A Simple Table Space Design for Retroactive Call Subsumption

Flávio Cruz and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto

15th Portuguese Conference on Artificial Intelligence

Tabling in Logic Programming

Tabled Resolution

- Tabling works by storing generated answers in a **table space** and then by reusing those answers in **similar calls**.

Tabling in Logic Programming

Tabled Resolution

- Tabling works by storing generated answers in a **table space** and then by reusing those answers in **similar calls**.
- First calls to tabled subgoals are considered **generators**, because they will generate answers through the execution of code.
- Similar calls are named **consumers**, since they will consume the answers generated by the corresponding similar subgoal, instead of being re-evaluated against the program clauses.

Tabling in Logic Programming

Tabled Resolution

- Tabling works by storing generated answers in a **table space** and then by reusing those answers in **similar calls**.
- First calls to tabled subgoals are considered **generators**, because they will generate answers through the execution of code.
- Similar calls are named **consumers**, since they will consume the answers generated by the corresponding similar subgoal, instead of being re-evaluated against the program clauses.
- Similar calls are found by using a **call similarity test** which determines if a subgoal will be a **generator** or a **consumer**.
- There are two popular similarity tests for subgoals:
  - **Call by variance** (or variant-based tabling).
  - **Call by subsumption** (or subsumption-based tabling).

Tabling in Logic Programming

Variant-Based Tabling

- Subgoal *A* is similar to *B* if they are the same by renaming the variables.

Example

p(f(X),1,Y) and **p(f(A),1,Z)** are **variant** because both can be transformed into **p(f(VAR0),1,VAR1)**.

Tabling in Logic Programming

Variant-Based Tabling

- Subgoal *A* is similar to *B* if they are the same by renaming the variables.

Example

p(f(X),1,Y) and **p(f(A),1,Z)** are **variant** because both can be transformed into **p(f(VAR0),1,VAR1)**.

- Implemented in most tabling systems: XSB Prolog, Yap Prolog, ...
- Relatively easy to implement efficiently.

Tabling in Logic Programming

Subsumption-Based Tabling

- Subgoal A is similar to B when A is more specific than B (or B is more general than A).

Example

p(f(X),1,f(a)) is more specific than **p(Y,1,Z)** because there is a substitution $\{\mathbf{Y = f(X),\ Z = f(a)}\}$ that makes **p(f(X),1,f(a))** an **instance** of **p(Y,1,Z)**.

Tabling in Logic Programming

Subsumption-Based Tabling

- Subgoal A is similar to B when A is more specific than B (or B is more general than A).

Example

p(f(X),1,f(a)) is more specific than **p(Y,1,Z)** because there is a substitution $\{\mathbf{Y = f(X),\ Z = f(a)}\}$ that makes **p(f(X),1,f(a))** an **instance** of **p(Y,1,Z)**.

- **Less code is executed** because subsumed subgoals can reuse answers instead of executing their own code.
- More answers are shared across subgoals, therefore there is **less redundancy in the table space**.

Subsumption-Based Tabling

Time Stamped Tries

- The most widely mechanism to support subsumption-based tabling is the **Time-Stamped Tries (TST)** approach that stores answers with **timestamp** information.
- The table space is based on **tries**, which are tree-based data structures where common prefixes are represented only once.
- Two levels of tries are used:
  - A **subgoal trie** for each tabled predicate.
  - An **answer trie** for each **subsuming subgoal**.

Subsumption-Based Tabling

Time Stamped Tries

- The most widely mechanism to support subsumption-based tabling is the **Time-Stamped Tries (TST)** approach that stores answers with **timestamp** information.
- The table space is based on **tries**, which are tree-based data structures where common prefixes are represented only once.
- Two levels of tries are used:
  - A **subgoal trie** for each tabled predicate.
  - An **answer trie** for each **subsuming subgoal**.
- At the end of each subgoal trie leaf, we may have a:
  - **Subsumptive subgoal frame** storing information about a subsuming (more general) subgoal which includes a **time stamped answer trie** with the answers for the subgoal.

Subsumption-Based Tabling

Time Stamped Tries

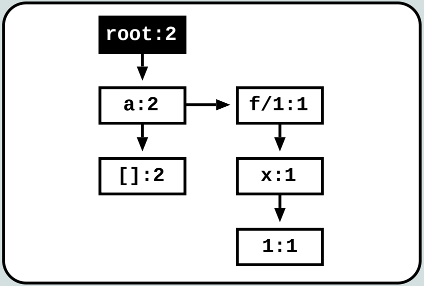
- The most widely mechanism to support subsumption-based tabling is the **Time-Stamped Tries (TST)** approach that stores answers with **timestamp** information.
- The table space is based on **tries**, which are tree-based data structures where common prefixes are represented only once.
- Two levels of tries are used:
  - A **subgoal trie** for each tabled predicate.
  - An **answer trie** for each **subsuming subgoal**.
- At the end of each subgoal trie leaf, we may have a:
  - **Subsumptive subgoal frame** storing information about a subsuming (more general) subgoal which includes a **time stamped answer trie** with the answers for the subgoal.
  - **Subsumed subgoal frame** storing information about a subsumed subgoal which includes a **timestamp** and a pointer to the **time stamped answer trie** of the corresponding subsuming subgoal.

