Incremental Evaluation of Lattice-Based Aggregates in Logic Programming Using Modular TCLP *

Joaquín Arias and Manuel Carro
joaquin.arias@{imdea.org,alumnos.upm.es},
manuel.carro@{imdea.org,upm.es}

IMDEA Software Institute and Universidad Politécnica de Madrid

Abstract. Aggregates are used to compute single pieces of information from separate data items, such as records in a database or answers to a query to a logic program. The maximum and minimum are well-known examples of aggregates. The computation of aggregates in Prolog or variant-based tabling can loop even if the aggregate at hand can be finitely determined. When answer subsumption or mode-directed tabling is used, termination improves, but the behavior observed in existing proposals is not consistent. We present a framework to incrementally compute aggregates for elements in a lattice. We use the entailment and join relations of the lattice to define (and compute) aggregates and decide whether some atom is compatible with (entails) the aggregate. The semantics of the aggregates defined in this way is consistent with the LFP semantics of tabling with constraints. Our implementation is based on the TCLP framework available in Ciao Prolog, and improves its termination properties w.r.t. similar approaches. Defining aggregates that do not fit into the lattice structure is possible, but some properties guaranteed by the lattice may not hold. However, the flexibility provided by this possibility justifies its inclusion. We validate our design with several examples and we evaluate their performance.

1 Introduction

Aggregates, in general and informally, are operations which take all the records in a database table or all the answers to a logic programming query and synthesize a result using these data items. Common aggregates include maximum, minimum, and the set of all answers, counting the number of solutions, or computing an average. A straightforward way to compute aggregates is to compute all solutions and then calculate the aggregate. However, this has several drawbacks. In some cases, computing an aggregate can be done without computing all possible answers: for example, if the operational semantics of the underlying language include mechanisms to avoid repeating useless computations [2,1]. Also, the computation of the aggregate may involve (recursively) using the aggregate itself (see Example 1), so computing a full aggregate-less model with a fixpoint procedure may simply be not correct, and several iterations of fixpoint procedures may be necessary.

Several tabling systems [14,11,16] include the so-called *modes*, which make it possible to implement some specific aggregates incrementally. However, while being very

* Work partially supported by EIT Digital (https://eitdigital.eu), MINECO project TIN2015-67522-C3-1-R (TRACES), and Comunidad de Madrid project S2018/TCS-4339 BLOQUES-CM co-funded by EIE Funds of the European Union. 1423419.

```
:- table dist/3.

dist(X, Y, D) :-
    dist(X, Z, D1),
    edge(Z, Y, D2),
    D is D1 + D2.
dist(X, Y, D) :-
    edge(X, Y, D).
```

```
:- table dist/3.

dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).
```

Fig. 1. Left-recursive distance traversal in a graph: Tabling (left) / TCLP (right).
Note: The symbols **#>** and **#=** are (in)equalities in CLP.

helpful in some situations, a careful examination of their behavior reveals inconsistencies with the LFP semantics which makes reasoning about simple programs unsound.

In this paper we present a semantics for a class of common aggregates, derived from an interpretation of their meaning in a lattice. This interpretation makes it possible to give them a consistent least fixed point semantics. We observe that it is possible to take advantage of existing implementation techniques for tabled logic programming and extend them in order to implement the additional machinery necessary for aggregates: tabling, in all of its variants, needs to store the answers returned by the different branches of the computation, which is a first step towards computing aggregates. We further develop this initial implementation by adding the necessary support (in the form of syntax and underlying infrastructure) to incrementally compute aggregates based on the answers that are added to the table.

In particular, we base our proposal in the Modular TCLP [2,1] framework, which already has infrastructure to perform tabling with constraints. This infrastructure includes the possibility of storing answers and using entailment between stored answers to increase expressiveness, termination properties, and speed of tabling.

In Section 2 we briefly describe the Modular TCLP interface. In Section 3 we present a semantics for aggregates based on entailment and/or join operation over a lattice which is consistent with the LFP semantics. In Section 4 we present the generic framework for lattice-based aggregates with an improvement in Modular TCLP which allows the combination of answers. In Section 5 we evaluate the expressiveness and performance of ATCLP versus Prolog and tabling. Finally, in Section 6, we offer some conclusions.

2 Background: Tabling and Constraints

Tabled Logic Programming with Constraints (TCLP) [2,12,5] improves program expressiveness and, in many cases, efficiency and termination properties. Let us consider a program to compute distances between nodes in a graph written using tabling and using TCLP (Fig. 1, left and right, resp.)

Tabling records the first occurrence of each call to a tabled predicate (the *generator*) and its answers. In variant tabling (the most usual form of tabling), when a call is found to be equal, modulo variable renaming, to a previous generator, the execution of the call is suspended and it is flagged as a *consumer* of the generator. For example

`dist(a,Y,D)` is a variant of `dist(a,Z,D)` if `Y` and `Z` are free variables. Upon suspension, execution switches to evaluating another, untried branch. A branch which does not suspend due to the existence of a repeated call can generate an answer for an initial goal. When a generator finitely finishes exploring all the clauses and all answers are collected, its consumers are resumed and are fed the answers of the generator. This may make consumers produce new answers which can in turn resume more consumers. This process finishes when no new answers can be generated — i.e., a fixpoint has been reached. Tabling is sound and, for programs with a finite Herbrand model, is complete .

