effort involved in defining an appropriate language bias is not negligible, but it is inherent in every inductive learning or discovery mechanism [Mit97, Mit99]. This effort is no greater in CID systems than in any other approach. An approach based on a language bias alone does not, in itself, make the specification of inductive tasks more cumbersome. If anything, it unifies concerns that would otherwise be split into modelling coherent compositions of language, search and pruning biases. However, in more complex cases, the use of language bias alone, may make the specification of the bias more complex, as its role becomes more wide-ranging and, in some sense, subtle interactions are likely to have be taken into account in specifying it.

As in the example above, a *languages bias* is a finite subset of the language $\mathcal{L}_{\mathcal{D}}$ of a CID $\mathcal{D}$. As a set of clauses, a language bias can be defined extensionally, by enumerating all its clauses, or intensionally, e.g., as a grammar with an associated generating mechanism. Other linguistic formalisms are also admissible, e.g., the DLAB language [DdR96], or a definite-clause grammar (DCG) [PW80] for Datalog (as shown in Section C.6). Both example formalisms are interesting insofar as they (among others) bring associated mechanisms with them that can automate and control the generation of the clauses in a language bias. This is instrumental in the context of a concrete implementation of CID systems as explained in Section 5.4.7. For the moment, for the sake of simplicity, assume that a language bias is specified given extensionally so that generators can be defined to scan it.

If all clauses in a language bias need to be exhaustively verified against premisses, then the cardinality of the language bias can lead to great inefficiency. However, the issue of improving efficiency is deferred to Section 5, where it can be discussed in the context of algebraic optimisation and compared to search-and-prune alternatives.

The next section describes more elaborate inductive CID inference rules that express classification, clustering and association.

4.2.1 Expressing Classification

Classification rules [BFOS84] predict, for any unseen instance, which one, among possibly many classes, that instance belongs to. A CID inference rule can capture the induction of classification rules in the context of a CID. This inference rule takes positive and negative examples O^+ and O^-, respectively, as premisses. Instances in O^+ are known to have been correctly assigned to a class, and instances in O^- are known not to have.

Both examples and classification rules can be represented in p-Datalog. Positive examples can be represented as a set of ground facts (as usual in inductive logic programming [Wro96]) annotated with a validity assessment that either is asserted as a result of appropriate domain knowledge or else is computed by a CID inference rule. For example, in Section 1.1, a classification model that can predict competences is needed to procure a candidate replacements for an expert. In this case, the positive examples O^+ might be represented as follows:

is_competent_in(fiona, epidemiology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(fiona, histology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(fiona, immunology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lewis, biochemistry)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lynda, molecular_biology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lynda, bioinformatics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(peter, cytology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(peter, histology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(sally, bioinformatics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(sally, genetics)$\leftarrow\!\!\dashv 1.0 \vdash$

Since negative facts are not representable in p-Datalog, negative examples are also represented as a separate set O^- of p-facts that is disjoint from O^+. Thus, the negative examples O^- might be represented as follows:

is_competent_in(fiona, genetics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(fiona, biochemistry)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(fiona, bioinformatics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lewis, immunology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lynda, genetics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(lynda, histology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(peter, biochemistry)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(peter, bioinformatics)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(sally, cytology)$\leftarrow\!\!\dashv 1.0 \vdash$
is_competent_in(sally, immunology)$\leftarrow\!\!\dashv 1.0 \vdash$

A classification rule can be represented as a p-rule of the form[12]

$$\mathit{is\_a(E,s)} \leftarrow\!\!\dashv C \vdash l_1, l_2, \ldots, l_n$$

and interpreted as follows: any instance of E is included in class s, if the conditions $l_1, l_2, \ldots, l_n$ are satisfied, and this implication is assumed to have a validity C. For example, let r[13] be:

is_competent_in(Researcher,bioinformatics)$\leftarrow$
  researcher(Researcher,_,good,_,_,_),
  writes(Researcher,Paper),
  paper(Paper,_,Venue),
  venue(Venue,bioinformatics_journal,_),
  refers_to(Paper,bioinformatics)

The rule r can be interpreted as follows: a researcher is competent in bioinformatics if he/she has good indicators of professional esteem and has authored a paper related to bioinformatics in the Bioinformatics journal. This classification rule can be one of several in a model that predicts the expertise of researchers, and the conditions denoted by the body literals justify the stated prediction. In this example, constant symbols in the extent of the *expertise*/1 relation represent the labels that define the possible target classes. A Datalog-based representation for classification rules is more expressive than the usual attribute-value representation. Moreover, this representation is versatile, since a single rule can embody knowledge that is generic enough to predict for many target classes, e.g., the rule r' below:

is_competent_in(Researcher,Expertise)$\leftarrow$
  researcher(Researcher,_,good,_,_,_),
  writes(Researcher,Paper),
  refers_to(Paper,Expertise)

The level of generalisation/specialisation expected from a model is implicitly determines by a language bias L. Thus, if a p-rule is to be induced as a classification rule from the examples above, it must be contained in the language bias L. L might specify that that the desired rules must have *is_competent_in*/2 in the head and a choice of *researcher*/6, *paper*/3, *writes*/2, *venue*/3, *is_pc_member_of*/2, *refers_to*/2 in the body. If so, L would constrain the inferential process into defining competence in terms of the academic activities of the candidate researchers. L might also specify where, and which, constants and variables may occur in a p-rule, e.g., that the second attribute of *is_competent_in*/2 is a variable. If so, L would contain r', but not r. Again, a language bias is shown to control the non-determinism associated

[12] This representation is only a notational convention for illustration purposes, other alternatives are equally admissible.
[13] This is an equivalent notational convention to is_a(Researcher,competent_in_bioinformatics).

with inductive inference. A practical and concise definition for L will be given in Example 5.13 in the context of a prototype CID engine.

A classification rule is only ejected if it can be used to deductively derive sufficiently many positive examples and sufficiently few negative ones, with respect to the given background data and knowledge in a CID. This type of covering approach [Mit82, Mic83] is also followed here.

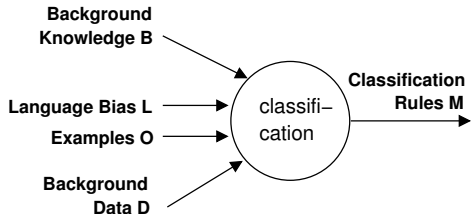
A score function is needed to quantify 'sufficiently many' and 'sufficiently few'. A classic score function for classification rules is *misclassification rate* [HMS01], which can be computed in terms of the number of true positives and true negatives in the *confusion matrix* [WF99] that can be computed for each classification model. Again, assume that the extent of a classification rule r, with respect to the background CID $\mathcal{D}$, is the set of deductively inferred ground facts X. Let $O^+ \cap X$ and $O^- \setminus X$ denote the set of true positive and true negatives derived by querying $\mathcal{D}$ with r. Then, the certainty value the rule can computed by the following function:

$$g_S(X, O^+, O^-) = \frac{|O^+ \cap X| + |O^- \setminus X|}{|O^+ \cup O^-|} \tag{2}$$

The function g_S is trivially monotone and continuous with respect to the validity assessments asserted with O^+ and O^-, insofar as its result is influenced by different validity assessments asserted to O^+ and O^-. Therefore, g_S qualifies as a score function that complies with the formal development described in Section 3.

Then, given a language bias L, the set of all classification rules inferrable from a CID $\mathcal{D} = \langle B, D\rangle$ (taken as background knowledge) that cover at least T^+ positive examples and at least T^- negative ones, coincides with the set denoted by the following expression:

$$\begin{array}{l} \mathit{classification\_rules}(L, O^+, O^-, \langle B, D\rangle, T^+, T^-) \stackrel{def}{=} \\ \cup\{\ h \mapsto g_S(X, O^+, O^-) \vdash l_1, l_2, \ldots, l_n \ \ [\![\\ \quad (h \leftarrow l_1, l_2, \ldots, l_n) \ \ \leftarrow L, \\ \quad X \equiv \mathit{query\_answers}(h \leftarrow l_1, l_2, \ldots, l_n, \langle B, D\rangle), \\ \quad |O^+ \ \cap \ X| \ \geq \ T^+, \\ \quad |O^- \setminus \ X| \ \geq \ T^-\} \end{array}$$



where *query_answers* is the monoid comprehension defined in Section 4.1.2 and both intersection and set cardinality are well-defined over monoid comprehensions, as shown in Section 2.3. Note that, first, a candidate generalisation rule is chosen from the language bias L. Then, using *query_answers*, against the background knowledge $\mathcal{D}$, the set X of inferrable consequences is obtained from $\mathcal{D}$ augmented with the candidate rule. A rule is considered to have matched the requirements for ejection if the cardinality of both the set $|O^+ \cap X|$ of true positives and $|O^- \setminus X|$ of true negatives are above the stipulated thresholds T^+ and T^-, respectively. If a classification rule fulfills the minimum coverage constraints, it is *ejected* with the certainty value computed as described in (2). Finally, note that *classification_rules* are fully specified in the monoid comprehension calculus.

Let $\mathcal{D}_0$ be the CID in Appendix A, let O^+ and O^- and L be as given above, and let the coverage thresholds $T^+ = T^- = 1$. Note that r is contained in L. Then, in the evaluation of *classification_rules*, r is eventually selected and its extent is computed as prescribed by *query_answers*. In this case, X contains:

is_competent_in(sally, genetics)$\mapsto 1.0 \vdash$

The cardinality of its intersection with O^+ is 1, while the cardinality of its set difference from O^- is 10 (since the intersection is empty). Hence, both coverage thresholds T^+ and T^- are satisfied and therefore r can be ejected as legitimate classification rules with validity assessment of 0.55 (i.e., the confusion matrix is $(1 + 10)/(10 + 10) = 11/20 = 0.55$ considering the cardinality 10 of both O^+ and O^-). In summary, each rule annotated generated from L is considered in turn and, if there is justification, it is annotated with the proper validity assessment and ejected.

The classification inference rule described above can subsume (or is equivalent to) those output by knowledge discovery tasks that are usually performed by inductive logic programming systems [NCdW97], or by propositional, rule-based classifiers [CN88]. The next section describes CID inference rules for clustering.

4.2.2 Expressing Clustering

Hierarchical conceptual clustering [Fis87] is one the most commonly used clustering techniques for non-numeric data [JMF99]. This technique starts by mapping all input instances into initial clusters. Then, it incrementally builds a new cluster by merging the two most similar clusters, provided that they have not been merged before, until there is no more clusters to merge, and a single one emerges as encompassing all instances. Hierarchical clusters can represent a conceptual taxonomy [Sow99] if nodes are interpreted as taxa, and edges as inclusion relationships between taxa.

A CID inference rule can capture the induction of clustering models, e.g., taxonomies, in the context of a CID. The clustering inference rule takes a set of instances O as premisses. Clustering instances are also represented as p-facts. For example, in Section 1.1, a clustering model was needed in order to contrast the profile of an expert and of his/hers candidate replacement. In that case, the instances O might be as follows:

{instance(peter, 3, regular, a6, research_assistant, p1, cytology)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$,
instance(fiona, 1, good, a5, consultant, p1, epidemiology)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$,
instance(lynda, 6, excellent, a7, head_of_lab, p1, molecular_biology)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$,
instance(zoe, 4, good, a1, research_assistant, p1, bioinformatics)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$,
instance(sally, 0, good, a2, research_assistant, p2, bioinformatics)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$,
instance(lewis, 1, good, a4, medical_doctor, p2, biochemistry)$\leftarrow\!\!\!\mid 1.0 \mid\!\!\!\rightarrow$}

Note that instances have no associated cluster yet: the first attribute value is assumed above to identify the instance itself, while all other attribute values denote features. These features are compared to gauge the similarity of clustering instances.

A taxon can be represented by a p-rule that defines the extent of a relation with schema *taxon(taxon_id, left_subtree, right_subtree)*, where the first attribute identifies the taxon, the second and third identify the subtaxa that were merged to build the current taxon. For example, that instances zoe and sally are included in a taxon t1 because both have in common good recognition, a research assistant position, and expertise in bioinformatics, despite different affiliations, and project allocations. This taxon might be represented as:

taxon(t1,zoe,sally)$\leftarrow\!\!\!\mid 0.8 \mid\!\!\!\rightarrow$
  researcher(Researcher,Number_of_Participations,good,Wage,Affiliation,research_assistant),
  participation(Researcher,Project,Allocation),
  is_competent_in(Researcher,bioinformatics)

It is formed under conditions, denoted by the body literals, that point to the data and knowledge item in a CID.

Observe that instances are represented as p-facts while taxa are represented as p-rules. To simplify the formulation of inference rules, instances are mapped to initial clusters of the form *taxon(s,nil,nil)*, where the first attribute identifies the instance, and the constant *nil* denotes that there are no subtaxa of this taxon, i.e., such a p-clause denotes a leaf in the taxonomy. Thus, the extent of *taxon*/3 represents a taxonomy as a binary tree, where edges denote the containment of taxa. The induced set of *taxon*/3 clauses characterises a conceptual model that clusters instances according to the degree of similarity between them.

A language bias L is necessary in order to relate the features that occur in each instance in O to the representation chosen for taxa. For example, let $L = \{r\}$ and let r be:

features(Number_of_Participations,Recognition,Affiliation,Position,Project,Expertise)$\leftarrow$
  researcher(Researcher,Number_of_Participations,Recognition,_,Affiliation,Position),
  participation(Researcher,Project,_),
  is_competent_in(Researcher,Expertise).

This rule projects features found in the relations of the CID system in Appendix A onto instances in O.

The following monoid expression can use a rule such as r to convert instances into leaf taxa.

$$
\begin{array}{l}
\mathit{leaf\_taxa}(\mathsf{features}(F_1, F_2, \ldots, F_n) \leftarrow l_1, l_2, \ldots, l_n, O) \stackrel{def}{=} \\
\quad \cup\{\mathit{substitute}((\mathsf{taxon(i,nil,nil)} \leftarrow\!\!\!\mid C \mid\!\!\!\rightarrow l_1, l_2, \ldots, l_n), \theta) \;\; [\![\\
\qquad (\mathsf{instance}(i, F'_1, F'_2, \ldots, F'_n) \leftarrow\!\!\!\mid C \mid\!\!\!\rightarrow) \leftarrow O \\
\qquad \theta \equiv mgu([\mathsf{features}(F'_1, F'_2, \ldots, F'_n)], [\mathsf{features}(F_1, F_2, \ldots, F_n)] \\
\quad \}
\end{array}
$$

The evaluation of *leaf_taxa({r},O)* yields for the fourth instance in O the following p-rule:

```
taxon(zoe,nil,nil)↤1.0⊢
  researcher(zoe,4,good,_,a1,research_assistant),
  participation(zoe,p1,_),
  is_competent_in(zoe,bioinformatics).
```

In summary, the language bias allows for background data and knowledge that is stored in or inferred from CID to be considered during the clustering process.

Another language bias N is needed for clustering so as to provide possible cluster names (examples for classification already convey all class labels, but not instances for clustering). Again, a set of facts N can define new names for non-leaf taxa. For example:

```
{taxon_name(t1)←,
taxon_name(t2)←,
taxon_name(t3)←,
taxon_name(t4)←,
taxon_name(t5)}
```

Score functions used in clustering are often based on distance [HMS01]. These functions assume clustering instances to be points in a multidimensional space. Thus, the distance between two clusters is captured by a function d, e.g., Euclidean distance. However, to transform a distance function into a function that computes validity assessments one needs the intuition that the greater the distance between two taxa, the less valid is their merge to yield a new taxon. Thus, the validity assessment can be, e.g., the inverse of the distance measure d:

$$g_K(d) \quad = \quad \frac{1}{1+d} \tag{3}$$

Like Function 2, g_K above is trivially monotone and continuous with respect to the validity assessments asserted in O. Therefore, g_K also qualifies as a score function that complies with the formal development described in Section 3.

In contrast with the classification case, the clustering inference rule is defined as one monoid expression that merges of two child taxa to yield their parent, and another functional-composition monoid expression that captures the iteration of the first expression until a complete taxonomy model is constructed. These two monoid expressions are described next.

Since a taxon cannot be merged more than once, a taxon will only have one parent. The following expression can be used to check whether a taxon t has already been merged within the set of taxa T:

$$\begin{array}{l} \mathit{not\_merged}(\mathsf{t},T) \stackrel{def}{=} \\ \quad \wedge\{\mathsf{taxon}(\_,\mathsf{t},\_) \neq A \wedge \mathsf{taxon}(\_,\_,\mathsf{t}) \neq A \;[\!]\; A \leftarrow T\} \end{array}$$

This expression evaluates to true if t does not occur as the left or right subtree of any other taxon in the set of known taxa T. If so, the taxon t is a candidate to be merged and hence participate in the formation of a new parent taxon.

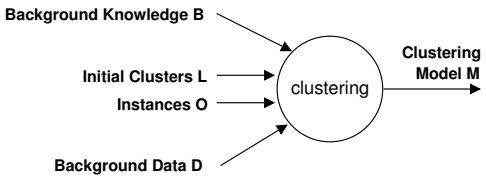
Then, given language biases N and L, a CID $\mathcal{D} = \langle B, D\rangle$, and a distance measure d, the set of parent taxa created at each step coincides with the result denoted by the following expression:

$$\begin{array}{l} \mathit{taxon}((\mathsf{taxon\_name}(\mathsf{t}),(\mathsf{features}(\mathsf{F}_1,\mathsf{F}_2,\ldots,\mathsf{F}_n)\leftarrow \mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n),T,\mathcal{D},d) \stackrel{def}{=} \\ \mathit{max}\{\mathit{substitute}((\mathsf{taxon}(t,s^1,s^2)\leftarrow g_K(d(K^1,K^2))\vdash \mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n),\theta^1\cap\theta^2) \;[\!] \\ \quad (\mathsf{taxon}(\mathsf{s}^1,\_,\_)\leftarrow C\vdash \mathsf{l}^1_1,\mathsf{l}^1_2,\ldots,\mathsf{l}^1_n) \leftarrow T, \\ \quad (\mathsf{taxon}(\mathsf{s}^2,\_,\_)\leftarrow C\vdash \mathsf{l}^2_1,\mathsf{l}^2_2,\ldots,\mathsf{l}^2_n) \leftarrow T, \\ \quad \mathsf{s}^1 \neq \mathsf{s}^2, \\ \quad \mathit{not\_merged}(\mathsf{s}^1,T), \\ \quad \mathit{not\_merged}(\mathsf{s}^2,T), \\ \quad \theta^1 \equiv \mathit{mgu}([\mathsf{l}^1_1,\mathsf{l}^1_2,\ldots,\mathsf{l}^1_n],[\mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n]) \\ \quad \theta^2 \equiv \mathit{mgu}([\mathsf{l}^2_1,\mathsf{l}^2_2,\ldots,\mathsf{l}^2_n],[\mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n]) \\ \quad K^1 \equiv \mathit{query\_answers}(\mathit{substitute}(\mathsf{features}(\mathsf{F}_1,\mathsf{F}_2,\ldots,\mathsf{F}_n)\leftarrow \mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n,\theta^1),\mathcal{D}), \\ \quad K^2 \equiv \mathit{query\_answers}(\mathit{substitute}(\mathsf{features}(\mathsf{F}_1,\mathsf{F}_2,\ldots,\mathsf{F}_n)\leftarrow \mathsf{l}_1,\mathsf{l}_2,\ldots,\mathsf{l}_n,\theta^2),\mathcal{D}) \\ \} \end{array}$$

This expression takes two taxa in T, identified by s^1 and s^2. If neither has been merged in T yet, it derives a new taxon t that contains s^1 and s^2. The common features of the new taxon t are defined by intersection of the two unifiers θ^1 and θ^2 that retain the features occurring in s^1 and s^2 respectively. The unifiers θ^1 and θ^2 are computed by the usual function *mgu* and their intersection is applied in the head of the comprehension using *substitute*. The last two lines are only necessary assuming that the distance measure d is, e.g., Euclidean distance, which is usually defined for tuples only. If d was defined for first-order representations, e.g., as in [KWH01], then there would be no need to join the relations $l_1, l_2, \ldots, l_n$ and project the features into a single relation. The evaluation of this expression yields all taxa that can be merged and selects the one with the largest validity annotation. It captures the semantics of the internal loop of the generic algorithm for agglomerative clustering [HMS01], replacing the removal of the merged clusters by the *not_merged* filter.

The taxonomy is induced by composing the CID inference rule above a sufficient number of times. This number is one less than the cardinality of the input instances O, i.e., $|O| - 1$. To capture this bounded iteration, a comprehension over a functional-composition monoid (similar to the one presented in Section 3.3.3) is defined. The iteration is controlled providing a list of $|O| - 1$ cluster names N to identify the new taxa. Thus, taxa are accumulated to constitute the taxonomy through the following expression.

$$
\begin{array}{l}
\mathit{taxonomy(N,L,O,\langle B,D\rangle,d)} \stackrel{def}{=} \\
\circ\{\lambda I.\mathit{taxon}(T,L,I,\langle B,D\rangle,d) \cup I \; [\![\\
\quad T \leftarrow N \\
\}(\mathit{leaf\_taxa(L,O)})
\end{array}
$$



Note that the expression *leaf_taxa* is used to generate the p-rule representation for the clustering instances in O.

