# *Description, Implementation, and Evaluation of a Generic Design for Tabled CLP*

Joaquín Arias[1,2]

Manuel Carro[1,2]

joaquin.arias@imdea.org, manuel.carro@{imdea.org,upm.es}
[1]*IMDEA Software Institute,* [2]*Universidad Politécnica de Madrid*

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

## Appendix A CLP Trees and TCLP Forests

Prolog and CLP follow a depth-first search strategy with chronological backtracking. The computation rule selects constraints and literals from the resolvent from left to right. Literals are resolved against the clauses of the program, selected from top to bottom. When a literal unifies with a clause head, it is substituted by the body of the clause after applying the unifier obtained from the literal-head unification. If a derivation branch fails because there are no more matching clauses or the constraint store is inconsistent, the evaluation backtracks to the youngest literal that has a candidate matching clause. Depth-first search is incomplete and in general not all answers can be computed. Moreover, there are programs with finite derivations for which logically equivalent programs produce infinite derivations. The use of TCLP can work around this issue in many cases.

We will show the CLP trees and the TCLP forests for the query `?- D #< 150, dist(a,Y,D).` for two logically equivalent versions of the `dist/3` program: with left recursion (Fig. 1, right) and with right recursion (Fig. 2, right). We use the graph in Fig. A 1, where the length of one of the edges is defined with constraints.

```
1  edge(a, b, 50).
2  edge(b, a,  D) :-
3       D #> 25,
4       D #< 35.
```

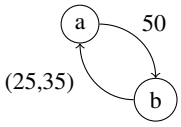



Fig. A 1: Graph definition. (25,35) is the open interval from 25 to 35.

Fig. A 2 and Fig. A 3 (top) are the CLP trees of the right- / left-recursive programs respectively. Fig. A 3 (bottom) and Fig. A 4 are the TCLP forest of the left- / right-recursive programs respectively. In these figures, the nodes of the trees represent the states (Def. 2) of the computation. A state is a tuple $\langle R, c \rangle$, where $R$ is a sequence of goals, $[g_1, g_2, \ldots, g_n]$ and $c$ is a conjunction of constraints. The numbers attached to each state indicates the order in which they are created.

On the one hand, Fig. A 2 shows a finite CLP tree which finds all the answers and Fig. A 3 (top) shows an infinite CLP tree caused by the left recursion. On the other hand, Fig. A

3 (bottom) and Fig. A 4 show that the TCLP forest for both programs are finite and all the answers to the query are found, since the use of tabling makes it terminate with left recursion as well.

### *A.1 CLP Tree of `dist/3` with Right Recursion*

Fig. A 2 shows the CLP tree of the query using the version of `dist/3` with right recursion (Fig. 2, right). We see that the evaluation of the recursive clause generates similar states (`s1`, `s4`, `s7` and `s10`), but in each iteration the domain of the constrained variable $\texttt{D2}_i$ is reduced. As a consequence, the constraint store in state `s13` is inconsistent and the evaluation of this derivation fails. The pending branches are evaluated upon backtracking. We explain now how we obtain some of the states; the rest are obtained similarly, so we will skip them:

**s1** the initial state is the representation of the query.

**s2i/ii** are obtained by resolving the literal `dist(a,Y,D)` against the two clauses of the program. The constraints $\texttt{Y}_1 = \texttt{Y} \wedge \texttt{D}_1 = \texttt{D}$ are added to the constraint store.

**s3** is obtained from the leftmost state `s2i` by adding the constraints of the resolvent $[\texttt{D1}_1\texttt{\#>0},\ \texttt{D2}_1\texttt{\#>0},\ \texttt{D}_1\texttt{\#=D1}_1\texttt{+D2}_1]$ to the constraint store.

**s4** is obtained by resolving the literal $\texttt{edge(a,Z}_1\texttt{,D1}_1\texttt{)}$. The constraint $\texttt{Z}_1 = \texttt{b} \wedge \texttt{D1}_1 = 50$ reduces the domain[1] of $\texttt{D2}_1$ to $\texttt{D2}_1 > 0 \wedge \texttt{D2}_1 < 100$.

**s7** is obtained by resolving the literal $\texttt{edge(b,Z}_2\texttt{,D1}_2\texttt{)}$. The constraint $\texttt{Z}_2 = \texttt{a} \wedge \texttt{D1}_2 > 25 \wedge \texttt{D1}_2 < 35$ reduces the domain of $\texttt{D2}_2$ to $\texttt{D2}_2 > 0 \wedge \texttt{D2}_2 < 75$.

**s10** is obtained by resolving the literal $\texttt{edge(a,Z}_3\texttt{,D1}_3\texttt{)}$. The constraint $\texttt{Z}_3 = \texttt{b} \wedge \texttt{D1}_3 = 50$ reduces the domain of $\texttt{D2}_3$ to $\texttt{D2}_3 > 0 \wedge \texttt{D2}_3 < 25$.

**s13** is obtained by resolving the literal $\texttt{edge(b,Z}_4\texttt{,D1}_4\texttt{)}$. The constraint $\texttt{Z}_4 = \texttt{a} \wedge \texttt{D1}_4 > 25 \wedge \texttt{D1}_4 < 35$ is inconsistent with the current constraint store, $\texttt{D} < 150 \wedge \texttt{D} > 125 + \texttt{D1}_4 + \texttt{D2}_4 \wedge \texttt{D2}_4 > 0 \wedge \ldots$. Its child is a fail node.