The program in Fig. 1 would always loop under SLD due to the left-recursive rule. Under tabling, a query such as `?- dist(a,Y,D),D<K` would terminate for acyclic graphs. In a cyclic graph, however, `dist/3` has an infinite Herbrand model: every cycle can be traversed repeatedly and create paths of increasing length. Therefore, that query will not terminate under variant tabling.

However, the integration of tabling and CLP makes it possible to execute the `dist/3`, right, using *constraint entailment* [4] to suspend calls which are more particular than previous calls, and to keep only the most general answers. The query `?- D #< K, dist(a,Y,D)` terminates under TCLP because by placing the constraint `D #< K` before `dist(a,Y,D)`, the search is pruned when the values in `D` are larger than or equal to `K`.

This illustrates the main idea underlying the use of entailment in TCLP: more particular calls (consumers) can suspend and later reuse the answers collected by more general calls (generators). In order to make this entailment relationship explicit, we will represent a TCLP goal as $\langle \mathtt{g}, \mathtt{c_g} \rangle$ where g is the call (a literal) and c_g is the projection of the current constraint store onto the variables of the call. For example, $\langle \mathtt{dist(a,Y,D)}, \mathtt{D} > 0 \wedge \mathtt{D} < 75 \rangle$ entails the goal $\langle \mathtt{dist(a,Y,D)}, \mathtt{D} < 150 \rangle$ because $(\mathtt{D} > 0 \wedge \mathtt{D} < 75) \sqsubseteq \mathtt{D} < 150$. We also say that the latter (the generator) is more general than the former (the consumer). All the solutions of a consumer are solutions for its generator, since the space of solutions of the consumer is a subset of that of the generator. However, not all answers from a generator are valid for its consumers. For example $\mathtt{Y} = \mathtt{b} \wedge \mathtt{D} > 125 \wedge \mathtt{D} < 135$ is a solution for our generator, but not for our consumer, since the consumer call was made under a constraint store more restrictive than the generator. Therefore, the tabling engine has to filter, via the constraint solver, the answers from the generator that are consistent w.r.t. the constraint store of the consumer.

Some tabling systems offer facilities that improve termination in this situation. Tabling engines that implement *mode-directed tabling* [6,17] and/or *answer subsumption* [13] can use policies other than being a variant to decide whether a call is a consumer and should be suspended. These are expressed by specifying the *modes* of some arguments. For example, the directive `:- table dist(_,_,min)` specifies the (aggregate) mode `min` for the third argument. The call will in this case terminate because only the shortest distance will be returned. However, note that the standard least fixpoint semantics (calculated by tabling) is not well-suited to programs with aggregates [8,9,15]. For example, let us consider the following program:

```
1 p(1).
```

```
2 p(0) :- p(1).
```

and let us assume that we want to minimize the (single) argument to `p/1`, i.e., we want to evaluate this program under the constraint that the argument of `p/1` has to be as small as possible. On the one hand, this means that only one literal (the `p(X)` having the smallest value for `X`) should be in the model. On the other hand, it turns out that neither {`p(0)`} nor {`p(1)`} are consistent with this intended semantics. For `p(0)` to be the literal with the minimum value, `p(1)` needs to be true. But then `p(1)` would be in the model and therefore it should be the minimum. This paradox points to the need of an ASP semantics for the general case (and clarifies why there is not an accepted, consistent semantics for aggregates in Prolog-based logic programming — see at the end of Section 3.1). We will present here an alternative, defensible meaning for a class of aggregates that can stay within the least fixpoint semantics.

3 Aggregates in Lattices

We consider first the case of aggregates that can be embedded into a lattice: the elements on which we operate can be viewed as points in a lattice whose structure depends on the particular aggregate we are computing, and where the aggregation operation can be expressed based on the partial order of the lattice. As an intuitive example, the minimum of a set of elements is the element x for which there is no other element y s.t. $y \sqsubseteq x$. This view gives rise to a view of aggregates returning designated representatives of a class.

3.1 Aggregates Based on Entailment

The simplest type of aggregation operations can be defined using only the $\sqsubseteq$ operation of the lattice. Since $\sqsubseteq$ is related to constraint entailment, we have used this name.

Definition 1 (Entailment-Based Aggregates).
Given a partial order relation $\sqsubseteq$ over a multiset S,[1] *the aggregate of S over $\sqsubseteq$, denoted as $Agg_{\sqsubseteq}$, is the set of more general values of S w.r.t. $\sqsubseteq$:*

$$Agg_{\sqsubseteq}(S) = \{x \in S \mid \nexists y \in S, y \neq x \cdot x \sqsubseteq y\}$$

`minimum` and `maximum` are two widely used entailment-based aggregates. But it is interesting to note that other policies that select a subset of answers to a query, such as `variant` or `subsumption`, can also be expressed as aggregates in a lattice.

Example 1 (`min`).
The minimum of a set of values is the least upper bound of the lattice ordered by `' > '`. The aggregate of S over `min` is defined as:

$$Agg_{min}(S) = \{x \in S \mid \nexists y \in S, y \neq x \cdot x > y\}$$

[1] This definition would usually be based on a set instead of a multiset. The reason to choose explicitly a multiset will be clear in Section 4.5, when we apply our implementation to operations that cannot be embedded in a lattice.