Finally, let $\mathcal{D}_0$ be the CID in Appendix A, and let N, L, O and d be as described above. Then, the evaluation of *taxonomy(N,L,O,*$\mathcal{D}_0$*,d)* infers clauses that include:

```
taxon(t1,zoe,sally)↤0.8⊢
  researcher(Researcher,Number_of_Participations,good,Wage,Affiliation,research_assistant),
  participation(Researcher,Project,Allocation),
  is_competent_in(Researcher,bioinformatics)
```

The complete taxonomy is depicted in Figure 3.

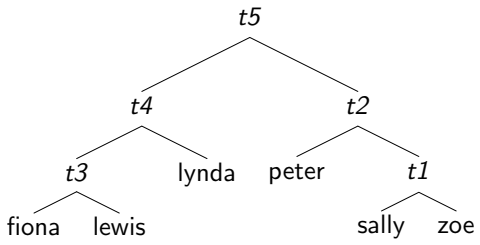


Figure 3: Taxonomy of Substitute Researchers

Both classification and clustering have been shown to be expressible as CID inference rules. Therefore, CID systems have thereby been shown to support both supervised and unsupervised knowledge discovery. CID systems can also support association analysis as shown in the next section.

4.2.3 Expressing Association

Association analysis finds co-occurrence patterns in data items. Association analysis can also be expressed as a CID inference rule. While there have been previous attempts to capture the original notion of association [AIS93, AS94] to a clausal setting [DT99, DT01, GdB02], these proposals are not satisfactory for the aims of this thesis for one main reason. A so-called relational association rule

[DT99, DT01, GdB02] of the form $h_1, h_2, \ldots, h_m \rightsquigarrow l_1, l_2, \ldots, l_n$ lacks a widely-accepted formal-logic interpretation[14] that might allow for the symbolic manipulation and automatic deployment of association models.

In contrast, an association rule inferred from a CID system is represented in p-Datalog. This choice implies an inductive leap from co-occurrence to implication. More precisely, in CID systems, an association rule takes the form:

$$h(Y_1, Y_2, \ldots, Y_m, a_1, a_2, \ldots, a_t) \dashv\vdash C \vdash l_1, l_2, \ldots, l_n$$

where Y_i denotes a variable, and a_j denotes a constant. Such an association rule is interpreted as stating that, with validity C, $a_1, a_2, \ldots, a_t$ co-occur frequently in instances identified by $Y_1, Y_2, \ldots, Y_m$ whenever the conditions denoted by $l_1, l_2, \ldots, l_n$ hold. In basket analysis terms, this can be rephrased as stating that the items $a_1, a_2, \ldots, a_t$ are frequently purchased together in baskets identified by $Y_1, Y_2, \ldots, Y_m$, if the conditions $l_1, l_2, \ldots, l_n$ hold. These conditions may refer to the purchase of some other items (as classically they do), or, more generally, to store, customer, promotions details, etc., taking full advantage therefore of the expressiveness of a Datalog representation to connect multiple relations.

For example, suppose that one wants to predict which positions in a project are more likely to be filled with researchers that reside in the same locality, or city of the project manager. Thus, let $\mathcal{D}_0$ be CID in Appendix A, and let O be a set of p-fact which associates project manager, manager address, project members, and member addresses, such as:

{addresses(fiona, manchester, fiona, manchester)$\dashv\vdash 1.0\vdash$,
addresses(fiona, manchester, lewis, london)$\dashv\vdash 1.0\vdash$,
addresses(fiona, manchester, sally, cambridge)$\dashv\vdash 1.0\vdash$,
addresses(lynda, manchester, fiona, manchester)$\dashv\vdash 1.0\vdash$,
addresses(lynda, manchester, lynda, manchester)$\dashv\vdash 1.0\vdash$,
addresses(lynda, manchester, peter, manchester)$\dashv\vdash 1.0\vdash$,
addresses(lynda, manchester, zoe, cambridge)$\dashv\vdash 1.0\vdash$}

Let L define a language bias for association that contains only clauses with relation name *addresses*/4 in the head literal, and with relation names *researcher*/7, *participation*/3 and *project*/4 in the body literals. The language bias L might contain clauses such as:

addresses(M,manchester,R,manchester)←researcher(R,_,_,_,_,research_assistant),
participation(R,P,_),project(P,_,M,_)
addresses(M,manchester,R,manchester)←researcher(R,_,_,_,_,consultant),
participation(R,P,_),project(P,_,M,_)

An association rule represents an attempt at generalising a conjunction $h(\ldots), l_1, l_2, \ldots, l_n$ to the implication $h(\ldots) \dashv\vdash C \vdash l_1, l_2, \ldots, l_n$. This generalisation leap would be logically sound if the conjunction of all literals always holds (it is false only if $l_1, l_2, \ldots, l_n$ holds but $h(\ldots)$ does not). Often this is not the case, but the generalisation can still be inductively justified if the conjunction above holds significantly often. The cardinality of the extent of the query $q(Y_1, Y_2, \ldots, Y_m) \leftarrow l_1, l_2, \ldots, l_n, h(Y_1, Y_2, \ldots, Y_m, a_1, a_2, \ldots, a_t)$ gives an absolute measure of significance for the conjunction. This measure taken in proportion to the cardinality of the query $q(Y_1, Y_2, \ldots, Y_m) \leftarrow l_1, l_2, \ldots, l_n$ gives a relative measure of significance for the conjunction, called *confidence*. Observe that, since it only contains an additional literal, the first query is always contained[15] in the second. A confidence measure C provides both a validity assessment for an association rule and an inductive justification for generalising the conjunction into an implication. This generalisation leap has the advantage of enabling further use of the discovered patterns, e.g., via deductive querying. In short, this section describes an extension of classical association rules which is inspired by, but not related to, previous proposals in the clausal setting.

Each of the rules in the language bias L, as illustrated above, gives rise to a pair of queries of the form: q(M,R)←researcher(...),participation(...),project(...) and q(M,R)←researcher(...),participation(...),project(...),addresses(...
Note that M and R are the variables occurring in *addresses*. The extent of those queries, say X_b and X_h, respectively, provide a measure of confidence [AS94] for association rules. Thus, the validity assessment of association rules is computed via the following function:

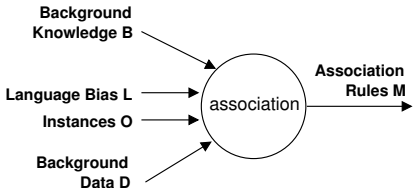
[14] Only the WARMR algorithm [DT99] provides one.
[15] See [AHV95] for more details about query containment.

$$g_A(X_h, X_b) = \frac{|X_h|}{|X_b|} \tag{4}$$

By an analogous argument to the one given for Equation 2, g_A also qualifies as a score function that complies with the formal development described in Section 3.

Let L be a language bias, let O be a set of instances where co-occurrence patterns are expected, let $\mathcal{D} = \langle B, D\rangle$ be a CID, and, finally, let T be a threshold on the support for the rules. Then, the set of all association rules inferrable from a CID $\mathcal{D}$ whose support with respect to $\mathcal{D}$ and O is greater than T coincides with the set denoted by the following expression:

$$\begin{array}{l} \mathit{association\_rules}(L,O,\langle B,D\rangle,T) \stackrel{def}{=} \\ \cup\{\, (h(Y,F) \mapsfrom g_A(X_h, X_h) \vdash l_1, l_2, \ldots, l_n) \;\; [\![\\ (h(Y,F) \leftarrow l_1, l_2, \ldots, l_n)) \quad \leftarrow L, \\ X_b \equiv \mathit{query\_answers}(q(Y) \leftarrow l_1, l_2, \ldots, l_n, \langle B, D\rangle), \\ X_h \equiv \mathit{query\_answers}(q(Y) \leftarrow l_1, l_2, \ldots, l_n, h(Y,F), \langle B, D\rangle), \\ |X_h| \geq T \\ \} \end{array}$$



What is being ascertained for an association rule is how frequent the query including the literal $h(Y,F)$ in the body is satisfied in proportion to the satisfaction of the query with that literal removed. This frequency imparts significance to the associations denoted in the derived rule. If $T > 0$, then X_h cannot be empty; in this case, $h(Y,F)$ must hold for some substitution of Y, and hence it can be trivially verified that the first and the second query logically imply the association rule.

For example, selecting the first candidate rule in L, it gives rise to the following queries:

q(M,R)←researcher(R,_,_,_,_,research_assistant),participation(R,P,_),project(P,_,M,_)
q(M,R)←researcher(R,_,_,_,_,research_assistant),participation(R,P,_),project(P,_,M,_),
addresses(M, manchester, R, manchester),

The extent of these queries can be determined by *query_answers*, resulting in X_b:

{q(fiona, sally)↤1.0⊢,
q(lynda, zoe)↤1.0⊢,
q(lynda, peter)↤1.0⊢}

and, respectively, in X_h:

{q(lynda, peter)↤1.0⊢}

Since the second query has a non-empty extent, then the rule above can be ejected as an association rule:

addresses(M, manchester, R, manchester)↤0.3⊢researcher(R,_,_,_,_,research_assistant),
participation(R,P,_),project(P,_,M,_)

The validity assessment is computed according to g_A to $1/3 = 0.33$, which is then rounded to 0.3 for a discrete validity domain ranging over deciles.

In summary, this section has shown the following:

1. popular data mining techniques can be characterised declaratively using the notion of a CID inference rule and making validity assessments correspond to the usual score functions [HMS01] adopted for them in the data mining literature;

2. declarative language biases can be used to specify preferences over knowledge models to the extent that it becomes possible to abstract over implementation details of the search strategies that enforce those preferences algorithmically;

3. p-Datalog can uniformly express discovered knowledge and, in a principled manner allow the fine-grained management of validity assessments.

Although issues relating to data selection, preparation and evaluation were overlooked in this section, they are an integral part of the knowledge discovery process [FPS96b]. Many data selection and preparation steps (e.g., feature construction, dimensionality reduction, aggregation and simple forms of discretisation [Pyl99, ZZY03]) can be specified using p-rules, and performed through deductive inference. Thus, data preparation can be expressed using monoid comprehensions that are nested, in turn, into the monoid comprehensions that denote knowledge discovery tasks. Although this strategy is hinted at throughout this section, a more detailed treatment of preparation and evaluation capabilities that are provided by CID systems would require a broader study, that is left to lie beyond the scope of this dissertation. The further pursuit of this study would demonstrate that CID systems can provide comprehensive and seamless support that CID systems to all stages of the knowledge discovery process.

5 Developing Combined Inference Database Systems

The purpose of this section is to demonstrate the feasibility of developing CID systems that implement the formalisation in Section 3, and hence are capable of performing tasks that compose CID inference rules such as those exemplified in Section 4. This section describes one development and deployment strategy, reporting on the experience acquired during the implementation of a proof-of-concept prototype. The strategy consists of choosing a user-interaction model and a functional architecture that are modelled on those of a classical database system. The reason for preferring such a strategy is to build, as much as possible, upon existing database technology, and in particular, upon the algorithms and optimisation techniques available for the monoid calculus and algebra. In short, the aim is to explore a possible road map for the engineering of practical CID systems.

5.1 A Usage Model

The first concern in the design of the prototype CID system was to define a usage model that enables users to interact with the system. The chosen usage model is inspired in standard database management system technology. It is depicted in the use case diagram in Figure 4. This usage model comprises three

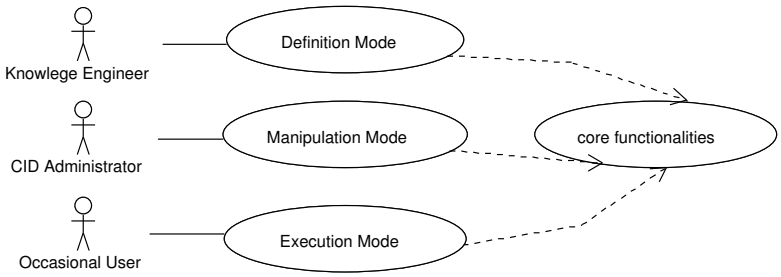


Figure 4: CID Usage Model

modes, each associated with a different class of stakeholders. The modes are described and exemplified by revisiting example in Section 1.1.

The definition mode enables knowledge engineers to define the capabilities of a CID system. More precisely, CID inference rules are developed and then incorporated, or *plugged*, into a particular CID system, so that they can be referred to in the composition of tasks later on.

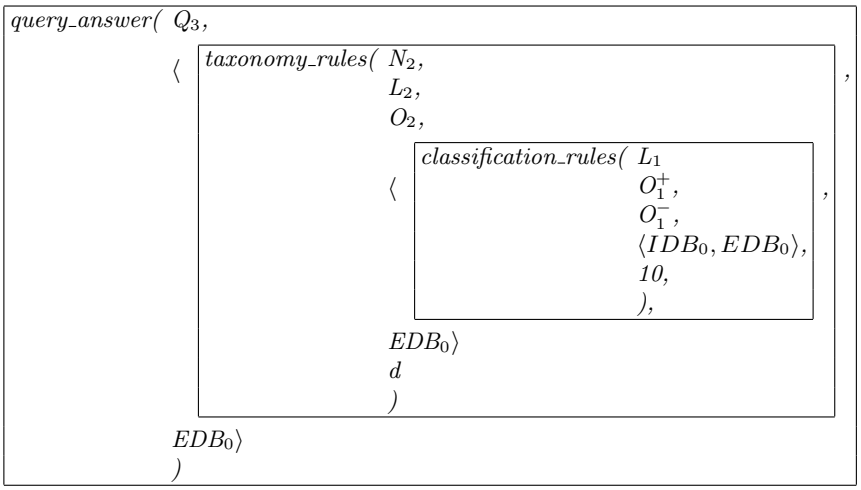
Example 5.1. In order to procure a candidate replacement for an expert, as presented in Section 1.1, a CID system should be capable of, at least, performing query answering, classification, and clustering. Hence, the deductive inference rules in Section 4.1.1 and the inductive ones in Sections 4.2.1 and 4.2.2, at least, should be part of the repertoire of such a CID system. In the definition mode, the knowledge engineer, having determined that the CID system should have those CID inference rules, plugs them, thereby defining the inferential capabilities to be used to address the problem. ◁

The manipulation mode enables database administrators to manage data and knowledge items stored in CID. Thus, activities such as insertion, update, deletion, load, back-up, restore, etc., of p-clauses are performed in this mode.

Example 5.2. Once the capabilities of the CID system in Example 5.1 are defined, the clauses defined in Appendix A can then be loaded to populate it. ◁

The execution mode is the main end-user mode. Users can define tasks that compose those CID inference rules that have been plugged, then submit such tasks for evaluation over the content of the database. The collection of p-clauses that results from task evaluation is returned to the end user. The end user can, in turn, decide whether to discard the results or assimilate them for further use.

Example 5.3. Taking the CID system defined in Example 5.1 and populated in Example 5.2, an end user can procure candidate replacements for an expert by submitting the CID task depicted in Figure 1, which is specified as the following a composition of CID inference rules:



Above, $\langle IDB_0, EDB_0\rangle$ is the CID in Appendix A. The parameters L_1, O_1^+ and O_1^- are the language bias, the positive and the negative examples in Section 4.2.1, respectively. The parameters N_2, L_2, O_2, and d are the cluster names, the language bias, the initial clusters, the inter-cluster distance in Section 4.2.2, respectively. Thus, all this variables are mapped to concrete collections of clauses. The query Q_3 retrieves candidate replacements for an expert based on a similarity model.

```
candidate_substitute(Candidate,Expert,Project)←
  researcher(Expert,_,_,_,_,_),
  participation(Expert,Project,_),
  researcher(Candidate,_,_,_,_,_),
  has_similar_profile_to(Candidate,Expert)
```

This expression puts together the solution presented in fragments throughout the examples in Section 4. The bounding box notation has the purpose of stressing the composition of CID inference rules, and is complementary to the circles and arrows notation in Figure 1. ◁

The **core functionalities** in Figure 4 comprise user and program interfaces, and storage and retrieval facilities that are shared by all modes.

5.2 Requirements

The evaluation criteria, viz., compositionality, closure, seamless evaluation and extensibility map to concrete user requirements in the context of the usage mode in Section 5.1 as follows:

U.1. To allow for extensibility, in definition mode, a CID system must enable knowledge engineers to express CID inference rules, and must enable them to incorporate (a process referred to as *plug*) those;

U.2. In the manipulation mode, a CID system must empower administrators to manipulate and make persist data and knowledge items,

U.3. To allow for compositionality and closure, in execution mode, a CID system must enable end users to compose plugged CID inference rules into tasks;

U.4. To allow for seamless evaluation, in execution mode, a CID system must provide a single underlying execution engine that is capable of evaluating any task, over the available data and knowledge managed (as in U.2).

In light of the approach outlined in Section 1 and formalised in Section 3, the user requirements above map to the following system requirements:

R.1. to parse and store CID inference rules represented as monoid comprehension expressions;

R.2. to store data and knowledge items represented as p-clauses;

R.3. to parse tasks and resolve references to inference rules that occur in them;

R.4. to compile, optimise and evaluate the tasks stemming from R.3 over data and knowledge items managed as in R.2.

This still leaves several alternatives as to how to deploy CID systems. These include, for example, as a standard database engine, as a service in a distributed environment, as a lightweight software agent, among others.

The remainder of this section describes the development of a prototype CID engine that is implemented as a classical database engine. This choice aims at investigating the feasibility of developing industrial-strength CID systems, which build, as much as possible, upon the existing core database technology, viz., compilation into a calculus, translation into a logical algebra and optimisation into a physical algebra for evaluation. Other alternatives may be more suitable in different contexts, but, for the proof-of-concept, the target community is that of database researchers and practitioners.

5.3 Representing p-Datalog Clauses

The second concern in designing a CID system is how to represent p-clauses. A convenient way is to do so directly in the monoid calculus using tuples and constants, which are primitive types. This has the advantage of simplifying the syntactical description of tasks, as well as allowing the reuse of parsing, storage, and retrieval designs already available in the literature.

In the monoid calculus, a p-clause $h \leftarrow\!\dashv C \vdash l_1, l_2, \ldots, l_n \langle f^{\wedge}, f^{\leftarrow}, f^{\vee} \rangle$ can be represented as a tuple of the form:

$\langle$ *head:*h *body:*$[l_1, \ldots, l_n]$ *validity: C combinators:* $\langle$*and:*$f^{\wedge}$ *implies:*$f^{\leftarrow}$ *or:*$f^{\vee}\rangle\rangle$

A literal $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_m)$, such as the head literal h or any of the body literals $l_1, \ldots, l_n$, is represented as:

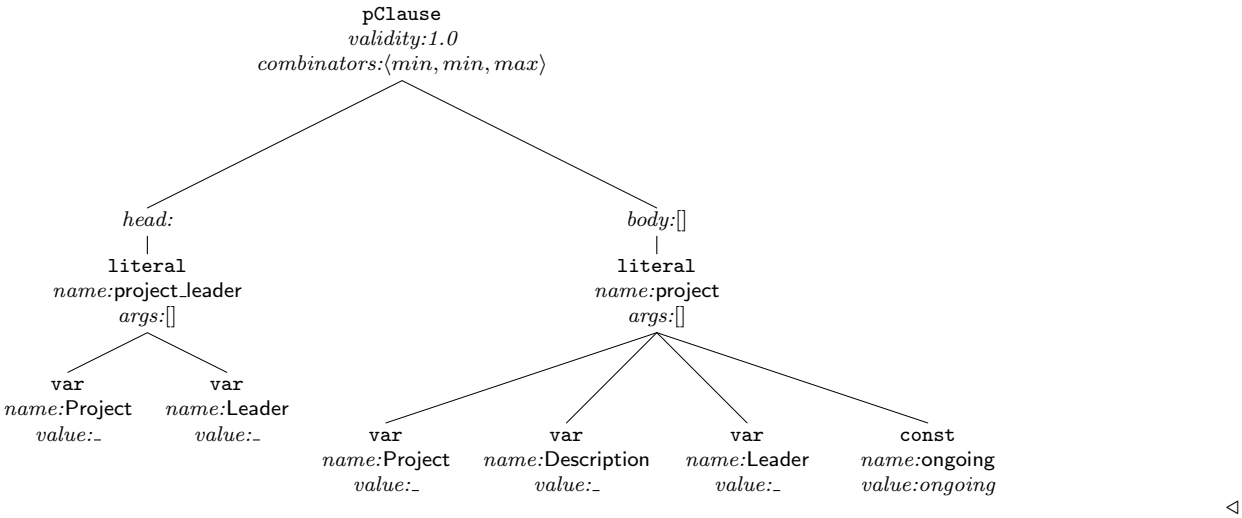
$\langle$ *name:*p *args:*$[\mathsf{t}_1, \ldots, \mathsf{t}_n]$ $\rangle$

A constant term t and a variable V are represented as $\langle$ *name:*t *value:*t $\rangle$ and $\langle$ *name:*V *value:*V $\rangle$, respectively. Thus, constant terms (and variables) in a p-Datalog language map to constant terms (and variables) in the monoid calculus, respectively.