**s14** is obtained, upon backtracking to the state `s11b` by resolving the literal $\texttt{edge(b,Y,D2}_3\texttt{)}$. However, it is also a failed derivation because the resulting constraint store is inconsistent.

**s15** is a final state of a successful derivation, obtained upon backtracking to the state `s8b` by resolving the literal $\texttt{edge(a,Y,D2}_2\texttt{)}$. The constraint $\texttt{Y} = \texttt{a} \wedge \texttt{D2}_3 > 25 \wedge \texttt{D2}_3 < 35$ is consistent with the constraint store.

**a1** is the first answer $\texttt{Y} = \texttt{a} \wedge \texttt{D} > 125 \wedge \texttt{D} < 315$, projected onto the variables of the query ($vars(Q) = \{\texttt{Y},\texttt{D}\}$).

**s16** is a final state obtained upon backtracking to the state `s5b`.

**a2** is the second answer, $\texttt{Y} = \texttt{a} \wedge \texttt{D} > 75 \wedge \texttt{D} < 85$.

**s17** is a final state obtained upon backtracking to the state `s2ii`.

**a3** is the third and last answer, $\texttt{Y} = \texttt{b} \wedge \texttt{D} = 50$

.

[1] We are considering a linear constraint solver over the rational numbers that from $\texttt{D} < 150 \wedge \texttt{D} = \texttt{D1}_1 + \texttt{D2}_1 \wedge \texttt{D1}_1 = 50$ it infers that $\texttt{D2}_1 < 100$

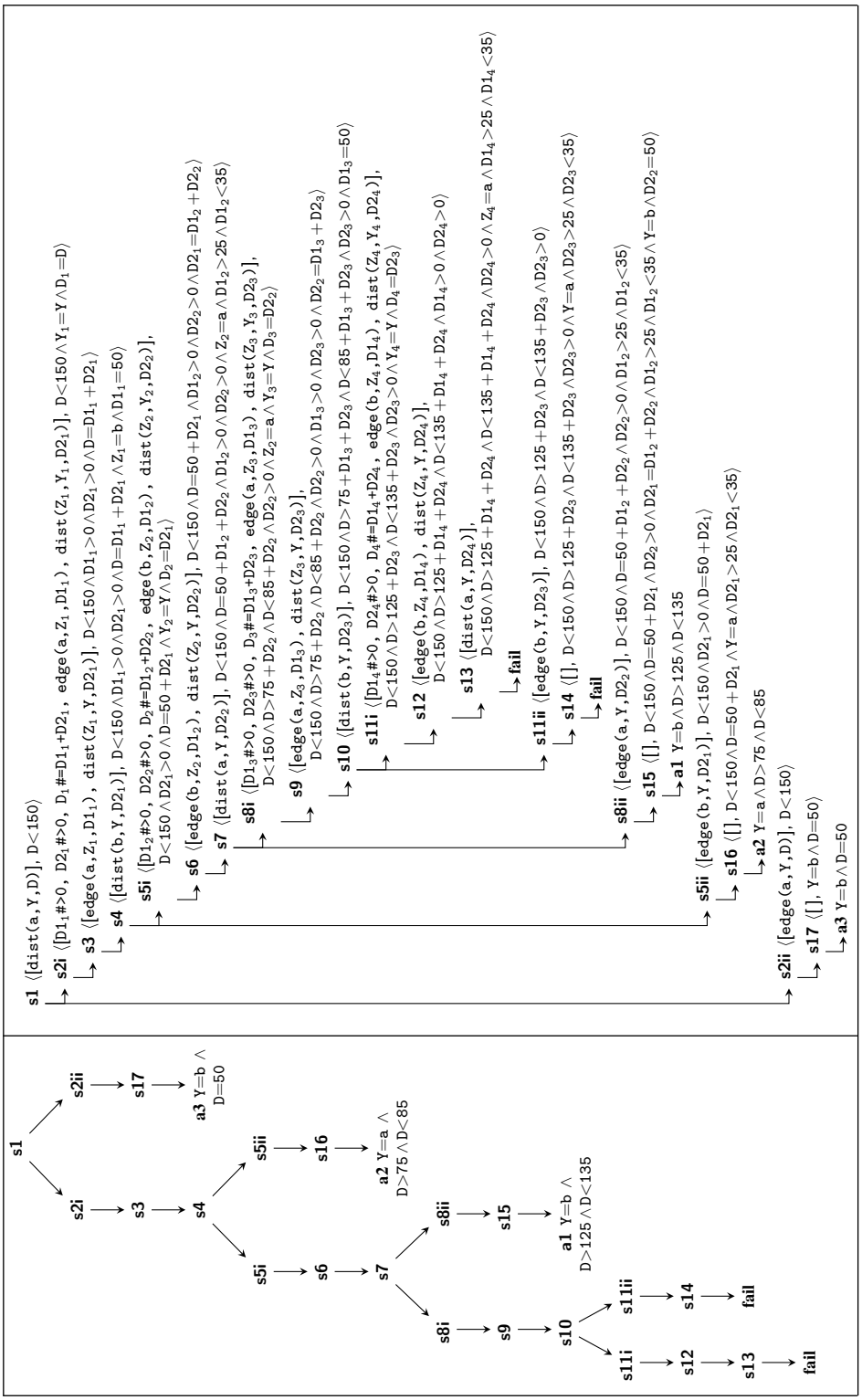



Fig. A 2: CLP tree of the query `?- D #< 150, dist(a,Y,D)` for `dist/3` with right recursion

