

# Tabling as a Library with Delimited Control

**Alexander Vandenbroucke**

---

Benoit Desouter - Marko Vandooren - Tom Schrijvers



## Tabling as a library with delimited control

BENOIT DESOUTER and MARKO VAN DOOREN

Ghent University, Belgium  
(e-mail: benoit.desouter,marko.vandooren@ugent.be)

TOM SCHRIJVERS

KU Leuven, Belgium  
(e-mail: tom.schrijvers@cs.kuleuven.be)

submitted 29 April 2015; revised 3 July 2015; accepted 15 July 2015

### Abstract

Tabling is probably the most widely studied extension of Prolog. But despite its importance and practicality, tabling is not implemented by most Prolog systems. Existing approaches require substantial changes to the Prolog engine, which is an investment out of reach of most systems. To enable more widespread adoption, we present a new implementation of tabling in under 600 lines of Prolog code. Our lightweight approach relies on delimited control and provides reasonable performance.

**KEYWORDS:** tabling, tabulation, delimited continuations, Prolog, logic programming

### 1 Introduction

Tabling is one of the most widely studied extensions to Prolog because it considerably raises the declarative nature of the language. Tabling takes away the sensitivity of SLD resolution to rule and goal ordering, and allows a larger class of programs to terminate. As an added bonus, the memoisation that is done by the tabling mechanism may drastically improve performance in exchange for more memory.

Given all these advantages, it may come as a surprise that many Prolog systems still do not support tabling. The reason for this is that existing implementations, such as those of Yap and XSB, require pervasive changes to the Prolog engine. This is a substantial engineering effort that is beyond most systems (Santos Costa *et al.* 2012).

Several works have already attempted to tackle this problem. Through the foreign function interface, Ramesh and Chen (1994) extend Prolog with tabling primitives implemented in C. A complicated program transformation introduces calls to these C routines at the appropriate points in tabled predicates. More recently, Guzmán *et al.* (2008) have addressed the performance bottlenecks of Ramesh and Chen's approach. But while their improvement is successful in terms of performance, it does require lower-level C primitives, changes to the WAM's memory management, and an even more complicated program transformation. These changes further increase the cost of porting and maintaining the mechanism, and the development effort

## Tabling as a Library with Delimited Control

Benoit Desouter

Marko van Dooren

Tom Schrijvers

Alexander Vandenbroucke

Ghent University, Belgium  
benoit.desouter@ugent.be  
marko.vandooren@ugent.be

KU Leuven, Belgium  
tom.schrijvers@kuleuven.be  
alexander.vandenbroucke@kuleuven.be

### Abstract

The logic programming language Prolog uses a resource-efficient SLD resolution strategy for query answering. Yet, its propensity for non-termination seriously detracts from the language's declarative nature. This problem is remedied by *tabling*, a modified execution strategy that allows a larger class of programs to terminate. Unfortunately, few Prolog systems provide tabling, because the documented implementation techniques are complex, low-level and require a prohibitive engineering effort.

To enable more widespread adoption, this paper presents a novel implementation of tabling for Prolog that is both high-level and compact. It comes in the form of a Prolog library that weighs in at under 600 lines of code, is based on delimited control and delivers reasonable performance.

### 1 Introduction

The essence of programming is crisply captured in Kowalski's adage  $\text{ALGORITHM} = \text{LOGIC} + \text{CONTROL}$  [1979]. The ideal of logic programming is that the programmer should be able to focus only on the first part, the problem logic, while the control should be supplied by the programming system.

Unfortunately, traditional Prolog systems fall short of reaching this ideal as programmers need to be constantly aware of Prolog's SLD resolution strategy [Kowalski, 1974], which processes rules in a "top-down, left-to-right" fashion. Consider for example the logic rules for computing the transitive closure of the  $e/2$  relation:

$$\begin{aligned} p(X, Y) &:- p(X, Z), e(Z, Y). \\ p(X, Y) &:- e(X, Y). \end{aligned}$$
$$e(1, 2) . e(2, 3) .$$

Although the logic is correct, Prolog diverges when resolving any  $p(X, Y)$  query. Unfortunately, it is up to the programmer to diagnose the left recursion in the first rule as the culprit, and to address the issue by eliminating the left recursion and by reordering the rules. Moreover, to handle a cycle in the graph,<sup>1</sup> the programmer needs to pollute his declarative

<sup>1</sup>e.g., obtained by adding the fact  $e(2, 1)$

model more thoroughly with control logic. This clearly goes against the grain of declarative programming.