Subsumption-Based Tabling

Time Stamped Tries

- Each subsumptive subgoal frame has a **global timestamp T**, that is incremented whenever a new answer is inserted. With a new answer, we set the answer trie path, from leaf to root, to **T+1**.

TST with answers <f(x),1> and <a,[]>

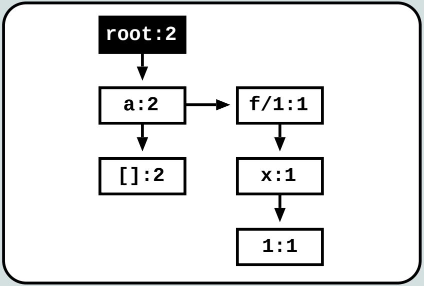


Subsumption-Based Tabling

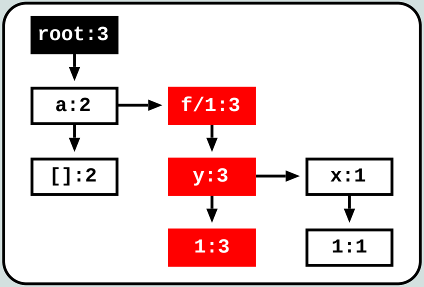
Time Stamped Tries

- Each subsumptive subgoal frame has a **global timestamp T**, that is incremented whenever a new answer is inserted. With a new answer, we set the answer trie path, from leaf to root, to **T+1**.

TST with answers <f(x),1> and <a,[]>



TST after inserting <f(y),1>

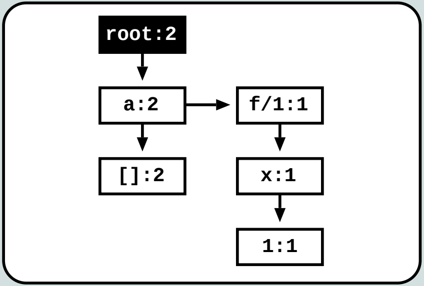


Subsumption-Based Tabling

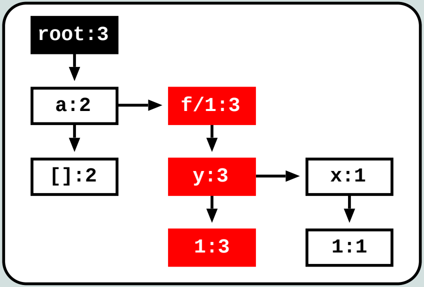
Time Stamped Tries

- Each subsumptive subgoal frame has a **global timestamp T**, that is incremented whenever a new answer is inserted. With a new answer, we set the answer trie path, from leaf to root, to **T+1**.

TST with answers <f(x),1> and <a,[]>



TST after inserting <f(y),1>



- Each subsumed subgoal frame uses its timestamp to retrieve new relevant answers as execution proceeds.

Subsumption-Based Tabling

Disadvantages

- The mechanisms used to support subsumption-based tabling are **harder to implement**.
- For example, in XSB Prolog, if a more general subgoal is called before specific subgoals, answer reuse will happen, but if specific subgoals are called before a more general subgoal, **no reuse will occur**.

Example

If **p(1,X)** is called **before** **p(X,Y)**, **p(1,X)** **will not reuse** the answers from **p(X,Y)**, but will execute code to generate its own answers.

Subsumption-Based Tabling

Retroactive Call Subsumption (RCS)

- We have developed a new resolution extension called **Retroactive Call Subsumption (RCS)** that supports subsumption-based tabling by allowing full sharing of answers among subsumptive subgoals, independently of the order they are called.

Example

If **p(1,X)** is called **before or after p(X,Y)**, **p(1,X)** **will reuse** the answers from **p(X,Y)**.

Subsumption-Based Tabling

Retroactive Call Subsumption (RCS)

- We have developed a new resolution extension called **Retroactive Call Subsumption (RCS)** that supports subsumption-based tabling by allowing full sharing of answers among subsumptive subgoals, independently of the order they are called.

Example

If **p(1,X)** is called **before or after p(X,Y)**, **p(1,X) will reuse** the answers from **p(X,Y)**.

- RCS selectively prunes the evaluation of a subgoal F when a more general subgoal G appears later on.