The minimum of a set of values is unique and, as aggregate, is a set: $Agg_{min}(\{2,3,4\}) = \{2\}$. Note that $Agg_{min}(\{2,3,4,5,6\}) = \{2\}$, as well. Therefore, one can view the aggregation of a set of values as another (potentially different) set that in some sense summarizes or represents the initial set of values. As such, several sets can have the same aggregate, or, conversely, a single aggregate can represent many initial sets. As we will see, we define $Agg_{min}(\{2,3,4\}) = \{x \mid x \geq 2\}$ as this brings interesting properties to aggregates that are compatible with the intuitive idea of what an aggregate is.

We will see how this definition of aggregates can be applied to the previous `min` case to generate a model that is compatible with the least fixpoint of a logic program. Let us consider the following variant of an example taken from [15].[2]

Example 2 (`p(min)`).

In the program below, `:- table p(min)` is intended to mean that we want to restrict the model of the program to the atoms that minimize the value of the single argument of `p/1`.

```
1 :- table p(min).
2 p(3).
3 p(2).
4 p(1) :- p(2).
5 p(0) :- p(3).
```

In absence of the `table` aggregate declaration, the set of answers would be `{p(0), p(1), p(2), p(3)}` and, therefore, the expected aggregated answer using the minimum should be `p(0)`. This is the model that ATCLP returns as the aggregated answer for the previous program and query. It also behaves consistently with an LFP semantics if `p(`k`)` is intended to mean `p(`x`)` s.t. $x \geq k$. In that case, using the clause `p(0):-p(3)` does not fall into a contradiction: if `p(`x`)` s.t. $x \geq 0$ is the model of the program, the atom `p(3)` is true under that model (because $3 \geq 0$). Therefore, `p(3)` can be used to support `p(0)`.

We want to note that the current state of affairs in other systems is far from being satisfactory. Following [15], none of the current *answer subsumption* implementations seems to behave correctly: XSB and B-Prolog return `p(1)`, and Yap, which uses batch scheduling[3] returns, on backtracking, `p(3)`, `p(2)`, and `p(1)` the *first time* the query is issued, and only `p(1)` in subsequent calls.

3.2 Aggregates Based on Join

Some interesting aggregates need to be based on an operation richer than the entailment, because they have to generate a new element based on previous elements. For these cases, we posit an aggregate similar to the one in Def. 1, but using the join operation instead of the entailment.

[2] The original example used `max`. For coherence with the rest of the cases in this paper, we have converted it to use `min`.

[3] Batch scheduling returns answers as soon as they are found.

Definition 2 (Join-Based Aggregates).
Given a join-semilattice domain D with a join operation $\sqcup$ (that is commutative, associative and idempotent), the aggregated value of any multiset $S \in D$ over $\sqcup$, denoted as $Agg_{\sqcup}$, is the least upper bound of S w.r.t. $\sqcup$:

$$Agg_{\sqcup}(S) = LUB_{\sqcup}(S)$$

The main difference w.r.t. entailment-based aggregates is that when using the join operator, the resulting aggregate could be a value that is not in S. In our case, it may not be a logical consequence of the program.

Example 3 (`min of pairs`).
Let us build on Example 1 and define the minimum of a set of pairs as element-wise minima. We define the join operation $(a_1,b_1) \sqcup (a_2,b_2) = (\min(a_1,a_2),\min(b_1,b_2))$. The aggregate value of $S = \{(a_i,b_i)\}, i = 1 \ldots n$ over this join operator is:

$$Agg_{\min}(S) = LUB_{min}(\{(a_i,b_i) \in S\}) = (min(a_i),min(b_i)) \text{ for } i = 1 \ldots n$$

Note that the minimum of a set of pairs using an entailment-based aggregate and an element-wise order (i.e., $(a_1,b_1) > (a_2,b_2) \leftrightarrow a_1 > a_2 \wedge b_1 > b_2$) can return a non-singleton set $Agg_{min}(\{(4,4),(4,2),(3,3)\}) = \{(4,2),(3,3)\}$ that defines a Pareto frontier. The join-based definition, however, returns a unique value which was not an element of the initial set: $Agg_{min}(\{(4,4),(4,2),(3,3)\}) = \{(3,2)\}$. Similarly to Def. 1, the model derived from a join-based aggregate is assumed to capture the constraint used to generate the aggregate – i.e., $Agg_{min}(\{(4,4),(4,2),(3,3)\}) \sqsubseteq (5,7)$.

4 The ATCLP Framework

We present here the ATCLP framework: how aggregated predicates are declared, how the aggregates are defined, and how the implementation works. This implementation is based on a program transformation that uses the underlying infrastructure of Modular TCLP. Finally, we present an extension to the Modular TCLP framework that makes it possible to combine answers and write aggregation operations that do not follow a lattice structure.