Example 5.4. The following p-clause defines a view whose extent contains the project leader of each ongoing project:

project_leader(Project,Leader)$\leftarrow\!\dashv 1.0 \vdash$
  project(Project,Description,Leader,ongoing) $\langle min, min, max \rangle$

Using the representation above, and labelling the nodes with their types, the p-clause above is parsed into the following abstract syntax tree:



◁

This representation allows CID inference rules in Section 4 to be rewritten into a form that is more amenable to parsing and manipulation, therefore satisfying Requirement U.2. For example, the extended elementary production in Section 3.3.1 is textually represented as in Figure 5. This is the actual notation

```
eep(IDB, EDB) as
SET{<head:substitute(Rule.head, Theta),
     body:[],
     validity:
        Rule.combinators.implies(
          Rule.validity,
          Rule.combinators.and(BAG{Fact1.validity,...,FactN.validity}),
     combinators:Rule.combinators> |
  Rule  <- IDB,
  Fact1 <- EDB,
  ...,
  FactN <- EDB,
  Theta ~ mgu(Rule.body,[Fact1.head,...,FactN.head]),
  Theta != nil
}
```

Figure 5: Textual representation of Extended Elementary Production

used in the CID engine prototype described in Appendix C.

Finally, collections of p-clauses become collection monoids, and validity domains scalar monoids.

5.4 Architecture

The third concern in the design of CID system is the functional architecture[16] of its engine. A layered architecture for a CID engine has proved adequate in the implementation of the prototype.

The proposed architecture is depicted in Figure 6. Boxes denote functional components, while arrows denote data flows between them. This architecture is centred around the **CID inference rules** store, i.e., the definition of the engine's capabilities.

There are two alternative flows downward from parser. The one on the left traverses the functional components that implement a stacked database engine [Gra93]. The one on the right, flows into a calculus interpreter directly, and hence bypasses the compilation and algebraic evaluation on the left. The compiler/optimiser component depicted in Figure 7 follows the same layered architecture of a classical database engine.

Two data stores, viz., **extensional database** and **intensional database**, taken together, denote the content of the CID (cf. Section 3.2). The **language bias** store denotes the generative mechanism

[16]This architecture covers the functionalities of the execution mode, since the implementation of the other two modes is comparatively trivial.

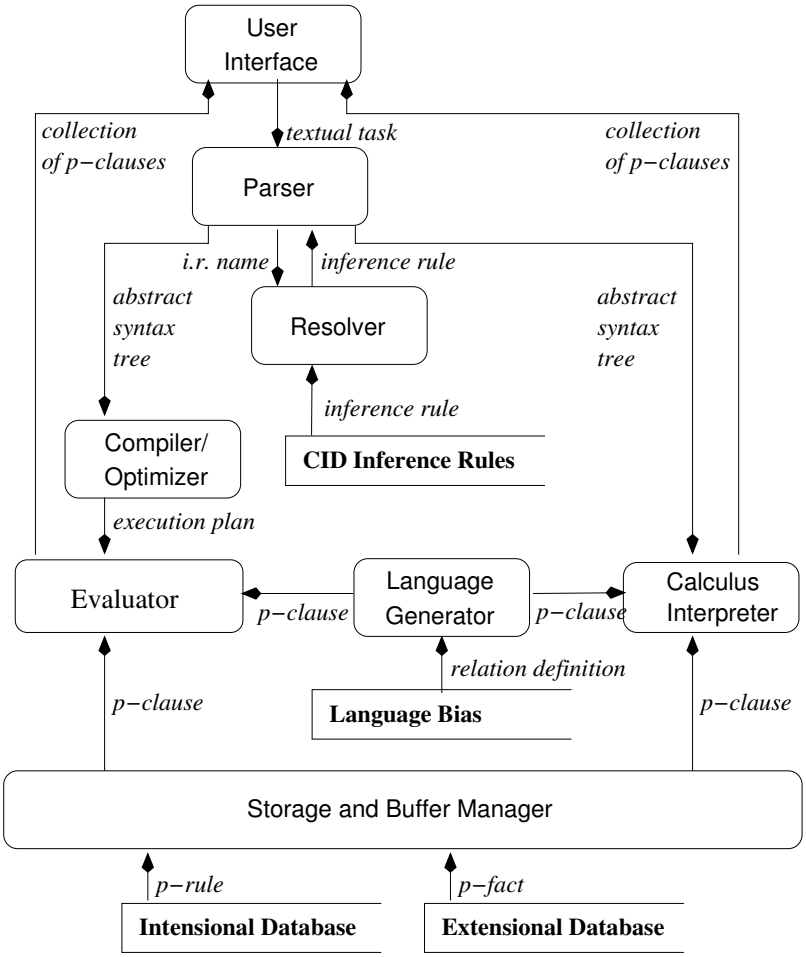


Figure 6: The Overall Architecture of a CID Engine

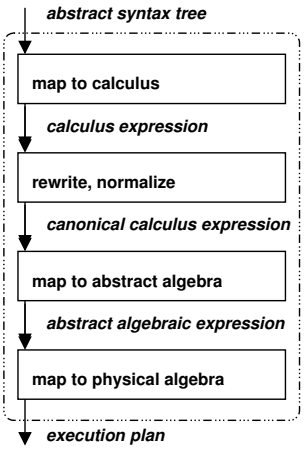


Figure 7: The Architecture of the Compiler/Optimiser Component

that, using extensional or intensional means, allows for subsets of interest of the language of the CID to be made available in the evaluation of CID tasks. It generates the extent of relations whose definition is neither in the **extensional** nor in the **intensional database**, and are, therefore, to be inductively discovered. Language bias definition and language generation are discussed in Section 5.4.7.

Example 5.3 is now developed to illustrate the functionalities of the components in Figure 6. In the examples that follow, only the relevant fragments of the actual inputs and outputs are shown, for the sake of clarity.

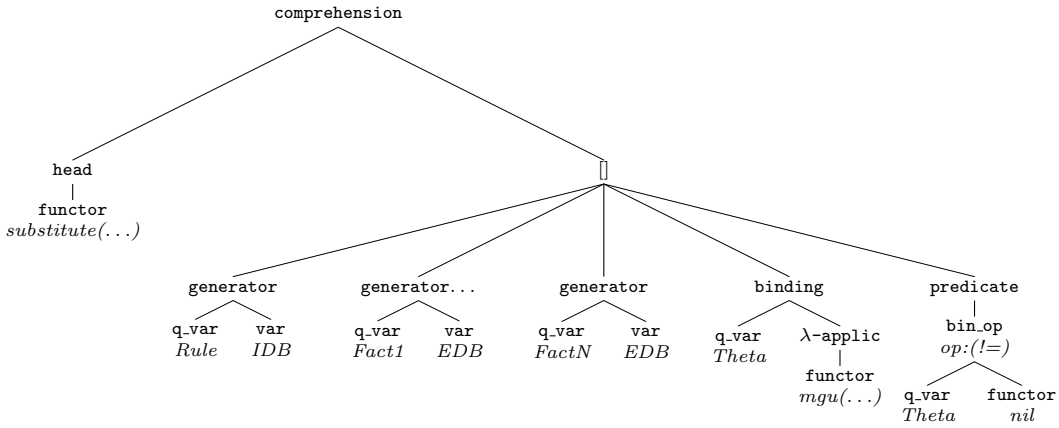
5.4.1 Command Line Interface

A *command line interface* is a simple, yet versatile, facility for ad-hoc user interaction. It implements the usage model in Section 5.1, and exposes a simple set of commands per mode (see Appendix C.4). Thus, a CID system can be configured and populated through the command line interface. CID tasks can also be specified and submitted for evaluation through the same interface.

5.4.2 Parsing

The prototype CID engine uses a generic *parser* for monoid calculus expressions, because CID tasks, CID inference rules and p-clauses can all be expressed as such and the proposal of an end-user language was left outside the scope of the research reported here. Thus, the textual form of a submitted task is an expression in the monoid calculus in which CID inference rules appear and are composed. The parser outputs an abstract syntax tree that becomes the internal representation shared both by the compiler/optimiser and by the calculus interpreter.

Example 5.5. Recall the textual representation of extended elementary production in Section 5.3. The parser maps it to the following abstract syntax tree:



◁

5.4.3 Name Resolution

The resolver component is invoked by the parser when a functor is recognised and checks whether it denotes a CID inference rule. When, in definition mode, a CID inference rule is plugged, it is parsed and assigned an identifying functor with which is associated a list of formal parameters. These parameters are variables that do not occur as range variables in generators and bindings. When, in execution mode, a CID task is evaluated, the parser identifies such functors and delegates to the resolver the retrieval request of the corresponding CID inference rule. Assuming that this process succeeds, the parser replaces the functor with the monoid calculus expression that is denoted by it and binds the formal parameters in the latter to the actual parameters in the former. This name resolution process is thus comparable to the resolution of a function call. Note that scoping rules apply, and variable renaming is used to distinguish nested lexical scopes.

Example 5.6. At the appropriate point in the parsing of the task in Example 5.3, the functor classification_rules is found as depicted in the fragment below:

```
...
Rule←  classification_rules( L1
                             O1+,
                             O1−,
                             ⟨IDB0, EDB0⟩,
                             10,
                             ),
...
```

This expression is then resolved into:

```
...
Rule <-  SET{   <head:Clause.head,
                body:Clause.body,
                validity:gS(X, O1+, O1−)
                combinators:<min,min,max>>
                ||
                Clause <- L1,
                X ∼ query_answers(Clause,⟨IDB0, EDB0⟩),
                set_cardinality(set_intersection(O1+,X)) ≥ 10,
                set_cardinality(set_difference(O1−,X)) ≥ 10
         }
...
```

The internal reference to the query_answers inference rule is, in turn, resolved into its definition (cf. Section 4.1.2) in terms of extended elementary production.

◁

The nested monoid comprehensions generated by the name resolution process are unnested during the logical optimisation and compilation steps in Figure 6. Depending on which CID tasks are composed, name resolution can give rise to redundant subexpressions (i.e., the same monoid comprehension, with the same actual parameters, occurring more than once), which cannot but be evaluated to the same results. Some initial ideas for dealing with such redundancies are discussed in Section 5.5.3.

By the end of the parsing process, all references to CID inferences rules will have been expanded into a single, possibly nested, monoid comprehension, with all formal parameters bound to actual ones given in the task. Once a CID task has been fully expanded into a single monoid comprehension, it can either be compiled into monoid algebra, as described next, or be translated into executable code in an appropriate target programming language, as described in Section 5.4.6.

5.4.4 Compilation and Optimisation

A CID task is denoted by an expression in the monoid calculus. The translation of one such expression into an equivalent expression in the monoid algebra can be performed exactly as described in [Feg98b, FM00].

Normalisation A task-denoting expression can be normalised into its canonical form using the terminating, confluent set of rewriting rules presented in [FM00]. The unnesting of most nested subexpressions (e.g., those that signal composition of CID inference rules) happens during this stage. Thus, the normalisation process plays a crucial role in the efficient evaluation of tasks.

Example 5.7. Consider again the fragment from Example 5.3 used in Example 5.6. The application of a single normalisation rule (rule N8 in [FM00]) to the expression depicted in Example 5.6 unnests the nested monoid comprehension introduced by the name resolution process to yield the following expression:

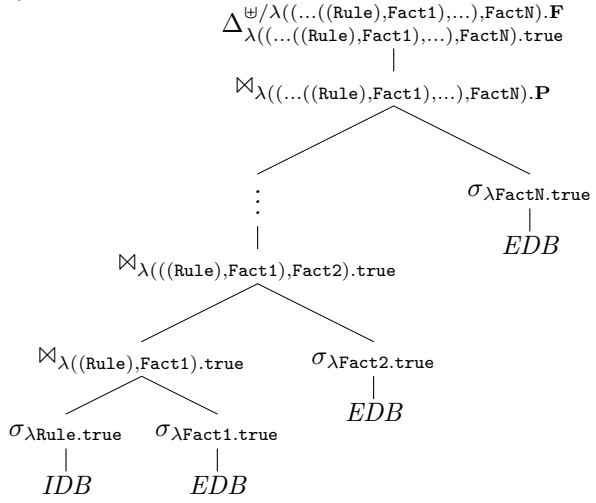
```
...
Clause <- L1,
X ~ query_answers(Clause,⟨IDB0, EDB0⟩),
set_cardinality(set_intersection(O1+,X)) ≥ 10,
set_cardinality(set_difference(O1−,X)) ≥ 10
Rule ~   <head:Clause.head,
          body:Clause.body,
          validity:gS(X,O1+, O1−)
          combinators:<min,min,max>>
...
```

Observe that there is one box less than in Example 5.6, i.e., the normalisation process acts to reduce the levels of nesting. ◁

Normalisation rules are applied until the expression in is in canonical form, i.e., it only contains generators over collections and predicates (possibly over nested comprehensions). In the prototype CID engine the normalisation process takes the abstract syntax tree output by the parser, and outputs a restructured tree that represents the original expression in canonical form.

Unnesting Expressions in canonical form are translated into an equivalent logical algebra, called the *monoid algebra*[17], using the algorithm in [FM00]. The translation algorithm unnests subexpressions in the head or in predicates of a canonical expression that the normalisation algorithm lets through. The outcome is a logical query evaluation plan.

Example 5.8. The `eep` *CID inference rule in Section 5.3 is translated into the following logical query evaluation plan:*



The generators in `eep` *have given rise to joins* ($\bowtie_p$) *and scans* (σ_p). *The eject function* $\boldsymbol{F}$ *in* `eep`, *i.e.:*

```
⟨head:substitute(Rule.head,mgu(Rule.body,[Fact1.head,...,FactN.head])),
body:[],
validity:Rule.combinators.implies(Rule.validity,
            Rule.combinators.and([Fact1.validity,...,FactN.validity])),
combinators:Rule.combinators⟩
```

has given rise to the function $\lambda((\ldots((\texttt{Rule}),\texttt{Fact1}),\ldots),\texttt{FactN}).\mathbf{F}$ *in the root reduce* (Δ). *The predicate* $\boldsymbol{P}$ *in* `eep`, *i.e.:*

```
mgu(Rule.body,[Fact1.head,...,FactN.head])!=nil
```

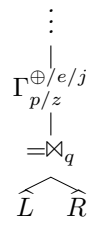
has given rise to the filtering predicate $\lambda((\ldots((\texttt{Rule}),\texttt{Fact1}),\ldots),\texttt{FactN}).\mathbf{P}$ *in the topmost join operator, because this is (bottom up) the earliest point in the logical plan where the existence of a unifier (mgu) can be verified (insofar as a rule and the facts it requires have been retrieved).*

[17] See the list of operators in the monoid algebra and the symbols that denote them are described in the Table of Symbols.

All scan operators, all join operators (but the topmost one), and the reduce operator include the vacuous predicate **true**, *because there are no more conditions to test. Observe the bottom-up flow that collects p-rules from IDB and enough p-facts from EDB until the root operator can construct a new p-fact, represented as the tuple* ***F*** *in the picture above.* ◁

More complex comprehension expressions give rise to correspondingly more complex executions plans with respect to the number of algebraic operators involved. Thus, complex compositions of CID inference rules often give rise to very complex logical plans.

Example 5.9. Some inductive CID inference rules, e.g., those described in Section 4, give rise to logical plans that contain the following fragment (with $\widehat{L}$ *and* $\widehat{R}$ *denoting the left and right subplans of their parent node):*



in which the left subplan $\widehat{L}$ *of the outer join* ($=\!\!\bowtie_q$) *contains a scan over a language generator, while the immediate, upstream operators scan p-facts that aim to match the body of the generated clause. The nest operator* ($\Gamma^{\oplus/e/j}_{p/z}$) *computes the extension of the generated clause and assembles the clause and its extension into a pair. Further up in the plan, a reduce operator can use this pair to compute the validity assessment for the generated clause.* ◁

A typical logical plan forms a tree with a reduce operator at the root, and scan operators at the leaves. In the internal nodes, inputs are combined using join and outer join operators, or pairs of nest and outer join operators applied to the results of unnesting CID inference rules.

Beyond the general optimisations described above, there is still much room for improvement before robustly efficient evaluation of CID tasks is achieved. Some specific optimisation techniques are described in Section 5.5 that go beyond what is reported in [FM00]. Cost-based physical optimisations still need to be investigated, since have been left outside the scope of the research reported here.

Mapping to a Physical Algebra The last step in the compilation is to map the logical execution plan into a physical plan consisting of the best possible operators in the physical algebra the CID engine makes available.

The prototype CID engine implements the physical operators listed in Table 2. The pseudocode

logical operators	physical operators
$\sigma_p(I)$	SCAN(p,I)
	INDEXED-SCAN(p,I)
$L \oplus R$	BAG-UNION(L,R)
$L \bowtie_p R$	NESTED-LOOP-JOIN(p,L,R)
$L=\!\!\bowtie_p R$	OUTER-NESTED-LOOP-JOIN(p,L,R)
$\Delta^{\oplus/e}_p(I)$	REDUCE($\oplus$,e,p,I)
$\Gamma^{\oplus/e/j}_{p/z}(I)$	NEST($\oplus$,e,j,p,z,I)
	PRINT(I)
	CACHE(d,I)
	PRUNE(p,I)
	REFINE(d,p,I)

Table 2: Correspondence Between Logical and Physical Operators

of all physical operators is given in Appendix C.7. The correspondence between logical[18] and physical operators is also shown in Table 2. For each logical operator there are, in principle, different semantically equivalent physical operators, one of which is chosen as the implementation of the logical operator in the physical plan. In the prototype, all physical operators implement the iterator interface described in

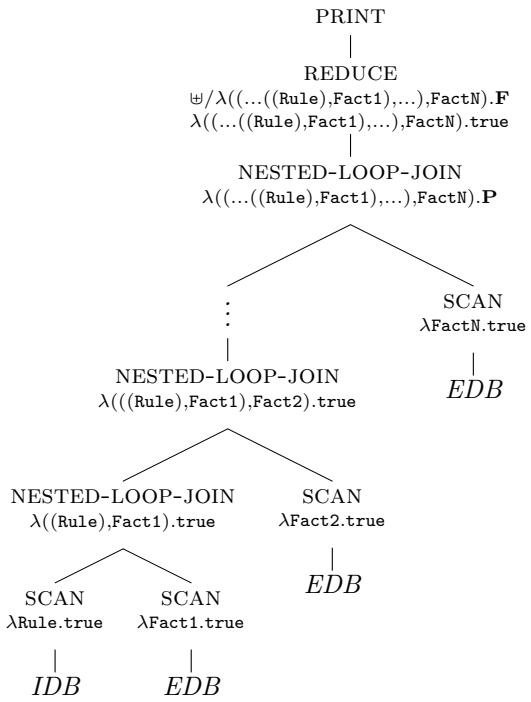
[18]Note that operators unnest (μ) and outer unnest ($=\!\!\mu$) are not discussed here, since they rarely occur in CID execution plans.

[Gra93], and hence, in general, physical plan evaluation exhibits pipelined parallelism. The iterator-based model is an approach physical plan evaluation whose main goal is to keep the memory requirements low. Insofar as most operators only keep minimal state (i.e., have constant space complexity for a low n) this is achieved. However, there are cases in which the operator is said to *block*. By that is meant that (typically in the opening phase) the operator consumes its (or one of its) inputs. For example, a hash join consumes one (typically, the smaller) of its inputs to populate a hash table against which the other input will have its tuples matched in normal iterator-based style. It is up to the physical optimiser to use cost parameters and decide that an operator, notwithstanding its blocking behaviour, is preferrable. There are also cases in which collection-valued attributes may cause the internal state to vary in size, but since attribute-valued collections are not normally expected to have high cardinality, this is not expected to jeopardise the achievement of the goal of keeping memory requirements low.

The implementations of SCAN, NESTED-LOOP-JOIN, REDUCE and PRINT are based on [Sam02]. SCAN retrieves p-clauses that are either managed by the storage component or lazily constructed by a language generator. The join operators (NESTED-LOOP-JOIN and OUTER-NESTED-LOOP-JOIN) are implemented as expected [GMUW01] and, of course, hash-based and sort-merge joins would also be possible. REDUCE and UNNEST need, in addition, to cache intermediate results so as to apply monoid merge operations correctly. For instance, $\Delta_p^{max/e}(I)$ requires that the input I is completely scanned before the single maximum of I that satisfies p can be returned. The physical operators described so far correspond one-to-one to logical operators, and hence offer no real choice during physical optimisation, but they are sufficient for the evaluation of any CID task.

The PRINT operator is introduced at the root of the execution plan simply as a means to print the result. Four additional operators, viz., INDEXED-SCAN, CACHE, PRUNE, and REFINE are optional, but enable further optimisations, as explained in Section 5.5. INDEXED-SCAN uses information on its predicate and a index on p-clauses in order to retrieve fewer clauses, viz., only those that match the selection predicate. CACHE caches intermediate results to avoid re-evaluating replicated subplans. REFINE and PRUNE work together in order to avoid generating clauses that can be inferred to have no impact on the outcome, thus speeding up the evaluation. These four operators are discussed in detail in Section 5.5.