Subsumption-Based Tabling

Retroactive Call Subsumption (RCS)

- We have developed a new resolution extension called **Retroactive Call Subsumption (RCS)** that supports subsumption-based tabling by allowing full sharing of answers among subsumptive subgoals, independently of the order they are called.

Example

If **p(1,X)** is called **before or after p(X,Y)**, **p(1,X) will reuse** the answers from **p(X,Y)**.

- RCS selectively prunes the evaluation of a subgoal F when a more general subgoal G appears later on.
- RCS works by pruning the execution branch of F and then by restarting the evaluation of F as a consumer. By doing that, we **save execution time** by not executing code that would generate a subset of the answers we can find by executing G.

Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- `path(X,3):`

`1. path(X, 3)`

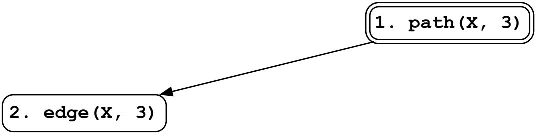
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3):



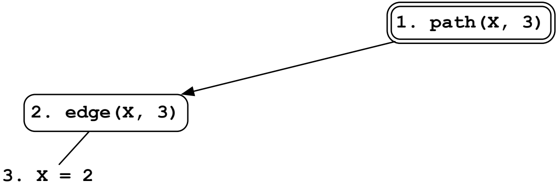
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>



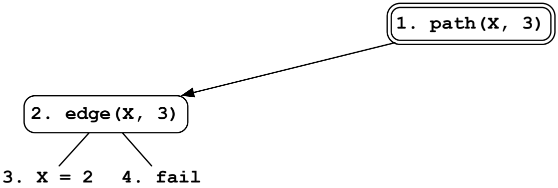
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>



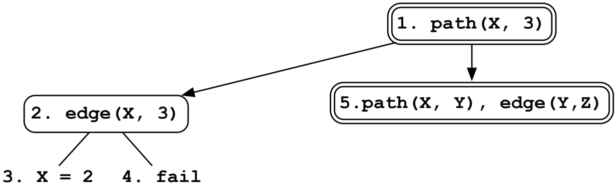
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y):



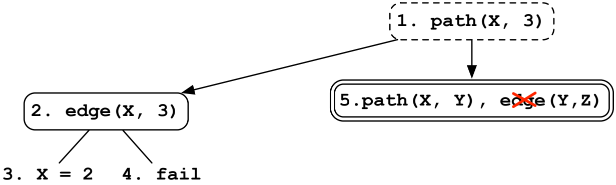
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y):



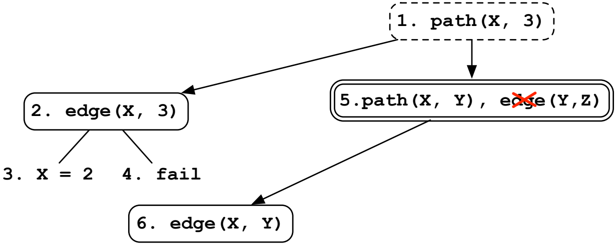
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y):



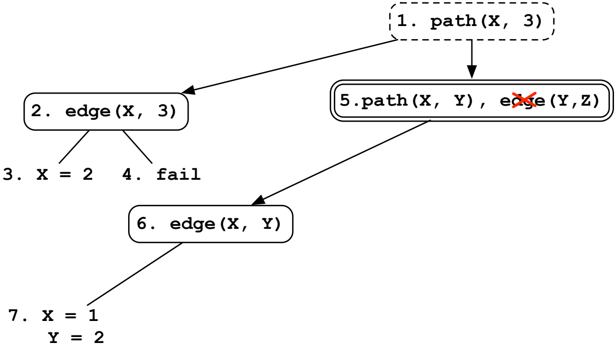
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2>



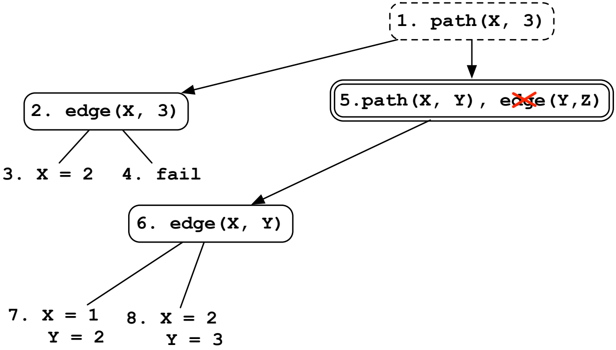
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>