*Tabling* is a variation on SLD resolution that does not get stuck in cyclic derivations, and thus captures the declarative least fixed-point semantics of logic programs more completely. It works by storing the intermediate answers to predicates (in a data structure known as a table), and reusing them instead of recomputing them, whenever possible. At the same time cyclic derivations are suspended and only resumed when new answers are available. This approach not only improves the termination behavior, but may also drastically improve performance—be it at the cost of increased memory usage.

Given these advantages, it may come as a surprise that not many Prolog systems support tabling. The reason for this is that existing implementations, such as those of Yap [Santos Costa *et al.*, 2012] and XSB [Swift and Warren, 2012], require pervasive changes to the Prolog engine, the Warren Abstract machine (WAM) [Warren, 1983; Ait-Kaci, 1999] or one of its variants. This is a substantial engineering effort that is beyond most systems [Santos Costa *et al.*, 2012].

Several attempts have been made to tame the complexity by means of transformations, calls to C routines and very specific changes to the WAM [Ramesh and Chen, 1994; Zhou *et al.*, 2000; Guo and Gupta, 2001, 2004]. Nevertheless, these approaches incur substantial technical debt, have a high maintenance and porting cost, and the development effort cannot be amortised over other features. In contrast, extension tables [Fan and Dietrich, 1992] provide a high-level tabling mechanism that is implemented directly in Prolog. However, the approach cannot achieve satisfactory performance as suspended goals are always re-evaluated.

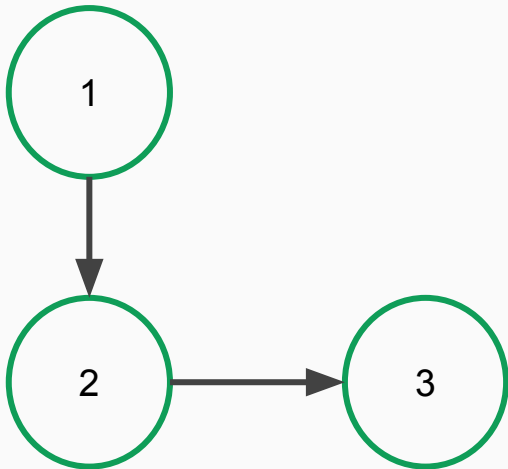
We improve upon the current state of the art with a novel lightweight implementation of tabling based on delimited control. Our approach comes in the form of a Prolog library that weighs in at less than 600 lines of code, delivers acceptable performance, and requires only a minimal extension to the Prolog system: *delimited control* [Schrijvers *et al.*, 2013b]. Moreover, the development effort of delimited control can be amortised over the range of high-level language features they enable, such as *effect handlers* [Plotkin and Pretnar, 2013].

The remainder of this paper provides a high-level overview of our contribution. We refer to the extended paper [Desouter *et al.*, 2015] for more details.

A green rectangular sign with rounded corners and a white border, mounted on two wooden posts. The sign features the word "Motivation" in a large, white, sans-serif font. The background is a bright blue sky with scattered white clouds.

**Motivation**

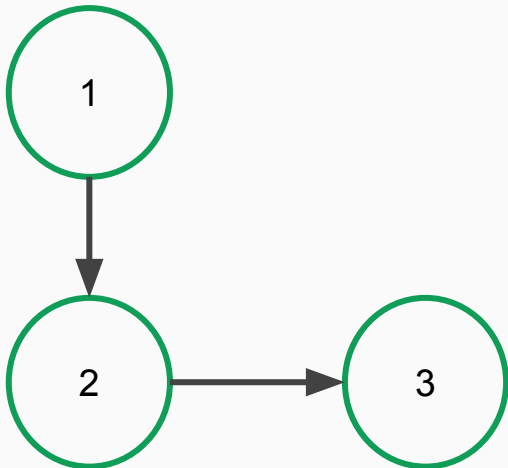
# Motivating example



# Motivating example

*% edges*

$e(1,2)$ .  $e(2,3)$ .