Example 5.10. The logical plan in Example 5.8 is translated into the following physical plan:



where the symbols $\boldsymbol{F}$ *and* $\boldsymbol{P}$ *denote the same function and predicate, respectively, as they do in Example 5.8.* ◁

5.4.5 Evaluation of Physical-Algebraic Plans

An iterator-based implementation of the physical operators implies a specific flow of control in the evaluator component. The semantics is that the evaluator receives a physical execution plan as input,

calls `open()` on the root operator, then, while there is input to consume, calls `next()` on it, and finally calls `close()` on the root to end the evaluation. Each call is propagated down the tree by a parent calling its children until, at the leaves, p-clauses start to flow upwards. Once the inputs at the leaves have been consumed, the clean-up process starts from the leaves and propagates up until the root is reached, whereby evaluation ends. The final output is a (possibly empty) collection of p-clauses.

Example 5.11. The evaluation of the execution plan in Example 5.10 over the CID in Appendix A, produces a set containing at least the following p-clause:

candidate_substitute(james,sarah,p1)$\hookleftarrow 0.9 \vdash$

◁

5.4.6 Evaluation of Calculus Expressions

It is easier to enhance the expressiveness of the monoid calculus than it is of the monoid algebra. For example, to capture recursion, a fixpoint computation can be used in wrapping a monoid comprehension. It is harder to make a comparable adjustment in the corresponding algebra, although the literature records how it can be done [BFP$^+$95, SP00]. The prototype described in this section does not include this capability in its logical and physical algebra, and hence it is only effective for hierarchical CIDs.

However, a monoid calculus expression is a comprehension expression. Since several programming languages make comprehensions first-class constructs, it is possible to evaluate a CID task involving recursion not by mapping them into an algebraic execution plan, but directly, as source code in an appropriate language. The drawback is that the execution will be inefficient. Since it is easy to do, the prototype CID engine is also capable of mapping CID inference rules into list comprehension in some language-specific syntax. In the case in hand, the target language is Python [vRe03, Bea01] but the translation target could be any other programming language that directly support comprehensions, e.g., Haskell [Bir98].

Example 5.12. Extended elementary production in Section 5.3 is translated into the Python code fragment below:

```
[ pclause(substitute(rule.head,theta),
        [],
        rule.combinators.implies(rule.validity,
          rule.combinators.and([fact1.validity,...,factn.validity]),
        rule.combinators)
   for rule in IDB
   for fact1 in EDB
   ...
   for factn in EDB
   for theta in [mgu(rule.body,[fact1.head,...,factn.head])]
   if  theta != None
]
```

This list comprehension can then be evaluated by a Python interpreter if `EDB` *and* `EDB` *have been properly assigned. Note that the last* `for` *loop acts as a binding action because Python syntax does not allow assignments inside list comprehensions. Otherwise the mapping is direct, and only the support for other monoids (e.g.,* $\cup$*,* $+$ *and max) needs to be programmed.* ◁

Observe that both the evaluation of algebraic execution plans and the interpretation of calculus expression satisfies the requirements U.3 and U.4 in Section 5.2.

5.4.7 Language Generators

Inductive inference yields p-clauses that are part of the language of a CID, but not part of its intensional database (see Section 3). Thus, the evaluation of a CID task in which inductive inference rules occur requires the systematic generation of such clauses (since they are not stored anywhere). The language generator component is included in the architecture exactly to generate lazily (i.e., upon request) the clauses in the language.

A convenient implementation of a language generator exposes the same iterator-based API (i.e., `open()`, `next()` and `close()`) that the storage component does. Thus, the same implementations of the operator SCAN can retrieve p-clauses either from the storage manager or from a language generator.

An execution plan to evaluate the composition of several inductive inference rules may need to scan different subsets of the language of the same CID. Note, however, that Datalog languages are domain-independent, i.e., they only differ in the lexicon that is implicit in the schema declarations. Other than that, a clause has a head literal and a possible empty list of literals, each of which has a relation name defined in terms of constant and variable names. Hence, the same language generator can generate different language biases, by taking as a parameter the appropriate lexicon. A straightforward way of implementing a language generator is an engine that parses and generates sentences defined by a DCG [PW80]. This idea has its roots in the fact that parsing a DCG is a simple form of deductive inference [PW83], therefore, inverting a parser to yield a generator can be understood as a mechanism for the inductive construction of sentences in the language defined by the grammar. DCG-based engines are versatile, insofar as they can parse or enumerate clauses according to a given order, or even to take a reference clause and generate only clauses that are greater than the reference under the given ordering. Other inputs such as typing information and argument-binding patterns can be easily incorporated into a DCG to control and reduce the number of clauses to be generated [MdR94]. Although other similarly concise formalisms have many interesting properties [Coh94, DdR96, dRBDL01], DCG-based engines are simple and powerful enough for the purposes of this dissertation. The DCG used to implement a language generator for Datalog clauses is listed in Appendix C.6.

Example 5.13. An engine that can interpret the DCG in Section C.6 can also generate all the language biases illustrated in Section 4.2. For instance, the DCG parameters to specify the language bias in Section 4.2.1 are the following:

```
'@bodysize'(3).
'@type'(name).
'@type'(keyword).
...
'@predicate'(head,is_competent_in,2,[+name,-keyword])
'@predicate'(body,researcher,7,
    [-name,-number_of_participations,-recognition,-wage,-affiliation,-position])
'@predicate'(body,paper,3,[-paper,-title,-venue])
'@predicate'(body,writes,2,[-name,-paper])
'@predicate'(body,is_pc_member_of,2,[-name,-venue])
'@predicate'(body,refers_to,2,[-paper,-keyword])
'@const'(recognition,good).
'@var'(name,Researcher).
'@var'(expertise,Expertise).
...
```

Thus, a DCG generator can generate the rules illustrated in Section 4.2.1 using the lexicon above, modulo variable renaming. Most of this specification can be drawn from the schema of a CID, as depicted in Figure 10.

◁

5.5 Preliminary Investigation on Optimisation

Performance and scalability issues are beyond the scope of this research. Nevertheless, this section provides some preliminary evidence that efficient evaluation is not precluded in principle.

The preliminary empirical explorations have shown that a naïvely-generated execution plan is almost never efficient. Issues arise because some characteristics of CID tasks and CID inference rules are not specifically catered for in the Fegaras-Maier work on monoid query processing. Section 5.5.1 explains what some of these issues are and Sections 5.5.2 to 5.5.4 discuss some initial strategies to address them.

5.5.1 Issues Regarding Efficient Evaluation of CID Tasks

The characteristics of inefficient execution plans include:

- no selectivity of scans;
- generation of a large number of replicated subplans;

- naïve, wasteful generation of clauses in inductive inference rules;

In the context of this dissertation, the first problem is especially acute in executions plans derived from extended elementary production, as in, e.g., Example 5.8. Notice how the extensional database (*EDB*) is scanned, as many times as there are literals in the body of each rule scanned from the intensional database (*IDB*). The first operator with some selectivity occurs only after that, viz., the join with the predicate that verifies the existence of a most general unifier. Thus, several combinations of literals are scanned and propagated upwards, before any test is effectively applied to discard them.

The second problem arises because one CID task may contain multiple references to the same CID inference rule, over the same CID, as mentioned in Section 5.4.3. For example, a functional-composition monoid expression exactly nests several applications of the same inference rule. The unfolding of *hierarchical_consequences* (recall Section 4.1.1) gives rise, after compilation and unnesting, to a number of SCAN operators that is exponential on the depth of the dependency paths in the intensional database.

The third problem is a consequence of the very conceptual design of CID systems. None of the search-based optimisation techniques used in machine learning and data mining algorithms are captured in inductive CID inference rules. Such techniques are often driven by the goal of reducing the expansion of the search space as it is traversed and possibly pruning it. This is typically done by controlling the language generator to avoid waste. However, CID engines implement the exhaustive, bottom-up, dataflow computation model of query execution. Thus, the evaluation of execution plans involving inductive inference typically scans larger subsets of the language than needed. This is often wasteful.

5.5.2 Pushing Down Unification-Based Predicates

One way to address the first problem in the previous section is to break up the *mgu* predicate, and push down the corresponding subpredicates to the SCAN operators. More precisely, if $mgu([L_1, \ldots, L_{i+1}], [R_1, \ldots, R_{i+1}])$ exists, then

$$mgu([L_1, \ldots, L_i], [R_1, \ldots, R_i]) \subseteq mgu([L_1, \ldots, L_i, L_{i+1}], [R_1, \ldots, R_{i+1}, R_{i+1}])$$

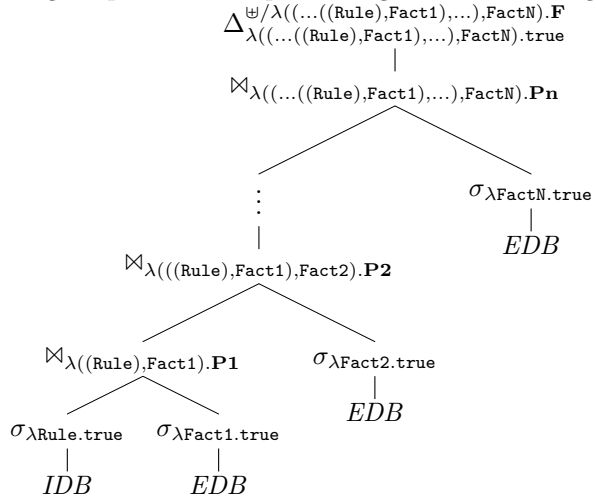
This property entails the following equation:

$$mgu([L_1, \ldots, L_n], [R_1, \ldots, R_n]) = mgu([L_1], [R_1]) \cup \ldots \cup mgu([L_1, \ldots, L_n], [R_1, \ldots, R_n])$$

Hence, $mgu([L_1, \ldots, L_n], [R_1, \ldots, R_n]) \neq \nabla$ is equivalent to:

$$mgu([L_1], [R_1]) \neq \nabla \wedge \ldots \wedge mgu([L_1, \ldots, L_n], [R_1, \ldots, R_n]) \neq \nabla$$

As a consequence, the logical plan in Example 5.9, e.g., can be rearranged into:



where **F** represents the same tuple as in Example 5.8, and the other symbols in bold are as follows[19]:

P1 represents `mgu(Rule.body[1:1],[Fact1.head])!=nil`
P2 represents `mgu(Rule.body[1:2],[Fact1.head,Fact2.head])!=nil`
...
PN represents `mgu(Rule.body[1:n],[Fact1.head,...,FactN.head])!=nil`

[19] The notation $L[1:i]$ is a shorthand for the first i elements of the list L.

In this way, the number of p-clauses flowing into successive joins is likely to be reduced, even though, the scan operators are still producing as many p-clauses. Observe that the predicates **P1**, **P2**, ... **PN** effect verifications that are redundant in principle but often useful in practice. This is because computing unification for Datalog clauses is comparatively inexpensive, so the cost of evaluating these predicates is likely to be offset by the gains resulting from pushing fewer p-clauses upwards.

This technique can be made more effective if a modified join algorithm is used, i.e., if the logical optimisation above can be complemented by a physical one. For example, assume that a NESTED-LOOP-JOIN operator in a given execution plan has as predicate **P3**, say:

```
λ((((Rule),Fact1),Fact2),Fact3).
  mgu(rule.body[1:3],[Fact1.head,Fact2.head,Fact3.head])
```

and assume that this join retrieves from its left input the tuple:

$$(((\mathsf{p(X,Y)} \leftarrow\!\!\shortmid 1.0 \vdash \mathsf{q(X)}, \mathsf{r(X,Z)}, \mathsf{s(Z,Y)}), \mathsf{q(2)} \leftarrow\!\!\shortmid 0.9 \vdash), \mathsf{r(2,5)} \leftarrow\!\!\shortmid 0.6 \vdash)$$

A partial application (i.e., a Currying [BSM98] of the predicate **P3** over the tuple above yields the following predicate **P3′**:

```
λFact3.mgu([q(X),r(X,Z),s(5,Y)],[q(2),r(2,5),Fact3.head])
```

For the partial substitution {X/2,Z/5}, **P3′** can be simplified to yield:

```
λFact3.mgu([s(5,Y)],[Fact3.head])
```

The new predicate **P3′** can be pushed to the right SCAN operator, before the right input is open for reading, insofar as it depends only on the right input now. In this way, **P3′** gives the right SCAN operator greater selectivity, therefore the right SCAN would pass possibly fewer p-clauses to its parent join. Even better, the INDEXED-SCAN operator can be implemented to exploit its selection predicate and clause indexing [DMC89]. In this example, the INDEXED-SCAN operator can make use of **P3′** and retrieve only p-facts with relation name $s/2$ and with the first argument instantiated with 5. Simply considering the relation name and one instantiated argument may drastically reduce the number of p-facts to be output by the SCAN, as done in typical logic programming interpretors.

Furthermore, pushing down unification-based predicates enables cost-based optimisations that address not only the cardinality of the relations, but also the co-occurrence of variables in literals in the body of a p-rule, by choosing the best order for multiple joins[Feg98a][20]. For example, join operations that correspond to literals that are likely to be totally or partially ground sooner should be pushed down to increase selectivity.

The strategy described above is an example of a CID-relevant logical optimisation strategy. The next two strategies exemplify CID-relevant physical optimisation strategies.

5.5.3 Caching Replicated Subplans

A caching strategy can address the second problem, viz., subplan replication. The obvious alternative, i.e., eliminating repetitions during the parsing, may both block possible logical optimisations and require the normalisation rules and the compilation algorithm to be rewritten. In order to avoid this extra effort, the parser allows the replication through, but signals its occurrence to the optimiser. The parser also labels comprehension generators in order to identify which CID inference rule they originally belong to. These labels are ignored during normalisation and compilation, but serve to identify SCAN operators that are replicated in the resulting execution plan. If there is replication, the optimiser will try to introduce a CACHE operator in the execution plan in order to avoid the re-evaluation of replicated subplans.

The CACHE operator evaluates its input subplan completely, once and only once, and caches its output, as an intermediate result. Any subsequent attempt to read the same input from the same instance of a CACHE operator does not cause its subplan to be re-evaluated, but instead, the cached results to be read, because this input was cached when it was first read. Reference counts keep track of how many times an input instance has been requested by parent operators, and removes it from the cache when it is going to be no longer read. In short, for scalability reasons, only inputs that have been requested at least once but not as many times as there are parent operators are kept in the cache.

```
cache(id,input,parents)

open()
    subplan = input.open()
    materlz = {} # local state
    return materlz.open() # return a new cursor crs

next(crs) # crs from the cursor pool
    cached = materlz.next(crs)
    if cached == null # not cached yet
       input = input.next()
       add (input,1) to materlz # cache as read by one
       return input
    else
       if cached.refcount + 1 == parents
           remove cached from materlz # no need to store
       else
           cached.refcount = cached.refcount + 1
       return cached.input

close(crs)
    materlz.close(crs)
    if materlz is empty # all inputs consumed by all parents
        input.close()
```

Figure 8: Pseudocode for the CACHE Operator

This implementation preserves pipelined parallelism. The pseudocode of the CACHE operator is given in Figure 8.

The CACHE operator is introduced in an execution plan by the exhaustive application of three transformation rules. These transformation rules are applied in order and exhaustively, from the leaves to the root.

The first one introduces a CACHE operator. The opportunity for doing so is characterised by the template to the left in Table 3 in which two SCAN operators were compiled from the same CID inference rule, as can be seen from the fact that both have been marked by the parser with the same † token, and in the subplan denoted by $\widehat{X}$ no other SCAN operator occur that is also marked with †. The new CACHE operator is introduced as shown to the right in Table 3 and also marked with †.

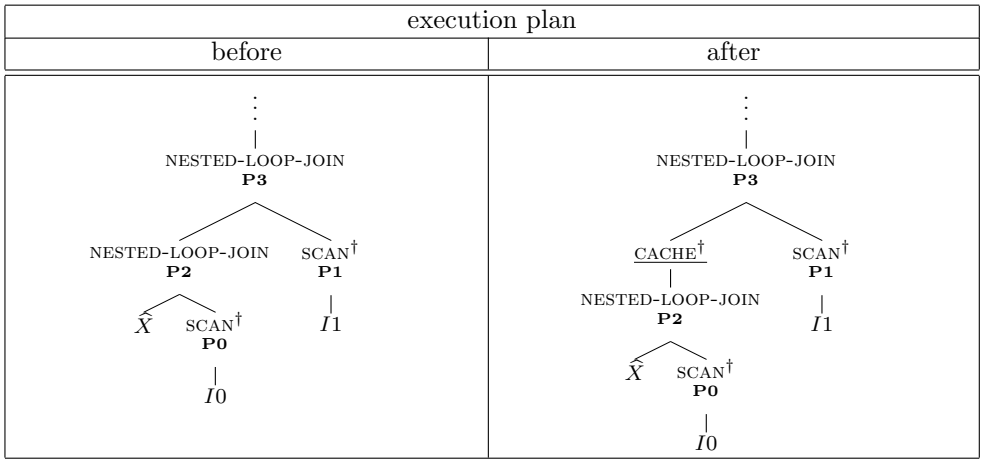


Table 3: Introduction of a CACHE Operator (First Rule)

Opportunity for caching may happen in more than one point of the execution plan and the optimiser introduces a CACHE operator in each of these. However, if different points are marked as associated to

[20]The reordering of join operators is valid because a bottom-up strategy for task evaluation is order-insensitive with regard to literals whose variable-sharing gives rise to joins.

the same CID inference, then the optimiser introduces the *same* instance of the CACHE operator. Thus, the topology of the plan changes from a tree to a graph in which some nodes are shared.

The second transformation rule pushes an introduced CACHE operator up, so as to ensure that it completely covers the largest possible replicated subplan. As depicted in Table 4, it promotes a CACHE operator by swapping it with its parent, provided that its sibling is also a SCAN marked by the same token. Eventually, after sufficient applications of this rule, the CACHE operator is the parent of a subplan that encompasses all SCAN operators associated with the same CID inference rule as desired.

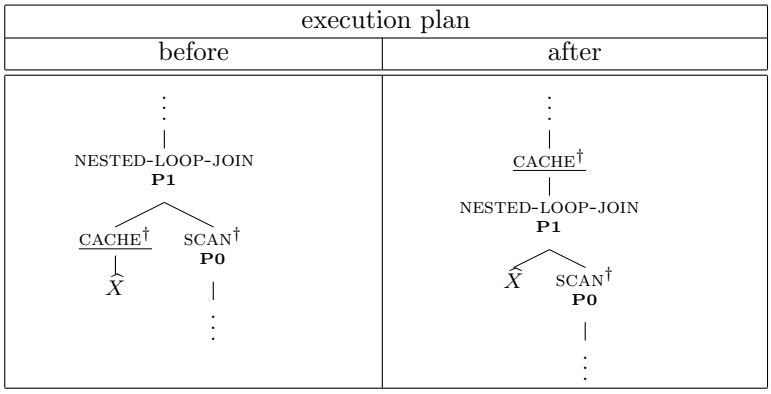


Table 4: Introduction of a CACHE Operator (Second Rule)

The third transformation rule is only applied if there are no more opportunities to apply the first and the second. It rearranges the execution plan so as to avoid that one CACHE operator occurs in the subplan of another. If there is a path in the execution plan between two CACHE operators, then the one above would redundantly store results that the one below has already stored. This is depicted in Table 5.

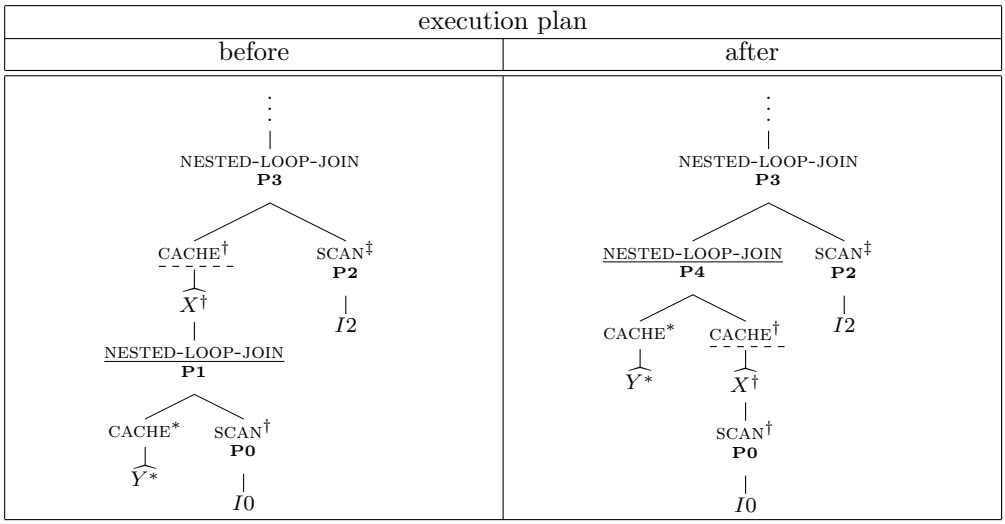


Table 5: Introduction of a CACHE Operator (Third Rule)

The path between two CACHE operators is reordered by promoting the underlined NESTED-LOOP-JOIN to lie above the two CACHEs, as depicted to the right in Table 5. This transformation requires that $X^\dagger$ contains no CACHE operator.