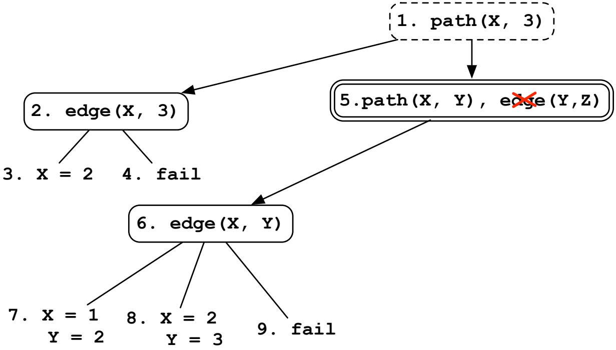
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>



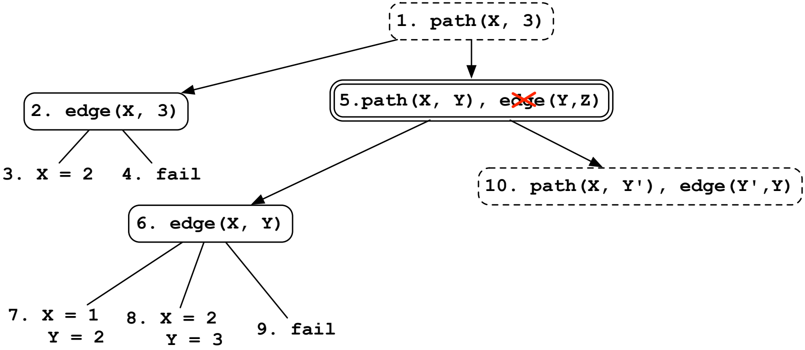
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>



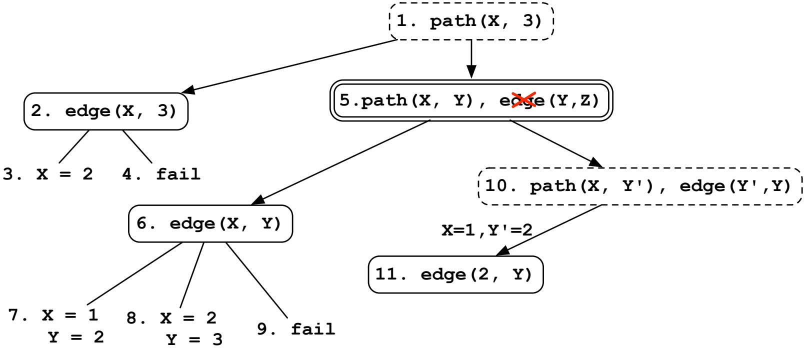
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>



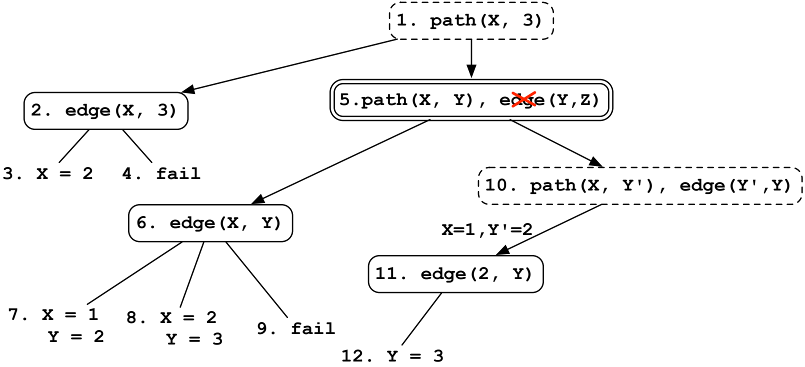
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



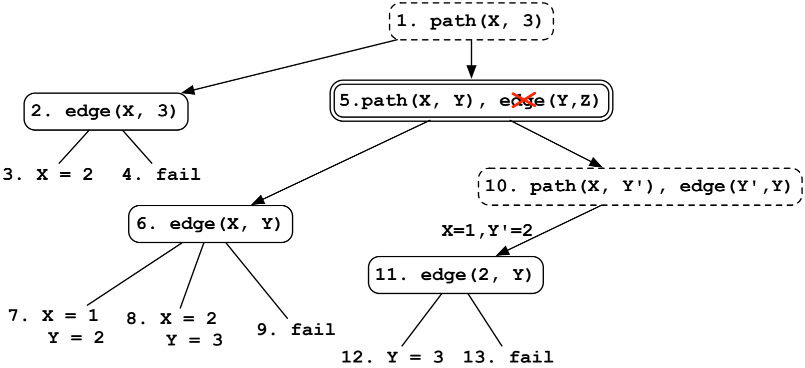
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3> (12) <X=1,Y=3>