### A.2 CLP Tree of `dist/3` with Left Recursion

Fig. A 3 (top) shows the CLP tree of the query to `dist/3` with left recursion (Fig. 1, right). We see that the recursive clause also generates similar states (`s1`, `s3`, `s5`, ...) but in this example the domain of the constrained variable $D1_i$ remains unchanged, and the evaluation therefore enters a loop. As before, we only explain how we obtain some of the states:

**s3** is obtained from the leftmost state s2i. The domain of $\texttt{D1}_1$ is $\texttt{D1}_1 > 0 \wedge \texttt{D1}_1 < 150$.
**s5** is obtained from the leftmost node s4i. The domain of $\texttt{D1}_2$ is $\texttt{D1}_2 > 0 \wedge \texttt{D1}_2 < 150$.
... the evaluation enters a loop.

Although the program that generates this CLP tree is logically equivalent to the previous one, this tree is infinite and no answers are found.

### A.3 TCLP Forest of `dist/3` with Left Recursion

Fig. A 3 (bottom) shows the TCLP forest for the query we have been using with the `dist/3` program written using left recursion (Fig. 1, right), where the set of tabled predicates is $Tab_P = \{\texttt{dist/3}\}$. The main point is that at state s3 the tabling engine detects that the evaluation of $\langle \texttt{dist(a,Z}_1\texttt{,D1}_1\texttt{)}, \texttt{D1}_1 > 0 \wedge \texttt{D1}_1 < 150 \rangle$ entails the generator $\langle \texttt{dist(a,V0,V1)}, \texttt{V1} < 150 \rangle$ and therefore it suspends the execution and waits until another generator feeds the suspended goal with answers. The evaluation of the state s2ii generates the first answer `a1` upon backtracking. Then, the tabling engine resumes the consumer with `a1` and generates `a2` which is used to generate `a3`. Finally, the evaluation fails after consuming `a3` and, since all the clauses have been evaluated and there are no more consumers to be resumed or answers to be consumed, the generator is marked as complete and all the answers are returned. We explain below how some of the states are obtained. The rest of the states are obtained similarly, so we skip them for brevity:

**s0** We omit the representation of the TCLP tree for the query $\tau_P(\texttt{dist(a,Y,D)}, \texttt{D} < 150)$ and its answer resolution.
**s1** the initial state of the TCLP tree $\tau_P(\texttt{dist(a,V0}_1\texttt{,V1}_1\texttt{)}, \texttt{V1}_1 < 150)$ is the renamed generator. (Def. 3).
**s2i/ii** are obtained by resolving the literal `dist(a,V0,V1)` against the two clauses of the program.
**s3** is obtained from the leftmost state s2i by adding the constraints to the constraint store as in the CLP tree.
**Ans(s1)** the tabled literal $\texttt{dist(a,Z}_1\texttt{,D1}_1\texttt{)}$ has to be resolved by answer resolution (Def. 3) using the answer from the current TCLP tree $\tau_P(\texttt{dist(a,V0}_1\texttt{,V1}_1\texttt{)}, \texttt{V1}_1 < 150)$ because, after renaming, the projection of the current constraint store onto the variables of the literal entails the projected constraint store of the generator: $\texttt{V1}_1 > 0 \wedge \texttt{V1}_1 < 150 \sqsubseteq \texttt{V1}_1 < 150$. Since the current TCLP tree is under construction and depends on itself, this branch derivation is suspended.
**s4** is a final state of a successful derivation. It is obtained, upon backtracking to the state s2ii, by resolving with `edge(a,V0,V1)`. The equations $\texttt{V0} = \texttt{b} \wedge \texttt{V1} = 50$ are consistent with the constraint store.