The tuple $((\ldots(V_1),\ldots),V_n)$ was formed by joins in the subplan Y^*, and joined with the output of SCAN†, say $V_1^\dagger$. Then this tuple is concatenated with more output in the subplan $X^\dagger$, giving rise to $((\ldots(((\ldots(V_1),\ldots),V_n),V_1^\dagger),\ldots),V_n^\dagger)$, and finally joined with the output of SCAN‡, say $V_1^\ddagger$. So, the new predicate **P4** is defined to be of the form $\lambda((\ldots(((\ldots(V_1),\ldots),V_n),V_1^\dagger),\ldots),V_n^\dagger).p1$ given that **P1** is $\lambda(((\ldots(V_1),\ldots),V_n),V_1^\dagger).p1$, and **P3** is $\lambda(((\ldots(V_1^\dagger),\ldots),V_n^\dagger),V_1^\ddagger).p3$. The test verified is still the same, i.e., $p1$, however the input to the underlined join was $(((\ldots(V_1),\ldots),V_n),V_1^\dagger)$ and is now $((\ldots(((\ldots(V_1),\ldots),V_n),V_1^\dagger),\ldots),V_n^\dagger)$ because it concatenates the outputs of both CACHE operators.

P3 still has inputs of the same cardinality, and **P4** still has inputs of the same cardinality of the inputs to **P1**.

Successive applications of this transformation rule reduce the depth of the execution plan by making some replicated subplans converge to a single CACHE operator, thereby changing the tree-like topology of the execution plan.

5.5.4 Reducing Clause Generation

Very many and very different optimisation techniques have been exploited in the machine learning and data mining literatures. This section shows how one of the most used optimisation techniques can also used in CID engines.

The *apriori* algorithm [AS94] relies on the following property: assume a partial order (e.g., based on size) over the inputs (e.g., item sets) such that if the measurement (e.g., confidence) assigned to one input A is not larger than an arbitrary threshold, then none of the other inputs that are larger than A will have measurements exceeding the same threshold. Formally, $A \subseteq B$ and $measure(A) \leq T$ implies $measure(B) \leq T$, since $measure(B) \leq measure(A)$. This property allows the early pruning of candidate inputs, i.e., before the costly measure computation for those inputs. The *apriori* technique was generalised for non-recursive Datalog clauses in [TUA+98], taking query containment (which is decidable in this case) as the partial order and confidence (as defined in [AS94]) as the measure. Even before [AS94], some inductive logic programming algorithms (e.g., [QCJ93, dRB93]) had already exploited the same property for pruning their language space. These algorithms assume a specialisation ordering over clauses [NCdW96], e.g., based on θ-subsumption [Plo70, Plo71], and take the number of examples covered as the measure, since specific (i.e., subsumed) clauses cover no more examples than those covered by their generalisations (i.e., those that subsume them).

The evaluation of an inductive CID inference rule can experience similar performance gains if the same property can be exploited. A CID engine can support an *apriori*-style optimisation provided the following conditions hold. Assume one ordering, e.g., query containment or θ-subsumption, and denote it by $\subseteq$. Let C and C' be validity assessments that are used to annotate $h \leftarrow l_1, l_2, \ldots, l_n$ and $h' \leftarrow l'_1, l'_2, \ldots, l'_m$, respectively. Assume the following property: if $h \leftarrow l_1, l_2, \ldots, l_n \subseteq h' \leftarrow l'_1, l'_2, \ldots, l'_m$ than $C \geq C'$. In this case, if C is below the acceptance threshold then evaluating $h' \leftarrow l'_1, l'_2, \ldots, l'_m$ is pointless, insofar as it will not make it pass the threshold either.

CID engines can exploit this property as follows. Firstly, the language generator component must generate the clauses in order, with respect to $\subseteq$. Secondly, given one reference clause, the language generator must only generate clauses that are greater than the given reference. DCG-based generators, as explained in Section 5.4.6, are capable of both. Thirdly, the PRUNE and REFINE operators, alluded in Table 2, must be introduced in the execution plan. The pseudocode of the operators is shown in Figure 9.

The PRUNE and REFINE operators are introduced together and co-operate with one another to control the generation of p-clauses from the language. Therefore, REFINE receives an identification token (i.e., the first parameter `id`), so that one can refer to its PRUNE counterpart. A stack is used to relate their states. This stack is initialised with the constant `NULL` and the bottom element in the assumed ordering $\subseteq$ (e.g., the empty clause $\square$), which mark the end and start, respectively, of the prune-refine process. The bottom element is popped first and expanded by the language generator, triggering the generation of all clauses. When no more clauses are pushed, eventually the stack will run empty with only the the constant `NULL` to mark the end of the exploratory generation process. PRUNE only outputs p-clauses that satisfy the predicate parameter `p` and keeps track of all such clauses in the stack. REFINE retrieves p-clauses from the language generator. However, it seeds this process, as follows. It invokes the primitive `next_candidate()` to pop the last clause that PRUNE has output so far from the latter's stack and then passes that p-clause on to the language generator as a reference for further generation. Thus, the language generator is repositioned to generate p-clauses that are larger than the reference it has received, because only these stand a chance of being accepted by PRUNE later on. Therefore, pruned p-clauses are not stored in PRUNE nor retrieved by REFINE, and their specialisations are never generated in the first place.

Both operators preserve pipelined parallelism, however the evaluation of query plan does become order dependent, i.e., if the language is generated in a different order, different results may be inferred. Therefore, the order in which the clauses are generated must satisfy the property that underpins this optimisation strategy, otherwise equivalence is lost.

The first opportunity for introducing PRUNE and REFINE is characterised by the following conditions:

```
PRUNE(pred,input)

open()
  initialise stack
    # null signals termination
  push(null,stack)
    # □ is bottom element of ⊆
  push(□,stack)
  return input.open()

next()
  input = input.next()
  if input <> null
     while not pred(input)
        input = input.next()
     push(input,stack)
  return input

close()
  release stack
  input.close()

next_candidate(d)
  candidate = top(stack)
  pop(stack)
  return candidate
```

```
REFINE(id,pred,input)

open()
  paired_prune = id
  largest = paired_prune.next_candidate()
  return input.open()

next()
  while largest <> null
     input = input.next(largest)
     while input <> null
        if pred(input)
             return input
        input = input.next(largest)
     largest = paired_prune.next_candidate()
  return null

close()
  input.close()
```

Figure 9: Pseudocode for the PRUNE and REFINE Operators

a. There exists a SCAN operator that scans a language generator, tests a predicate, say **P0**, of the form $\lambda(\ldots(R),\ldots).p_0$, and outputs, a p-clause, say R.

b. There exists a REDUCE operator that tests R to verify whether its validity is greater than a threshold M. For example,REDUCE tests a predicate, say **P1**, of the form: $\lambda(\ldots(R),\ldots)(p_1 \wedge p_2)$, where p_2 is of the form $R.validity > T$.

Assume that the execution plan to the left in Table 6 satisfies both conditions. Then, a new PRUNE is introduced as the parent of the REDUCE and a new REFINE replaces the SCAN, as depicted to the right in Table 6. An identifier (denoted by †) is associated to PRUNE and passed as a parameter to REDUCE, so that reduce can invoke `next_candidate()` on its counterpart PRUNE.

The predicates in the operators involved are changed as follows:

a′. PRUNE has a predicate **P2** of the form $\lambda(\ldots(R),\ldots)p_2)$, where p_2 is of the form $R.validity > T$. Thus, PRUNE takes from REDUCE the responsibility of checking validity thresholds.

b′. REDUCE has now a predicate **P1′** of the form $\lambda(\ldots(R),\ldots).p_1$.

c′. REFINE has the same predicate **P0** of the replaced SCAN.

The second opportunity for introducing PRUNE and REFINE is very similar. The only difference is that a NEST operator verifies the validity of generated p-clause. This is depicted in Table 7.

In both cases, the filtering is delegated to the PRUNE so that the REFINE can make use of that to control the language generator.

This section has described three non-conflicting optimisation strategies that can be jointly applied to improve the performance and scalability of CID tasks.

5.5.5 Improvements Obtained

In order to illustrate the possible gains with optimisations such as described in this section, consider the CID inference rule that induces classification rules in Section 4.2.1. It nests inside itself an expression,

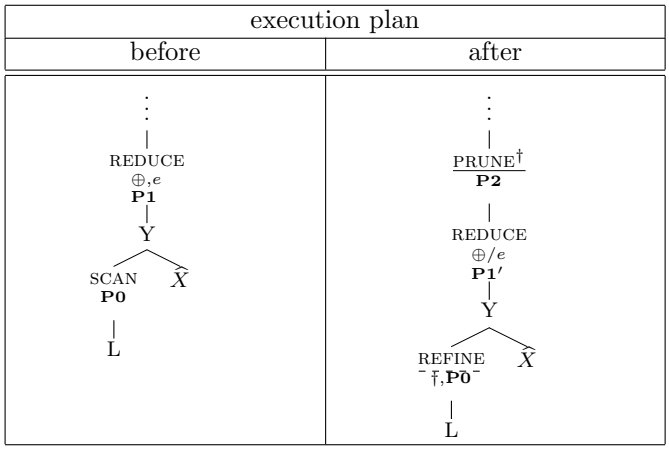


Table 6: Introduction of PRUNE and REFINE Operators (First Case)

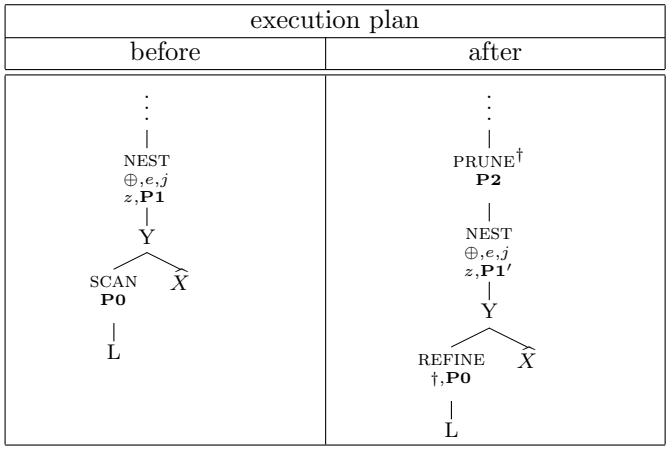


Table 7: Introduction of PRUNE and REFINE Operators (Second Case)

viz., *query_answering*, which unfolds into extended elementary production. The number of scans that would be necessary to evaluate this rule would be proportional to:

$$|L||IDB|^{(2^p-1)}|EDB|^{n2^{(p-1)}}$$

where n denotes the average size of rule body in IDB, while p denotes the longest path in the dependency graph in IDB. This implies that the number of data retrievals from EDB is a function of the cardinality of L and of IDB, of the length of the dependency path between rules in IDB, and the length of their bodies. Some of these factors have an exponential impact, therefore the level of nesting CID inference rules can easily make evaluation intractable.

However, the impact caused by the term n on the cardinality of EDB can be progressively minimised by exploiting unification properties. The impact caused by the term p can be reduced by caching the results of replicated subplans instead of effecting wasteful re-evaluations. Thus, the term p becomes the constant 1. The impact caused by the cardinality of L can be reduced by exploiting the inverse correlation between orderings over clauses and over their respective validity assessments. The combined effect of all these techniques is to make the whole approach tractable in practice.

6 Related Work

Some extensions of SQL, and even new SQL-like languages, with constructs that denote mining algorithms have been proposed [HFW+96, IV99, MPC96]. Since these approaches involve hard-wired knowledge discovery functionalities, they deliver modest results in terms of closure, compositionality and seamlessness.

The use of inductive inference in deductive databases to derive information in intensional form was first suggested in [Ber93], but its technical development was never pursued further. In another approach, [Man97] hints of the idea of combining data and knowledge stocks into a class of information management tools called *inductive databases* but provides no formalization. The inductive database approach underpins [BKM98] and [LdR03], but, at the moment, it is still unclear the what is the nature of the task of, and the effort involved in, formalizing and implementing this class of systems. The architectural design described in [Wu00] is comparable to that used in prototyping the CID engine [AF03]. However, a CID engine is underpinned by single database calculus and algebra and this leads to closure, compositionality and seamlessness in potentially scalable evaluation (e.g., via parallelism and distribution) that would be hard to achieve through the separate engines used in [Wu00]. [MPC98] presents better developed processing and optimization approaches. However, they are specifically geared towards association analysis, whereas CIDS are more comprehensive and extensible. The proposal in [JLN00] was the first to encompass several data mining techniques, as CIDS do. [JLN00] provides a constraint-based algebra to combine the results derived by different invocations of data mining algorithms, however, this comes short of a principled approach to the composition of deductive and inductive steps, as provided in CIDS. In general, the lack of a reconciliation of the inductive and deductive methods into a unified foundation has led all these proposals to lack both flexibility and a clear route for incorporation into mainstream database technology.

More closely related to CIDS is our own previous work on applying specific combinations of query answering and knowledge discovery techniques to specific problem areas. For example, an extension of inductive relations of [Ber93] was implemented as a prototype knowledge management platform [AF02b] and later used to characterize web-services substitutivity [AF02a]. However, neither [AF02b] nor [AF02a] have a database-centred operational semantics and, hence, no clear path to scalability. Moreover, unlike these, CIDS represent in the underlying logic the distinction between outcomes with different validity assessments. The user-defined assimilation policies of [AF02b] and [AF02a] only allow limited closure.

The state-of-the-art for commercial database systems is to equate data mining algorithms to user-defined functions invokable in SQL queries. IBM Intelligent Miner allows classification and clustering models coded in XML to be deployed and applied to row sets. Microsoft SQL Server with OLE DB DM provides a comparable mechanism, viz., *prediction join*, to build and apply classification models. In SPSS Clementine, one can specify compositions of preparation, mining, and visualization steps, but no more than one mining step. In all the above, compositionality and closure are very restricted and evaluation is far from seamless across the two inference modes. Levels of coupling are often so loose that significant engineering effort is required for interoperation to be possible.

7 Conclusions

This paper has described in some detail a concrete proposal for a class of logic-based databases that integrate querying data and discovering knowledge. A database-centred view of the problem has been adopted throughout that succeeds in providing a unified semantics of both deductive (e.g., query answering) and inductive tasks (e.g., classification, clustering, association, etc.) as expressions in a database calculus that has an equivalent logical algebra. The calculus builds upon Fegaras and Maier's monoid comprehension calculus for which efficient physical algebras exist. We have built an evaluator for the calculus and a compiler into the algebra, for which an evaluator also exists. The preliminary results are encouraging in that they indicate specific directions to be pursued in developing physical optimizations, but this remains a topic for further investigation.

Database research faces the challenge of eliciting novel and potentially useful information from very many, very large, databases, especially in sciences such as particle physics, astronomy, molecular biology and the sciences related to climate. Much knowledge that lies in the data is, in general, irretrievable via classical query languages. Algorithm collections bundled up as data mining tools can serve the needs of many users but the loose coupling, lack of closure and impediments to fluent exploration set a limit to what they can deliver to all but the most specialist users. This paper has aimed to show that database calculi and algebras can be developed that remove the need for coupling, deliver closure, and hence, are more likely to better meet the needs of the majority of users.

Although we have built a prototype CID system, much work remains to be done to render CIDS truly efficient. In the short term, we plan to identify and study in detail optimization opportunities and to devise concrete data structures and algorithms that might offer options to the optimizer. In the medium term, we plan to deploy CIDS as a conservative extension to OGSA-DQP [AMG+03], a distributed query processor for the Grid which we have developed jointly with colleagues, and which uses the same monoid-based query processing approach that give CIDS their operational semantics, thereby facilitating this extension task in a very significant way. In the long term, we aim to deploy this implementation of CIDS in ongoing work in e-Science, specifically in post-genomic bioinformatics [SRG03], in which we are involved and of which the current OGSA-DQP is already a component.

References

[AE82] Krzysztof R. Apt and Maarten H. Van Emden. Contributions to the theory of logic programming. *Journal of the Association of Computing Machinery*, 29(3):841–862, July 1982.

[AF02a] Marcelo A. T. Aragão and Alvaro A. A. Fernandes. Characterizing Web Service Substitutivity with Combined Deductive and Inductive Engines. In *Proceedings of the ADVIS'02*, volume 2457 of *Lecture Notes in Computer Science*, pages 244–254. Springer-Verlag, 2002.

[AF02b] Marcelo A. T. Aragão and Alvaro A. A. Fernandes. Inductive-Deductive Databases for Knowledge Management. In *Proceedings of the ECAI KM&OM'02*, pages 11–19, 2002.

[AF03] Marcelo A. T. Aragão and Alvaro A. A. Fernandes. Combining Query Answering and Knowledge Discovery. Technical report, University of Machester, 2003. Submitted for publication.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Pub., September 1995. ISBN: 0201537710.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):207–216, June 1993.

[AMG+03] M. Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, and Jim Smith. Service-based distributed querying on the grid. In *Proceedings of the 1st Inter'l Conference on Service Oriented Computing (ICSOC'03)*, 2003.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499. Morgan Kaufmann, September 1994.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles and Techniques and and Tools*. Addison Wesley Pub., November 1985. ISBN: 0201100886.

[Bax03] Andreas D. Baxevanis. The Molecular Biology Database Collection: 2003 update. *Nucleic Acids Research*, 31(1):1–12, 2003.

[Bea01] David Beazley. *Python Essential Reference*. New Riders, June 2001. ISBN: 0735710910.

[Bea03] David Beazley. PLY (Python Lex-Yacc). available at `http://systems.cs.uchicago.edu/ply/`, 2003.

[Ber93] Francesco Bergadano. Inductive database relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):969–971, 1993.

[BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Kluwer Academic Pub., January 1984. ISBN: 0412048418.

[BFP+95] Maria L. Barja, Alvaro A. A. Fernandes, Norman W. Paton, M. Howard Williams, Andrew Dinn, and Alia I. Abdelmoty. Design and implementation of ROCK & ROLL: a deductive object-oriented database system. *Information Systems*, 20(3):185–211, March 1995.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Pub., 2nd edition, 1998. ISBN: 0134843460.

[BK03] François Bry and Peer Kröger. A computational biology database digest. *Distributed and Parallel Databases*, 13:7–42, 2003.

[BKM98] Jean-François Boulicaut, Mika Klemettinen, and Heikki Mannila. Querying inductive databases: A case study on the MINE RULE operator. In Mukesh K. Mohania and A. Min Tjoa, editors, *Proceedings of the 1st International Conference in Data Warehousing and Knowledge Discovery (DaWaK'99)*, volume 1510 of *Lecture Notes in Computer Science*, pages 194–202, Nantes, France, September 1998. Springer-Verlag.

[BSM98] Richard Bird, Thomas E. Scruggs, and Margo A. Mastropieri. *Introduction to Functional Programming*. Prentice Hall Pub., 2nd edition, April 1998. ISBN: 0134843460.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in Computer Science. Springer-Verlag, Heidelberg, 1990. ISBN: 0387517286.

[CN88] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261, 1988.

[Coh94] William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 1994.

[Das92] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison Wesley Pub., Cambridge, 1992. ASIN: 0201568977.

[DdR96] Luc Dehaspe and Luc de Raedt. Dlab: A declarative language bias formalism. In Zbigniew W. Ras and Maciej Michalewicz, editors, *Proceedings of the 9th International Symposium on Foundations of Intelligent Systems (ISMIS '96)*, Lecture Notes in Computer Science, pages 613–622, Poland, June 9-13 1996. Springer-Verlag.

[DL01] Sašo Džeroski and Nada Lavrač, editors. *Relational Data Mining*. Springer-Verlag, Berlin, September 2001. ISBN: 3540422897.

[DMC89] Bart Demoen, Andre Marien, and Alain Callebaut. Indexing Prolog clauses. In *Proceedings of the North American Conference on Logic Programming*, volume 42(2), pages 1001–1012, 1989.

[dRB93] Luc de Raedt and Maurice Bruynooghe. A theory of clausal discovery. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1058–1063. Morgan Kaufmann, 1993.

[dRBDL01] Luc de Raedt, Hendrik Blockeel, Luc Dehaspe, and Wim Van Laer. Three companions for data mining in first order logic. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 105–139. Springer-Verlag, 2001.

[DT99] Luc Dehaspe and Hannu Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, January 1999.

[DT01] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, September 2001.

[Feg98a] Leonidas Fegaras. A new heuristic for optimizing large queries. In Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon, editors, *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA '98)*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer-Verlag, 1998.

[Feg98b] Leonidas Fegaras. Query unnesting in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, volume 27(2) of *ACM SIGMOD Record*, pages 49–60, New York, June 1–4 1998. ACM Press.

[FHV92] Ronald Fagin, Joseph Y. Halpern, and Moshe Y. Vardi. What is an inference rule? *The Journal of Symbolic Logic*, 57(3):1018–1045, 1992.

[Fis87] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning Journal*, 2(2):139–172, February 1987.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.

[FPS96a] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 1, pages 1–35. MIT Press, Cambridge, March 1996.