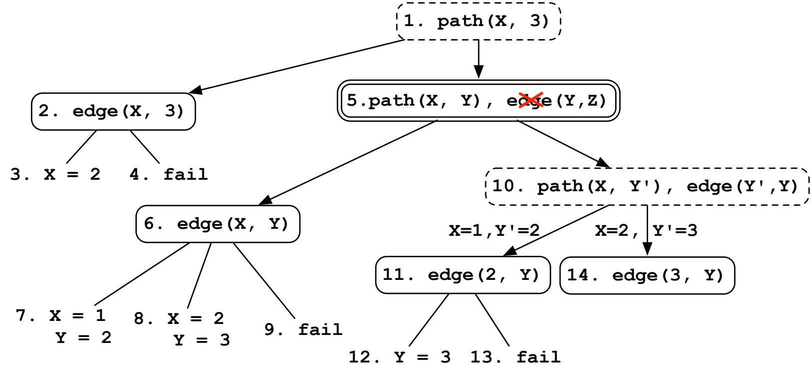
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



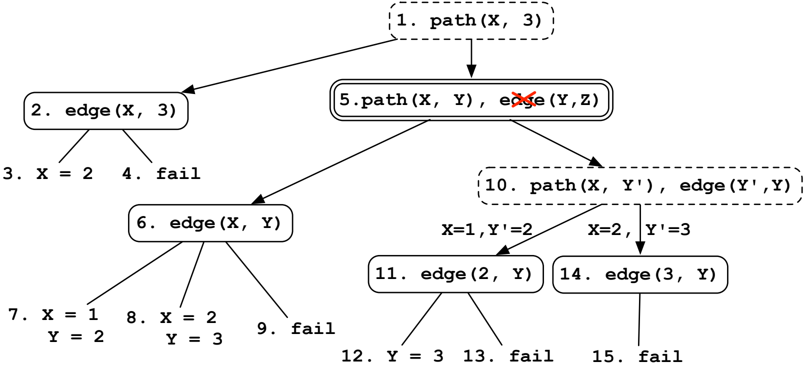
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



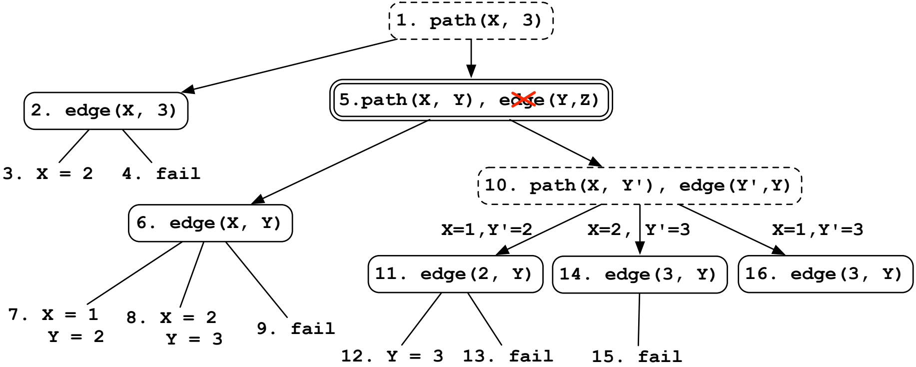
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3> (12) <X=1,Y=3>



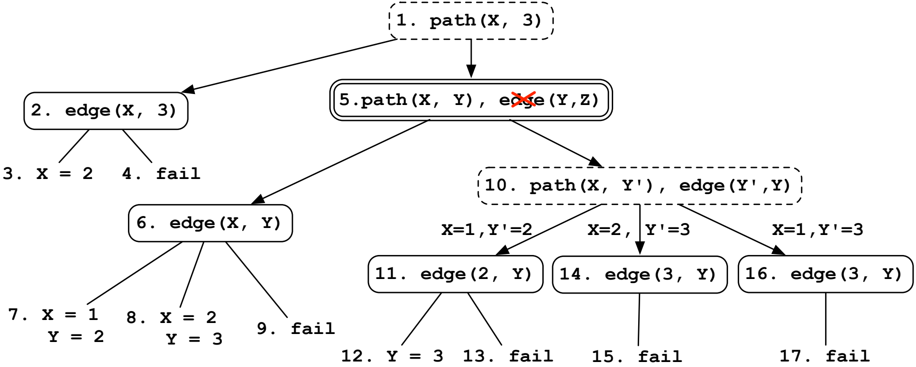
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