# Motivating example

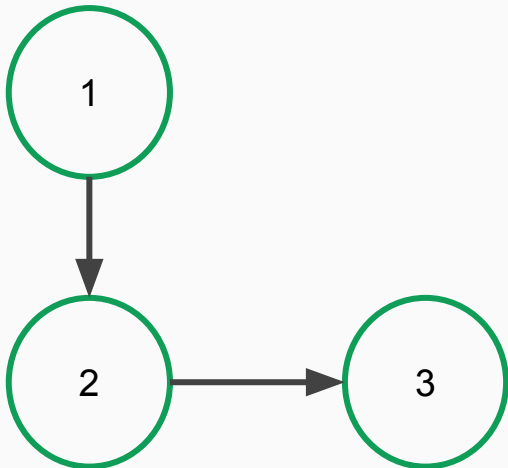
*% edges*

$e(1,2)$ .  $e(2,3)$ .

*% transitive & reflexive closure*

$\text{connected}(X,Y) \text{ :- connected}(X,Z), e(Z,Y)$ .

$\text{connected}(X,X)$ .



# Motivating example

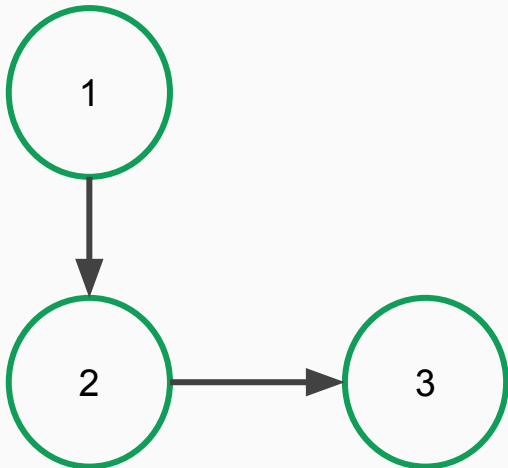
*% edges*

`e(1,2). e(2,3).`

*% transitive & reflexive closure*

`connected(X,Y) :- connected(X,Z),e(Z,Y).`

`connected(X,X).`



?- `connected(X,Y).`

**Expected:**

`connected(1,1);`

`connected(1,2);`

`connected(1,3);`

`connected(2,2);`

`connected(2,3);`

`connected(3,3).`

# Motivating example

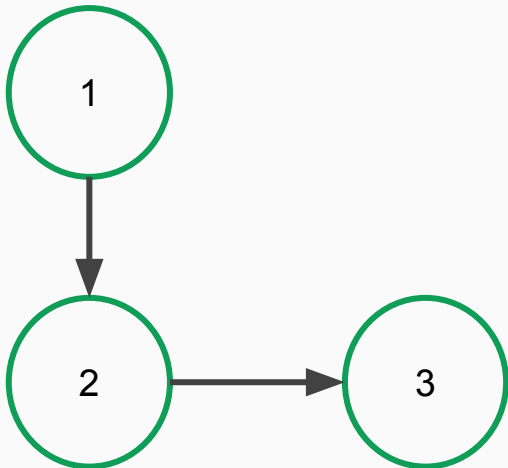
*% edges*

$e(1,2)$ .  $e(2,3)$ .

*% transitive & reflexive closure*

$\text{connected}(X,Y) :- \text{connected}(X,Z), e(Z,Y)$ .

$\text{connected}(X,X)$ .



?-  $\text{connected}(X,Y)$ .  
**Error: Out of Stack**



# Motivating example

*% edges*

`e(1,2). e(2,3).`

*% transitive & reflexive closure*

`connected(X,Y) :-`

`connected(X,Z), e(Z,Y).`

`connected(X,X).`

*% edges*

`e(1,2). e(2,3).`

*% transitive & reflexive closure*

`connected(X,X).`

`connected(X,Y) :-`

`connected(X,Z), e(Z,Y).`

*% edges*

`e(1,2). e(2,3).`

*% transitive & reflexive closure*

`connected(X,Y) :-`

`e(X,Z), connected(Z,Y).`

`connected(X,X).`

*% edges*

`e(1,2). e(2,3).`

*% transitive & reflexive closure*

`connected(X,X).`

`connected(X,Y) :-`

`e(X,Z), connected(Z,Y).`

# Motivating example

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure  
connected(X,X).  
connected(X,Y) :-
```

```
    connected(X,Z), e(Z,Y).
```

```
connected(X,X).
```

```
connected(X,X).
```

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
    connected(X,Z), e(Z,Y).
```