4.1 From Lattices to Constraints

We built our system upon the infrastructure used in Modular TCLP [2] to handle constraints. Indeed, many of the operations are similar: entailment in a lattice can be handled similarly (from an implementation point of view) to entailment in a constraint system and the implementation of the join operation can also be executed in the same places where previous, less general answer constraints are discarded in a TCLP system. We are looking at the aggregate operations in a lattice as a counterpart of similar operations among constraints, including the removal of answers that, from the point of view of the aggregates, are entailed by other answers.

```
:- use_package(tclp_aggregates).
:- table dist(_,_,min).

dist(X,Y,D) :-
     edge(X,Y,D).
dist(X,Y,D) :-
     edge(X,Z,D1),
     dist(Z,Y,D2),
     D is D1 + D2.

entails(min,A,B) :- A >= B.

edge(a,b,10).
...
```

```
:- include(aggregate_rt).
:- table '$dist'/3.

dist(X,Y,A1) :-
     put(V1,(min,F1)),
     '$dist'(X,Y,V1),
     ( var(A1) -> A1=F1
     ; entails(min,A1,F1) ).
'$dist'(X,Y,V1) :-
     get(V1,(min,A1)), A1 = D,
     edge(X,Y,D).
'$dist'(X,Y,V1) :-
     get(V1,(min,A1)), A1 = D,
     edge(X,Z,D1),
     dist(Z,Y,D2),
     D is D1 + D2.
...
```

Fig. 2. Left: minimum distance traversal program using aggregates. Right: transformation of the program.

4.2 Design of the ATCLP Interface

ATCLP provides a directive to declare the aggregated predicates and a generic interface designed to facilitate the use of different user-defined aggregates.

For homogeneity, aggregated predicates are declared with the same directive used by mode-directed tabling: `:- table p(`$\mathtt{agg}_1$`,...,`$\mathtt{agg}_n$`)`, where $\mathtt{agg}_i$ denotes the aggregate used for the i^{th} argument. For the arguments that should be evaluated under variant tabling, we use the mode `'_'`.

Fig. 2, left, shows the minimum distance traversal program using aggregates. The directive `:- use_package(tclp_aggregate)` initializes the TCLP engine, and the directive `:- table path(_,_,min)` states that the answers of `path/3` should be aggregated using the `min` of its third argument. The aggregation operation is defined as an entailment, by specifying with the predicate `entails/3` when two values are entailed from the point of view of `min` (the first argument to `entails/3`). Note that the rest of the program remains as in Fig. 1. The entailment and join operations for a given aggregate are provided by the user with predicates that implement these operations. The two predicates that a user can define are:

- `entails(Agg,A,B)` defines an entailment-based aggregate. It succeeds when the answer `A` entails the answer `B` w.r.t. the aggregate `Agg`, e.g., when $A \sqsubseteq_{Agg} B$.
- `join(Agg,A,B,New)` defines a join-based aggregate. It returns in `New` the combination of the answers `A` and `B` w.r.t. the aggregate `Agg`, e.g. $New = A \sqcup_{Agg} B$.

Examples of Entailment-Based Aggregates

Example 4 (Implementation of `min`).
The implementation of Example 1 would be complete by providing the `entails/3` predicate as:

```
entails(min, A, B) :- A >= B.
```

In order to further clarify the relationship between the aggregates and the model of the program where they appear, we show now a program that captures the semantics of the program in Example 2.

Example 5 (interpretation of `p(min)`).
The code below exemplifies how Example 2 is expected to behave under ATCLP, according to Def. 1 and the entailment definition in Example 4:

```
p(X) :- entails(min,X,3).
p(X) :- entails(min,X,2).
p(X) :- entails(min,X,1), p(Y), entails(min,2,Y).
p(X) :- entails(min,X,0), p(Y), entails(min,3,Y).
```

With this interpretation, `p(2)`, inferred by the second clause, is more general than `p(3)`, inferred from the first clause, since $\{x \geq 3\} \sqsubseteq \{x \geq 2\}$. `p(3)` is therefore discarded when the second clause is executed and only `p(2)` remains in the model (which, in our implementation, lives in the answer table of the tabling engine). After this, the first entailment goal of the third clause succeeds, `p(Y)` then succeeds with `Y=2` followed by `entails(min,2,2)`, which also succeeds because $2 \leq 2$, and `p(1)` is inferred. At this point, `p(2)` is discarded because `p(1)` is more general: $\{x \geq 2\} \sqsubseteq \{x \geq 1\}$. Finally, the first entailment goal in the last clause succeeds and the rest of the clause succeeds as well because we had `p(1)` and $3 \geq 1$. `p(0)` is then inferred and `p(1)` is discarded because it entails `p(0)`, i.e., `p(1)` $\sqsubseteq_{p(min)}$ `p(0)`.
The interpretation of a query is similar to that of a body goal: `?- p(2)` is to be understood as `?- p(X), entails(min,2,X)` which in our example succeeds because `p(X)` returns `X=0` and `entails(min,2,0)` succeeds because $2 \geq 0$.

As noted before, this interpretation extends the range of atoms which are true to include some that were not in the program without the aggregate declaration. The model for the latter was {`p(0)`, `p(1)`, `p(2)`, `p(3)`}, but the intended meaning of `?- p(X)` under the new semantics is {`p(X)` | `entails(min,X,0)`}, and therefore the query `?- p(5)` also succeeds. While this may seem strange, we also want to note that by seeing aggregates as constraints defining a domain for a variable plus a value to *anchor* these constraints, this interpretation is similar to an answer in a CLP system or to the behavior of subsumption tabling in the Herbrand domain, as the following example highlights:

Example 6 (`p(sub)`).
In the program below, `:- table p(sub)` means that we want to keep the more general answers.

```
:- table p(sub).
p(f(X,Y)).
p(f(g(Z),a)).

:- use_module(terms_check).
entails(sub,A,B) :-
      instance(A,B).
```

```
path(X,Set) :-
   setof(Y, path_(X,Y), Set).
path_(X,Y) :- edge(X,Y).
path_(X,Y) :- edge(X,Z), path_(Z,Y).
```

```
:- table path(_,set).
path(X,[Y]) :- edge(X,Y).
path(X,Ys) :- edge(X,Z),
              path(Z,Ys).
```

```
edge(a,b).  edge(b,c).  edge(b,a).  edge(c,d).
```

Fig. 3. Set of reachable nodes from a given node.

