# An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives

Miguel Areias and Ricardo Rocha
CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto, Portugal
*miguel-areias@dcc.fc.up.pt* *ricroc@dcc.fc.up.pt*

PADL 2010, Madrid, Spain, January 2010

# Tabling in Logic Programming

- Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with **redundant sub-computations** and **recursion**.
- Implementations of tabling are currently available in systems like XSB Prolog, Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog.
- In these implementations, we can distinguish two main categories of tabling mechanisms:
  - **Suspension-Based Tabling**: can be seen as a sequence of sub-computations that can be suspended and later resumed, when necessary, to compute fix-points (XSB Prolog, Yap Prolog, Mercury and Ciao Prolog).
  - **Linear Tabling**: can be seen as a single execution tree where tabled subgoals use iterative computations, without requiring suspension and resumption, to compute fix-points (B-Prolog and ALS-Prolog).

# Linear Tabling

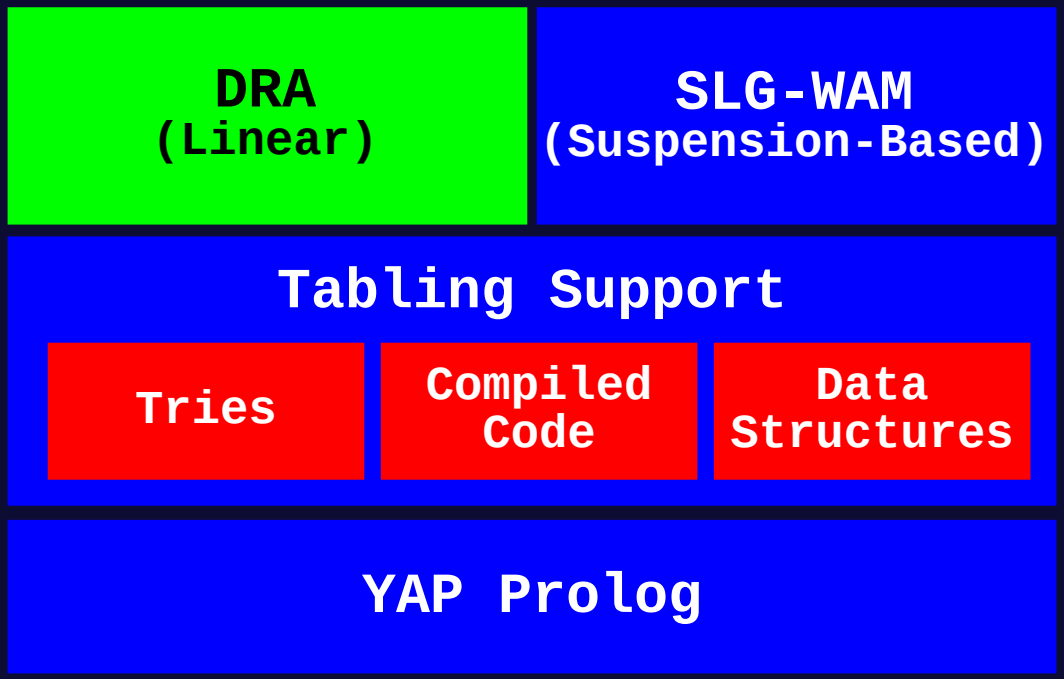
- Arguably, the two most well-known linear tabling proposals are:
  - **SLDT (Zhou *et al.*, B-Prolog)**: repeated calls, **the followers**, execute from the backtracking point of the former call. A follower is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a fix-point is reached.
  - **DRA (Guo and Gupta, ALS-Prolog)**: tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls, the **looping alternatives**. It then uses the looping alternatives to repeatedly recompute them until reaching a fix-point.

# Motivation

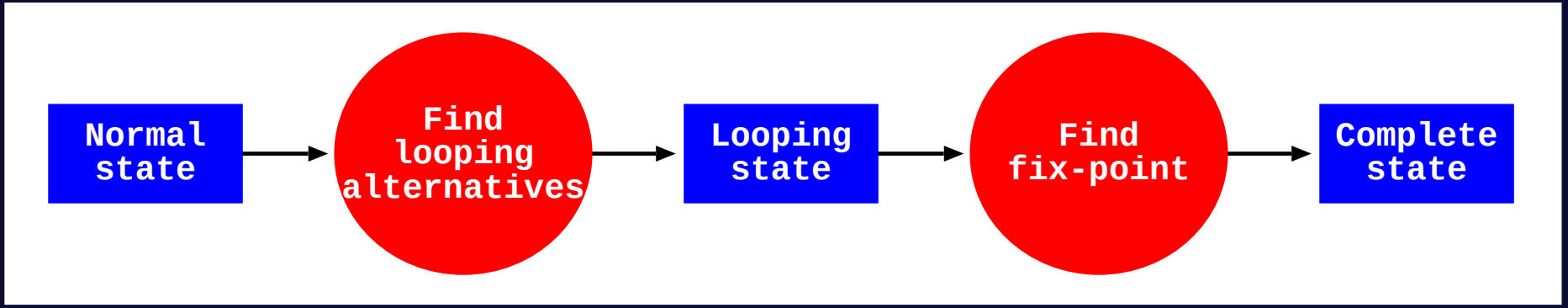
- Suspension-based mechanisms are considered to be more **complicated** to implement but, on the other hand, they are considered to obtain **better results**.
- However, to the best of our knowledge, no rigorous and fair comparison between suspension-based and linear tabling was yet been done in order to better understand the advantages and weaknesses of each mechanism.
- The main reason for this is that **no single Prolog system simultaneously supports both mechanisms** and thus, the available comparisons between both mechanisms cannot be fully dissociated from the strengths and weaknesses of the base Prolog systems on top of which they are implemented.

# Our Proposal

- In this work, we present a new and efficient implementation of linear tabling, but for that we have extended an already existent suspension-based implementation, the tabling engine of **Yap Prolog**.
- Our linear tabling implementation is based on the **DRA technique** but it innovates by considering a strategy that **schedules the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of Yap**.

