Implementation of Mixed-Integer Programming in LogicBlox

The Language Team

LogicBlox, Inc.

1 Mathematical and Mixed-Integer Programming

Mathematical Programming (MP) is a technique for finding the minimum or the maximum of (i.e., optimizing) a given function, referred to as the *objective function*, by systematically choosing real or integer values of the function's variables from a value domain defined by a set of *constraints*. We are interested in MP problems in which (1) the objective function is linear, and (2) all constraints are linear equalities or inequalities. MP is extensively applied in business, economics, as well as engineering, in particular in the areas of transportation, energy, telecommunication, and manufacturing. The technique has proved useful in modeling diverse types of problems related to planning, routing, scheduling, assignment, and design.

Definition 1. *An MP program with n variables and m constraints has the form:*

$$
\begin{aligned}
&\textit{Minimize } f(\bar{x})\\
&\textit{subject to } A_1 x = b_1\\
&\qquad\qquad A_2 x \leq b_2\\
&\qquad\qquad l_i \leq x_i \leq u_i \textit{ for } i = 1 \ldots n\\
&\qquad\qquad x_j \textit{ is an integer for all } j \in D \textit{ s.t. } D \subseteq \{1 \ldots n\}
\end{aligned}
$$

where:

$$
\begin{aligned}
f(\bar{x}) &= k_1 x_1 + \ldots + k_n x_n + k,\\
A_1 &\textit{ is a } m_1 \times n \textit{ matrix,}\\
A_2 &\textit{ is a } m_2 \times n \textit{ matrix, and}\\
m_1 + m_2 &= m.
\end{aligned}
$$

If all the variables can be rational (i.e., $D = \emptyset$), the program defines a *linear programming problem*, which can be solved in polynomial time. In practice, linear programs can be solved efficiently for reasonable-sized problems, or even for large problems with special structure. However, when some or all of the variables must be integer-typed—corresponding to *mixed-integer programming (MIP)* and *integer programming* respectively—the problem becomes NP-complete.

In the reminder we focus on MIP problems, which we may also refer to as *optimization problems*.

Example 1 (Diet Problem). Given a set *FOOD* of foods, a set *NUTR* of nutrients, the amount $amt_{n,f}$ of each nutrient n in each food f, the cost $cost_f$ of each food

f, and the minimum daily requirement $nutrLow_n$ of each nutrient n, find the minimal cost of a meal that comprises certain non-negative quantity Buy_f of each food $f \in FOOD$ and satisfies the nutrition requirements. Formally, the problem is defined as follows.

$$\begin{aligned} &\text{Minimize } \textstyle\sum_{f \in FOOD} cost_f \times Buy_f \\ &\text{subject to } \textstyle\sum_{f \in FOOD} amt_{n,f} \times Buy_f \geq nutrLow_n, \forall n \in NUTR \\ &\qquad\qquad Buy_f \geq 0, \forall f \in FOOD \end{aligned}$$

In the definition, the predicates *FOOD* and *NUTR* represent *index sets*; the predicates *cost*, *amt*, and *nutrLow* represent *parameters*, and the predicate *Buy* represents a *variable*. Parameters and variables range over one or more index sets. Additionally, program definition may contain *constants* represented as numerical values (such as 0 here) or predicates defined by the programmer. The objective function as well as the constraints may be defined using arbitrarily nested arithmetic expressions.

The implementation of support for MIP in LogicBlox required us to:

- extend the LogicBlox language (which we call $\texttt{Datalog}^{\text{LB}}$) with features necessary to represent MIP problems,
- build libraries for compiling and interpreting the new lanugage features, and
- integrate the library with external optimization software to enable finding problem solutions.

The following sections give a detail description of each of these enhancements. Section 2 identifies *Existential Datalog*—written $\texttt{Datalog}^{\exists}$—a subset of $\texttt{Datalog}^{\text{LB}}$ with features necessary and sufficient for expressing MIP problems. The main difference between $\texttt{Datalog}^{\text{LB}}$ and $\texttt{Datalog}^{\exists}$ is that the latter allows the use of existentially quantified variables in the heads of the rules. Section 3 shows two practical database applications of $\texttt{Datalog}^{\exists}$—other than support for MIP—enabled by this new feature. Section 4 presents the main components of our MIP framework—the compiler and interpreter for $\texttt{Datalog}^{\exists}$—and outlines the interaction between LogicBlox and the external solvers. Section 5 compares our approach to related work. Finally, the Appendix **??** contains several use cases illustrating different aspects of the LogicBlox MIP framework.

2 $\texttt{Datalog}^{\exists}$

Notations We use, possibly subscripted, x, y, z to denote logical variables, s, t, u to denote arbitrary terms (variables or constants), $\bar{s}$, $\bar{t}$, $\bar{u}$ to denote vectors of terms, a, b, c, to denote arbitrary constant values, k, n, m to denote numerical values, f, g to denote arbitrary function symbols, and p, q to denote logical predicates. Specific constants are either numbers (e.g., 3 or 3.14), strings (e.g., *"foo"*), or boolean constants (i.e., *true* or *false*). For an arbitrary expression e, we use $vars(e)$ to denote the set of variables appearing in e.