Without the aggregate declaration, the set of answers for the query `?- p(X)` is `{p(f(X,Y)),p(f(g(Z),a))}`. In the Herbrand domain with subsumption tabling, the answer `A = f(X,Y)` covers the answer `A = f(g(Z),a)`. Therefore, the expected aggregated answer using subsumption is `p(f(X,Y))`. Note that the query `?- p(f(1,g(-1)))` succeeds under ATCLP, but also in Prolog under the standard LFP semantics, even if the literal was not present in the set of answers obtained without the aggregate declaration. Therefore, our interpretation of the meaning of a model for a program with aggregates can be viewed as an extension of the Herbrand model with subsumption for constraint domains.

An Example of Join-Based Aggregates

Example 7 (`path(set)`).

Let us consider a program to compute the set of nodes that are reachable from a given node in a graph. Fig. 3 shows, on the left, a simple Prolog program and, on the right, an ATCLP program using the `set` aggregate (see below). While both seem to have the same expressiveness, the Prolog program would loop for graphs with cycles and cannot to answer some queries that the ATCLP program can (see at the end of this example). Adding tabling to the Prolog program helps in this case, but note that mixing all-solution predicates and tabling does not always work, as the suspension and resumption mechanism of tabling interacts with the usual failure- and assert-driven implementations of `setof/3` and similar predicates.

The `set` aggregate generates sets from the union of subsets. It can therefore generate values that are not logical consequences of the program without aggregates. Assuming that we have a library implementing basic operations on sets (e.g., Richard O'Keefe's well-known `ordset.pl`), we can define the `set` aggregate as:

```
:- use_module(library(sets)).
entails(set, SetA, SetB) :- ord_subset(SetA, SetB).
join(set, SetA, SetB, NewSet) :- ord_union(SetA, SetB, NewSet).
```

Note that in this case we define both the entailment and the join (although the former can be defined in terms of the latter).

This example returns the set (as an ordered list without repetitions) `L=[a,b,c,d]` for the query `?- path(a,L)`. Moreover, if we want to know which nodes can reach a set of nodes, the query `?- path(X,[a,d])` returns `X=a` and `X=b` under ATCLP, which neither Prolog nor tabling can if `setof/3` is used.

In general, for lattice-based aggregates, `entails/3` can be defined in terms of `join/4` or vice versa. However, join-based aggregates allows us to aggregate the answers in a unique value, and in some cases its gain in efficiency, in space, and time comes with a loss of precision. Nevertheless, there are applications where this trade-off can remain feasible, e.g., abstract interpretation and stream data analysis.

4.3 Implementation Sketch

In this section we present the program transformation used to execute programs with aggregates and we describe how ATCLP is implemented using Modular TCLP as underlying infrastructure.

Modular TCLP: Modular TCLP is a tabling engine that handles constraints natively. It can use constraint entailment to perform suspension and to save and return only the most general answers to a query. Its modularity comes from the existence of a generic interface with constraint solvers that defines what operations a constraint solver needs to provide to the tabling engine [2]. By extending the code (written in Prolog) that calls these external solver operations, we can *hack* the existing TCLP engine to execute aggregates as described before.

Program transformation: Fig. 2, right, shows the transformation applied to the predicate `dist/3`. The original entry point is rewritten to call an auxiliary predicate where the aggregated arguments are substituted by attributed variables [7]. These are later on caught by the tabling engine [5] and their execution is derived to the TCLP code written in Prolog. The auxiliary predicate corresponds to the original one, but the original arguments are retrieved from the attributed variables with `get/2`. The attributes are tuples of the form $(\texttt{Agg}_i,\texttt{F}_i)$, where $\texttt{Agg}_i$ is the aggregate mode declared for that argument and $\texttt{F}_i$ is a fresh variable where the aggregated value will be collected. Once the auxiliary predicate collects the aggregated answer, it is either returned (if called with an unbound variable) or checked for entailment against the value in the corresponding argument.

ATCLP Internals The TCLP tabling engine calls interface predicates from constraint solvers whose implementation depends on that solver. When this interface is used to implement aggregates, its implementation is always the same and ultimately calls the user-provided `entails/3` and `join/4` predicates. Fig. 4 shows the implementation of this interface, under the simplifying assumption that we are aggregating over a single variable. This implementation merely recovers information related to which aggregate is being used and which variables are affected, and passes it to and from the join and entailment operations.