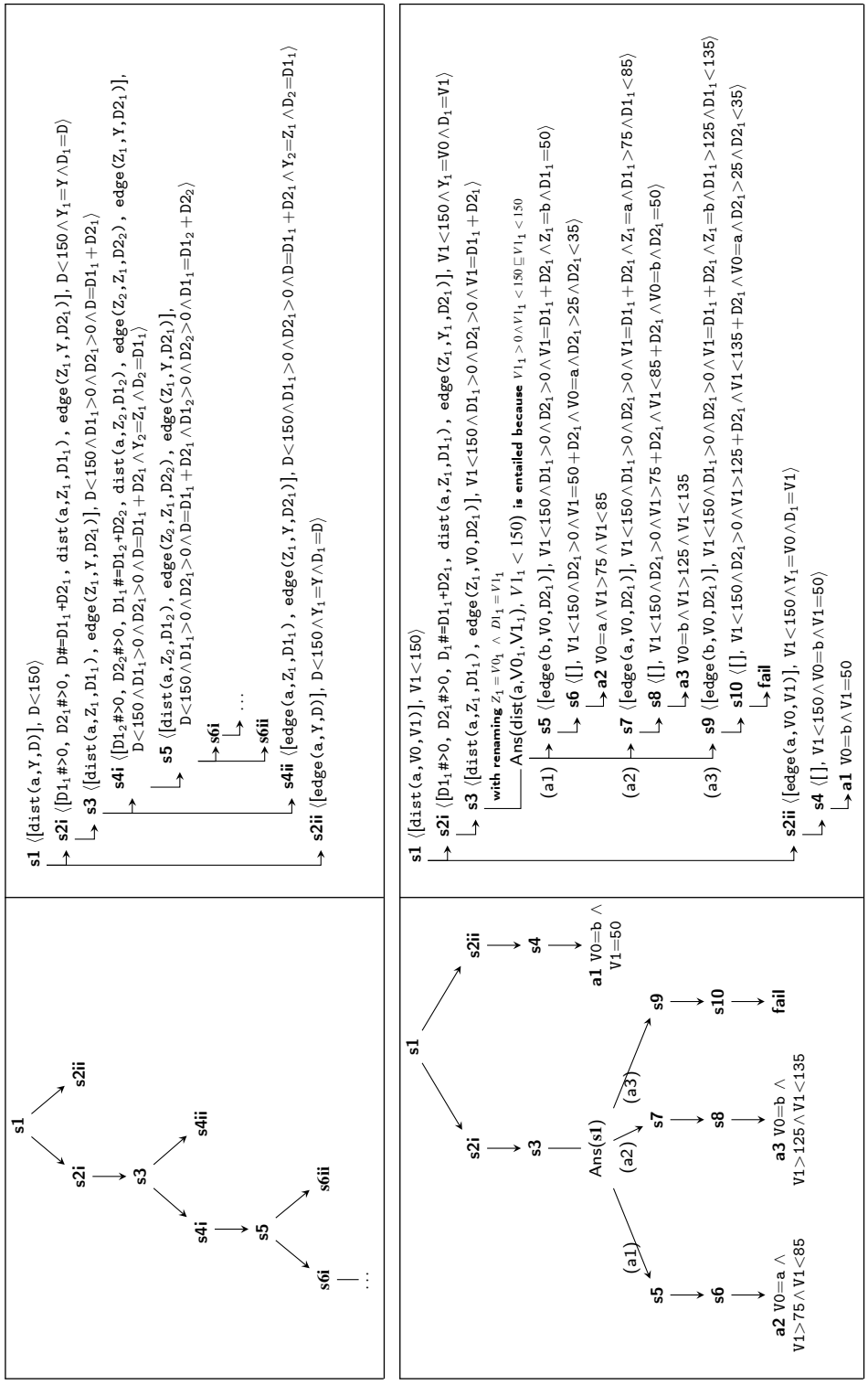



Fig. A 3: CLP tree (top) and TCLP-forest (bottom) of the query
`?- D #< 150, dist(a,Y,D)` for `dist/3` with left recursion

**a1** is the first answer, $\texttt{V0} = \texttt{b} \wedge \texttt{V1} = 50$ (Def. 4). Since is the first one, it is also the more general one.

**s5** is obtained from the state s3 (because there are no more branches) by answer resolution consuming a1 (Def. 4).

**s6** is a final state obtained by resolving the literal $\texttt{edge(b,V0,D2}_1\texttt{)}$.

**a2** is the second answer, $\texttt{V0} = \texttt{a} \wedge \texttt{V1} > 75 \wedge \texttt{V1} < 85$. It is neither more particular nor more general than a1.

**s7** is obtained from the state s3 by consuming a2.

**s8** is a final state.

**a3** is the third answer, $\texttt{V0} = \texttt{b} \wedge \texttt{V1} > 125 \wedge \texttt{V1} < 135$. It is neither more particular nor more general than a1 or a2.

**s9** is obtained from the state s3 by consuming, a3.

**s10** is a failed derivation because the resulting constraint store is inconsistent, $\texttt{V1} < 150 \wedge \ldots \wedge \texttt{V1} > 125 + \texttt{D2}_1 \wedge \texttt{D2}_1 > 25$. Its child is a fail node.

Note that the CLP execution entered a loop when resolving the state s3. Under TCLP, answer resolution avoids looping and the resulting TCLP forest is finite and complete (i.e., the leaves of the trees are either fail nodes or answers).

### *A.4 TCLP Forest of `dist/3` with Right Recursion*

Fig. A 4 shows the TCLP forest corresponding to querying the right recursive `dist/3` program (Fig. 2, right). This example is useful to show how the algorithm works with mutually dependent generators[2] and to see why not all the answers from a generator may be directly used by its consumers.

Unlike the left-recursive version, which shows only one TCLP tree (Fig. A 3, bottom), Fig. A 4 has two TCLP trees (one for each generator). That is because the left recursive version only sought paths from the node a, but the right recursive version creates a new TCLP tree at the state s4 to collect the paths from the node b, since `edge(a,b)` had been previously evaluated at state s3. As before we only explain how we obtain some of the states:

**s1** the TCLP tree $\tau_P(\texttt{dist(a,V0,V1)}, \texttt{V1} < 150)$ is created.

**s4** is obtained by resolving the literal $\texttt{edge(a,Z}_1\texttt{,D1}_1\texttt{)}$.

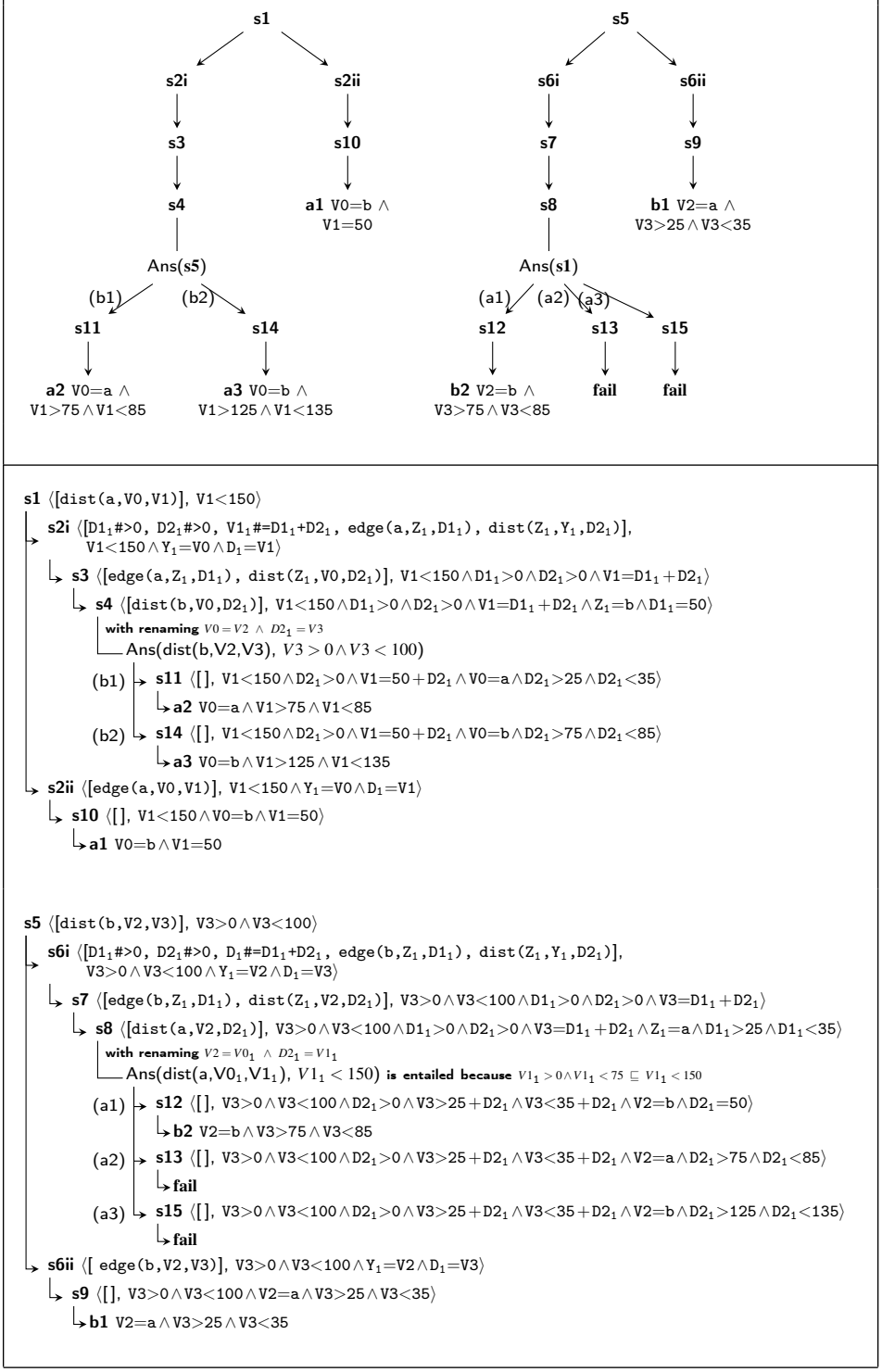
**Ans(s5)** the tabled literal $\texttt{dist(b,V0,D2}_1\texttt{)}$ is a new generator and a new TCLP tree $\tau_P(\texttt{dist(b,V2,V3)}, \texttt{V3} > 0 \wedge \texttt{V3} < 100)$ is created (Def. 3).

**s5** is the root node of the new TCLP tree.

**s6i/ii** are obtained by resolving the literal `dist(b,V2,V3)` against the clauses of the program.

**s8** is obtained by resolving the literal $\texttt{edge(b,Z}_1\texttt{,D1}_1\texttt{)}$.
In the state s8, the call $\langle \texttt{dist(a,V2,D2}_1\texttt{)}, \texttt{D2}_1 > 0 \wedge \texttt{D2}_1 < 75 \rangle$ is suspended because it entails the former generator $\langle \texttt{dist(a,V0}_1\texttt{,V1}_1\texttt{)}, \texttt{V1}_1 < 150 \rangle$.

[2] I.e., generators which consume answers from each other.




**s1** ⟨[dist(a,V0,V1)], V1<150⟩
- **s2i** ⟨[$D1_1$#>0, $D2_1$#>0, $V1_1$#=$D1_1$+$D2_1$, edge(a,$Z_1$,$D1_1$), dist($Z_1$,$Y_1$,$D2_1$)], V1<150∧$Y_1$=V0∧$D_1$=V1⟩
  - **s3** ⟨[edge(a,$Z_1$,$D1_1$), dist($Z_1$,V0,$D2_1$)], V1<150∧$D1_1$>0∧$D2_1$>0∧V1=$D1_1$+$D2_1$⟩
    - **s4** ⟨[dist(b,V0,$D2_1$)], V1<150∧$D1_1$>0∧$D2_1$>0∧V1=$D1_1$+$D2_1$∧$Z_1$=b∧$D1_1$=50⟩
      - with renaming $V0 = V2 \wedge D2_1 = V3$
        - Ans(dist(b,V2,V3), $V3 > 0 \wedge V3 < 100$)
      - (b1) **s11** ⟨[], V1<150∧$D2_1$>0∧V1=50+$D2_1$∧V0=a∧$D2_1$>25∧$D2_1$<35⟩
        - **a2** V0=a∧V1>75∧V1<85
      - (b2) **s14** ⟨[], V1<150∧$D2_1$>0∧V1=50+$D2_1$∧V0=b∧$D2_1$>75∧$D2_1$<85⟩
        - **a3** V0=b∧V1>125∧V1<135
- **s2ii** ⟨[edge(a,V0,V1)], V1<150∧$Y_1$=V0∧$D_1$=V1⟩
  - **s10** ⟨[], V1<150∧V0=b∧V1=50⟩
    - **a1** V0=b∧V1=50