A Datalog$^{\exists}$ program is a collection of clauses defining a set of logical predicates—called the *intensional predicates*—in terms of the externally provided data—called the *extensional predicates*. A predicate is identified by a name p, and is associated with an arity k and an *interpretation*—a finite set of tuples of length k. A clause is either (1) a *fact assertion* (*axiom*), which asserts the existence of a tuple in the interpretation of one or more predicates, (2) a *rule* (*theorem*), which defines the interpretation of a predicate p in terms of the interpretations of some predicates $q_1, ...q_n$, or (3) a *constraint* (*denial*), which disallows a certain situation. A *satisfying interpretation* for a program is a set of interpretations for all the program's predicates such that all assertions, definitions and constraints are true. The satisfying interpretations for the extensional predicates are given immediately by the assertions. The goal of a Datalog computation is to calculate satisfying interpretations for the intensional predicates. When more than one satisfying interpretations exist, we require that the computation finds a *minimal* interpretation, that is, an interpretation that has no proper subset which also satisfies the program.

Example 2. The following program defines the intensional predicate `ancestor` in terms of the extensional predicate `parent`. The fact assertions in lines 1-2 require the interpretation of `parent` to be: {(*"bob"*, *"dave"*), (*"dave"*, *"mary"*)}.

```
1   parent(“bob”,“dave”).
2   parent(“dave”,“mary”).
3   ancestor(x,y) <- parent(x,y).
4   ancestor(x,y) <- parent(x,z), ancestor(z,y).
5   !( parent(x,y), !ancestor(x,y) ).
```

The meaning of the recursive definition in line 4 is that, for any assignment of values to the variables `x` $= a$, `y` $= b$, and `z` $= c$, if both `parent(`a`,`c`)` and `ancestor(`c`,`b`)` are true, then `ancestor(`a`,`b`)` must also be true. Thus, the tuple set {(*"bob"*, *"dave"*), (*"dave"*, *"mary"*), (*"bob"*, *"mary"*)} forms the minimal satisfying interpretation for `ancestor`.

The meaning of the constraint in line 5 is that there are no assignments `x` $= a$ and `y` $= b$ such that `parent(`a`,`b`)` is true but `ancestor(`a`,`b`)` is false. The computed interpretation satisfies this constraint because every tuple in the interpretation of `parent` is also in the interpretation of `ancestor`.

2.1 Syntax

The grammar for Datalog$^{\exists}$ is shown in Figure 1. A program P consists of zero or more clauses. A clause S is either (1) a fact assertion (denoted by a conjunction), (2) a rule (denoted by an atom—the *head* of the rule, representing the predicate being defined—followed by '`<-`', followed by a conjunction—the *body* of the rule), or (3) a constraint (denoted by a negated conjunction). A conjunction C is a literal or a sequence of literals separated by the conjunction symbol ','. A literal L is an atom, possibly preceded by the negation symbol '!'. An atom A is either a predicate name p applied to a parenthesized list of terms—called the atom's *arguments*—or an equality judgment. The number of arguments must match the

P	$\equiv$	S^*	Program
S	$\equiv$	$C.$	Fact assertion (Axiom)
	$\mid$	$p(T^*)$ `<-` $C.$	Rule
	$\mid$	$!(C).$	Constraint
C	$\equiv$	$L \mid L, C$	Conjunction
L	$\equiv$	$A \mid !A$	Literal
A	$\equiv$	$p(T^*)$	Atom
	$\mid$	$T_1 = T_2$	Equality judgement
T	$\equiv$	$x \mid a$	Term

Fig. 1. `Datalog`$^\exists$ syntax

arity of p. An equality judgment asserts that two terms are always equal. A term T is either a constant or a variable. A constant a is either a number, a string (enclosed in double quotes), or a boolean constant (either `true` or `false`). A variable x is an arbitrary identifier, beginning with a letter and containing no spaces. Note that a predicate may appear in the head of more than one rule. For any intensional predicate p, the set of all rules with p in their heads is called the *procedure* (or *definition*) of p.

Standard Datalog requires that any variable that occurs in the head of a rule must also occur in the rule's body. In `Datalog`$^\exists$ we relax this restriction, and call any variable occurring only in the head of a rule an *existential* variable, and any rule with at least one existential variable an *existential* rule.

2.2 Semantics

The *logical semantics* of a `Datalog`$^\exists$ program is given by a conjunction of closed well-formed formulas of first-order logic, one formula for each clause. Each clause represents a logical implication (the conjunction symbol, negation symbol, variables and constants are given their standard logical meanings):

- a fact assertion $L^*.$ represents logical implication $\texttt{true} \Rightarrow \langle \exists \bar{y} :: L^* \rangle$, where $\bar{y} = vars(L^*)$;
- a rule A `<-` $L^*.$ represents logical implication $\langle \forall \bar{x} :: L^* \Rightarrow \langle \exists \bar{y} :: A \rangle \rangle$, where $\bar{x} = vars(L^*)$, and $\bar{y} = vars(A) - \bar{x}$;
- a constraint $!(L^*).$ represents logical implication $\langle \forall \bar{x} :: L^* \rangle \Rightarrow \texttt{false}$, where $\bar{x} = vars(L^*)$.