ATCLP uses two objects: the aggregated argument (`V`) and the aggregate mode and the value for the argument `(Agg,A)`. There are three main phases in the execution of ATCLP:

Call entailment: the TCLP engine invokes `store_projection(+V,-(Agg,A))` to retrieve the representation of the aggregated arguments of a new call. Then `call_entail(+(Agg,A),+(Agg,B))` is called to check whether the new call `A`

```
store_projection(V, (Agg,A))              :- get(V, (Agg,A)).
call_entail((_  ,_), (_  ,B))             :- var(B),!.
call_entail((Agg,A), (Agg,B))             :- entails(Agg,A,B).
answer_compare((Agg,A), (Agg,B),'=<')     :- entails(Agg,A,B),!.
answer_compare((Agg,A), (Agg,B), '>')     :- entails(Agg,B,A),!.
answer_compare((Agg,A), (Agg,B),'$new'((Agg,New))) :- join(Agg,A,B,New).
apply_answer(V, (Agg,B))      :- get(V,(Agg,A)), \+ ground(A), A = B, !.
apply_answer(V, (Agg,B))      :- get(V,(Agg,A)), entails(Agg,A,B).
```

Fig. 4. Simplified ATCLP interface with the constraint tabling engine.

entails a previous generator `B`. It succeeds if `B` is a variable or if `A` $\sqsubseteq_{Agg}$ `B`. If so, the new call suspends and consumes answers from the generator; otherwise, the new call is marked as a new generator.

Answer entailment: the TCLP engine invokes `store_projection(+V,-(Agg,A))` to retrieve the representation of aggregated arguments of a new answer. Then it invokes `answer_compare(+(Agg,A),+(Agg,B),-Res)` to compare the new answer `A` against a previous answer `B`. If `A` $\sqsubseteq_{Agg}$ `B`, the predicate succeeds with `Res='=<'`; conversely, if `B` $\sqsubset_{Agg}$ `A`, the predicate returns `Res='>'`. This entailment check discards/removes more particular answers from the answer table. When the entailment check fails, and if the join operator of the aggregate mode `Agg` is implemented, the predicate returns `Res = '$new'(New)`, where `New = A` $\sqcup_{Agg}$ `B`. Otherwise, `answer_compare/3` fails and the new answer is stored in the answer table of the generator.

Answer consistency: In constraint tabling, answers from a generator may not be directly applicable to a consumer: if the environment of the consumer is more restrictive than that of the generator, the generator's answers have to be filtered by applying the constraints in the consumer environment to generate compatible answers. The TCLP engine invokes `apply_answer(+V,+(Agg,B))`. When `A` (the aggregate value of `V`) is a variable, `B` is returned as the aggregated answer. Otherwise, entailment is checked: if `A` entails `B`, `appl_answer/2` succeeds, and it fails otherwise.

4.4 Adapting the Answer Management of TCLP

The Modular TCLP framework further rewrites the program in Fig. 2, right, to add at the end of each clause a call to the predicate `new_answer/0` (Fig. 6), which saves answers in the answer table.

This rewritten predicate is called through a meta-predicate `tabled_call/1` (Fig. 5), that executes the call entailment phase. `store_projection/2` retrieves the current value of the aggregate, and `call_entail/2` detects if the current call entails a previous generator by comparing their projections, i.e., their aggregates. If that is the case, the call is suspended by `suspend_consumer/1`; otherwise, the new call is made a generator and executed (with `save_generator/3` and `execute_generator/2` resp.) When the generators terminate and/or the consumers are resumed, answer consistency is checked and `apply_answer/2` applies all the answers collected during the execution of the generator.

```
tabled_call(Call) :-
    call_lookup_table(Call, Vars, Gen),
    'store_projection'(Vars, ProjStore),
    (   projstore_Gs(Gen, List_GenProjStore),
        member(ProjStore_G, List_GenProjStore),
        'call_entail'(ProjStore, ProjStore_G) ->
        suspend_consumer(Call)
    ;   save_generator(Gen, ProjStore_G, ProjStore),
        execute_generator(Gen, ProjStore_G),
    ),
    answers(Gen, ProjStore_G, List_Ans),
    member(Ans, List_Ans),
    projstore_As(Ans, List_AnsProjStore),
    member(ProjStore_A, List_AnsProjStore),
    'apply_answer'(Vars, ProjStore_A).
```

Fig. 5. `tabled_call`

```
1  new_answer :-
2      answer_lookup_table(Vars, Ans),
3      'store_projection'(Vars, ProjStore),
4      (   projstore_As(Ans, List_AnsProjStore),
5          member(ProjStore_A, List_AnsProjStore),
6          'answer_compare'(ProjStore, ProjStore_A, Res),
7          (    Res == '=<'                      % Discard ProjStore
8          ;    Res ==  '>',                     % Remove ProjStore_A
9               remove_answer(ProjStore_A),
10              fail
11         ;    Res == '$new'(NewProjStore),     % Save NewProjStore
12              remove_answer(ProjStore_A),
13              save_answer(Ans, NewProjStore)
14         ), !
15     ;   save_answer(Ans, ProjStore)           % Save ProjStore
16     ), !, fail.
```

Fig. 6. Extended implementation of `new_answer/0`.

`new_answer/0` (Fig. 6) collects the answers executing the answer entailment phase. Lines 7 to 10 perform the entailment check while lines 11 to 13 can join an incoming answer with previous answers into a new answer, and remove the previous answers [12]. This is used to *combine* two points `A1` and `A2` of a lattice into `A1` $\sqcup$ `A2` and, for example, store abstractions of answers. Such abstraction may lose some precision, but this can be acceptable for some applications (e.g., in abstract interpretation).

4.5 Non-Lattice Aggregates

We presented aggregates that are defined over lattices where the join operation is commutative, associative, and idempotent. However, there are many common aggregates that can be implemented using ATCLP but that do not satisfy some of the properties listed above. As a consequence, their execution may not completely align with LFP semantics. This is the case of `sum`, which can be defined using the join operator, but which does not have a sound definition for entailment.