[FPS96b] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[FSRM00] Leonidas Fegaras, Chandrasekhar Srinivasan, Arvind Rajendran, and David Maier. λ-DB: An ODMG-based object-oriented DBMS. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, volume 29(2), page 583. ACM Press, 2000.

[GdB02] Bart Goethals and Jan Van den Bussche. Relational association rules: getting warmer. In D. Hand, R. Bolton, and N. Adams, editors, *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, volume 2447 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems Implementation*. Prentice Hall Pub., 2000. ISBN: 0130402648.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Pub., October 2001. ISBN: 0130319953.

[GNU03] GNU. available at `http://www.gnu.org/copyleft/gpl.html`, 2003.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HFW+96] Jiawei Han, Yongjian Fu, Wei Wang, Jenny Chiang, Osmar R. Zaïane, and Krzysztof Koperski. DBMiner: interactive mining of multiple-level knowledge in relational databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 550–550, Montreal, Canada, June 1996. ACM Press.

[HMS01] David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, August 2001. ISBN: 026208290X.

[IA93] Peter Idestam-Almquist. Generalization under implication by recursive anti-unification. In P. Utgoff, editor, *Proceedings of the 10th International Conference on Machine Learning*, pages 151–158. Morgan Kaufmann, 1993.

[IV99] Tomasz Imielinski and Aashu Virmani. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, April 1999.

[JLN00] Theodore Johnson, Laks V. S. Lakshmanan, and Raymond T. Ng. The 3W model and algebra for unified data mining. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 21–32, Cairo, Egypt, September 2000. Morgan Kaufmann Pub.

[JMF99] Anil K. Jain, M. N. Murty, and Patrick J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[KWH01] Mathias Kirsten, Stefan Wrobel, and Tamas Horvath. Distance based approaches to relational learning and clustering. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 213–232. Springer-Verlag, September 2001.

[Lan96] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann, San Francisco, 1996. ISBN: 1558603018.

[LdR03] Sau Dan Lee and Luc de Raedt. An algebra for inductive query evaluation. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM'03)*, Melbourne, USA, 2003.

[LS01] Laks V. S. Lakshmanan and Fereidoon Sadri. On a theory of probabilistic deductive databases. *Theory and Practice of Logic Programming*, 1(1):5–42, 2001.

[LSV01] Laks V. S. Lakshmanan and Nematollaah Shiri-Varnaamkhaasti. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.

[Man97] Heikki Mannila. Inductive databases and condensed representations for data mining. In Jan Maluszynski, editor, *Proceedings of the 1997 International Logic Programming Symposium*, volume 13, pages 21–30, Long Island, October 1997. MIT Press.

[MdR94] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

[Mic83] Ryszard S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161, 1983.

[Min96] Jack Minker. Logic and databases: A 20 year retrospective. In Dino Pedreschi and Carlo Zaniolo, editors, *Proceedings of the International Workshop on Logic in Databases (LID'96)*, volume 1154 of *Lecture Notes in Computer Science*, pages 3–57, San Miniato, Italy, July 1996. Springer-Verlag.

[Mit82] Thomas M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982. Also appears in Readings in Artificial Intelligence, Webber and Nilsson (eds.), Tioga Press, 1981, pp. 517-542.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw Hill Pub., March 1997. ISBN: 0070428077.

[Mit99] Thomas M. Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11):30–36, November 1999.

[Mow97] Abbe Mowshowitz. Virtual organizations. *Communications of the ACM*, 40(9):30–37, September 1997.

[MPC96] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A new SQL-like operator for mining association rules. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 122–133, Mumbai (Bombay), India, 3–6 September 1996. Morgan Kaufmann.

[MPC98] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A tightly-coupled architecture for data mining. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 14th International Conference on Data Engineering (ICDE'98)*, pages 316–323. IEEE Computer Society, 1998.

[NCdW96] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Least generalizations and greatest specializations of sets of clauses. *Journal of Artificial Intelligence Research*, 4:341–363, 1996.

[NCdW97] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1997. ISBN: 3540629270.

[Obj00] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, v. 1.5 edition, mar 2000. `http://www.omg.org/technology/documents/formal/uml.htm`.

[OBS99] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, US, 1999. Software available at `http://www.sleepycat.com/`.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and David Michie, editors, *Machine Intelligence*, volume 5, pages 153–163, Edinburgh, 1970. Edinburgh University Press.

[Plo71] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and David Michie, editors, *Machine Intelligence*, volume 6, pages 101–124, Edinburgh, 1971. Edinburgh University Press.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite clauses for language analysis. *Artificial Intelligence*, 13:231–278, 1980.

[PW83] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, jun 1983.

[Pyl99] Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Pub., March 1999. ISBN: 1558605290.

[QCJ93] J. Ross Quinlan and R. Mike Cameron-Jones. FOIL: A midterm report. In Pavel B. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning (ECML'93)*, volume 667, pages 3–20, Vienna, Austria, April 1993. Springer-Verlag.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sam02] Sandra F. M. Sampaio. *A performance investigation of query processing in parallel object databases*. PhD thesis, The University of Manchester, 2002.

[Sow99] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. PWS Pub. Co, August 1999. ISBN: 0534949657.

[SP00] Pedro R. Falcone Sampaio and Norman W. Paton. Query Processing in DOQL: A Deductive Database Language for the ODMG Model. *Data & Knowledge Engineering*, 35(1):1–38, October 2000.

[SRG03] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. myGrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19(1), 2003.

[SV97] Nematollaah Shiri-Varnaamkhaasti. *Towards a Generalized Theory of Deductive Databases with Uncertainty*. PhD thesis, Concordia University, Montréal, Canada, 1997.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/Fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of the 2nd International Logic Programming Conference (ICLP'84)*, pages 127–138, Uppsala, Sweden, July 1984.

[TUA+98] Shalom Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In Laura M. Haas and Ashutosh Tiwary, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 1–12, Seattle, USA, June 1998. ACM Press.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association of Computing Machinery*, 23(4):733–742, October 1976.

[vRe03] Guido van Rossum and Fred L. Drake Jr. (eds.). *Python Documentation*. Python Labs, version 2.3 edition, 2003. available at `http://www.python.org/doc`.

[WF99] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Pub., October 1999. ISBN: 1558605525.

[Wro96] Stefan Wrobel. Inductive logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 151–189. CSLI Publications, Berlin, German, 1996.

[Wro01] Stefan Wrobel. Inductive logic programming for knowledge discovery in databases. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 74–101. Springer-Verlag, September 2001.

[Wu00] Xindong Wu. Building intelligent learning database systems. *AI Magazine*, 21(3):59–65, September 2000.

[ZZY03] Shichao Zhang, Chengqi Zhang, and Qiang Yang. Data preparation for data mining. *Applied Artificial Intelligence*, 17(5-6):375–381, 2003.

A Motivating Example: The CID Content

A.1 Schema

The language of the CID used in the motivating example in Chapters 1 to 5 is defined by the schema in Figure 10 and explained as follows.

researcher/7 describes the current or potential participants of scientific task forces.

research_unit/7 are of type university department, a research lab, a governmental agency.

project/4 describes the project details.

participates_in/3 relates a researcher and the position he/she fills in a project.

expertise/1 is keyword that denotes expertise of interest.

requires/2 relates a project and the expertise it might require.

paper/3 describes a scientific or technical article.

writes/2 relates a paper such as above to one or more authors.

venue/3 describes a venue where papers are published.

is_pc_member_of/2 relates a researcher and a venue that he/she participated in the program committee.

refers_to/2 relates the paper keywords to expertises.

Types	
type	**description**
name	is a persons name.
past_experience	is the number of past participation in similar endeavours.
recognition	is either excellent, good or regular, denoting an assessment of the professional record.
compensation	is the contract wage in thousands of pounds/year.
affiliation	is the researcher's current affiliation.
job_title	is the position occupied by a researcher, e.g., researcher_assistant.
unit	is a unit of an institution, e.g., a lab or a department.
type	is either university, hospital or private_lab.
institution	is the institution name.
address	is a mail address.
telephone	is a contact telephone number.
url	is internet URL.
project_code	is an assigned identification.
description	is textual description.
status	is either ongoing, or complete.
keyword	denotes an expertise, e.g., bioinformatics.
bib_reference	is a bibliographic paper reference.
paper_title	is the paper title.
venue_acronym	is the venue abbreviation.
venue_title	is the venue full title.
date	is the usual date type.

Relations

researcher(name,past_experience,recognition,availability,compensation,affiliation,job_title),
research_unit(affiliation,unit,type,institution,address,telephone,url)
participation(name,project,function)
project(project,description,name,status)
expertise(keyword)
requires(project_code,keyword)
paper(bib_reference,paper_title,venue_acronym)
writes(name,bib_reference)
venue(venue_acronym,venue_title,date)
is_pc_member_of(name,venue_acronym)
refers_to(bib_reference,keyword)
is_competent_in(name,keyword)
has_similar_profile(name,name)

Figure 10: Data Schema for Procuring Candidate Replacement

is_competent_in/2 relates a researcher and an expertise that he/she can be inferred to be competent at (unlike those relations before, this one is not yet defined, extensionally or intensionally, therefore target for inductive discovery).

has_similar_profile/2 relates two researchers of similar profile, i.e., with similar past participation, recognition, position and competences (again this relation is also target for inductive discovery).

A.2 EDB

```
expertise(pharmacology) ←1.0→
expertise(immunology) ←1.0→
expertise(epidemiology) ←1.0→
expertise(biochemistry) ←1.0→
expertise(citology) ←1.0→
expertise(histology) ←1.0→
expertise(proteomics) ←1.0→
expertise(genetics) ←1.0→
expertise(molecular_biology) ←1.0→
expertise(bioinformatics) ←1.0→

researcher(peter,3,regular,  20,a3,research_assistant) ←1.0→
researcher(fiona,1,good,     32,a5,consultant) ←1.0→
researcher(james,0,good,     23,a6,senior_researcher) ←1.0→
researcher(lynda,6,excellent,36,a7,head_of_lab) ←1.0→
researcher(zoe,  4,good,     20,a1,research_assistant) ←1.0→
researcher(sally,0,good,     24,a2,research_assistant) ←1.0→
researcher(lewis,1,good,     32,a4,consultant) ←1.0→

research_unit(a1,genomics,lab,ebi,cambridge,221,www_ebi_ac_uk) ←1.0→
research_unit(a2,macromolecule,lab,sanger,cambridge,223,www_sanger_ac_uk) ←1.0→
research_unit(a3,informatics,school,ed,edinburgh,982,www_inf_ed_ac_uk) ←1.0→
research_unit(a4,medicine,department,ic,london,213,www_ic_ac_uk) ←1.0→
research_unit(a5,pathology,clinic,cmcs,manchester,270,www_cmmc_nhs_uk) ←1.0→
research_unit(a6,computer_science,department,man,manchester,275,www_cs_man_ac_uk) ←1.0→
research_unit(a7,biology,department,man,manchester,278,www_biomed_man_ac_uk) ←1.0→



project(p1,sars_epidemic,lynda,ongoing) ←1.0→
project(p2,diabetes_control,fiona,ongoing) ←1.0→

participation(peter,p1,research_assistant) ←1.0→
participation(fiona,p1,consultant) ←1.0→
participation(fiona,p2,project_leader) ←1.0→
participation(zoe,p1,research_fellow)  ←1.0→
participation(lynda,p1,project_leader) ←1.0→
participation(sally,p2,research_assistant) ←1.0→
participation(lewis,p2,consultant) ←1.0→

requires(p1,immunology) ←0.3→
requires(p1,genetics) ←0.7→
requires(p1,bioinformatics) ←0.6→
requires(p1,molecular_biology) ←0.5→
requires(p2,immunology) ←0.3→
requires(p2,epidemiology) ←0.4→
requires(p2,biochemistry) ←0.9→

paper(t11,v2,citology_explained) ←1.0→
paper(t13,v2,immunology_in_24hours) ←1.0→
paper(t15,v2,epidemiology_explained) ←1.0→
paper(t17,v2,histology_in_24hours) ←1.0→
paper(t19,v2,biochemistry_explained) ←1.0→
paper(t21,v3,proteomics_explained) ←1.0→
paper(t23,v1,genetics_for_dummies) ←1.0→
paper(t25,v3,molecular_biology_unleashed) ←1.0→
paper(t27,v3,molecular_biology_explained) ←1.0→
paper(t28,v1,bioinformatics_in_24hours) ←1.0→
paper(t29,v1,bioinformatics_in_24hours) ←1.0→
```

```
writes(fiona,t13) ⟻1.0⊢
writes(fiona,t15) ⟻1.0⊢
writes(peter,t11) ⟻1.0⊢
writes(peter,t17) ⟻1.0⊢
writes(lewis,t19) ⟻1.0⊢
writes(james,t21) ⟻1.0⊢
writes(zoe,t23) ⟻1.0⊢
writes(sally,t23) ⟻1.0⊢
writes(lynda,t25) ⟻1.0⊢
writes(james,t27) ⟻1.0⊢
writes(zoe,t28) ⟻1.0⊢
writes(sally,t29) ⟻1.0⊢

venue(v1,bioinformatics,2003) ⟻1.0⊢
venue(v2,ismb,2003) ⟻1.0⊢
venue(v3,ieee_csb,2003) ⟻1.0⊢

is_pc_member_of(peter,v1) ⟻1.0⊢
is_pc_member_of(fiona,v1) ⟻1.0⊢
is_pc_member_of(james,v2) ⟻1.0⊢
is_pc_member_of(lynda,v3) ⟻1.0⊢
is_pc_member_of(zoe,v2) ⟻1.0⊢
is_pc_member_of(sally,v3) ⟻1.0⊢
is_pc_member_of(lewis,v1) ⟻1.0⊢

refers_to(t11,cytology) ⟻0.6⊢
refers_to(t13,immunology) ⟻0.3⊢
refers_to(t15,epidemiology) ⟻0.4⊢
refers_to(t15,histology) ⟻0.9⊢
refers_to(t17,histology) ⟻0.8⊢
refers_to(t19,biochemistry) ⟻0.3⊢
refers_to(t21,proteomics) ⟻0.8⊢
refers_to(t23,genetics) ⟻0.5⊢
refers_to(t25,molecular_biology) ⟻0.6⊢
refers_to(t27,molecular_biology) ⟻1.0⊢
refers_to(t28,bioinformatics) ⟻0.9⊢
refers_to(t29,bioinformatics) ⟻0.7⊢
```

A.3 IDB

```
relevant_paper(Description,Title,Author,Venue_Name,Year) ⟻1.0⊢
   project(Project,Description,_,_),
   requires(Project,Expertise),
   paper(Paper,Venue,Title),
   writes(Author,Paper),
   venue(Venue,Venue_Name,Year),
   refers_to(Paper,Expertise)
```

B Semantics of CIDs: Proofs

B.1 Model-theoretic Semantics

Lemma B.1. *Let $\mathcal{D}$ be any CID, and $\mathbb{V}_{\mathcal{D}}$ be the set of all extended validity valuations acceptable to $\mathcal{D}$, then $\langle \mathbb{V}_{\mathcal{D}}, \oplus, \otimes, \geq \rangle$ is a complete lattice.*

Proof. *The lemma follows from the fact that $\mathbb{V}_{\mathcal{D}}$ is a pointwise extension of the validity domain of $\langle \mathbb{C}, \oplus, \otimes, \geq \rangle$ which is assumedly a complete lattice.* □

Lemma B.2. *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{I}_{\mathcal{D}} \rangle$ be a CID system, let u and v be any two extended validity valuations, each of which satisfies $\mathcal{S}$, then $u \otimes v$ is also an extended validity valuation that satisfies $\mathcal{S}$.*

Proof. *Let ρ be any inference rule in $\mathcal{I}_{\mathcal{D}}$ which uses the function g_ρ to compute the validity assessment for its consequences. Let $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n$ be ground instances of, respectively, p-clauses $S_1, S_2, \ldots, S_n$ in*

$\mathcal{L}_\mathcal{D}$. *Assume that* $S_1, S_2, \ldots, S_n$ *can be taken as premisses for* ρ. *Let* $\{\!\{\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n\}\!\}$ *be a bag if ground instances of* $S_1, S_2, \ldots, S_n$. *Since* u *satisfies* $\mathcal{S}$, *by Definition 3.1, it is true that:*

$$u(Q) \geq g_\rho(u(\widehat{S}_1), u(\widehat{S}_2), \ldots, u(\widehat{S}_n))$$

Since, by hypothesis, g_ρ *is a monotone function, it holds that:*

$$g_\rho(u(\widehat{S}_1), u(\widehat{S}_2), \ldots, u(\widehat{S}_n)) \geq g_\rho((u \otimes v)(\widehat{S}_1), (u \otimes v)(\widehat{S}_2), \ldots, (u \otimes v)(\widehat{S}_n))$$

therefore,

$$u(Q) \geq g_\rho((u \otimes v)(\widehat{S}_1), (u \otimes v)(\widehat{S}_2), \ldots, (u \otimes v)(\widehat{S}_n))$$

The same argument holds for:

$$v(Q) \geq g_\rho((u \otimes v)(\widehat{S}_1), (u \otimes v)(\widehat{S}_2), \ldots, (u \otimes v)(\widehat{S}_n))$$

These two results together imply, by the properties of a lattice, that:

$$(u \otimes v)(Q) \geq g_\rho((u \otimes v)(\widehat{S}_1), (u \otimes v)(\widehat{S}_2), \ldots, (u \otimes v)(\widehat{S}_n))$$

This, in turn, implies that $(u \otimes v)$ *is an extended validity valuation that satisfy the bag of ground instances* $\{\!\{\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n\}\!\}$. *Moreover, it follows from Definition 3.1 that:*

$$\begin{aligned} u(Q) &\geq f^\vee(\{\!\{g_\rho(u(\widehat{S}_1), u(\widehat{S}_2), \ldots, u(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \\ v(Q) &\geq f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \end{aligned}$$

therefore,

$$(u \otimes v)(Q) \geq f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\})$$

Since $\widehat{S}_1, \widehat{S}_2, \ldots, \widehat{S}_n$ *are arbitrary ground instances of* $S_1, S_2, \ldots, S_n$, *then* $(u \otimes v)$ *satisfy* $S_1, S_2, \ldots, S_n$. *Further, since* $S_1, S_2, \ldots, S_n$ *are arbitrary premisses and* ρ *is an arbitrary inference rule, then it holds that* $(u \otimes v)$ *satisfy* $\mathcal{S}$. □

Theorem B.1. *Let* $\langle \mathcal{D}, \mathcal{I}_\mathcal{D} \rangle$ *be a CID system, and* $\mathbb{V}_\mathcal{D}$ *the set of all extended validity valuations that are acceptable to* $\mathcal{D}$, *then,* $\otimes\{v \,[\![\, v \leftarrow \mathbb{V}_\mathcal{D}, \models_v^{[\mathcal{I}_\mathcal{D}]} \mathcal{D}\}$ *is the least extended validity valuation that satisfies* $\mathcal{D}$.

Proof. *Let* $v_\top$ *be them greatest extended validity valuation in* $\mathbb{V}_\mathcal{D}$; v *also satisfies* $\mathcal{D}$, *by construction, therefore the comprehension above is not empty. Thus, the theorem holds as a consequence of Lemma 3.2 and the definition of the least extended validity valuation.* □

Claim. Assume now that a CID system $\mathcal{P} = \langle \langle IDB, EDB \rangle, \{eep\} \rangle$ where *eep* is extended elementary production (See Section 3.3.1). Assume also that v is an extended validity valuation that satisfies $\mathcal{P}$. An extended validity valuation v_{HB} that satisfies the corresponding p-Datalog program *IDB* can be obtained simply by reducing the range of v to the Herbrand base of $\mathcal{L}_\mathcal{P}$.