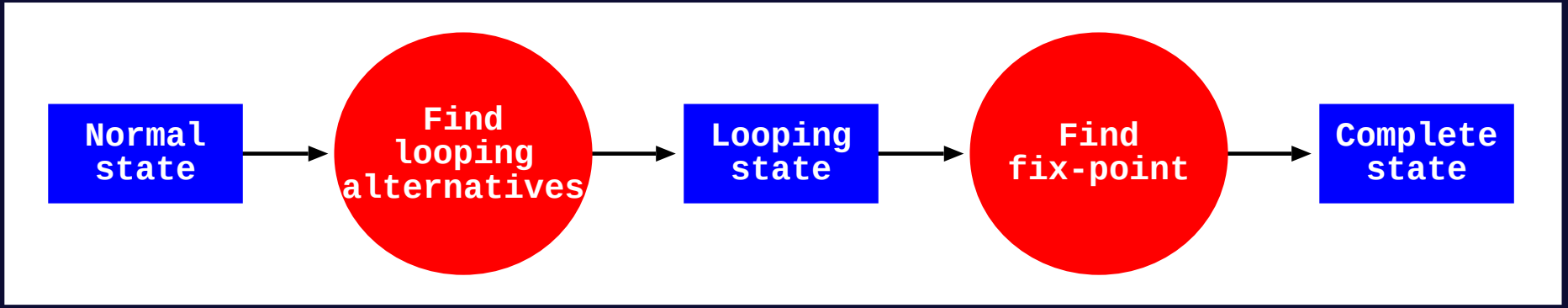


# Dynamic Reordering of Alternatives



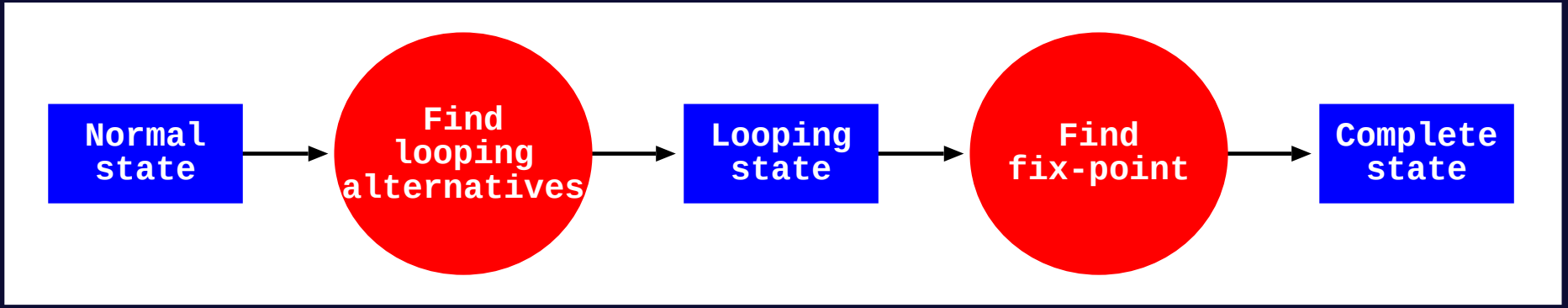

- Consider a tabled subgoal call C. Initially, C enters in **normal state** where it is allowed to explore the matching clauses as in standard Prolog.

# Dynamic Reordering of Alternatives



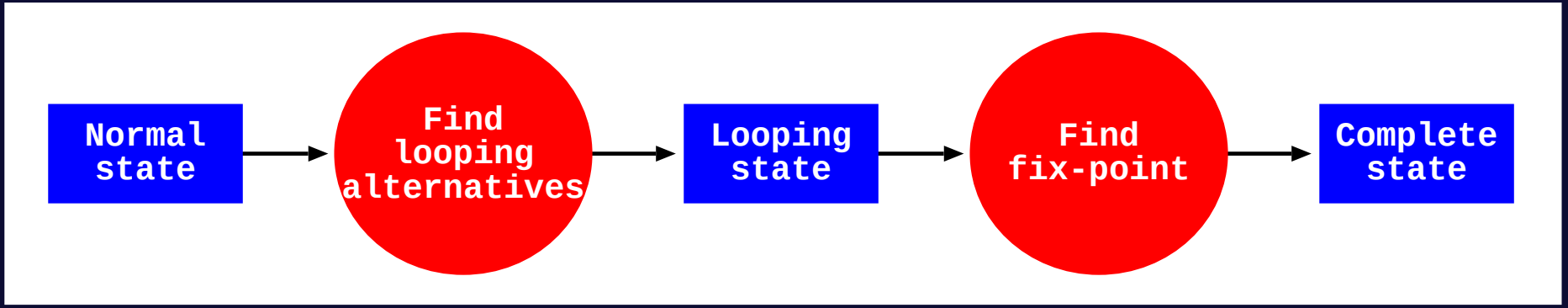

- Consider a tabled subgoal call C. Initially, C enters in **normal state** where it is allowed to explore the matching clauses as in standard Prolog.
- In normal state, if a repeated call is found then the current clause for the first call to C will be memorized as a **looping alternative**.

# Dynamic Reordering of Alternatives




- Consider a tabled subgoal call C. Initially, C enters in **normal state** where it is allowed to explore the matching clauses as in standard Prolog.
- In normal state, if a repeated call is found then the current clause for the first call to C will be memorized as a **looping alternative**.
- Next, after exploring all the matching clauses, C goes into the **looping state**. From this point, it keeps trying the looping alternatives repeatedly until reaching a fix-point.

# Dynamic Reordering of Alternatives




- Consider a tabled subgoal call C. Initially, C enters in **normal state** where it is allowed to explore the matching clauses as in standard Prolog.
- In normal state, if a repeated call is found then the current clause for the first call to C will be memorized as a **looping alternative**.
- Next, after exploring all the matching clauses, C goes into the **looping state**. From this point, it keeps trying the looping alternatives repeatedly until reaching a fix-point.
- If no new answers are found during one cycle of trying the looping alternatives, then we have reached a fix-point and we can say that C is **completely evaluated**.

# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

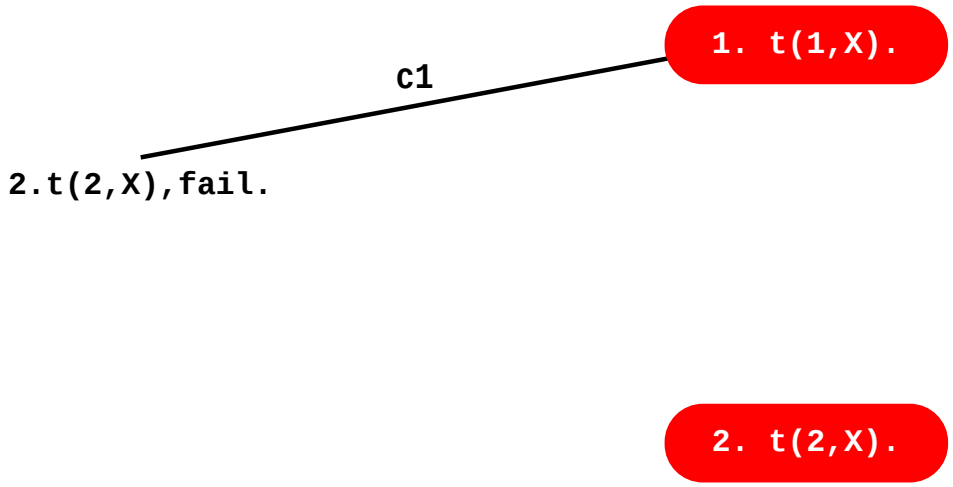
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | |
| | | |

1. t(1,X).

# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

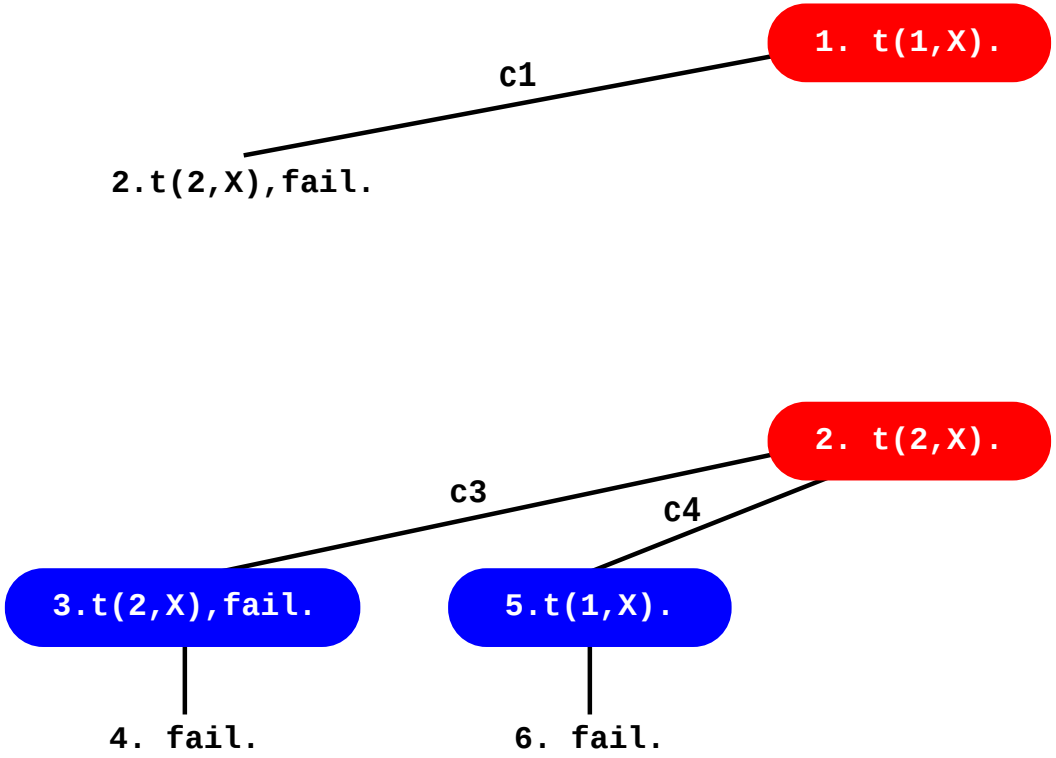
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | |
| 2. t(2,X) | | |



# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

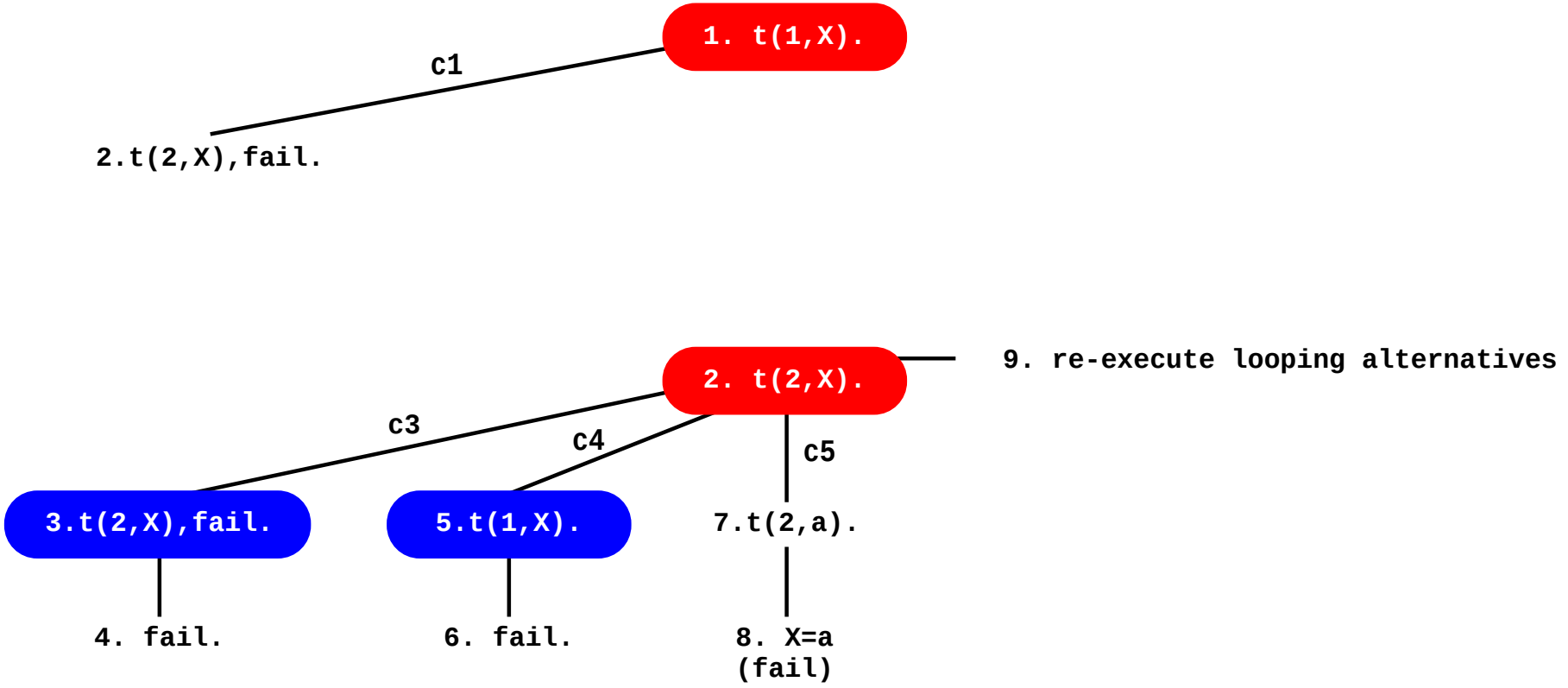
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | 5. t(1,X):-t(2,X),fail. (c1) |
| 2. t(2,X) | | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |



# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

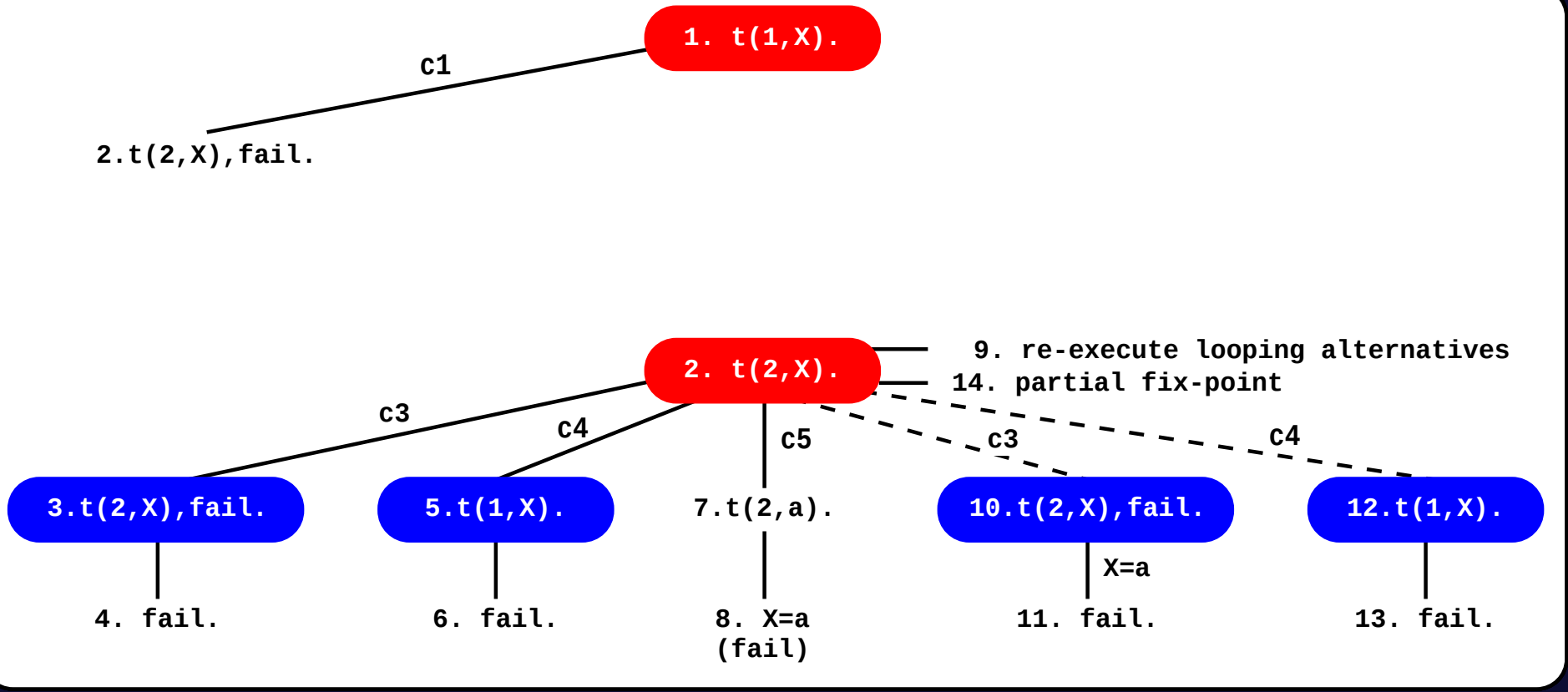
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | 5. t(1,X):-t(2,X),fail. (c1) |
| 2. t(2,X) | 8. X=a | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |



# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

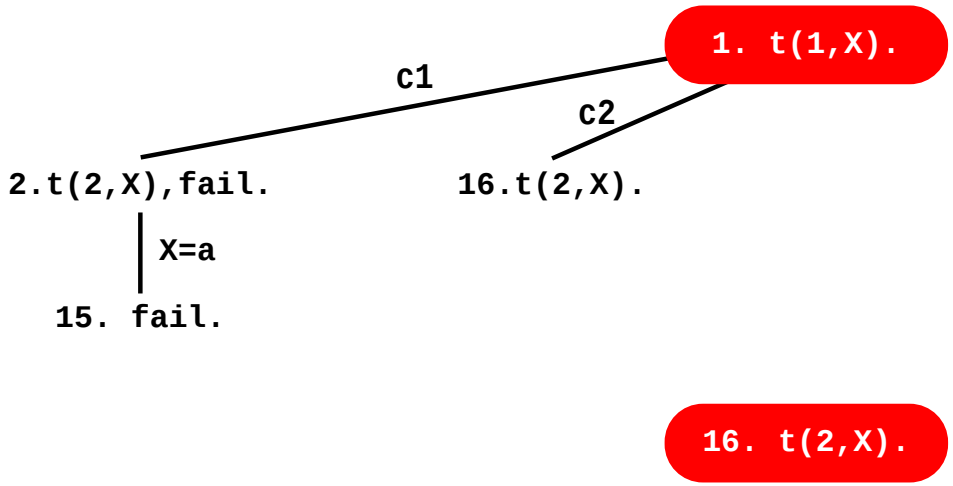
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | 5. t(1,X):-t(2,X),fail. (c1) |
| 2. t(2,X) | 8. X=a | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |



# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

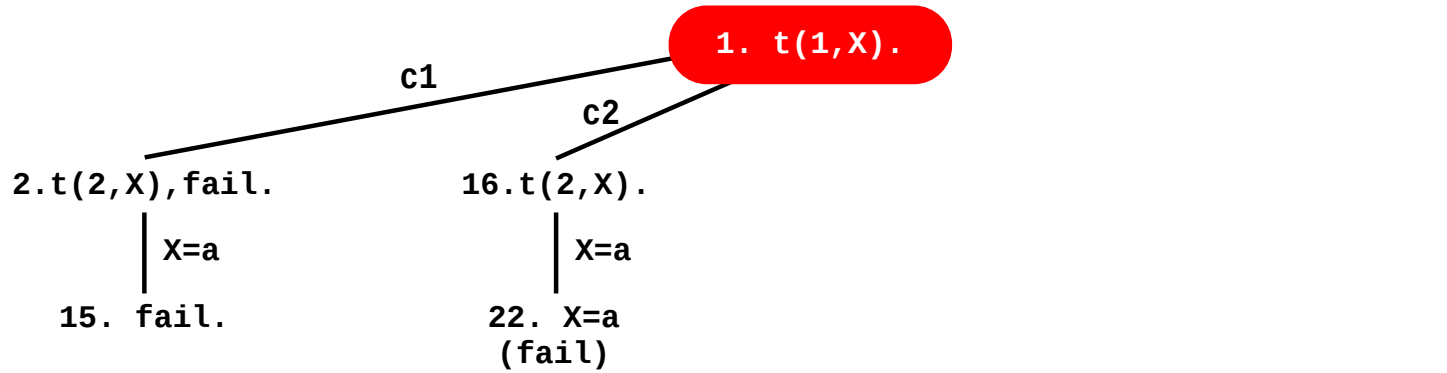
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | | 5. t(1,X):-t(2,X),fail. (c1) |
| 2. t(2,X) | 8. X=a | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |

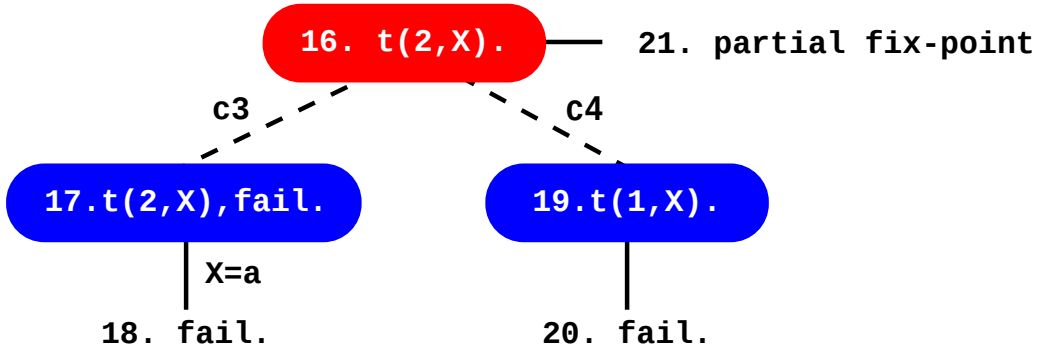


# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | 22. X=a | 5. t(1,X):-t(2,X),fail. (c1)<br>19. t(1,X):-t(2,X). (c2) |
| 2. t(2,X) | 8. X=a | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |

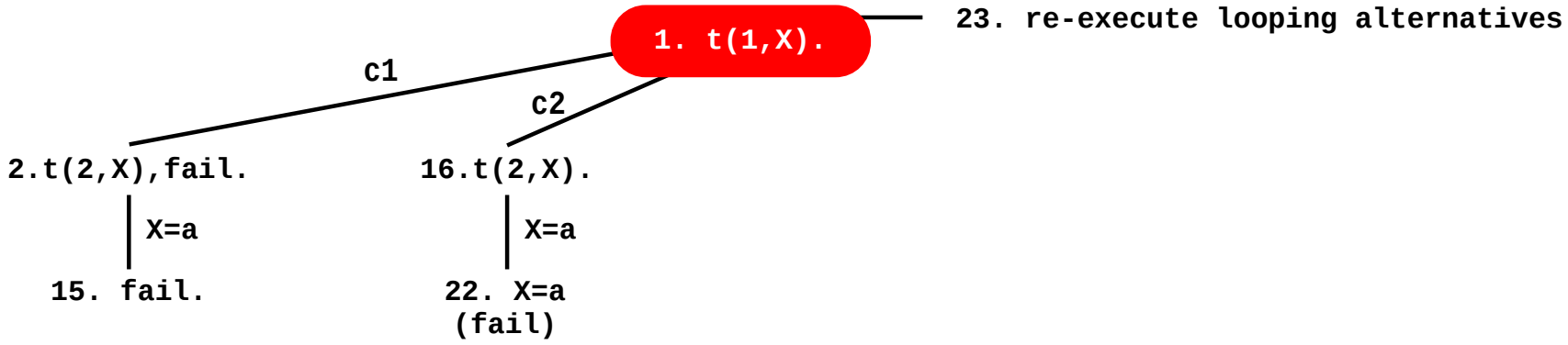


# An Evaluation Example

```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

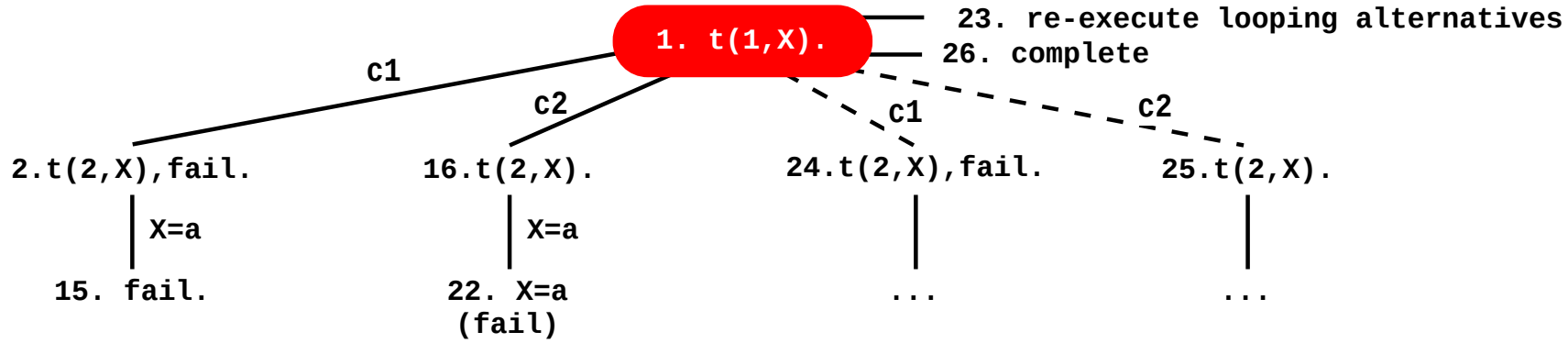
| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | 22. X=a | 5. t(1,X):-t(2,X),fail. (c1)<br>19. t(1,X):-t(2,X). (c2) |
| 2. t(2,X) | 8. X=a | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |



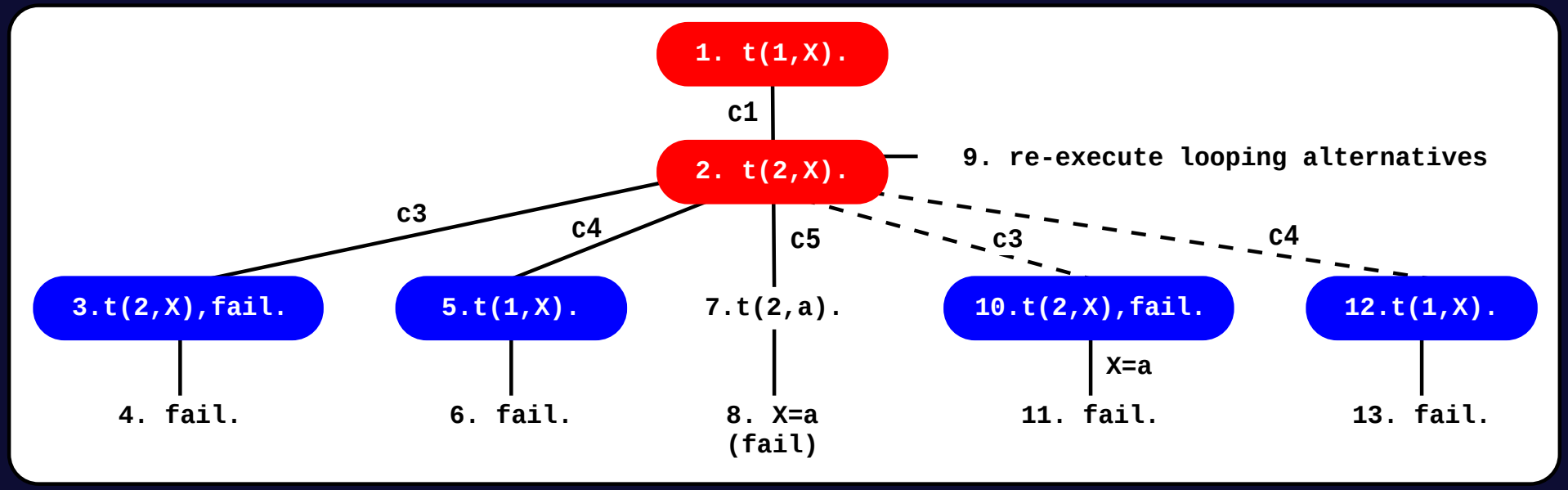
# An Evaluation Example

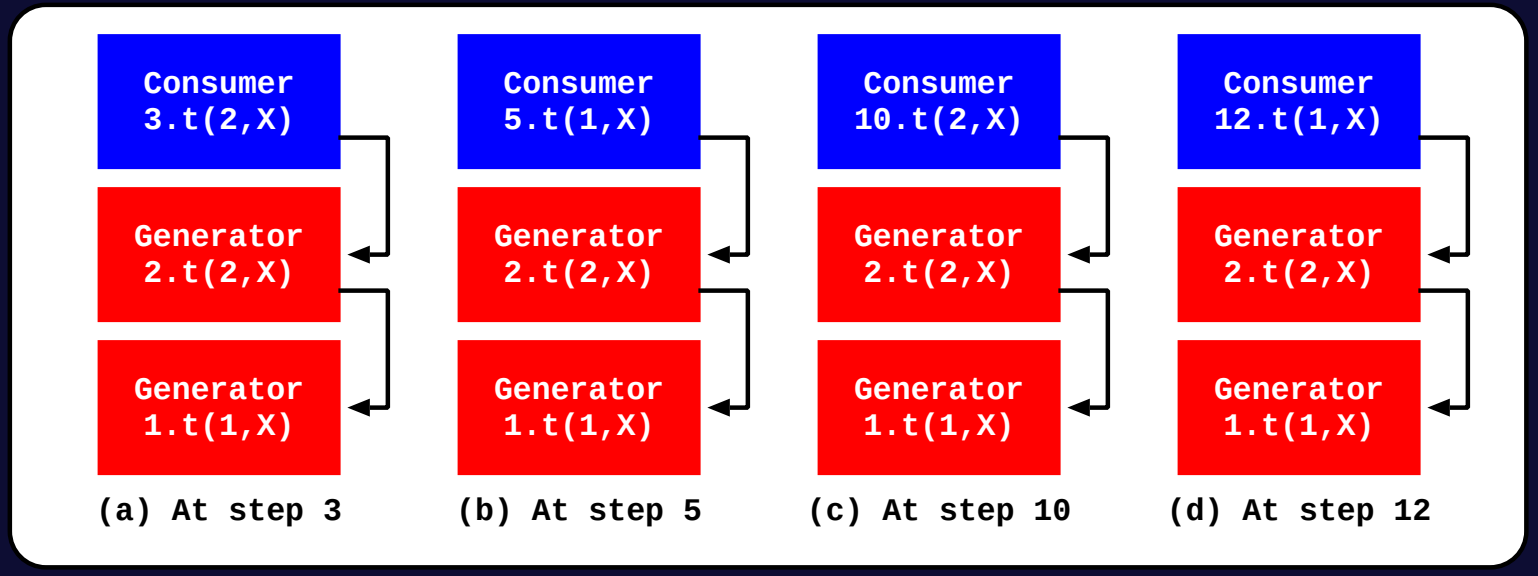
```
:- table t/2.
t(1,X):-t(2,X),fail.      (c1)
t(1,X):-t(2,X).           (c2)
t(2,X):-t(2,X),fail.      (c3)
t(2,X):-t(1,X).           (c4)
t(2,a).                   (c5)
```

| Call | Answers | Looping Alternatives |
|---|---|---|
| 1. t(1,X) | 22. X=a<br>26. complete | 5. t(1,X):-t(2,X),fail. (c1)<br>19. t(1,X):-t(2,X). (c2) |
| 2. t(2,X) | 8. X=a<br>26. complete | 3. t(2,X):-t(2,X),fail. (c3)<br>5. t(2,X):-t(1,X). (c4) |

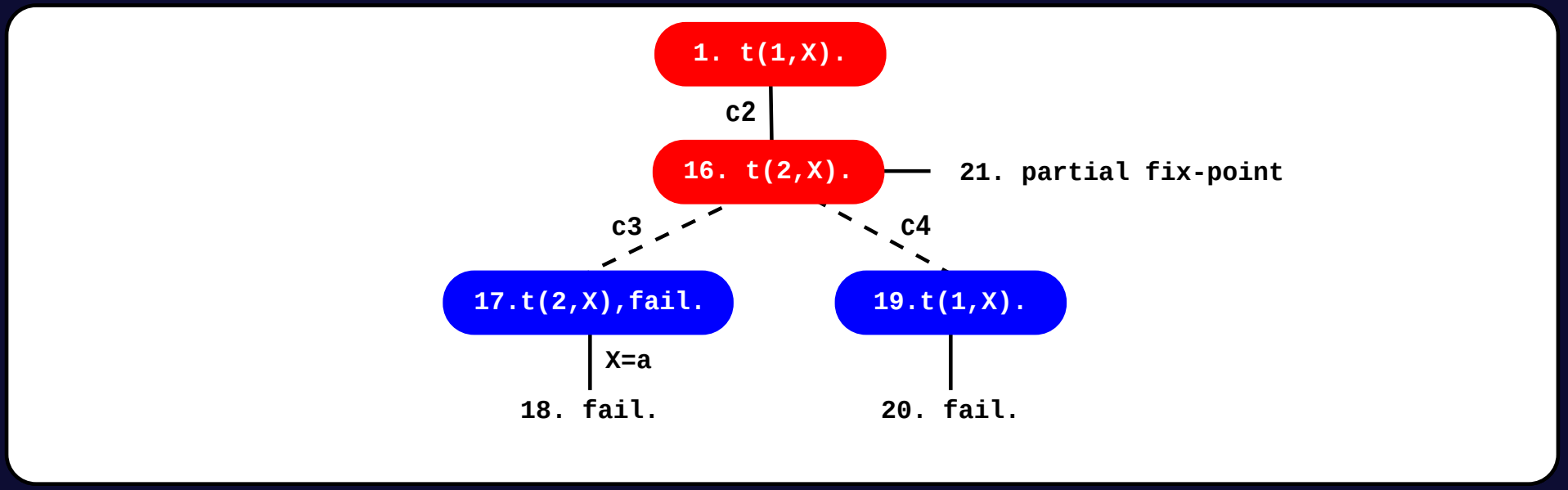


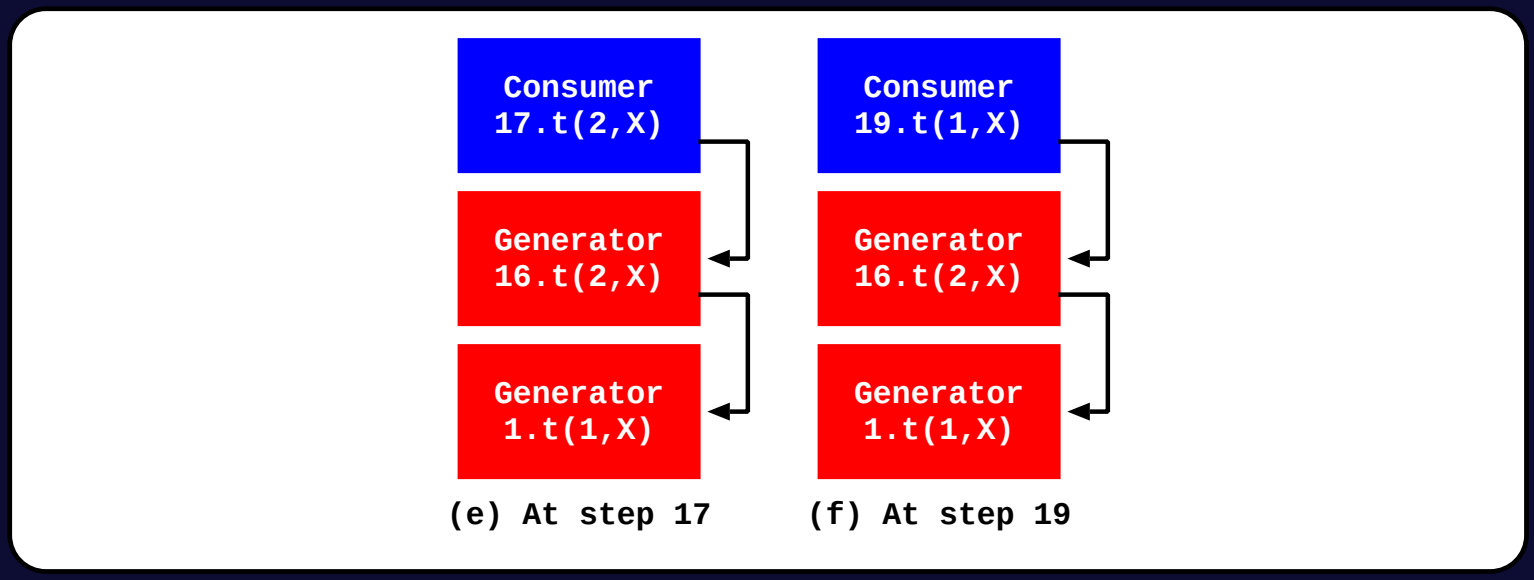
# Choice Point Stack



# Choice Point Stack



# Re-Computation Issues

- Remember that, in looping state, we keep trying the looping alternatives repeatedly until **reaching a partial fix-point** in the evaluation of the corresponding tabled call.

# Re-Computation Issues

- Remember that, in looping state, we keep trying the looping alternatives repeatedly until **reaching a partial fix-point** in the evaluation of the corresponding tabled call.
- Reaching a partial fix-point beforehand can be completely **useless for non-leader calls** when later the leader call re-executes itself its looping alternatives, which in turn leads the non-leader calls to re-execute again their looping alternatives.

# Re-Computation Issues

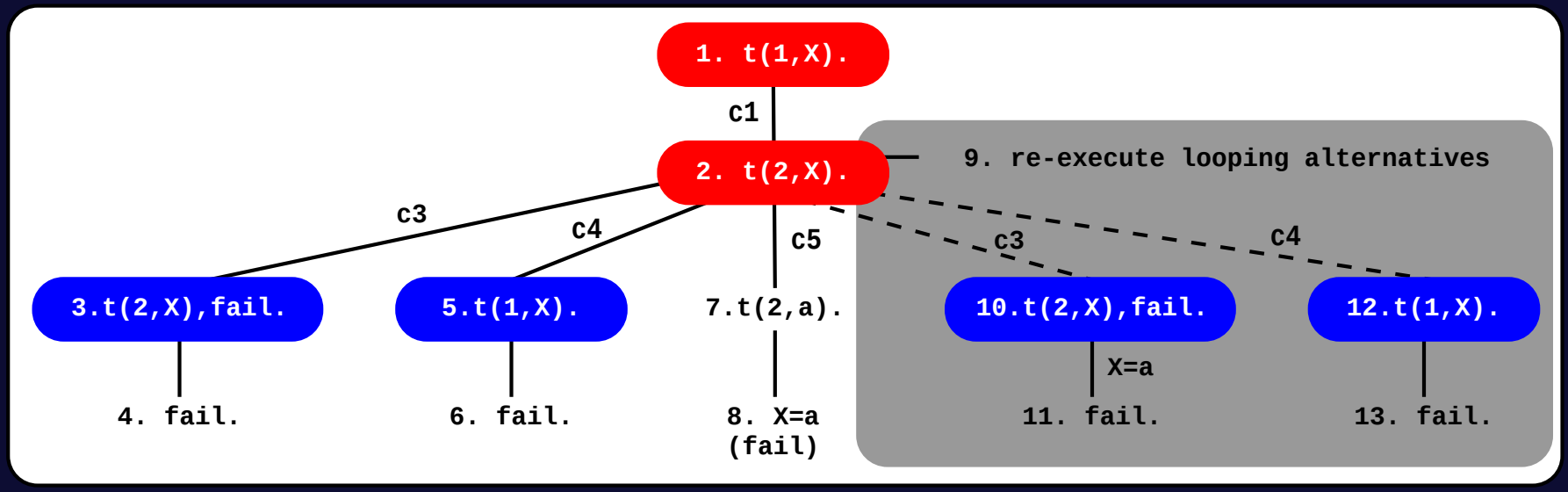
- Remember that, in looping state, we keep trying the looping alternatives repeatedly until **reaching a partial fix-point** in the evaluation of the corresponding tabled call.
- Reaching a partial fix-point beforehand can be completely **useless for non-leader calls** when later the leader call re-executes itself its looping alternatives, which in turn leads the non-leader calls to re-execute again their looping alternatives.
- We innovate by considering a strategy that schedules the re-evaluation of tabled calls in a **similar manner to the suspension-based strategies of Yap**:

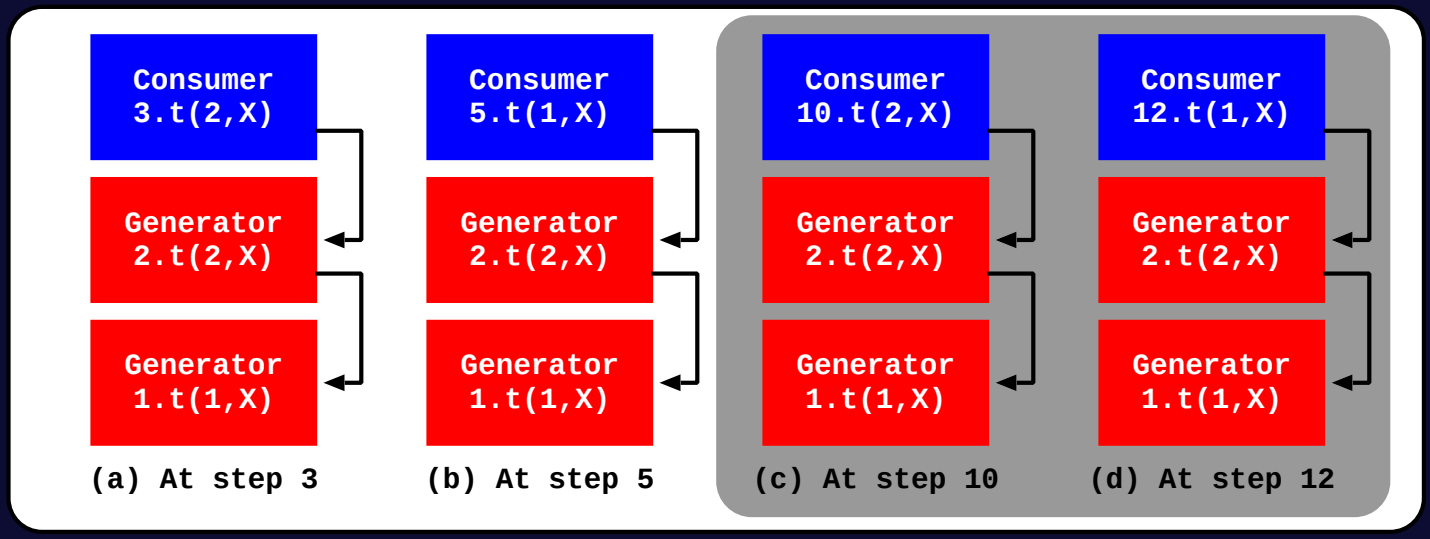