Example 8 (probability of paths in a graph).

Let us consider a (cyclic) graph where each edge has a transition probability. We want the probability `P` of reaching a node `N` from another node `a`. `P` is the sum of the transition probabilities of all possible paths from `a` to `N`. Then, on one hand we have to multiply the probability of every traversed edge to calculate the probability of a path and, on the other hand, we have to add probabilities for every path. We define an aggregate (resp., `sum` and `thr(Epsilon)`) for each of these.

Incrementally adding path probabilities (in general, numbers) is easy by adding every new answer to the previous value. This behaves as expected when we have a finite set of answers to add. For non-cyclic graphs, the model is finite and computing all the paths and their sum is possible. However, in case of cycles, edges within loops may have to be traversed an unbounded number of times, and their contribution to the final solution decreases with every loop.

A possible strategy is to *discard* edges when their contribution goes below a certain user-defined threshold. With a somewhat ad-hoc reading of this condition, we can say that new solutions with a difference small enough w.r.t. existing solutions entail these previous solutions and therefore they ought not to be taken into account. This can be expressed in our framework by defining another aggregate that decides, via entailment, when further advancing in a path does not contribute enough.

```
1  :- table reach(_,sum).
2  :- table path(_,_,thr(0.001)).
3
4  entails(sum,_,_) :- fails.
5  join(sum, A, B, C) :-
6      C  is A + B.
7  entails(thr(Epsilon), A, B):-
8      A < Epsilon * B.
9  reach(N,P) :- path(a,N,P).
10
11 path(X,Y,P) :-
12         edge(X,Y,P).
13 path(X,Y,P) :-
14         edge(X,Z,P1),
15         path(Z,Y,P2),
16         P is P1 * P2.
```

In this example, for each node `N`, the predicate `reach(_,sum)` aggregates in its second argument the sum of the transition probabilities of the paths from `a` to `N`. Note we want to add all distances; therefore we define the entailment of `sum` to be always false. Since cyclic graphs have infinitely many paths, we have implemented the `threshold` aggregate, denoted by `thr(Epsilon)` to discard paths between `X` and `Y` whose relative contribution to the final results w.r.t. the contribution of another path falls below `Epsilon`.

5 Experimental Evaluation

We will now evaluate the expressiveness and performance of ATCLP w.r.t. pure Prolog and tabling. The ATCLP framework presented in this paper is based on TCLP, that is in turn implemented in Ciao Prolog. The examples, benchmarks, and a Ciao Prolog distribution including the libraries and frameworks presented in this paper are available at

	Prolog	tabling	ATCLP
3x3	1051	**167** (2)	359 **(1)**
4x4'	> 5 min	**10166** (130)	15194 **(30)**
4x4"	> 5 min	out of mem.	**134918** **(252)**

Table 1. Run time (ms), between parentheses the memory usage (in Mb) for Minimax with different initial boards.

`http://www.cliplab.org/papers/padl2019-atclp/`.[4] All the experiments were performed on a Mac OS X 10.13.6 laptop with a 2 GHz Intel Core i5. Times are given in milliseconds.

We will first evaluate some implementations of the well-known minimax algorithm applied to (an extended version of) TicTacToe. Our starting point is the Prolog version from [10,3] that uses `bagof/3` to collect the possible movements from a TicTacToe position and selects the best one. Thanks to the expressiveness of ATCLP, our code for the core minimax procedure (below) is considerably more compact (i.e., less number of predicates and arguments per predicate) than the equivalent Prolog or tabling code.

```
:- table minimax(_, first, best).

minimax(Pos, NextPos, (Pos,Val)) :-
    move(Pos, NextPos),                  % Chose a move
    minimax(NextPos, _, (NextPos, Val)).
minimax(Pos, Pos, (Pos,Val)) :-
   \+ move(Pos, _),                      % Final position
   utility(Pos,Val).                     % Calculate score

entails(best,(Pos,ValA),(Pos,ValB)) :-
    min_to_move(Pos), ValA >= ValB.      % Minimizing
entails(best,(Pos,ValA),(Pos,ValB)) :-
    max_to_move(Pos), ValA =< ValB.      % Maximizing
entails(first,_,_) :- true.              % Chose first best option
```

The ATCLP code chooses the best movement by applying the `best` aggregate which discards movements with worst (resp., best, depending on the current player) value. The infrastructure for aggregates transparently keeps track of gathering solutions and retains only the most relevant one at each moment. Note that we are using two different aggregates functions in the same predicate: `best` takes care of minimization/maximization and `first` retains only the first solution found among those with the same score.

We compared execution time and memory usage in two scenarios: determining the best initial movement for a 3×3 TicTacToe board and determining the best movement for a 4×4 TicTacToe starting at two different positions. In all three cases the remaining game tree was completely explored. The results (Table 1) show that the Prolog version is the slowest, with the tabling version being faster than the ATCLP version. However,

[4] Stable versions of Ciao Prolog are available at `http://www.ciao-lang.org`. However, ATCLP is still in development and not fully available yet in the stable versions.