Proof. *The claim above holds if for every ground instance* $h \leftarrowtail C_0 \vdash l_1, l_2, \ldots, l_n$, $l_1 \leftarrowtail C_1 \vdash$, $l_2 \leftarrowtail C_2 \vdash, \ldots, l \leftarrowtail C_n \vdash$ *of p-clauses in* $\mathcal{L}_\mathcal{D}$:

$$\models_v^{[eep]} \{\!\{(h \leftarrowtail C_0 \vdash l_1, l_2, \ldots, l_n), (l_1 \leftarrowtail C_1 \vdash), (l_2 \leftarrowtail C_2 \vdash) \ldots, (l_n \leftarrowtail C_n \vdash)\}\!\} \tag{5}$$

implies that

$$\models_{v_{HB}} (h \leftarrowtail C_0 \vdash l_1, l_2, \ldots, l_n) \tag{6}$$

However, from Definition 3.1 and (5) it holds that

$$v(h \leftarrow) \geq g_{eep}(v(h \leftarrow l_1, l_2, \ldots, l_n), v(l_1 \leftarrow), v(l_2 \leftarrow), \ldots, v(l_n \leftarrow)$$

and by definition of g_{eep},

$$g_{eep} = f^{\leftarrow}(v(h \leftarrow l_1, l_2, \ldots, l_n), f^\wedge(v(l_1 \leftarrow), v(l_2 \leftarrow), \ldots, v(l_n \leftarrow))$$

and hence,

$$v(h \leftarrow) \geq f^{\leftarrow}(v(h \leftarrow l_1, l_2, \ldots, l_n), f^\wedge(v(l_1 \leftarrow), v(l_2 \leftarrow), \ldots, v(l_n \leftarrow))$$

since v *is, by assumption, acceptable to* $\langle IDB, EDB \rangle$, $v(h \leftarrow) = C_0$ *which implies from the definition of satisfiability in [LSV01] in (6). The same argument extends to every subset of p-clauses in,* $\mathcal{L}_\mathcal{D}$, *and, in turn, to every p-Datalog program. Therefore,* v_{HB} *from the Herbrand base of a p-Datalog program* $\mathcal{P}$ *to the same image* $\mathbb{C}$ *is also a validity valuation that satisfies* $\mathcal{P}$. □

Corollary B.1. *Let $\mathcal{D} = \langle IDB, EDB\rangle$ and $\mathcal{S} = \langle \mathcal{D}, \{eep\}\rangle$ be, respectively, a CID and a CID system in which every p-clause in $\mathcal{L}_\mathcal{D}$ is annotated with maximum validity $\top$, and let $v : \mathcal{L}_\mathcal{D} \rightarrow \{\top, \bot\}$ be an extended validity valuation. If v satisfies $\mathcal{D}$ it also gives rise to a Herbrand interpretation for the deductive database that corresponds to $\mathcal{D}$.*

Proof. *This holds because: (1) v assign $\top$ to all ground facts in EDB (because if v satisfies $\mathcal{D}$, then v is assumed to be acceptable to $\mathcal{D}$) (2) v assigns no less than $\top$ to the deductive eep-inferrable consequences of $\mathcal{D}$ (because of Definition 3.1 and because the validity combinators are bounded from above, thus, assigning $\top$ to all eep-inferrable consequences of $\mathcal{D}$). Moreover, the least extended validity valuation w assigns maximum validity assessment to (and only to) those facts that are deductive consequences of $\mathcal{D}$, therefore w can be equated to the minimal model of $\mathcal{D}$.* □

B.2 Fixpoint Semantics

Lemma B.3. *The immediate consequence operator $T_\mathcal{D}$ is monotone and continuous.*

Proof. *Let $\mathcal{D} = \langle IDB, EDB\rangle$ be a CID and $\mathcal{D} = \langle \mathcal{D}, \mathcal{I}_\mathcal{D}\rangle$ a CID system. The first step is to show that $T_\mathcal{D}$ is monotone. Let u and v be any valuations acceptable to $\mathcal{D}$ such that $u \leq v$. Then, it should hold that $T_\mathcal{D}(u) \leq T_\mathcal{D}(v)$. Let Q be any clause in $\mathcal{L}_\mathcal{D}$, and $f^\vee$ be the disjunctive combinator associated with Q. Then, by Definition 3.2, for any CID inference rule $\rho \in \mathcal{I}_\mathcal{D}$,*

$$\begin{aligned} T_\mathcal{D}(u)(Q) &= f^\vee(\{\!\{g_\rho(u(\widehat{S}_1), u(\widehat{S}_2), \ldots, u(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \\ T_\mathcal{D}(v)(Q) &= f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \end{aligned}$$

Since $u \leq v$, it holds that $u(A) \leq v(A)$, for every clause in $A \in \mathcal{L}_\mathcal{D}$. In particular, $u(S_k) \leq v(S_k)$, for $1 \leq k \leq n$. Since, $f^\leftarrow$ and g_ρ are assumed to be monotone, it holds that:

$$\begin{aligned} & f^\vee(\{\!\{g_\rho(u(\widehat{S}_1), u(\widehat{S}_2), \ldots, u(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \\ \leq & \\ & f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \end{aligned}$$

Therefore,

$$T_\mathcal{D}(u)(Q) \leq T_\mathcal{D}(v)(Q)$$

whereby, $T_\mathcal{D}$ is concluded to be monotone.

The next step is to show that $T_\mathcal{D}$ is continuous. Let $v_0 \leq v_1 \leq \ldots$ be any chain of valuations acceptable to $\mathcal{D}$, and Q be any clause in $\mathcal{L}_\mathcal{D}$. It holds that $T_\mathcal{D}$ is continuous, if it holds that:

$$T_\mathcal{D}(\oplus\{v_j(Q) \,[\![\, j \geq 0\}) \quad = \quad \oplus\{T_\mathcal{D}(v_j)(Q) \,[\![\, j \geq 0\} \tag{7}$$

Firstly, consider the left hand side of equation (7). By Lemma 3.1, since the set $\mathbb{V}_\mathcal{D}$ of valuations of $\mathcal{D}$, ordered by $\leq$, is a complete lattice, the chain $v_0 \leq v_1 \leq \ldots$ has a least upper bound, say v. That is, $v = \oplus\{v_j \,[\![\, j \geq 0\}$. Since, every CID inference rule $\rho \in \mathcal{I}_\mathcal{D}$ take only a finite set of premisses, say $S_1, S_2, \ldots, S_n$ to infer the consequence Q, and p-Datalog languages are function free, then there is finite number of ground instances of $S_1, S_2, \ldots, S_n$. Moreover, since the same disjunctive combinator, say $f^\vee$, is associated with Q, then it holds, by Definition 3.2 that:

$$T_\mathcal{D}(v)(A) = f^\vee(\{\!\{g_\rho(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\})$$

Secondly, consider the right hand side of equation (7), it is shown next to be identical to the last expression above. By Definition 3.2,

$$\begin{aligned} & \oplus\{T_\mathcal{D}(v_j)(Q) \,[\![\, j \geq 0\} = \\ & \oplus\{f^\vee(\{\!\{g_\rho(v_j(\widehat{S}_1), v_j(\widehat{S}_2), \ldots, v_j(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \,[\![\, j \geq 0\} \end{aligned}$$

Since, $f^\vee$ and g_ρ are assumed to be monotone, it holds that, the merge operator $\oplus$ can be pushed inside the expression while maintaining equality. Doing so, it holds that:

$$\begin{aligned} & \oplus\{f^\vee(\{\!\{g_\rho(v_j(\widehat{S}_1), v_j(\widehat{S}_2), \ldots, v_j(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \,[\![\, j \geq 0\} = \\ & f^\vee(\{\!\{g_\rho(\oplus\{v_j \,[\![\, j \geq 0\}(\widehat{S}_1), \ldots, \oplus\{v_j \,[\![\, j \geq 0\}(\widehat{S}_n)) \mid S_1, \ldots, S_n \in \mathcal{L}_\mathcal{D}\}\!\}) \end{aligned}$$

However, v is the least upper bound of the chain, then $v = \oplus\{v_j \parallel j \geq 0\}$, hence,

$$T_{\mathcal{D}}(v)(A) = f^{\vee}(\{\!\{g_{\rho}(v(\widehat{S}_1), v(\widehat{S}_2), \ldots, v(\widehat{S}_n)) \mid S_1, S_2, \ldots, S_n \in \mathcal{L}_{\mathcal{D}}\}\!\})$$

which makes both side of equation (7) identical and concludes that continuity also holds. □

Lemma B.4. *Let $\mathcal{D} = \langle\langle IDB, EDB\rangle, \mathcal{I}_{\mathcal{D}}\rangle$ be a CID system and v be any extended validity valuation acceptable to $\mathcal{D}$. Then $\models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}$ if and only if $T_{\mathcal{D}}(v) \leq v$.*

Proof. *The lemma follows from Definitions 3.1 and 3.2 upon noticing that $lfp(T_{\mathcal{D}}) = \otimes\{v \parallel v \leftarrow \mathbb{V}_{\mathcal{D}}, \models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}\}$, as demonstrated in Theorem B.2.* □

B.3 Semantic Equivalence

Theorem B.2. *Let $\mathcal{D} = \langle\langle IDB, EDB\rangle, \mathcal{I}_{\mathcal{D}}\rangle$ be a CID system. Then*

$$lfp(T_{\mathcal{D}}) = \otimes\{v \parallel v \leftarrow \mathbb{V}_{\mathcal{D}}, \models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}\}$$

Proof. *This theorem is analogous to the van Emden-Kowalski theorem in standard logic programming [vEK76], so the same argumentation used there can be followed here assuming the immediate consequence operator is monotone and continuous (cf. lemma B.3).* □

Theorem B.3. ***(Soundness)*** *Let $\mathcal{S} = \langle\mathcal{D}, \mathcal{I}_{\mathcal{D}}\rangle$ be a CID system. Let Q be any p-clause in the outcome of a CID task. Then, if $Q \notin \mathcal{D}$ is annotated with validity assessment C then $C \leq lfp(T_{\mathcal{D}})(Q)$.*

Proof. *Suppose k is the height of CID task Ξ over $\mathcal{D}$. The proof of soundness follows from induction on k. Let $\mathcal{D} = \langle IDB, EDB\rangle$, and $f^{\vee}$ be the disjunctive combinator associated with Q.*

Base case *$k = 1$. Since height of Ξ is 1, it can have only one internal node that denotes, say $\Phi(\rho)$ where $\rho \in \mathcal{I}_{\mathcal{D}}$. Thus, ρ which chooses the premisses scanning subsets of $\mathcal{L}_{\mathcal{D}}$. If Q with validity assessment C is contained in the result of Ξ, $Q \notin \mathcal{D}$, so, Q is inferred by ρ, and aggregated by Φ, in such way that $C = f^{\vee}(\{\!\{C_1, C_2, \ldots, C_m\}\!\})$ where each C_j is computed using g_{ρ} from an alternative bag of premisses $S_1^j, S_2^j, \ldots, S_n^j$ from $\mathcal{D}$. Since, all candidate premisses are also in $\mathcal{L}_{\mathcal{D}}$, then by Definition 3.4, $lfp(T_{\mathcal{D}})(Q) = f^{\vee}(X)$, where $\{\!\{C_1, C_2, \ldots, C_m\}\!\} \subseteq X$. From this it holds that $C \leq lfp(T_{\mathcal{D}})(Q)$, upon noting that $f^{\vee}$ and $T_{\mathcal{D}}$ are monotone.*

Inductive Hypothesis *Suppose that for any p-clause Q annotated with validity assessment C that is contained in result of a CID task of height at most k, $C \leq lfp(T_{\mathcal{D}})(Q)$.*

Inductive Step *Suppose Ξ is of height $k+1$, and its outermost monoid comprehension (i.e., the root node) denotes $\Phi(\rho)$ where $\rho \in \mathcal{I}_{\mathcal{D}}$. Thus, ρ chooses its premisses scanning collections $D_1, D_2, \ldots, D_n$ containing the consequences that are either in $\mathcal{D}$ or inferred by CID tasks of height at most k (i.e.,* consequences *of $\mathcal{D}$). If Q with validity assessment C is contained in the result of Ξ, Q is inferred by ρ and aggregated by Φ, in such way that $C = f^{\vee}(\{\!\{C_1, C_2, \ldots, C_m\}\!\})$ where each C_j is computed using g_{ρ} from an alternative bag of premisses $S_1^j, S_2^j, \ldots, S_n^j$ for $0 < j \leq m$ and $0 \leq i \leq n$. By the inductive hypothesis, any clause S' drawn from any D_i is annotated with a validity assessment C' such that $C' \leq lfp(T_{\mathcal{D}})(S')$, in particular if S_i^j is a premiss in one alternative derivation j of Q. Then it follows from the inductive hypothesis and Definition 3.4 that $lfp(T_{\mathcal{D}})(Q) = f^{\vee}(X)$, where $\{\!\{C_1, C_2, \ldots, C_m\}\!\} \subseteq X$. Also by the inductive hypothesis it is assumed that $C_j \leq lfp(T_{\mathcal{D}})(S_i^j)$, for $0 < j \leq m$. Since, g_{ρ} and $T_{\mathcal{D}}$ are monotone, $f^{\vee}$ is monotone and bounded-below, then $C_j \leq C = f^{\vee}(\{\!\{C_1, C_2, \ldots, C_m\}\!\})$ entails that $C \leq lfp(T_{\mathcal{D}})(Q)$.*

□

Theorem B.4. ***(Completeness)*** *Let $\mathcal{S} = \langle\mathcal{D}, \mathcal{I}_{\mathcal{D}}\rangle$ be a CID system. Let Q be any p-clause such that $lfp(T_{\mathcal{D}})(Q) = T_{\mathcal{D}}^k(Q)$, for some integer k. If $Q \notin \mathcal{D}$, then there exists a CID task that, if evaluated, will contain Q and will annotate Q with a validity value C such that $lfp(T_{\mathcal{D}})(Q) \leq C$.*

Proof. *The proof of completeness proceeds by induction on the iteration step k. Without loss of generality, assume that $lfp(T_{\mathcal{D}})(Q) \neq \bot$; otherwise, the result trivially holds. Let $\mathcal{D} = \langle IDB, EDB\rangle$, and $f^{\vee}$ be the disjunctive combinator associated with Q.*

Base case *For $k = 1$, $lfp(T_{\mathcal{D}})(Q) = T_{\mathcal{D}}(v)(Q)$, for the least extended validity valuation v of $\mathcal{D}$, i.e., $v = \otimes\{v \parallel v \leftarrow \mathbb{V}_{\mathcal{D}}, \models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}\}$. In this case, at least one CID inference rule must be applicable, let it be $\rho \in \mathcal{I}_{\mathcal{D}}$, to infer Q as consequence, otherwise $lfp(T_{\mathcal{D}})(Q) = \bot$, since, by assumption, $Q \notin \mathcal{D}$. By Definition 3.2, there exists, at least, one bag of premisses, say $\{\!\{S_1^j, S_2^j, \ldots, S_n^j\}\!\}$ for $0 < j \leq m$ contained in $\mathcal{L}_{\mathcal{D}}$ that allow ρ to infer Q. Then, a CID task can be construed as follows. It has only one internal node (also the root node), viz., the monoid comprehension expression that denotes $\Phi(\rho)$. Generator domains in the monoid comprehension that denotes ρ scan either IDB or EDB for premisses, the filters implement the formal justification prescribed by ρ. The head of the monoid comprehension that denotes ρ ejects Q with validity assessment computed using g_ρ. More precisely, for each alternative derivation j of Q from premisses $S_1^j, S_2^j, \ldots, S_n^j$, the validity assessment of Q is $C_j = g_\rho(\{\!\{C_1^j, C_2^j, \ldots, C^j{}_n\}\!\})$ for $0 < j \leq m$. In the head of Φ, all alternative validity assessments C_j for Q are aggregated by $f^\vee$ so as to $C = f^\vee(\{\!\{C_1, C_2, \ldots, C_m\}\!\})$ be the definitive validity assessment computed for Q. Hence, the constructed CID task annotates Q with C, which is the same value as $T_{\mathcal{D}}(v)(Q)$, therefore, $lfp(T_{\mathcal{D}})(Q) \leq C$ holds.*

Inductive Hypothesis *Suppose that for some $k > 1$, whenever $lfp(T_{\mathcal{D}})(Q) = T_{\mathcal{D}}^k(Q)$, then, there exists a CID task that contains a p-clause Q, and annotates Q with a validity assessment C and $lfp(T_{\mathcal{D}})(Q) \leq C$.*

Inductive Step *Suppose $lfp(T_{\mathcal{D}})(Q) = T_{\mathcal{D}}^{k+1}(v)(Q)$, for the least extended validity valuation v of $\mathcal{D}$, i.e., $v = \otimes\{v \parallel v \leftarrow \mathbb{V}_{\mathcal{D}}, \models_v^{[\mathcal{I}_{\mathcal{D}}]} \mathcal{D}\}$. In this case, at least one CID inference rule must be applicable, let it be $\rho \in \mathcal{I}_{\mathcal{D}}$, to infer Q as consequence, otherwise $lfp(T_{\mathcal{D}})(Q) = \bot$, again, by assumption, $Q \notin \mathcal{D}$. By Definition 3.2, there exists, at least, one bag of premisses, say $\{\!\{S_1^j, S_2^j, \ldots, S_n^j\}\!\}$ for $0 < j \leq m$ contained in $\mathcal{L}_{\mathcal{D}}$ that allow ρ to infer Q. Since $lfp(T_{\mathcal{D}})(Q) = T_{\mathcal{D}}^{k+1}(v)(Q)$, it must be that $lfp(T_{\mathcal{D}})(S_i^j) = T_{\mathcal{D}}^k(v)(S_i^j)$, for $0 < j \leq m$ and $0 \leq i \leq n$. Thus, by the inductive hypothesis, there exists a CID task say Ξ_i which contains in its results all premiss S_i^j annotated with validity assessment C_i^j. Then, a CID task can be construed as follows. It has as the root node the monoid comprehension that denotes $\Phi(\rho)$. Generator domains in the monoid comprehension that denotes ρ scan either IDB or EDB for premisses, the filters implement the formal justification prescribed by ρ. The head of the monoid comprehension that denotes ρ ejects Q with validity assessment computed using g_ρ. More precisely, for each alternative derivation j of Q from premisses $S_1^j, S_2^j, \ldots, S_n^j$, the validity assessment of Q is $C_j = g_\rho(\{\!\{C_1^j, C_2^j, \ldots, C^j{}_m\}\!\})$ for $0 < j \leq m$. In the head of Φ, all alternative validity assessments C_j for Q are aggregated by $f^\vee$ so as to $C = f^\vee(\{\!\{C_1, C_2, \ldots, C_m\}\!\})$ be the definitive validity assessment computed for Q. Also by the inductive hypothesis, it is assumed that $lfp(T_{\mathcal{D}})(S_i^j) \leq C_i^j$. Since g_ρ, $f^\vee$ and $T_{\mathcal{D}}$ are monotone, then it must hold also that $lfp(T_{\mathcal{D}})(Q) \leq C$.*

□

C A Prototype CID System

A CID prototype was built from scratch instead of adapting or extending existing software. The main reasons ware to have full control of the source code, and to be able to quickly prototype and explore ideas within the resources and time scope of this research. Thus, open source deductive databases were not considered, since those would likely be prone to deductive inferences, while a more neutral approach is more desirable. The natural candidate for extension would be **λ-DB** [FSRM00] for its complete support for monoid comprehensions, however, complex software package with more features and more stringent requirement of robustness and reliability than necessary for a quick prototype of CID (e.g., the **shore** object management store, and the support for OQL, and ODMG language bindings). Nevertheless, **λ-DB** provided the inspiration and some algorithms that were used in the implementation of the prototype, but no actual code.

C.1 The Modules

A high-level view of its implementation is depicted as a package diagram [Obj00] in Figure 11 which matches the architecture in depicted Figure 6.

The packages denote high-level components while arrows denote functional dependency between packages, i.e., class or functions in one package refer to class or functions in another. Dependency arrows provide a measure for coupling between the components of the prototype.

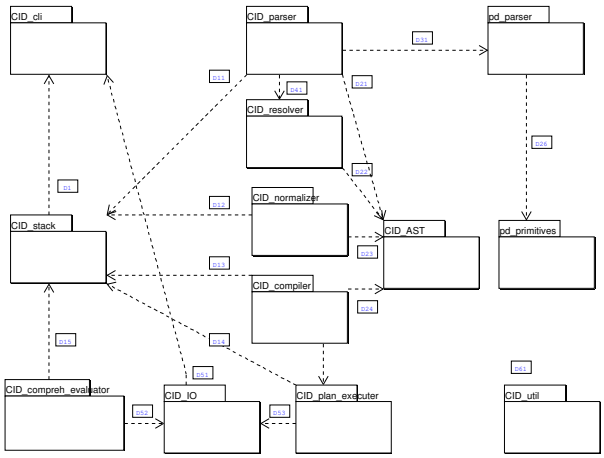


Figure 11: The Modules that compose the CID Prototype

1. `CID_cli` implements a simple command line shell that allow the functionalities of CID engine invoked according to the usage model in Section 5.1.

2. `CID_stack` is the central component and defines the API of the CID engine, including the basic functionalities: parsing, normalizing, compiling, and evaluating tasks.

3. `CID_util` has a miscellaneous of auxiliary classes and functions, e.g., for handling dictionaries, stacks, that are useful in other modules.

4. `CID_parser` implements a parser for monoid comprehension expressions, as in Section 5.4.2 (cf. Section 5.4.2).

5. `CID_AST` contains all the classes that define nodes of the abstract syntax tree, providing a unique API for all subsequent processing steps.

6. `CID_resolver` enables the parser to retrieve a plugged CID inference rules when it is referred to in the task being parsed (cf. Section 5.4.3).

7. `CID_normaliser` contains the normalisation rules (in [FM00]) that transform monoid comprehension expressions into their canonical form by restructuring the abstract syntax tree created in (4) (cf. Section 5.4.4)

8. `CID_compiler` contains the compilation and unnesting algorithm (in [FM00]) that translates monoid comprehensions expression into monoid algebra execution plans (cf. Sections 5.4.4). It also contains physical optimisation procedures (cf. Section 5.4.4).

9. `CID_plan_executer` contains the definition of the logical operators in the monoid algebra and their physical counter-parts. The physical operators implement a pipelined iterator based semantics that retrieve p-clauses from `CID_IO` (cf. Section 5.4.5).