As mentioned before, the goal of a `Datalog`$^\exists$ computation is to find satisfying interpretations for the intensional predicates. The algorithm used to find such interpretations can be considered the *operational semantics* of `Datalog`$^\exists$. The operational semantics of `Datalog`$^\exists$ is harder than that of standard Datalog because the values for the existential variables must be chosen to avoid violating the constraints. In general, this problem has been shown to be intractable [3]. Thus, we now consider three tractable subsets of the `Datalog`$^\exists$ syntax—identified by the restrictions they impose on the use of the existential rules—that supply features enabling representation of useful database applications such as: (1) value

invention, (2) dynamic choice, and (3) mixed-integer programming. We illustrate each of these applications in Section 1. Before that, Sections 2.3 and 2.4 introduce several features of $\texttt{Datalog}^{\exists}$ relevant to this discussion.

2.3 Syntactic Sugar

Positive Constraints Many constraints are most easily understood in the positive form (e.g., *all Foo's are Bar's*) as opposed to the negative form (*no Foo's are not Bar's*). $\texttt{Datalog}^{\exists}$ facilitates expressing positive constraints as clauses of the form 'C_1 `->` C_2.'. Intuitively, the intended meaning of such a clause is that, for any assignment of values to the variables $\bar{x}$ of C_1 that makes C_1 true, the same assignment of values to the variables $\bar{y}$ of C_2 will make C_2 true (we require that $\bar{y} \subseteq \bar{x}$). Formally, the positive constraint 'C_1 `->` C_2' has the same meaning as the negative constraint '$!(C_1, !C_2)$'.

Positive constraints commonly act as *type declarations*.

Example 3. The constraint '`parent(x,y) -> person(x),person(y)`' states that, whenever `x` is the parent of `y`, `x` is a person, and `y` is a person.

The simplest form of a type declaration, '$p(x_1, \ldots, x_k)$ `->` .', introduces a predicate p to the environment and assigns it arity k. LogicBlox requires that every predicate used in the program is introduced with a type declaration.

Functions A predicate p is a *function* if it is governed by a *functional dependency constraint* '$p(\bar{x}, y_1), p(\bar{x}, y_2)$ `->` $y_1 = y_2$'. A functional dependency constraint on p is implicit if every atom using p is always written in the *functional form* '$p[\bar{t}] = u$'. Note that the length of $\bar{t}$, $|\bar{t}|$, is $k - 1$, where k is the arity of p (when $|\bar{t}| = 0$, p is a singleton and represents a user-defined constant).

Example 4. The constraint '`father[p]=f -> person(p),person(f).`' declares a function capturing the father relation.

2.4 Arithmetic

$\texttt{Datalog}^{\text{LB}}$ provides a standard library of mathematical and utility predicates (many of which are functions). The starting point of the library is a set of unary predicates denoting native types, including `string(x)`, `boolean(x)`, and three groups of numeric domains represented by unary predicates with names of the form `float[`i`](x)` (floating-point), `int[`i`](x)` (integer), and `uint[`i`](x)` (unsigned integer), where i is the type's bit-width. These predicates may be used like any other predicates, except that they are read-only and *infinite* (i.e., considered by the system to be too large for its data to be persisted on disk). The library also contains a variety of read-only, infinite functions representing standard arithmetic and string operations, as well as special syntax for aggregations such as `total` and `count`.

Bounds constraints A type declaration for a predicate p may place restrictions on the values that can be assigned to p's arguments. Such bounds constraints are expressed as arithmetic inequality constraints. In general, any inequality relating a predicate argument to a value representing a numeric native type[1] is a valid bounds constraint.

Example 5. The constraint '`age(p,n) -> person(p),int[32](n),n>=0.`' states that a person's age must be non-negative.

3 Practical Applications of $\texttt{Datalog}^{\exists}$

We now highlight two practical database applications of $\texttt{Datalog}^{\exists}$.

3.1 Value Invention

Support for existential rules enables $\texttt{Datalog}^{\text{LB}}$ to represent, crucial for many database applications, *invented values* [1]. For instance, a common database pattern provides a hidden stable key (HSK) representing a domain entity. In tables referring to the entity, the HSK replaces the user-visible data that identifies the entity, thus allowing to modify the user-visible data without the need to also modify every entity reference. To support value invention, Datalog has been enhanced with an extra-logical operator `new` [1]. $\texttt{Datalog}^{\exists}$ provides an alternative, natural way to express this feature.

Example 6. Consider the axiom '`person(x), name[x]=`*"dave"*`.`', which asserts that there is a value a for `x` such that `person(`a`)` and `name(`a`)` $=$ *"dave"*. Thus, the axiom invents the new value a.