# Re-Computation Issues

- Remember that, in looping state, we keep trying the looping alternatives repeatedly until **reaching a partial fix-point** in the evaluation of the corresponding tabled call.
- Reaching a partial fix-point beforehand can be completely **useless for non-leader calls** when later the leader call re-executes itself its looping alternatives, which in turn leads the non-leader calls to re-execute again their looping alternatives.
- We innovate by considering a strategy that schedules the re-evaluation of tabled calls in a **similar manner to the suspension-based strategies of Yap**:
  - **Rule 1**: The fix-point check for completion is only done by leader calls.
  - **Rule 2**: Only first calls to tabled subgoals allocate generator choice points.

# Re-Computation Issues

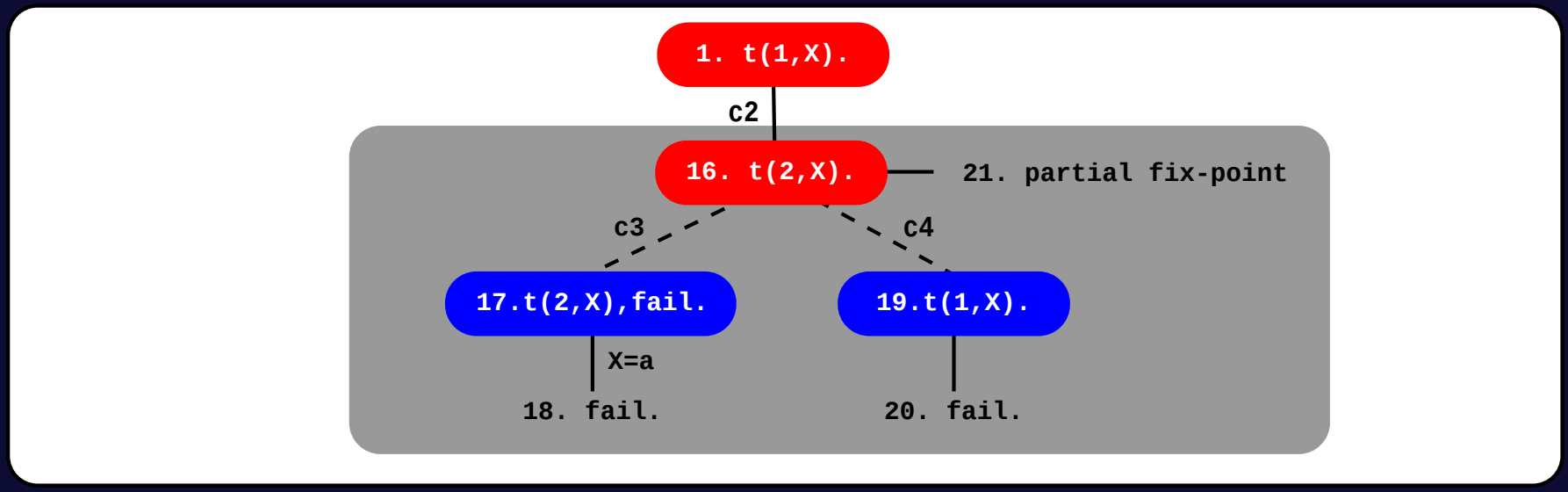
➤ **Rule 1**: The fix-point check for completion is only done by leader calls.

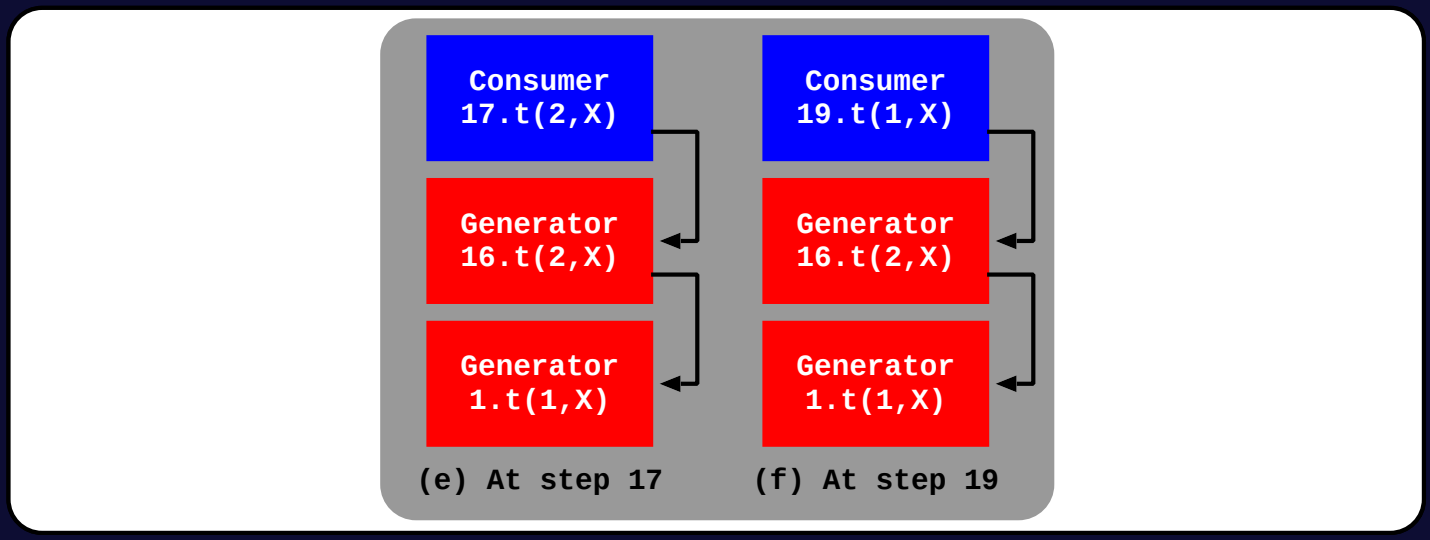


# Re-Computation Issues

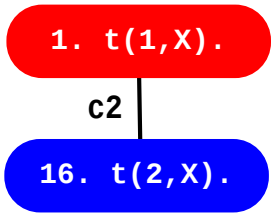
- **Rule 1**: The fix-point check for completion is only done by leader calls.

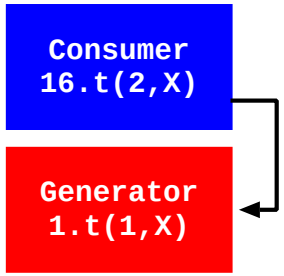


# Re-Computation Issues

- **Rule 2**: Only first calls to tabled subgoals allocate generator choice points.




(e) At step 16

# Re-Computation Issues

- To correctly support this strategy, we need to consider that:
  - a tabled call is a first call **every time we re-start a new round** over the looping alternatives for the leader call.

# Re-Computation Issues

- To correctly support this strategy, we need to consider that:
  - a tabled call is a first call **every time we re-start a new round** over the looping alternatives for the leader call.
  - the leader call must re-execute its looping alternatives when **new answers were found for any tabled call** during the last traversal of the looping alternatives (and not only when new answers were found for the leader call).

# Re-Computation Issues

- To correctly support this strategy, we need to consider that:
  - a tabled call is a first call **every time we re-start a new round** over the looping alternatives for the leader call.
  - the leader call must re-execute its looping alternatives when **new answers were found for any tabled call** during the last traversal of the looping alternatives (and not only when new answers were found for the leader call).
- To efficiently implement it, we use two chains of tabled calls:

# Re-Computation Issues

- To correctly support this strategy, we need to consider that:
  - a tabled call is a first call **every time we re-start a new round** over the looping alternatives for the leader call.
  - the leader call must re-execute its looping alternatives when **new answers were found for any tabled call** during the last traversal of the looping alternatives (and not only when new answers were found for the leader call).
- To efficiently implement it, we use two chains of tabled calls:
  - One is used when propagating dependencies to traverse the tabled calls in order **to mark the looping alternatives and to mark the non-leader calls**.

# Re-Computation Issues

- To correctly support this strategy, we need to consider that:
  - a tabled call is a first call **every time we re-start a new round** over the looping alternatives for the leader call.
  - the leader call must re-execute its looping alternatives when **new answers were found for any tabled call** during the last traversal of the looping alternatives (and not only when new answers were found for the leader call).
- To efficiently implement it, we use two chains of tabled calls:
  - One is used when propagating dependencies to traverse the tabled calls in order **to mark the looping alternatives and to mark the non-leader calls**.
  - The other one is used by the leader call to traverse the tabled calls in order **to mark them for re-evaluation or as completed**.

# Experimental Results: Yap / Yap+DRA

| Predicate | Pyramid | | Cycle | | Grid | | MC | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 1000 | 1500 | 30 | 40 | IP | LE | SV |
| left_first | 0.67 | 0.73 | 0.68 | 0.72 | 0.52 | 0.58 | 0.55 | 0.56 | 0.53 |
| left_last | 0.63 | 0.62 | 0.64 | 0.67 | 0.54 | 0.52 | 0.56 | 0.51 | 0.54 |
| right_first | **0.99** | **1.03** | 0.59 | 0.69 | 0.16 | 0.12 | – | – | – |
| right_last | **1.00** | **0.99** | 0.72 | 0.74 | 0.17 | 0.13 | – | – | – |
| double_first | 0.49 | 0.53 | 0.58 | 0.58 | 0.56 | 0.59 | – | – | – |
| double_last | 0.51 | 0.51 | 0.57 | 0.58 | 0.56 | 0.56 | – | – | – |

- Yap is around 1.5 to 2 times faster than Yap+DRA.
- Yap+DRA scales well when we increase the complexity of the problem.

# Experimental Results: XSB / Yap+DRA

| Predicate | Pyramid | | Cycle | | Grid | | MC | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 1000 | 1500 | 30 | 40 | IP | LE | SV |