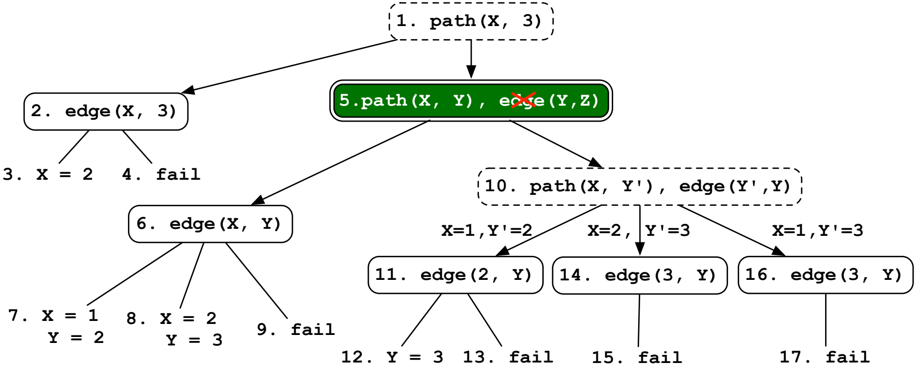
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



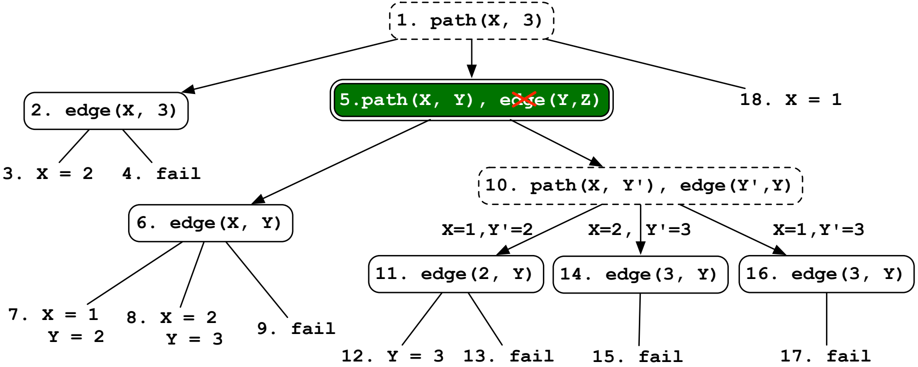
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2> (18) <X=1>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3> (12) <X=1,Y=3>



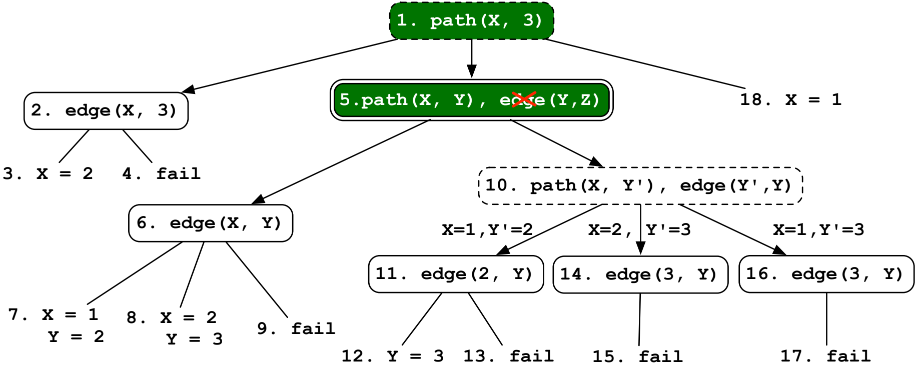
Retroactive Call Subsumption

Program

```
path(X,Z):- edge(X,Z).
path(X,Z):- path(X,Y), edge(Y,Z).
edge(1,2).
edge(2,3).
```

Table Space

- path(X,3): (3) <X=2> (18) <X=1>
- path(X,Y): (7) <X=1,Y=2> (8)<X=2,Y=3>
  (12) <X=1,Y=3>



Retroactive Call Subsumption

Table Space

- How do we design a table space that makes it efficient to **transform generators into consumers**?
- How do we guarantee that the newly transformed consumers do **not consume answers that were already generated** by them previously.

Retroactive Call Subsumption

Table Space

- How do we design a table space that makes it efficient to **transform generators into consumers**?
- How do we guarantee that the newly transformed consumers do **not consume answers that were already generated** by them previously.
- A possible design is to merge the answer tries of the subsumed subgoals.
  - This is a complex operation that would require the additional insertion in each subsumed answer of the ground terms in the call (note that the tables only store the answers for the variables in the subgoal call).

Retroactive Call Subsumption

Table Space

- How do we design a table space that makes it efficient to **transform generators into consumers**?
- How do we guarantee that the newly transformed consumers do **not consume answers that were already generated** by them previously.
- A possible design is to merge the answer tries of the subsumed subgoals.
  - This is a complex operation that would require the additional insertion in each subsumed answer of the ground terms in the call (note that the tables only store the answers for the variables in the subgoal call).
- We propose a simpler design: the **Single Time Stamped Trie**.

Retroactive Call Subsumption

Single Time Stamped Trie (STST)

- **Only a single time stamped trie** is used to store all answers (of all subgoals calls) for a predicate.
- No more variable substitutions are considered and **all terms in an answer are inserted into the STST**.