Value invention may be also used in rules:

Example 7. The following program defines the intensional predicate `out` in terms of the extensional predicate `in`:

```
1  in[x]=v -> t(x), string(v).
2  out[y]=v -> u(y), string(v).
3  out[y]=v <- in[x]=v.
```

The rule (line 3) contains the existential variable `y`. Since the constraint in line 2 requires that every value of `y` is associated with at most one value for `v`, a new value for `y` must be invented for each value of `v` in the predicate `in`.

Furthermore, we may mix value invention with recursion:

Example 8. The following program gives a logical definition of the set of natural numbers, for which no finite model exists:

[1] Bounds constraints with values computed by another predicate are not yet supported

```
1   nat(x0), zero(x0).           // Zero exists.
2   !( zero(x0), succ[y]=x0 ).   // Zero is not a successor.
3   succ[x]=y -> nat(x), nat(y).
4   succ[x]=y <- nat(x).         // Every nat has a unique successor.
5   pred[y]=x <- succ[x]=y.      // Every nat has a unique predecessor.
```

Our implementation of Datalog$^{\exists}$ executes this program until no more values fit in the memory, at which point the execution aborts with an error.

3.2 Greedy Choice

Adding extra-logical notation to Datalog to support expressing efficient greedy choice algorithms has also been proposed [7]. The same semantics may be captured naturally in Datalog$^{\exists}$.

Example 9. The following program assigns random values to a set of entities:

```
1   molecule(x), id[x]=i -> uint[32](i).
2   molecule(x), id[x]=1.
3   molecule(x), id[x]=2.
4   molecule(x), id[x]=3.
5   position[x]=v -> molecule(x), float[32](v), v≥0.0, v≤1.0.
6   position[x]=v <- molecule(x).
```

The logical meaning of the existential rule in line 6 is that every molecule must have a position. The declaration for `position` (line 1) requires the position to be a floating-point number between 0 and 1. Thus, the system assigns a random position between 0 and 1 to each molecule.

Allowing recursion increases the expressive power of the language:

Example 10. The following program computes a rooted spanning tree from a graph (given by the predicate `edge`) and a specified root:

```
1   edge(n1,n2) -> node(n1), node(n2).
2   root[]=n -> node(n).                      // Root must be supplied.
3   parent[n1]=n2 -> edge(n2,n1), !(n1 = n2).
4   parent[n2]=n1 <- root[]=n1, edge(n1,_).   // Choose a child of root.
5   parent[n2]=n1 <- parent[n1]=_, edge(n1,_). // Recursively choose a child.
```

It should be emphasized that the greedy choice algorithms of [7] represent a tractable subset of the overall problem of finding interpretations for Datalog$^{\exists}$ programs, which, as mentioned before, is intractable in general.

4 Datalog$^{\exists}$-based MIP Framework

The specification of an MIP problem in LogicBlox consists of the Datalog$^{\exists}$ encoding of the problem's mathematical definition, and associated with the problem extensional database (i.e., the collection of predicates that define the index sets, parameters, user-defined constants, and the expected valuations of

the problem variables). The actual specifications for a set of example problems can be found in the MIP test suite directory[2]. Every test suite includes a file named `suite.program`, containing the problem definition, one or more files named `test*.init`, each providing an extensional database for the problem, and one or more files named `text*.assert`, each specifying the instructions for running the suite with the extensional database given by the corresponding `test*.init` file.

Example 11. Figure 2(a) lists the $\texttt{Datalog}^{\exists}$ specification of the Diet problem. The index sets and parameters are declared in lines 1-2 and 4-6 respectively. The variable *Buy* is declared and defined in lines 8-9. Note that the declaration of *Buy* (line 8) includes the bounds constraint $Buy(f) \geq 0, \forall f \in FOOD$. The objective function and the direction of the optimization are identified by the *objective function axiom* in line 11, and defined by the aggregation function `total` (written using the infix operator '`+=`') applied over the cost of all purchased food items (line 12). The constraint expressing the problem's nutritional requirements is defined in lines 14-15 in terms of the intensional predicate `totalNutr` (line 14).

Figure 2(b) lists the excerpt of the Diet problem's initialization file. The axioms and rules in lines 1-17 define the problem's extensional database, whereas the rules in lines 18-19 provide the expected values for the variable predicate.

To enable solving MIP problems represented in $\texttt{Datalog}^{\exists}$, we have implemented a framework integrating LogicBlox with external, open-source Optimization Services (OS) [5] project. Developed under COIN-OR [4], the OS project provides a set of standards and libraries for representing problem instances, results, solver options, and communication between clients and solvers in a distributed environment using either Web Services or directly linked libraries. Integration with OS enables the use of a wide variety of publicly available as well as commercial solvers (e.g., CBC, CLP, Bonmin, Cplex, Symphony and more) as plug-ins, without the need to customize the interaction with each solver separately. In this section we describe our framework in detail.