10. `CID_compreh_evaluator` maps monoid comprehensions into Python comprehension and invoke them, thereby providing an alternative to evaluate tasks without normalizing and compile them into monoid algebra (cf. Section 5.4.6).

11. `CID_IO` implements a simple API wrapping the basic primitives of storage and retrieval, and also allowing the load, back-up, restore, and save a CID into textual files (cf. Section 5.4.5).

12. `pd_primitives` contains the classes that define p-clauses as a type (cf. Section 5.3).

13. `pd_parser` implements a simple parser for p-Datalog clauses.

C.2 Dependencies and Interactions

The dependency arrows in Figure 11 are described as follows.

1. The arrow labeled D1 denotes the link between the command line interface and API to the basic functionalities of the CID engine.
2. Arrows from D11 to D15 denote the functional dependency the API in (1) and the modules that implement each functionality.
3. Arrows from D21 to D24 denote dependency between the modules in (2) and the internal representation for monoid comprehensions in `CID_AST`.
4. The arrow D31 denotes that the parser for monoid comprehensions may use the parser for p-clauses, when a clause is identified as a term in a monoid comprehension expression.
5. The arrow D41 denotes the use that a parser may do of the resolver that retrieves plugged CID inference rules.
6. Arrows D51 to D53 denote the use of the storage and retrieval primitives provided by `CID_IO`.
7. The label D61 denotes widespread dependency on the basic functions and classes in `CID_util`, although the corresponding arrows are omitted.

Note that CID engine was implemented to be independent of p-Datalog clauses, insofar as it can be regarded as general purpose database engine based on monoid comprehension calculus and algebra. In a sense, a this implementation tries to resemble a stripped version of **λ-DB**, so as to give evidence that a CID can be built upon existing technology.

C.3 Data Structures

In the implementation of a CID engine prototype, the representation of p-clauses as specified in Section 5.3 is implemented, in an object-oriented style, as depicted in class diagram [Obj00] in Figure 12.

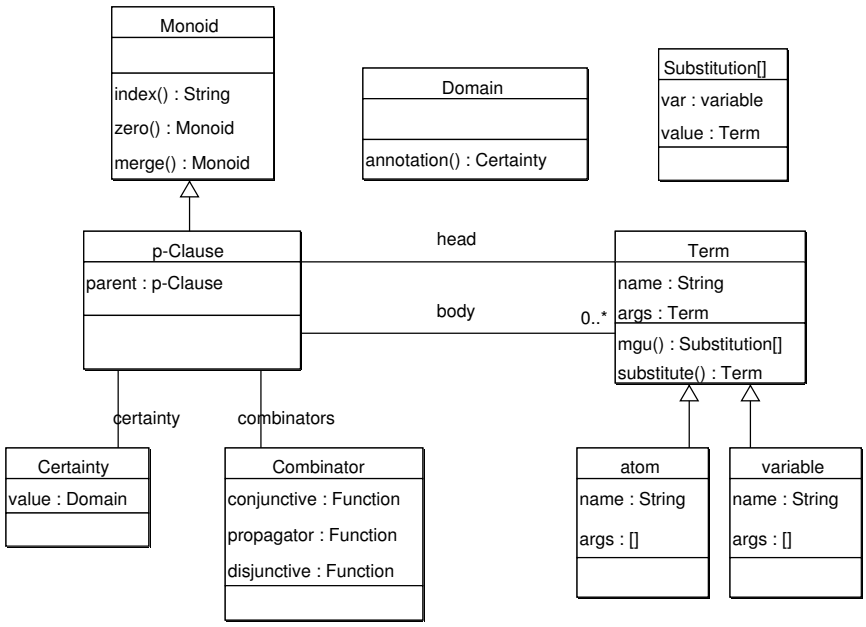


Figure 12: CID Primitive Types

C.4 The Command Line Interface

This section shown a sample of command exposed by the command-line interface of the prototype CID engine.

```
CID - [Version 0.9 from 03/03/2003]
Started on Thu, 27 May 2004 01:29:43 BRT
(CID) help

Documented commands (type help <topic>):
========================================
definition  execution  help  leave  manipulation  note  quit

(CID) help definition
(*H*) Type 'definition' if you want to enter the definition mode
(CID) help manipulation
(*H*) Type 'manipulation' if you want to enter the manipulation mode
(CID) help execution
(*H*) Type 'execution' if you want to enter the execution mode
(CID) definition
(-D-) help

Documented commands (type help <topic>):
========================================
backup  clear   display  help    leave  note  pwd   recover
cd      delete  dump     insert  load   plug  quit  unplug

(-D-) help load
(*H*) Type 'load <filename>' if you want to append
(*H*) the contents of <filename> to the global CID_BUFFER.
(-D-) help plug
(*H*) Type 'plug [<filename>]' if you want to add
(*H*) the contents of the global CID_BUFFER to the working
(*H*) set of inference rules, backing these up onto <filename>
(*H*) (The current default <filename> is 'tmp/cid_saved_inference_rules'.
(-D-) help unplug
(*H*) Type 'unplug' if you want to revert to
(*H*) an empty working set of inference rules
(-D-) help display
(*H*) Type 'display' if you want to see the entire contents of
(*H*) the plugged inference rules
(-D-) quit
(CID) manipulation
(-M-) help

Documented commands (type help <topic>):
========================================
backup  cd     delete   download  help    leave  note  quit     uncan
can     clear  display  dump      insert  load   pwd   recover  upload

(-M-) help insert
(*H*) Type 'insert <expr>' if you want to add <expr> to
(*H*) the contents of the global CID_BUFFER.
(-M-) help delete
(*H*) Type 'delete <key>' if you want to remove the expression
(*H*) with key <key> from the contents of the global CID_BUFFER.
(-M-) help upload
(*H*) Type 'upload [i|e]' if you want to make
(*H*) the contents of the global CID_BUFFER become either
(*H*) the _i_ntensional or the _e_xtensional database.
(-M-) help download
(*H*) Type 'download [i|e]' if you want to set
(*H*) the contents of the global CID_BUFFER to become either
(*H*) the _i_ntensional or the _e_xtensional database.
(-M-) help display
(*H*) Type 'display [i|e|t]' if you want to see the entire contents of
(*H*) the _i_ntensional, the _e_xtensional database or canned _t_asks.
(-M-) quit
(CID) execution
(-E-) help

Documented commands (type help <topic>):
========================================
cd     delete   display  enable    evaluator  help    leave  note  quit  set
```

```
clear  disable  dump     evaluate  get        insert  load   pwd   save

(-E-) help evaluator
(*H*) Type 'evaluator <C|A>' if you want to troggle between
(*H*) a monoid Calculus or a monoid Algebra evaluator engine.
(-E-) help evaluate
(*H*) Type 'evaluate' to evaluate the tasks currently in the GLOBAL BUFFER.
(-E-) help set
(*H*) Type 'set <arg>=<val>' if you want to bind <arg> to <val>.
(-E-) help get
(*H*) Type 'get <arg>' if you want to see what <val> is <arg> bound to.
(*H*) Type 'get' only if you want to see all bindings <val> == <arg>.
(-E-) help cd
(*H*) Type 'cd <directory>' if you want to change
(*H*) the folder where files are loaded, restored from,
(*H*) and backed-up, saved to.
(-E-) help pwd
(*H*) Type 'pwd' if you want to see the folder
(*H*) where files are loaded, restored from,
(*H*) and backed-up, saved to.
(-E-) help save
(*H*) Type 'save <file_name>' if you want to save the results of
(*H*) evaluating of the tasks in the GLOBAL BUFFER into a file.
(-E-) help display
(*H*) Type 'display' if you want to display the results of
(*H*) evaluating of the tasks in the GLOBAL BUFFER.
(-E-) quit
```

C.5 Abstract Syntax

The internal representation used for CID inference rules in the prototype is described by the following grammar.

```
named_task : declaration ≡ task
task : collection_monoid_comprehension
 | scalar_monoid_comprehension
 | logic_monoid_comprehension
 | ( collection_monoid_comprehension collection_monoid task )
 | ( scalar_monoid_comprehension scalar_monoid task )
 | ( logic_monoid_comprehension logic_monoid task )
 | functional_monoid_comprehension ( expression )
collection_monoid_comprehension :
   collection_monoid ⊕{ scalar_expression ⟦ qualifier_list }
 | collection_monoid ⊕{ collection_expression ⟦ qualifier_list }
scalar_monoid_comprehension :
   scalar_monoid ⊕{ scalar_expression ⟦ qualifier_list }
logic_monoid_comprehension :
   logic_monoid ⊕{ logic_expression ⟦ qualifier_list }
functional_monoid_comprehension :
   functional_monoid ⊕{ lambda_expression ⟦ qualifier_list }
qualifier_list : qualifier
 | qualifier_list , qualifier
qualifier : generator
 | binding
 | predicate
generator : var ⟵ variable
 | var ⟵ collection_expression
binding : var ≡ expression
predicate : logic_expression
 | ¬ logic_expression
collection_monoid : ∪
 | ⊎
 | ⧺
scalar_monoid : +
 | max
logic_monoid : ∧
 | ∨
functional_monoid : ∘
expression : collection_expression
 | scalar_expression
 | logic_expression
 | functional_expression
```

```
collection_expression : collection_monoid_comprehension
 | collection_monoid ⊕{ }
 | collection_monoid ⊕{ scalar_expression_list }
 | collection_monoid ⊕{ collection_expression_list }
 | IF logic_expression THEN collection_expression ELSE collection_expression
 | ( collection_expression collection_monoid collection_expression )
 | ( variable collection_monoid collection_expression )
 | functor
scalar_expression : scalar_monoid_comprehension
 | IF logic_expression THEN scalar_expression ELSE scalar_expression
 | NULL
 | ▽
 | ⊥
 | ⊤
 | variable
 | functor
 | tuple
 | numeric_expression
logic_expression : logic_monoid_comprehension
 | comparison
 | TRUE
 | FALSE
functional_expression : lambda_expression ( expression )
lambda_expression : λ quantified_var_list . expression
numeric_expression : variable
 | number
 | numeric_expression + numeric_expression
 | numeric_expression − numeric_expression
 | numeric_expression * numeric_expression
 | numeric_expression / numeric_expression
comparison : scalar_expression < scalar_expression
 | scalar_expression > scalar_expression
 | scalar_expression ≤ scalar_expression
 | scalar_expression ≥ scalar_expression
 | scalar_expression = scalar_expression
 | scalar_expression ≠ scalar_expression
tuple : ⟨ ⟩
 | ⟨ expression_list ⟩
declaration : const ( expression_list )
functor : path_expression ( expression_list )
path_expression : const
 | path_expression . const
variable : var
 | var . path_expression

collection_expression_list : collection_expression
 | collection_expression_list , collection_expression
scalar_expression_list : scalar_expression
 | scalar_expression_list , scalar_expression
expression_list : expression
 | expression_list , expression
quantified_var_list : var
 | quantified_var_list , var
var : '[A-Z][\w_]*'
const : '[a-z][\w_]*'
number: '\d+(\.\d+)?'
```

The parsing of a monoid comprehension expression, according to the grammar above, yields an abstract syntax tree that is acted upon by all subsequent steps in its evaluation.

C.6 A DCG for the Language Generator

The language generator component is defined by the following definite clause grammar:

```
/* grammar rules (fixed part) */
datalog_clause(datalog_clause(head(Head),body(Body))) -->
  head(head(Head),HVarsP,HVarsM), [':-'],
   {'@bodysize'(N)},
  body(N,[Head],body(Body),HVarsM,HVarsP).
head(head(HeadLit),HVarsP,HVarsM)    -->
  hlit(HeadLit,HVarsP,HVarsM).
body(0,RBody,body(Body),_VarsP,_VarsM)           --> [].
body(N,Body,body(NewBody),VarsSoFarP,VarsSoFarM) -->
   blit(BodyLit,VarsSoFarP,LitVarsP,VarsSoFarM,LitVarsM),
   {Nminus1 is N - 1,
     \+member(BodyLit,Body)}, %% avoid duplicates.
   body(Nminus1,[BodyLit|Body],body(NewBody),LitVarsP,LitVarsM).
head_literal(literal(Lit),HVarsP,HVarsM)         -->
  literal(head,Lit,[],HVarsP,[],HVarsM).
body_literal(literal(Lit),VarsSoFarP,VarsP,VarsSoFarM,VarsM) -->
  literal(body,Lit,VarsSoFarP,VarsP,VarsSoFarM,VarsM).

literal(Pos,predicate(symbol(P/0),args([])),VarsP,VarsP,VarsM,VarsM) -->
  predicate(Pos,P,0,[]).
literal(Pos,predicate(symbol(P/A),args(Args)),VarsSP,VarsP,VarsSM,VarsM)-->
  predicate(Pos,P,A,Args),
  ['('],
  terms(A,Args,VarsSP,VarsP,VarsSM,VarsM),
  [')'].
terms(1,[Arg],VarsSoFarP,NewVarsP,VarsSoFarM,NewVarsM) -->
  term(Arg,VarsSoFarP,NewVarsP,VarsSoFarM,NewVarsM).
terms(N,[Arg|Args],VarsSoFarP,NewVarsP,VarsSoFarM,NewVarsM) -->
  term(Arg,VarsSoFarP,ArgVarsP,VarsSoFarM,ArgVarsM),
   {dec(N,Nminus1)},
  terms(Nminus1,Args,ArgVarsP,NewVarsP,ArgVarsM,NewVarsM).
term(const(Type,C),VarsP,VarsP,VarsM,VarsM)             -->
  const(Type,C).
term(var(Type,Mode,Var),VarsP,NewVarsP,VarsM,NewVarsM)  -->
  var(Type,Mode,Var,VarsP,NewVarsP,VarsM,NewVarsM).
predicate(Pos,P,A,Args)                        --> [P],
   {'@predicate'(Pos,P,A,Args),length(Args,A)}.
```

Note that the declaration of types and modes (viz., *input* and *output*) is meant to help reducing significantly the cardinality of the set of clauses generated [MdR94].

```
const(Type,C)                                  --> [C],
 {'@const'(Type,ConstList),
   '@type'(Type),
   member(C,ConstList)}.
var(Type,+,Var,VarsP,VarsP,VarsM,VarsM)        --> [Var],
 {member(Var,VarsP),
   '@var'(Type,VarList),
   '@type'(Type),
   member(Var,VarList)}.
var(Type,-,Var,VarsP,VarsP,VarsM,VarsM)        --> [Var],
 {member(Var,VarsM),
   '@var'(Type,VarList),
   '@type'(Type),
   member(Var,VarList)}.
var(Type,+,Var,VarsP,[Var|VarsP],VarsM,VarsM) --> [Var],
 {'@var'(Type,VarList),
   '@type'(Type),
   next(VarsP,VarsM,Var,VarList)
   }.
var(Type,-,Var,VarsP,VarsP,VarsM,[Var|VarsM]) --> [Var],
 {'@var'(Type,VarList),
   '@type'(Type),
     next(VarsM,VarsP,Var,VarList)
   }.

/* input parameters */
/* control parameters */
'@bodysize'(N).
'@type'(Type).
/* lexicon schema */
'@const'(Type,ConstList).
'@var'(Type,VarList).
'@predicate'(Pos,P,A,Args).
```

C.7 Physical-Algebraic Operators

The pseudocode for the physical operators used to implemented the prototype is listed below.
`print(arg)`

```
  open()
    return arg.open()

  next(crs)
    input = arg.next(crs)
    if input <> null
         print input
    return

  close(crs)
    arg.close(crs)
```

```
scan(pred,arg)

  open()
    return arg.open()

  next(crs)
    input = arg.next(crs)
    while input <> null
        if pred(input)
            break
        input = arg.next(crs)
    return input

  close(crs)
    arg.close()


bag_union(left_arg,right_arg)

  open()
    lcrs = left_arg.open()
    rcrs = right_arg.open()
    current = left_arg           # status
    return (lcrs,rcrs)

  next((lcrs,rcrs))
    if current == left_arg
        input = left_arg.next(lcrs)
        if input == null # left exhausted
            current = right_arg
            input = right_arg.next(rcrs)
    else
        input = right_arg.next(rcrs)
    if input <> null
          return input

  close((lcrs,rcrs))
    left_arg.close(lcrs)
    right_arg.close(rcrs)


reduce(⊕,eject,pred,arg)

  open()
    output = Z⊕          # status
    crs = arg.open()
    input = arg.next(crs)
    if not ⊕ is collection
        while input <> null
            if pred(input)
                  output = output ⊕ U⊕(eject(input))
            input = arg.next(crs)
        arg.close(crs)
        output = [output]          # transform scalar into collection monoid
    return output.open()

  next(crs)
    return output.next()

  close(crs)
    output.close()
```

```
nested_loop(pred,left_arg,right_arg)

  open()
    lcrs = left_arg.open()
    rcrs = right_arg.open()       # status
    current = left_arg.next(lcrs) # status
    return lcrs

  next(lcrs)
    if current == null
        return null
    r = right_arg.next(rcrs)
    while TRUE
        if r == null             # right exhausted
            right_arg.close(rcrs)
            current = left_arg.next(lcrs)
            if current == null
                return null      # both exhausted
            rcrs = right_arg.open()
            r = right_arg.next(rcrs)
        else
            (larg,rarg) = merge_tuple(current,r)
            if pred((larg,rarg))
                    return (larg,rarg)
            r = right_arg.next(rcrs)

  close(lcrs)
    left_arg.close(lcrs)
    right_arg.close(rcrs)
```

```
outer_nested_loop(pred,left_arg,right_arg)

  open()
    lcrs = left_arg.open()
    rcrs = right_arg.open()       # status
    current = left_arg.next(lcrs) # status
    any = FALSE                   # status
    return lcrs

  next(lcrs)
    if current == null
        return null
    r = right_arg.next(rcrs)
    while TRUE
        if r == null   # right exhausted,
            if not any  # no right found for the current left
                    input = merge_tuple(current,null) # outer left
            else
                input = null
            right_arg.close(rcrs)
            current = left_arg.next(lcrs)
            any = FALSE
            rcrs = right_arg.open()
            if input <> null
                return input
        elif not current
            (larg,rarg) = merge_tuple(current,r)
            if pred((larg,rarg))
                    any = TRUE
                return (larg,rarg)
        if current == null
            return null
        r = right_arg.next(rcrs)

  close((lcrs,rcrs))
    left_arg.close(lcrs)
    right_arg.close(rcrs)
```

```
nest(⊕,eject,proj,pred,zeros,arg)

  open()
    hashtable =
    crs = arg.open()
    input = arg.next(crs)
    while input <> null        # cache in hashtable
        if input <> null and not zeros(input)
            if pred(input)
                key = proj(input)
                bucket = hashtable.get(key)
                if bucket exists
                    ((elm),monoid) = bucket
                    hashtable[key] = ((elm),monoid ⊕ 𝒰⊕(eject(input)))
                else
                    hashtable[key] = ((key),𝒰⊕(eject(input)))
        input = arg.next(crs)
    arg.close(crs)
    if hashtable is not empty
        materlz = list of hashtable.values() # status
    else
        materlz = []
    return materlz.open()

  next(crs)
    return materlz.next()

  close(crs)
    materlz = []



cache(id,input,parents)

  open()
      subplan = input.open()
      materlz =                  # local state
      return materlz.open() # return a new cursor crs

  next(crs)                        # crs from the cursor pool
      cached = materlz.next(crs)
      if cached == null    # not cached yet
         input = input.next()
         add (input,1) to materlz  # cache as read by one
         return input
      else
         if cached.refcount + 1 == parents # no need to store
             remove cached from materlz
         else
             cached.refcount = cached.refcount + 1
         return cached.input

  close(crs)
      materlz.close(crs)
      if materlz is empty # all inputs consumed by all parents
          input.close()
```

```
prune(pred,arg)

  open()
    initialise stack
      # null signals termination
    push(null,stack)
      # □ is bottom element of ⊆
    push(□,stack)
    return arg.open()

  next()
    input = arg.next()
    if input <> null
       while not pred(input)
          input = arg.next()
       push(input,stack)
       return input

  close()
    release stack
    arg.close()

  next_candidate(d)
    candidate = top(stack)
    pop(stack)
    return candidate

refine(id,pred,arg)

  open()
    paired_prune = id
    largest = paired_prune.next_candidate()
    return arg.open()

  next()
    while largest <> null
       input = arg.next(largest)
       while input <> null
          if pred(input)
             return input
          input = arg.next(largest)
       largest = paired_prune.next_candidate()
    return null

  close()
    arg.close()
```

C.8 Final Remarks

It was developed using the **Python** [vRe03, Bea01] programming language, version 2.3. The package **PLY** [Bea03] was used for the implementing the up parser described in Section 5.4.2, using a left recursive technique [ASU85] in a **Lex/Yacc** fashion [Joh79]. The storage and buffer management was developed using the **Python** library that interfaces **Berkeley DB** [OBS99] API. Thus, p-clauses are stored as objects (cf. Section 5.3) in persistent files managed by **Berkeley DB** version 4.1. Likewise plugged CID inference rules are stored, as parsed into abstract syntax trees (cf. Section 5.4.2), by **Berkeley DB**.

The prototype is available under GPL license [GNU03].