Example

If **p(X,3)** is called and an answer **X=2** is found then the entry stored in the STST will be <**2,3**>.

Single Time Stamped Trie

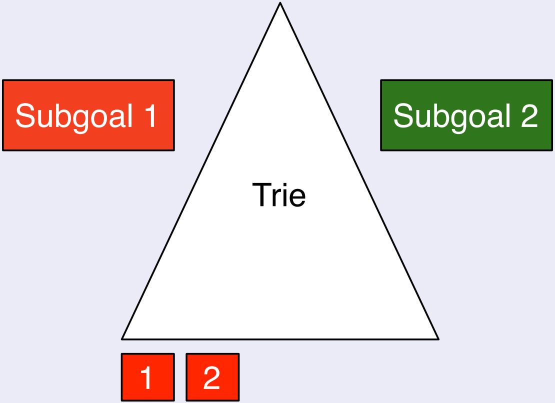
Inserting Answers

- All subgoal frames include a **timestamp field TS** in such a way that, at any time, each subgoal frame contains the list of relevant answers in the STST **older than the current timestamp TS**.
- When a generator inserts new answers, we thus **increment the STST timestamp** and **the subgoal frame TS field**.

Single Time Stamped Trie

Inserting Answers

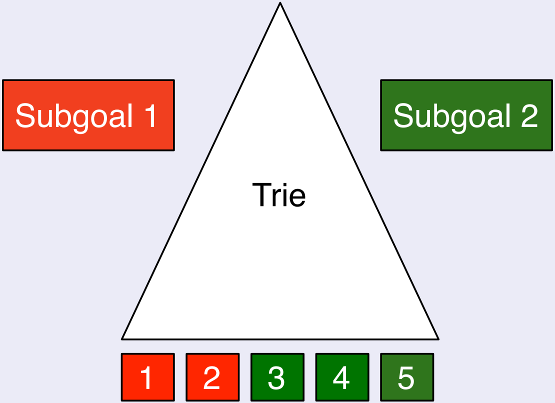
- However, several subgoals may be inserting answers into the STST and it may be difficult to determine if an answer is new or repeated for a subgoal if it is already on the STST.



Single Time Stamped Trie

Inserting Answers

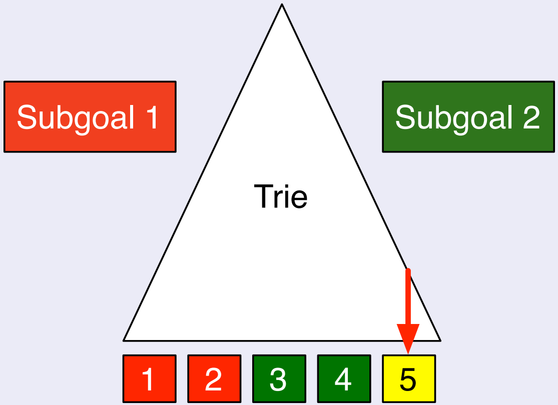
- However, several subgoals may be inserting answers into the STST and it may be difficult to determine if an answer is new or repeated for a subgoal if it is already on the STST.



Single Time Stamped Trie

Inserting Answers

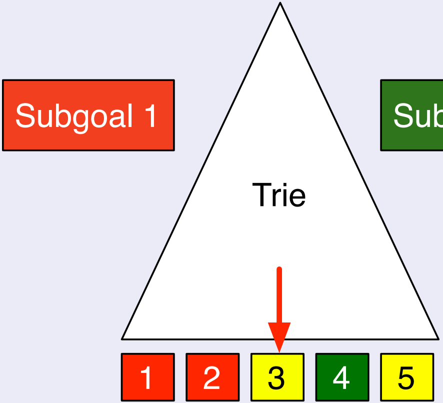
- However, several subgoals may be inserting answers into the STST and it may be difficult to determine if an answer is new or repeated for a subgoal if it is already on the STST.



Single Time Stamped Trie

Inserting Answers

- However, several subgoals may be inserting answers into the STST and it may be difficult to determine if an answer is new or repeated for a subgoal if it is already on the STST.



Single Time Stamped Trie

Inserting Answers

- For each subgoal, we keep a **pending answer index** with the answers older than the subgoal frame timestamp TS that were not found yet.

Single Time Stamped Trie

Inserting Answers

- For each subgoal, we keep a **pending answer index** with the answers older than the subgoal frame timestamp TS that were not found yet.
- If a subgoal **generates an answer younger than TS**, we collect all the relevant answers newer than TS and add them to the pending answer index.