The LogicBlox MIP framework comprises two components: the compiler and the interpreter. The compiler transforms a high-level $\texttt{Datalog}^{\exists}$ program into an intermediate form closely following the specification from Definition 1. At run time, the interpreter evaluates the intermediate representation and generates a *problem instance*—a matrix representing bounds and constraints on problem variables, and passes the instance to the OS interface for selected solver. OS invokes the solver, which computes the solution to the problem. Finally, OS returns the problem solution to LogicBlox run time, which reads the values computed by the solver and adds them to the appropriate predicates thus producing the model.

4.1 Compiler

The compiler transforms a program written in $\texttt{Datalog}^{\exists}$ into a mathematical problem description expressed in terms of the intermediate *MIP language*. defined in Figure 3. A problem Π is a triple $\langle \delta, \xi, \psi \rangle$ consisting of a direction δ

[2] `LogicBlox/BloxUnit/testsuites/application-optimization/datalog-existential`

```
1   NUTR(x) ->.
2   FOOD(x) ->.
3
4   amt[n,f]=a -> NUTR(n), FOOD(f), float[64](a).
5   nutrLow[n]=b -> NUTR(n), float[64](b).
6   cost[f]=c -> FOOD(f), float[64](c).
7
8   Buy[f]=b -> FOOD(f), float[64](b), b >= 0.
9   Buy[f]=_ <- FOOD(f).
10
11  lang:solver:minimal(‘totalCost).
12  totalCost[]+=cost[f]*Buy[f].
13
14  totalNutr[n]+=amt[n,f]*Buy[f].
15  NUTR(n) -> nutrLow[n] <= totalNutr[n].
```

(a)

```
1   +NUTR(x), +NUTR:name(x:"Prot").
2   +NUTR(x), +NUTR:name(x:"Iron").
3   ...
4   +FOOD(x), +FOOD:name(x:"QP").
5   +FOOD(x), +FOOD:name(x:"MD").
6   ...
7   +nutrLow(x;55) <- NUTR:name(x:"Prot").
8   +nutrLow(x;100) <- NUTR:name(x:"Iron").
9   ...
10  +cost(f;1.84) <- FOOD:name(f:"QP").
11  +cost(f;2.19) <- FOOD:name(f:"MD").
12  ...
13  +amt(n,x;a) <- NUTR:name(n:"Prot"), FOOD:name(x:"QP"), a=28.
14  +amt(n,x;a) <- NUTR:name(n:"Prot"), FOOD:name(x:"MD"), a=24.
15  ...
16  +amt(n,x;a) <- NUTR:name(n:"Iron"), FOOD:name(x:"QP"), a=20.
17  +amt(n,x;a) <- NUTR:name(n:"Iron"), FOOD:name(x:"MD"), a=20.
18  ...
19  +Buy_expected(f;4.38525) <- FOOD:name(f:"QP").
20  +Buy_expected(f;0) <- FOOD:name(f:"MD").
21  ...
```

(b)

Fig. 2. LogicBlox representation of the Diet problem: the definition encoded in $\texttt{Datalog}^{\exists}$ (a) and a fragment of the extensional database (b)

Π	$\equiv$	$\langle \delta, \xi, \psi \rangle$	Program
δ	$\equiv$	$min \mid max$	Optimization direction
ξ	$\equiv$	$v_{\bar{x}} \mid r_{\bar{x}} \mid \xi_1 \oplus \xi_2 \mid \sum_{x \in I} \xi$	Expression
$\oplus$	$\equiv$	$* \mid + \mid -$	Arithmetic operator
ψ	$\equiv$	$\xi_1 \diamond \xi_2 \mid \psi_1 \wedge \psi_2 \mid \forall_{x \in I}.\psi$	Formula
$\diamond$	$\equiv$	$\leq \mid \geq \mid \neq \mid =$	Comparison operator
I			Index set predicate name
v			Variable predicate name
r			Parameter predicate name
x			Index variable name

Fig. 3. MIP language syntax

(*"minimize"* or *"maximize"*), an objective function expression ξ, and a constraint formula ψ. An expression ξ is a variable predicate v applied to a set of index variables, a parameter predicate r applied to a set of index variables, or a set of expressions governed by an arithmetic operator. A formula ψ is a comparison between two expressions, a conjunction of formulas, or a universal quantification ranging over an index variable x drawn from an index set I. We assume that the sets of predicate names representing variables, parameters and index sets are disjoint and known in advance. In practice, they can be easily computed by analyzing the source program.

Figure 4 defines the simplified version of our MIP compiler. We assume that (1) all programs are in conjunctive normal form, (2) conjunction is truly commutative, and (3) bound variables in the rules can be renamed as needed. Based on the definition in Figure 4, the compiler may accept $\texttt{Datalog}^{\exists}$ specifications of non-linear constraints which many solvers cannot handle. Such specifications are rejected by the interpreter, which raises a run-time error.

Given a $\texttt{Datalog}^{\exists}$ program P, the compiler first identifies the predicate defining the objective function (as specified by the objective function axiom), and compiles its body into the expression ξ. The optimization direction is also derived form the objective function. In the interest of space, we omit the details of this compilation step.