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,Y) :-
```

```
    e(X,Z), connected(Z,Y).
```

```
connected(X,X).
```

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
    e(X,Z), connected(Z,Y).
```

# Motivating example

```
% edges
```

```
e(1
```

```
% t ?- connected(X,Y).
```

```
con
```

```
c
```

```
connected(X,X).
```

**Error: Out of Stack**

```
% edges
```

```
e(1
```

```
% t ?- connected(X,Y).
```

```
con
```

```
e(
```

```
connected(X,X).
```

**Error: Out of Stack**

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
    connected(X,Z), e(Z,Y).
```

```
% edges
```

```
e(1,2). e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
    e(X,Z), connected(Z,Y).
```

# Motivating example

```
% edges
```

```
e(1,2).
```

```
% transitive & reflexive closure  
?- connected(X,Y).
```

**Error: Out of Stack**

```
connected(X,X).
```

```
connected(X,X).
```

```
connected(X,X).
```

```
% edges
```

```
e(1,2).
```

```
% transitive & reflexive closure  
?- connected(X,Y).
```

**Error: Out of Stack**

```
connected(X,X).
```

```
connected(X,X).
```

```
connected(X,X).
```

```
% edges
```

```
e(1,2).
```

```
% transitive & reflexive closure  
?- connected(X,Y).
```

```
connected(1,1);
```

```
connected(2,2);
```

```
connected(3,3);
```

```
connected(1,2);
```

```
<continues forever>
```

```
connected(X,X).
```

```
% edges
```

```
e(1,2).
```

```
e(2,3).
```

```
% transitive & reflexive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
e(X,Z), connected(Z,Y).
```

# Motivating example

```
% edges
```

```
e(1
```

```
% t ?- connected(X,Y).
```

```
Error: Out of Stack
```

```
con
```

```
c
```

```
connected(X,X).
```

```
% edges
```

```
e(1
```

```
% t ?- connected(X,Y).
```

```
Error: Out of Stack
```

```
con
```

```
e(
```

```
connected(X,X).
```

```
% edges
```

```
e(1
```

```
?- connected(X,Y).
```

```
connected(1,1);
```

```
connected(2,2);
```

```
connected(3,3);
```

```
connected(1,2);
```

```
<continues forever>
```

```
connected(X,_) ; e(2,1).
```

```
% edges
```

```
e(1
```

```
?- connected(X,Y).
```

```
connected(1,1);
```

```
connected(2,2);
```

```
connected(3,3);
```

```
connected(1,2);
```

```
<continues forever>
```

```
e(X,_) ; connected(2,1).
```

# Motivating example

```
% edges  
e(1,2). e(2,3).
```

```
% transitive closure  
connected(X,Y) :-  
    connect(X,Y).  
connected(X,X).
```

```
% edges  
e(1,2).
```

```
% transitive closure  
connected(X,X).  
connected(X,Y) :-  
    connected(X,Z), e(Z,Y)
```

```
% edges  
e(1,2).
```

```
% transitive closure
```

**Logically** all programs are  
equivalent  
**Operationally** they behave  
differently

```
% transitive closure  
connected(X,X).  
connected(X,Y) :-  
    e(X,Z), connected(Z,Y).
```

# Motivating example

```
:- table connected/2.  
% edges  
e(1,2). e(2,3).  
% transitive & reflexive closure  
connected(X,Y) :-  
    connected(X,Z),e(Z,Y).  
connected(X,X).
```

```
:- table connected/2.  
% edges  
e(1,2). e(2,3).  
% transitive & reflexive closure  
connected(X,X).  
connected(X,Y) :-  
    connected(X,Z),e(Z,Y).
```

```
:- table connected/2.  
% edges  
e(1,2). e(2,3).  
% transitive & reflexive closure  
connected(X,Y) :-  
    e(X,Z), connected(Z,Y).  
connected(X,X).
```

```
:- table connected/2.  
% edges  
e(1,2). e(2,3).  
% transitive & reflexive closure  
connected(X,X).  
connected(X,Y) :-  
    e(X,Z), connected(Z,Y).
```

# Motivating example

```
:- table connected/2.  
% e ?- connected(X,Y).  
e(1 connected(1,1);  
% t connected(2,2);  
con connected(3,3);  
c connected(1,2);  
con connected(1,3);  
connected(2,3);
```