**s5** ⟨[dist(b,V2,V3)], V3>0∧V3<100⟩
- **s6i** ⟨[$D1_1$#>0, $D2_1$#>0, $D_1$#=$D1_1$+$D2_1$, edge(b,$Z_1$,$D1_1$), dist($Z_1$,$Y_1$,$D2_1$)], V3>0∧V3<100∧$Y_1$=V2∧$D_1$=V3⟩
  - **s7** ⟨[edge(b,$Z_1$,$D1_1$), dist($Z_1$,V2,$D2_1$)], V3>0∧V3<100∧$D1_1$>0∧$D2_1$>0∧V3=$D1_1$+$D2_1$⟩
    - **s8** ⟨[dist(a,V2,$D2_1$)], V3>0∧V3<100∧$D1_1$>0∧$D2_1$>0∧V3=$D1_1$+$D2_1$∧$Z_1$=a∧$D1_1$>25∧$D1_1$<35⟩
      - with renaming $V2 = V0_1 \wedge D2_1 = V1_1$
        - Ans(dist(a,$V0_1$,$V1_1$), $V1_1 < 150$) is entailed because $V1_1 > 0 \wedge V1_1 < 75 \sqsubseteq V1_1 < 150$
      - (a1) **s12** ⟨[], V3>0∧V3<100∧$D2_1$>0∧V3>25+$D2_1$∧V3<35+$D2_1$∧V2=b∧$D2_1$=50⟩
        - **b2** V2=b∧V3>75∧V3<85
      - (a2) **s13** ⟨[], V3>0∧V3<100∧$D2_1$>0∧V3>25+$D2_1$∧V3<35+$D2_1$∧V2=a∧$D2_1$>75∧$D2_1$<85⟩
        - **fail**
      - (a3) **s15** ⟨[], V3>0∧V3<100∧$D2_1$>0∧V3>25+$D2_1$∧V3<35+$D2_1$∧V2=b∧$D2_1$>125∧$D2_1$<135⟩
        - **fail**
- **s6ii** ⟨[ edge(b,V2,V3)], V3>0∧V3<100∧$Y_1$=V2∧$D_1$=V3⟩
  - **s9** ⟨[], V3>0∧V3<100∧V2=a∧V3>25∧V3<35⟩
    - **b1** V2=a∧V3>25∧V3<35

Fig. A 4: TCLP-forest of the query `?- D #< 150, dist(a,Y,D)` for `dist/3` with right recursion

**Ans(s1)** the tabled literal $\texttt{dist(a,V2,D2}_1\texttt{)}$ is resolved with answer resolution (Def. 3) using the answers from the previous TCLP tree $\tau_P(\texttt{dist(a,V0}_1\texttt{,V1}_1\texttt{)},\texttt{V1}_1 < 150)$ because the renamed projection[3] of the current constraint store onto the variable of the literal entails the projected constraint store of the generator: $(\texttt{V1}_1 > 0 \wedge \texttt{V1}_1 < 75) \sqsubseteq \texttt{V1}_1 < 150$. Since the initial TCLP forest is under construction and depends on itself, the current branch derivation is suspended.
This suspension also causes the former generator to suspend at the state s4.

**s9** is a final state obtained upon backtracking to the state s6ii.

**b1** is the first answer of the second generator.
At this point the suspended calls can be resumed by consuming the answer b1 or by evaluating s2ii. The algorithm first tries to evaluate s2ii and then it will resume s4 consuming b1.

**s10** is a final state obtained upon backtracking to the state s2ii.

**a1** is the first answer of the first generator: $\texttt{V0} = \texttt{b} \wedge \texttt{V1} = 50$.

**s11** is a final state obtained from the state s4 by consuming b1.

**a2** is the second answer of the first generator: $\texttt{V0} = \texttt{a} \wedge \texttt{V1} > 75 \wedge \texttt{V1} < 85$.

**s12** is a final state obtained from the state s8 by consuming a1.

**b2** is the second answer of the second generator.

**s13** is a failed derivation obtained from s8 by consuming a2. It fails because the constraints $\texttt{V0} = \texttt{a} \wedge \texttt{V1} > 75 \wedge \texttt{V1} < 85$ are inconsistent with the current constraint store. Note that the projection of the constraint store of s8 onto V1 is $\texttt{V1} > 0 \wedge \texttt{V1} < 75$. Its child is a fail node.

**s14** is a final state obtained from the state s4 by consuming b2.

**a3** is the third answer of the first generator: $\texttt{V0} = \texttt{b} \wedge \texttt{V1} > 125 \wedge \texttt{V1} < 135$.