| left_first | 0.56 | 0.58 | 0.78 | 0.69 | 0.66 | 0.65 | 1.05 | **1.52** | 0.80 |
| left_last | 0.58 | 0.62 | 0.68 | 0.79 | 0.66 | 0.63 | 1.05 | **1.50** | 0.69 |
| right_first | **1.32** | **1.44** | 1.05 | 1.03 | 0.29 | 0.23 | – | – | – |
| right_last | **1.36** | **1.34** | 1.01 | 0.98 | 0.30 | 0.24 | – | – | – |
| double_first | 0.89 | 0.90 | 0.98 | $r.e.$ | 1.01 | $r.e.$ | – | – | – |
| double_last | 0.90 | 0.89 | 0.97 | $r.e.$ | 0.99 | $r.e.$ | – | – | – |

- In general, the difference between XSB and Yap+DRA is clearly smaller.
- Surprisingly, Yap+DRA obtains better results than XSB for the right recursive definitions with the pyramid configurations and for the left recursive definitions with the model checking specifications.

# Experimental Results: B-Prolog / Yap+DRA

| Predicate | Pyramid | | Cycle | | Grid | | MC | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 1000 | 1500 | 30 | 40 | IP | LE | SV |
| left_first | 1.93 | 2.62 | 1.70 | 2.20 | 2.71 | 3.65 | 3.61 | **10.52** | **9.61** |
| left_last | 1.65 | 2.27 | 1.74 | 1.98 | 2.33 | 3.39 | 3.61 | **10.18** | **9.43** |
| right_first | 1.66 | 1.96 | 1.84 | 2.15 | 1.42 | 1.47 | – | – | – |
| right_last | 1.55 | 1.76 | 1.89 | 2.12 | 1.44 | 1.44 | – | – | – |
| double_first | 3.20 | 4.21 | 2.93 | 3.73 | 2.81 | 4.00 | – | – | – |
| double_last | 3.31 | 4.28 | 2.80 | 3.63 | 2.77 | 3.86 | – | – | – |

- Yap+DRA is always faster than B-Prolog in all experiments.
- The ratio over Yap+DRA shows a generic tendency to increase as the complexity of the problem also increases.

# Conclusions and Further Work

- We have presented a new and very efficient implementation of linear tabling, based on DRA technique, that shares the underlying execution environment and most of the data structures used to implement suspension-based tabling in Yap.
- The results obtained are very interesting and very promising. In our experiments, the commonly referred weakness of linear tabling of doing a huge number of redundant computations for computing fix-points was not such a problem.
- We thus argue that an efficient implementation of linear tabling can be a good and first alternative to incorporate tabling support into a Prolog system.
- Further work will include exploring the impact of applying our proposal to more complex problems. We also plan to expand our approach to support different linear tabling proposals like the SLDT strategy.