Single Time Stamped Trie

Inserting Answers

- For each subgoal, we keep a **pending answer index** with the answers older than the subgoal frame timestamp TS that were not found yet.
- If a subgoal **generates an answer younger than TS**, we collect all the relevant answers newer than TS and add them to the pending answer index.
- If a subgoal **generates an answer older than TS**, we lookup in the pending answer index:
  - If it is there, the answer is new.
  - If not, the answer is repeated.

Single Time Stamped Trie

Other Considerations

- When we turn a generator into a consumer we can **safely move** all the answers in the pending answer index **to the answer return list** and continue using the timestamp field TS for retrieving new relevant answers.

Single Time Stamped Trie

Other Considerations

- When we turn a generator into a consumer we can **safely move** all the answers in the pending answer index **to the answer return list** and continue using the timestamp field TS for retrieving new relevant answers.
- When a subgoal is first called we can select all the relevant answers already on the STST and **start using them before executing any code**.

Single Time Stamped Trie

Other Considerations

- When we turn a generator into a consumer we can **safely move** all the answers in the pending answer index **to the answer return list** and continue using the timestamp field TS for retrieving new relevant answers.
- When a subgoal is first called we can select all the relevant answers already on the STST and **start using them before executing any code**.
- When the most general subgoal completes, we can throw away the subgoal trie and use the **compiled tries** optimization for future calls to this predicate.

Experimental Results

Programs Taking Advantage of RCS

- Previous results for RCS showed good performance for programs where subsuming subgoals were called after subsumed subgoals.

Program	Var/RCS	Sub/RCS
double_first	1.07	1.09
double_last	1.05	1.10
reach_first	2.54	1.76
reach_last	3.22	1.87
fib	1.95	2.02
flora	3.17	1.17
big	13.26	13.66

Experimental Results

Programs Taking Advantage of RCS

- Previous results for RCS showed good performance for programs where subsuming subgoals were called after subsumed subgoals.

Program	Var/RCS	Sub/RCS
double_first	1.07	1.09
double_last	1.05	1.10
reach_first	2.54	1.76
reach_last	3.22	1.87
fib	1.95	2.02
flora	3.17	1.17
big	13.26	13.66

- In this work, we are interested in measuring the time and space impact of the STST design in benchmarks that do not take advantage of RCS evaluation.

Experimental Results

Benchmarks

- We took the program that computes the reachability between two nodes in a graph and transformed it:

Original

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

Transformed

```
path(f(X),f(Z)):- path(f(X),f(Y)),
                  edge(f(Y),f(Z)).
path(f(X),f(Z)):- edge(f(X),f(Z)).
```

Experimental Results

Benchmarks

- We took the program that computes the reachability between two nodes in a graph and transformed it:

Original

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

Transformed

```
path(f(X),f(Z)):- path(f(X),f(Y)),
                  edge(f(Y),f(Z)).
path(f(X),f(Z)):- edge(f(X),f(Z)).
```

- We used several variations of this program (storing different number of consumers) and two queries: **path(X,Y)** for the Original version and **path(f(X),f(Y))** for the Transformed version.
- Our goal is to force the STST to store more terms than those that are needed with variant and non-retroactive subsumptive tabling.

Experimental Results

Execution Time

	Original		Transformed	
	RCS/Var	**RCS/Sub**	**RCS/Var**	**RCS/Sub**
Average	1.04	1.07	1.25	1.37

Experimental Results

Execution Time

	Original		Transformed	
	RCS/Var	**RCS/Sub**	**RCS/Var**	**RCS/Sub**
Average	1.04	1.07	1.25	1.37

- Since the STST stores all the arguments of an answer in the trie and not only the substitutions, the **insertion and loading of the extra f/1 functors** are the primary causes for these overheads.
- The number of consumer nodes can also reduce the performance of the STST design since we need to **unify extra terms for loading answers** from the trie.

Experimental Results

Memory Usage

	Transformed	
	Var/RCS	**Sub/RCS**
Average	1.890	0.957

Experimental Results

Memory Usage

	Transformed	
	Var/RCS	**Sub/RCS**
Average	1.890	0.957

- Variant-based tabling requires, on average, 1.89 times more memory than RCS. This is because, with variant-based tabling, **there is no sharing of answers between subgoals**.
- Subsumption-based tabling requires, on average, 96% of the memory used by RCS. The natural **trie indexing structure** tends to minimize the memory overhead **as more terms are stored in the STST**.

A Simple Table Space Design for RCS

Conclusions

- We presented a simple, compact and practical table space design for RCS.
- Our proposal innovates by having only a single answer trie per predicate, making it easier to share answers across subgoals for the same predicate.
- Previous good results for RCS show that STST performs well in practical programs taking advantage of RCS.
- Due to its design, STST sometimes may store more terms than necessary.
  - This affects execution time on the worst case.
  - However, memory overhead is not as important given the nature of tries.