**s15** is a failed derivation obtained from s8 by consuming a3. Its child is a fail node.

This example illustrates why left recursion reduces the execution time and memory requirements when using tabling / TCLP: left recursion will usually create fewer generators. We have also seen that using answers from a more general call, as in the answer resolution of state s8 (i.e., the constraint store of the consumer $\texttt{V1}_1 > 0 \wedge \texttt{V1}_1 < 75$ is more particular than the constraint store of the generator $\texttt{V1}_1 < 150$), makes it necessary to filter the correct ones (i.e., answer resolution for a2 and a3 failed). This is not required in variant tabling because the answers from a generator are always valid for its consumers.

## Appendix B Step by Step Execution of `dist/3` under TCLP($\mathbb{Q}$)

The trace below shows the step by step execution of the TCLP version of the left recursive distance traversal program in Fig. 5, left, with the query `?- D #< 150, dist(a,Y,D).` using the graph in Fig. A 1. In this example we are using the TCLP($\mathbb{Q}$) interface (Section 4.3). Each step is annotated with the labels used in Fig. 8. The execution starts with the query `?- D #< 150, dist(a,Y,D)`:

[3] The projection of $\texttt{V3} > 0 \wedge \texttt{V3} < 100 \wedge \texttt{D1}_1 > 0 \wedge \texttt{D2}_1 > 0 \wedge \texttt{V3} = \texttt{D1}_1 + \texttt{D2}_1 \wedge \texttt{Z}_1 = \texttt{a} \wedge \texttt{D1}_1 > 25 \wedge \texttt{D1}_1 < 35$ onto $\texttt{D2}_1$ is $\texttt{D2}_1 > 0 \wedge \texttt{D2}_1 < 75$. After renaming $\texttt{D2}_1 = \texttt{V1}_1$, the resulting projection is $\texttt{V1}_1 > 0 \wedge \texttt{V1}_1 < 75$.

**0** the constraint `D#<150` in the query is added to the current store (state `s0`). Then $\langle$`dist(a,Y,D), D < 150`$\rangle$ is called and the tabling engine takes the control of the execution calling `tabled_call(dist_aux(a,Y,D))`.

**1** `call_lookup_table/3` initializes and saves (after renaming) `dist_aux(a,V0,V1)`, because it is the first occurrence, and returns `Vars=[D]` and `Gen=$1`, where `$1` is the reference for this generator.

**2** `store_projection([D],ProjStore)` returns `ProjStore=([V1],[V1#<150])`.

**3** `member/2` fails because the list of projected constraint stores associated to `Gen=$1` is empty.

**4** `save_generator/3` saves `([V1],[V1#<150])` in the list of projected constraint stores associated to `Gen=$1` (state `s1`).

**7** `execute_generator/2` evaluates the generator against the first clause of `dist_aux/3` and adds the body of the clause to the resolvent of the state `s2i`. Then the constraints of the resolvent, `[D1#>0, D2#>0, D#=D1+D2]`, are added to the constraint store (state `s3`) and $\langle$`dist(a,Z,D1), D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D = D1 + D2`$\rangle$ is called.

**0** the tabling engine reenters the tabled execution with `tabled_call(dist_aux(a,Z,D1))`.

**1** `call_lookup_table(dist_aux(a,Z,D1),Vars,Gen)` returns `Vars=[D1]` and `Gen=$1`, the reference to the previous generator, `dist_aux(a,V0,V1)`.

**2** `store_projection([D1], ProjStore)` returns `ProjStore=([V1], [V1#>0, V1#<150])`. For clarification, the projection of the current constraint store `D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D = D1 + D2` onto `D1` is `D1 > 0` $\wedge$ `D1 < 150`.

**3** `member/2` retrieves the projected constraint store `ProjStore_G=([V1], [V1#<150])`.

**5** `call_entail/2` succeeds because (`D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D = D1 + D2`) $\sqsubseteq$ `D1 < 150`.

**6** `suspend_consumer/1` suspends the current call `dist_aux(a,Z,D1)` (state s3, waiting for the answer of the current TCLP tree, `Ans(s1)`).

**7** The evaluation of the generator backtracks to evaluate the other clause (state s2ii). Now the current constraint store is `D#<150` and the call $\langle$`edge(a,Y,D), D < 150`$\rangle$ unifies with `edge(a,b,50)` (state `s4`). The first answer is found and `new_answer/0` is invoked to collect the answer.

**8** `answer_lookup_table/2` stores the Herbrand constraints[4] of the answer, `Y = b` $\wedge$ `D = 50`, returning `Vars=[]` and `Ans=$a1`, where `$a1` is the reference for this answer.

**9** `store_projection/2` returns `ProjStore=([],[])`.

**10** `member/2` fails because the list of projected constraint stores associated to `$a1` is empty.

**11** `save_answer/2` saves `([],[])` in the list of the answer constraints associated to `$a1` (state `a1`). The first answer is collected.

**14** the tabling engine resumes the goal suspended at state `s3` and `member/2` retrieves the Herbrand constraints `Y = b` $\wedge$ `D = 50` and the answer constraint `([],[])`.

[4] In solvers written in Prolog and implemented using attributed variables, such as CLP($\mathbb{Q}$) and CLP($\mathbb{R}$), it is usual that variables lose their association with the constraints where they appeared when these variables become ground. As ground terms do not have attributes attached, `D = 50` is handled as part of the Herbrand constraints.

```
fib(N,F) :-
    ⋮
    F #= F1 + F2,
    fib(N1, F1),
    fib(N2, F2).
```

```
fib(N,F) :-
    ⋮
    fib(N1, F1),
    fib(N2, F2),
    F #= F1 + F2.
```

Fig. C 1: Two versions of `fib/2`: $\mathbb{Q}$ and $\mathbb{R}$ (left) vs. $\mathbb{D}_{\leq}$ (right).