In the next step, the compiler examines all constraints in P, and identifies the ones that mention the variable predicates (judgment $P \vdash C$, which we omit for lack of space). Identification of these constraints involves checking whether their definitions depend on existential variable predicates[3]. The body of each identified constraint is then separately compiled using the relation $P \vdash P \Longrightarrow \psi$ into an MIP language constraint formula. The resulting constraint formula ψ is a top-level conjunction of all such formulas.

The relation $P \vdash C \Longrightarrow \psi$ is relatively simple because it assumes that the program is in disjunctive normal form. The base case looks for a constraint definition that contains a negated comparison '$!\diamond(r_1, r_2)$' among its body atoms. The other body atoms of such a definition (assumed to be positive literals to avoid

[3] A definition depends on a predicate p if either (1) its body contains p, or (2) its body contains a predicate that depends on p.

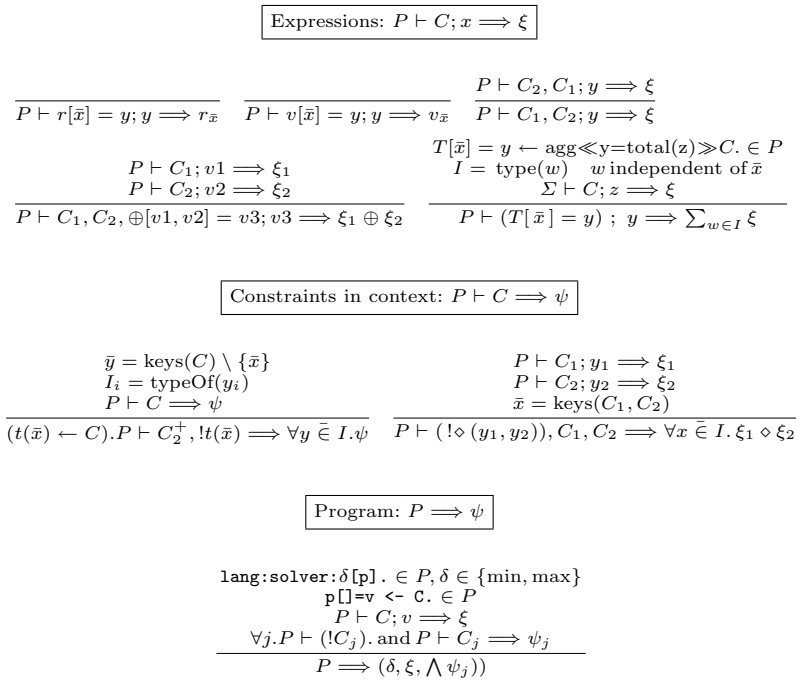

Fig. 4. LogicBlox MIP compiler

potential disjunctive constraints) are then used to construct expressions ξ_1 and ξ_2 from r_1 and r_2, respectively. The relation also builds a universal quantification over the key variables that occur in the definition (the details of the relation identifying key variables are omitted). Disjunctive constraints are avoided by the requirement that constraint definition may contain only one comparison operator (i.e., C_2^+ is a positive conjunction of atoms).

Note that the expression compiler judgment $P \vdash C; v \Longrightarrow \xi$ consumes *all* atoms in C when producing ξ, thereby rejecting clauses with extraneous operators that cannot be converted into the intermediate form.

In cases when a temporary predicate t is used to define a constraint (such as `totalNutr` in Figure 2), the compiler traces back through the definition of t until it encounters the constraint.

4.2 Interpreter

The intermediate representation $\mathit{\Pi} \equiv \langle \delta, \xi, \psi \rangle$ produced by the compiler is integrated with the rest of the original $\texttt{Datalog}^{\exists}$ program in the form of a special *MIP clause* defining the variable predicates $v_1, \ldots, v_n$. Recall that both the objective function expression ξ and the constraint formula ψ contain variable predicates, parameter predicates and universal quantification defined over index sets. The values of the index sets and the parameter predicates are computed at

run time by the LogicBlox engine. Once these values are computed, our interpreter evaluates the MIP clause, and produces a *solver instance* of the problem, that is later passed to an external solver.

In `Datalog`$^{\exists}$, as well as in the intermediate specification, the unknown values for which we are solving are organized into indexed families stored in variable predicates $v_{\bar{x}}$. For example, the predicate `Buy[x]` contains a set of unknown values, one unknown for each food `x`. When building a solver instance, the interpreter flattens all such predicates into a single variable set, and then sequences the variables in the set by mapping each one to a unique natural number. The number of variables depends on the size of the index sets, and is not available until run time, when all elements of the index sets are known.