	Prolog	Tabling	ATCLP
game_data_01	8062.49	14.66	**2.89**
game_data_02	> 5 min.	37.59	**4.87**
game_data_03	> 5 min.	1071.26	**19.61**
game_data_04	> 5 min.	4883.00	**23.21**

Table 2. Run time (ms) comparison for *Games* with different scenarios.

the ATCLP version behaves considerably better than tabling in terms of table memory consumption (between parentheses, in Mb). This is because viewing aggregates as constraints automatically stops the search as soon as the value of an aggregate is worse than a previously found one. That makes the ATCLP version to terminate for cases where regular tabling runs out of memory.

The second benchmark is the *Game* problem presented in the LP/CP contest of ICLP 2015 (`http://picat-lang.org/lp_cp_pc/Games.html`). The problem can be seen as a graph traversal where the movements represent a decision regarding whether to repeat the same game or play a new one. There are two parameters to optimize: `T`, the remaining money, and `F`, the fun we have had (which can be negative). The final goal have as much fun as possible, for which one has to keep as much money as possible. The core of the algorithm, where we again want to stress its compactness, follows:

```
1  :- table total_fun(max).
2  total_fun(F) :-
3      reach(initial,end,_,F).
4
5  :- table reach(_,_,max,max).
6  reach(GameA,GameB,T,F) :-
7      edge(GameA,GameB,T,F).
8  reach(GameA,GameB,Tf,Ff) :-
9      reach(GameA,GameZ,T1,F1),
10     edge(GameZ,GameB,T2,F2),
11     Ff is F1 + F2,
12     Tm is T1 + T2, Tm >= 0,
13     ( cap(Cap), Tm > Cap ->
14     Tf is Cap ; Tf is Tm ).
```

We developed three versions of a program to solve this problem using Prolog, tabling, and ATCLP. Table 2 shows that the ATCLP on-the-fly aggregate computation performs better than either Prolog or tabling, since ATCLP does not try to evaluate states where `T` and `F` are worse than in states already evaluated.

6 Conclusion and Future Work

We have presented a framework to implement a type of aggregates, defined over a lattice structure, whose behavior is consistent with the least fixpoint semantics. We provide an interface so that final users can define the basic operations on which the aggregates are built. We validated the flexibility and expressiveness of our framework through several examples; we also evaluated their performance in a couple of benchmarks, which showed a positive balance between memory consumption and execution speed.

Among the immediate future plans, we want to work on increasing the performance of the system and improve the user interface. In many cases, the `entails/3` and `join/4` predicates can directly be generated from a mode definition by providing

a predicate name. While this will not enhance performance or give more flexibility, it will make using the ATCLP interface more user-friendly. We also plan to include with ATCLP a library of commonly-used aggregate functions.

References

1. Arias, J., Carro, M.: Description and Evaluation of a Generic Design to Integrate CLP and Tabled Execution. In: Int'l. Symposium on Principles and Practice of Declarative Programming. pp. 10–23. ACM (Sept 2016)
2. Arias, J., Carro, M.: Description, Implementation, and Evaluation of a Generic Design for Tabled CLP. Theory and Practice of Logic Programming (to appear) (2018)
3. Bratko, I.: Prolog programming for artificial intelligence. Pearson education (Jan 2001)
4. Chico de Guzmán, P., Carro, M., Hermenegildo, M.V., Stuckey, P.: A General Implementation Framework for Tabled CLP. In: Int'l. Symposium on Functional and Logic Programming (FLOPS'12). pp. 104–119. No. 7294 in LNCS, Springer Verlag (May 2012)
5. Cui, B., Warren, D.S.: A system for Tabled Constraint Logic Programming. In: Int'l. Conference on Computational Logic. LNCS, vol. 1861, pp. 478–492 (Dec 2000)
6. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Software: Practice and Experience (1), 75–94 (Jan 2008)
7. Holzbaur, C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification. In: Int'l. Symposium on Programming Language Implementation and Logic Programming. pp. 260–268. No. 631 in LNCS, Springer Verlag (Aug 1992)
8. Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In: Int'l. Simposium on Logic Programming. pp. 387–401. Citeseer (Oct 1991)
9. Pelov, N., Denecker, M., Bruynooghe, M.: Well-Founded and Stable Semantics of Logic Programs with Aggregates. Theory and Practice of Logic Programming (3), 301–353 (Jun 2007)
10. Picard, G.: Artificial intelligence - implementing minimax with prolog. `https://www.emse.fr/~picard/cours/ai/minimax/`
11. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming (1-2), 5–34 (Jan 2012)
12. Schrijvers, T., Demoen, B., Warren, D.S.: TCHR: a Framework for Tabled CLP. Theory and Practice of Logic Programming (4), 491–526 (Jul 2008)
13. Swift, T., Warren, D.S.: Tabling with answer subsumption: Implementation, applications and performance. In: Logics in Artificial Intelligence. vol. 6341, pp. 300–312 (Sept 2010)
14. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming (1-2), 157–187 (Jan 2012)
15. Vandenbroucke, A., Pirog, M., Desouter, B., Schrijvers, T.: Tabling with Sound Answer Subsumption. Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (5-6), 933–949 (Oct 2016)
16. Zhou, N.: The Language Features and Architecture of B-Prolog. Theory and Practice of Logic Programming (1-2), 189–218 (Jan 2012)
17. Zhou, N.F., Kameya, Y., Sato, T.: Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In: Int'l. Conference on Tools with Artificial Intelligence. pp. 213–218. No. 2, IEEE (Oct 2010)