**15** `apply_answer/2` adds the answer to the current constraint store (state `s5`).

**7** the execution continues resolving $\langle \texttt{edge(b,Y,D2)}, \ldots \wedge \texttt{D2} < 100 \rangle$ which unifies with the clause `edge(b,a,D2):- D2#>25, D2#<35` (state `s6`). The second answer is found.

**8** `answer_lookup_table/2` stores the Herbrand constraints of the answer, $\texttt{V0} = \texttt{a}$, returning `Vars=[D]` and `Ans=$a2`.

**9** `store_projection/2` returns `ProjStore=([V1], [V1#>75,V1#<85])`.

**10** `member/2` fails because the list of projected constraint stores associated to `Ans=$a2` is empty.

**11** `save_answer/2` saves `([V1],[V1#>75,V1#<85])` in the list of answer constraints associated to `$a2` (state `a2`). The second answer is collected.

**14, 15, 7** the tabling engine resumes the suspended goal at state `s3` and consumes the second answer following the same steps as with the first one and generating the states `s7` and `s8`. The third answer has been found.

**8, 9, 10, 11** the answer is collected and `([V1],[V1#>125,V1#<135])` is saved in the list of answer constraint associated to `$a3` (state `a3`).

**14, 15** the tabling engine resumes the suspended goal at state `s3` and consumes the third answer.

**7** the execution fails resolving $\langle \texttt{edge(b,V0,D2}_1\texttt{)}, \texttt{V1} < 150 \wedge \ldots \wedge \texttt{D1}_1 < 135 \rangle$ (states `s9` and `s10`)

**14, 15** the generator has exhausted all the answers and it does not have any more dependencies, so `complete/0` marks the generator as complete. The query retrieves the answers from the generator one by one and returns them.

## Appendix C Comparison of Mod TCLP using $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{D}_{\leq}$

This section highlights that the modularity of TCLP makes it possible to choose the most adequate constraint solver for the specific problem, and that decision should not always be based solely on the performance of the constraint solver, but also on its expressiveness and/or precision. Since TCLP, unlike CLP, uses entailment checking extensively to decide whether to suspend and save / discard answers or not, the performance of entailment is more relevant than in CLP. It also makes its soundness (which can be challenged by e.g. numerical accuracy) critical, as incorrect entailment results can lead to non-termination or to unexpected termination.

We use the doubly recursive Fibonacci program (Fig. C 1). It is well-known that tabling reduces its complexity from exponential to linear, but, in addition, CLP makes it possible to run exactly the same program `fib/2` *backwards* to find the index of some Fibonacci

| | Mod TCLP($\mathbb{R}$) | Mod TCLP($\mathbb{Q}$) | Mod TCLP($\mathbb{D}_{\leq}$) |
|---|---|---|---|
| `fib(N, 832040)` | **25** | 61 | 147 |
| `fib(N, 28657)` | **16** | 40 | 69 |
| `fib(N, 610)` | **8** | 19 | 24 |
| `fib(N, 89)` | **5** | 12 | 13 |

Table C 1: Run time (ms) comparison for the `fib/2` using $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{D}_{\leq}$.

number by generating a system of equations whose solution is the index of the given Fibonacci number. Under CLP, the size of this system of equations grows exponentially with the index of the Fibonacci number. However, under TCLP, entailment makes redundant equations not to be added and solving them becomes less expensive.

We have run this benchmark using $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{D}_{\leq}$. Due to the characteristics of $\mathbb{D}_{\leq}$ (Section 5.1), the program for this constraint system is slightly different from the ones for $\mathbb{Q}$ and $\mathbb{R}$ (Section 4.3). In these two, constraints are placed before the recursive calls. However, $\mathbb{D}_{\leq}$ can have at most two variables per constraint, and therefore we had to move the constraint `F #= F1 + F2` to the end of the clause (Fig. C 1). This can be detrimental to the performance of Mod TCLP($\mathbb{D}_{\leq}$), as value propagation in the constraints is less effective.

Table C 1 shows the experimental results. First, note that the Mod TCLP($\mathbb{D}_{\leq}$) version is slower than any of the other two. While the implementation of CLP($\mathbb{D}_{\leq}$) is comparatively faster than CLP($\mathbb{R}$) and CLP($\mathbb{Q}$), moving the `F #= F1 + F2` to the end of the clause (which is necessary to satisfy the instantiation requirements of $\mathbb{D}_{\leq}$) reduces its usefulness to prune the generation of redundant constraints.

Additionally, note that although the solvers for $\mathbb{R}$ and $\mathbb{Q}$ are practically the same, Mod TCLP($\mathbb{R}$) is fastest in all cases, since it uses directly CPU floating point numbers while CLP($\mathbb{Q}$) implements rational numbers by software. However, there is a drawback: floating point arithmetic is not accurate, and when CLP($\mathbb{R}$) approximates its results, it can cause (depending on the particular program) non-termination. That would be the case for a query such as `?- fib(N, 23416728348467685)`, which terminates correctly with Mod TCLP($\mathbb{Q}$), but it does not (in under five minutes) with Mod TCLP($\mathbb{R}$), since the termination condition never holds.