Consider MIP problem with a constraint '$\psi = \forall x \in I_1. \forall y \in I_2. v_x + w_{xy} \leq 0$' with two variable predicates, v_x and w_{xy}, where $x \in I_1$ and $y \in I_2$. Suppose that the LogicBlox engine evaluates the index sets to be: $I_1 = \{a, b\}$ and $I_2 = \{c, d\}$. Expanding ψ yields the mapping $D = \{v_a \mapsto 1, w_{a,c} \mapsto 2, w_{a,d} \mapsto 3, v_b \mapsto 4, w_{b,c} \mapsto 5, w_{b,d} \mapsto 6\}$. The solver instance for the problem consists of:

- the mapping D of the unknowns to unique natural numbers,
- an objective function vector of coefficients of size $|D|$, which represents the value of the objective function $p_1 x_1 + \cdots + p_{|D|} x_{|D|}$, and
- a constraint matrix with row dimension $|D|$ and two bounds vectors, L and H, providing lower and upper bounds for each row in the constraint matrix. Together, the constraint matrix and the bounds vectors represent a system of equations of the form

$$\begin{array}{ccc} L_1 & \leq p_{11} v_{n1} + \cdots + p_{1|D|} v_{1|D|} & \leq H_1 \\ \vdots & \vdots & \vdots \\ L_n & \leq p_{n1} v_{n1} + \cdots + p_{n|D|} v_{n|D|} & \leq H_n \end{array}$$

The solver instance is produced by the MIP interpreter. The relevant definitions of the interpreter are shown in Figure 5(a). The interpreter consists of four semantic functions:

Variables The function $\mathcal{V}[\![\cdot]\!]\rho$, based on the mapping D, evaluates a variable expression $v_{\bar{x}}$ into a column index n. This computes the index of the column in the constraint matrix that corresponds to a particular variable for a given index value. The index variables $\bar{x}$ are evaluated in an environment ρ which maps index variables to their values.

Parameters The function $\mathcal{R}[\![\cdot]\!]\rho$ evaluates a parameter predicate $r_{\bar{x}}$ into a parameter value by looking $r_{\bar{x}}$ up in the predicate r, based on the values of the index variables $\bar{x}$ in the environment ρ.

Expressions The function $\mathcal{E}[\![\cdot]\!]\rho$ evaluates expressions into *row* vectors of the constraint matrix. For example, the expression $v_{\bar{x}} * r_{\bar{x}}$ evaluates to a row which contains the value of the parameter r at the index value of $v_{\bar{x}}$, calculated using $\mathcal{V}[\![\cdot]\!]$, and the value 0 at all other index values. Note, however, that multiplication of two variables, $v_{\bar{x}} * v'_{\bar{x}}$, evaluates to failure (denoted

$$
\begin{array}{ll}
\mathcal{V}[\![v[\bar{x}]]\!] & : \mathrm{Env} \to \mathbb{N} \\
\mathcal{V}[\![v[\bar{x}]]\!]\,\rho & = D(x_{\rho(x_1),\ldots,\rho(x_n)}) \\
 & \\
\mathcal{R}[\![r[\bar{x}]]\!] & : \mathrm{Env} \to \mathbb{R} \\
\mathcal{R}[\![r[\bar{x}]]\!]\,\rho & = c, \\
 & \quad \text{where } P(\rho(x)_1, \ldots, \rho(x_n)) = c
\end{array}
\qquad
\begin{array}{ll}
\mathcal{E}[\![\xi]\!] & : \mathrm{Env} \to \mathbb{R}_{\perp}^{|D|} \\
\mathcal{E}[\![v_{\bar{x}} * r_{\bar{y}}]\!]\rho & = [0 \cdots \mathcal{R}[\![r_{\bar{y}}]\!]_{\mathcal{V}[\![v_{\bar{x}}]\!]} \cdots 0] \\
\mathcal{E}[\![v_{\bar{x}} * v'_{\bar{y}}]\!]\rho & = \perp \\
\mathcal{E}[\![\xi_1 + \xi_2]\!] & = \mathcal{E}[\![\xi_1]\!] + \mathcal{E}[\![\xi_2]\!] \\
\mathcal{E}[\![\sum_{x \in I} \xi]\!] & = \sum_{j=0}^{n} \mathcal{E}[\![\xi]\!](\rho | x \mapsto c_j) \\
 & \quad \text{where } I = \{c_0, \ldots, c_n\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{F}[\![\psi_1 \wedge \psi_2]\!]\rho & = \mathcal{F}[\![\psi_1]\!]\rho \,\bar{\wedge}\, \mathcal{F}[\![\psi_2]\!]\rho \\
\mathcal{F}[\![\forall i \in I.\psi]\!]\rho & = \bar{\bigwedge}_{j=0}^{n} \left(\mathcal{F}[\![\psi_j]\!](\rho | i \mapsto c_j)\right) \\
 & \quad \text{where } I = \{c_0, \ldots, c_n\}
\end{array}
\qquad
\begin{array}{l}
\mathcal{F}[\![\xi \leq c]\!]\rho = (\mathcal{E}[\![\xi]\!]\rho, (-\infty, c)) \\
\mathcal{F}[\![\xi \geq c]\!]\rho = (\mathcal{E}[\![\xi]\!]\rho, (c, \infty))
\end{array}
$$

(a)

$$
\left(\begin{pmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \vdots & \vdots \\ v_{n1} & \cdots & v_{nd} \end{pmatrix}, \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}\right) \bar{\wedge} \left(\begin{pmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \vdots & \vdots \\ v_{m1} & \cdots & v_{md} \end{pmatrix}, \begin{pmatrix} c_1 \\ \vdots \\ c_m \end{pmatrix}\right) = \left(\begin{pmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \vdots & \vdots \\ v_{n1} & \cdots & v_{nd} \\ \vdots & \vdots & \vdots \\ v_{(n+m)1} & \cdots & v_{(n+m)d} \end{pmatrix}, \begin{pmatrix} c_1 \\ \vdots \\ c_n \\ \vdots \\ c_{(n+m)} \end{pmatrix}\right)
$$

(b)

Fig. 5. LogicBlox MIP interpreter (a) and the auxiliary matrix operator $\bar{\wedge}$ (b)

$\perp$) because it would result in a non-linear constraint that most solvers do not handle. The expression $\xi_1 + \xi_2$ is implemented by point-wise addition of two subexpression vectors. Finally, the evaluation of a sum $\sum_{x \in I} \xi$ iterates over the index set I, binds the index variable x to each value $c_j \in I$ in the environment, and evaluates the body, summing up point-wise all the intermediate vector results.

Constraint formulas The function $\mathcal{F}[\![\cdot]\!]\rho$ evaluates constraint formulas. The base case is the comparison formula $\xi \diamond c$ (we assume that the algebraic manipulation has placed a constant value at the right-hand side of the formula). The function uses $\mathcal{E}[\![\xi]\!]$ to construct a row for the expression ξ, and the constant c to construct either the upper or the lower bound, depending on the operator $\diamond$.

To evaluate a conjunction of two formulas, the interpreter builds two constraint matrices, one for each subformula, and then combines them together using the auxiliary operator $\bar{\wedge}$ on matrices defined in Figure 5(b). Similarly, to evaluate universal quantification, the interpreter builds a matrix for each substitution instance of the index variables, and then joins the submatrices in the same way as during conjunction evaluation.

Once the abstract interpreter successfully terminates (without the error value $\perp$), the resulting constraint matrices along with the objective function and bounds vectors are sent to an OS library which then invokes the selected solver. If successful, the solver returns a vector which maps each unknown to a value.

Finally, the reverse mapping D^{-1} maps each unknown into the correct fact in the unknown predicate, and the control returns to LogicBlox run-time system.

5 Related Work

Extending Datalog with value invention has been subject to previous research [1, 2, 8]. Datalog$^{\pm}$ [3] is a family of Datalog variants for answering queries over ontologies. Datalog$^{\pm}$ extends standard Datalog with negative constraints, a generalized form of functional dependencies, and stratified negation, and admits existentially quantified head variables in rules, in which the bodies comply with restrictions imposed to ensure decidability and tractability. According to its general definition, `Datalog`$^{\exists}$ is intractable. However, we ensure tractability by restricting this general definition based on the intended application of our language. For instance, the `Datalog`$^{\exists}$ subset for representing MIP problems coincides with Datalog$^{\pm}$'s sub-language of weakly guarded tuple-generating dependencies. The two formalisms are closely related. The differences between them are dictated by their different application domains.

The idea of using Datalog for MP problems has been also considered before. The approach most closely related to ours is $\mathcal{NP}$ Datalog [6], a Datalog variant aimed at expressing NP search and MP problems. The language supports constraints, restricted forms of stratified negation, and exclusive disjunction in rule heads as the means for expressing non-deterministic choice. $\mathcal{NP}$ Datalog queries are translated into OPL programs and executed in the ILOG OPL Development Studio. By contrast, our approach represents non-deterministic choice by means of existentially quantified variables in rule heads, and translates problem specifications into low-level matrix representation rather than high-level programs. Furthermore, the integration with OS enables us to use a variety of open-source as well as proprietary solvers.

References

1. Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1):62–124, 1991.
2. Luca Cabibbo. The expressive power of stratified logic programs with value invention. *Inf. Comput.*, 147(1):22–56, 1998.
3. Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: A unified approach to ontologies and integrity constraints. In *12th International Conference on Database Theory (ICDT)*, 2009. Invited survey paper.
4. COmputational INfrastructure for Operations Research (COIN-OR). `http://www.coin-or.org/`.
5. Robert Fourer, Jun Ma, and Kipp Martin. Optimization Services: A Framework for Distributed Optimization. Draft submitted; available at `http://www.optimizationservices.org`, 2008.
6. Sergio Greco, Irina Trubitsyna, and Ester Zumpano. $\mathcal{NP}$ Datalog: A logic language for np search and optimization queries. In *Ninth International Database Engineering and Applications Symposium (IDEAS)*, pages 344–353, 2005.

7. Sergio Greco and Carlo Zaniolo. Greedy algorithms in datalog. *Theory Pract. Log. Program.*, 1(4):381–407, 2001.
8. Gabriel M. Kuper and Moshe Y. Vardi. The logical data model. *ACM Trans. Database Sys.*, 18:86–96, 1993.