```
:- table connected/2.  
% e ?- connected(X,Y).  
e(1 connected(1,1);  
% t connected(2,2);  
con connected(3,3);  
e( connected(1,2);  
con connected(1,3);  
connected(2,3);
```

```
:- table connected/2.  
% e ?- connected(X,Y).  
e(1 connected(1,1);  
% t connected(2,2);  
con connected(3,3);  
con connected(1,2);  
c connected(1,3);  
connected(2,3);
```

```
:- table connected/2.  
% ?- connected(X,Y).  
e( connected(1,1);  
% connected(2,2);  
co connected(3,3);  
co connected(1,2);  
e connected(1,3);  
connected(2,3);
```



# Motivating example

```
:- table connected/2.
```

```
% edges
```

```
e(1,2). e(2,3)
```

```
% transitive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
connected(X,Z),
```

```
:- table connected/2
```

```
% edges
```

```
e(1,2). e(2,3)
```

```
% transitive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
connected(X,Z),
```

With *tabling* the operational behaviour *is* identical.

```
:- ta
```

```
% edges
```

```
e(1,2). e(2,3)
```

```
% transitive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
connected(X,Z), e(Z,Y)
```

```
% transitive closure
```

```
connected(X,X).
```

```
connected(X,Y) :-
```

```
e(X,Z), connected(Z,Y).
```



## Flora 2 - Knowledge Base System

Constraint Problems

Ontologies, Semantic web, ...

# Motivation

- ★ Tabling makes logic programming more declarative:  
clause and goal ordering agnostic
- ★ Tabling can make the program (asymptotically) more efficient by memoisation of the results

# Tabling support is lacking

**YAP**rolog



*xsb*



Supports Tabling



**SWI** Prolog

**SICS**<sup>4</sup>tus

BinProlog



...

Doesn't support tabling

# Implementing tabling is Hard

Implementing  
tabling ~~is~~ Hard  
was

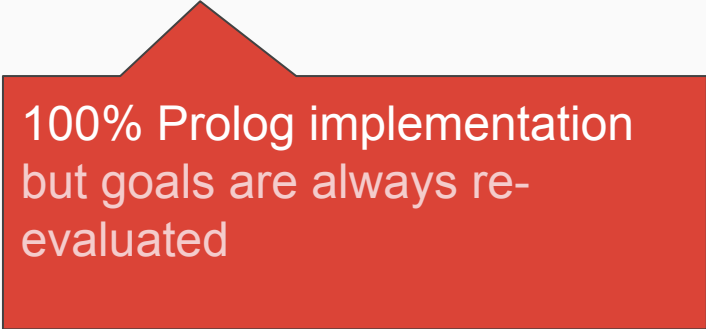
A photograph of a grand, multi-level library. The upper level is filled with tall wooden bookshelves packed with books. A central aisle on the upper level is flanked by metal railings. Below, a lower level features a long, narrow display case with a red carpeted floor. The display case contains several books and documents. The overall atmosphere is one of a well-preserved, historical library.

# Tabling Library in Prolog

## Extension Tables

engineering effort: **small**

performance: **poor**



100% Prolog implementation  
but goals are always re-  
evaluated



# Tabling Library – Earlier Approaches

Hybrid Approach:

Transformation + Special Extensions

engineering effort: **significant**

performance: **good**

- specific extensions
- complex transformation
- low level memory management

Ramesh, R. & Chen, W., A Portable Method for Integrating SLG Resolution into Prolog Systems (1994)

De Guzmán et al., An Improved Continuation Call-Based Implementation of Tabling (2007)

## Our Approach: Delimited Control

engineering effort: **low**

performance: **acceptable**

- simple shallow program transformation
- small generic extensions  
effort can be amortised

Schrijvers, T. et al., Delimited Continuations for Prolog (2013)

Felleisen, M., The Theory and Practice of First-class Prompts (1988)

Danvy, O. & Filinski, A., Abstracting Control (1990)

Plotkin, G. & Pretnar, M., Handling Algebraic Effects (2013)

# Aside: Delimited Continuations

Two Primitives:

`shift/1`



suspend computation

`reset/3`



capture suspended  
computation



**users of tabling never see this!**

# Aside: Delimited Continuations

Two Primitives:

`shift/1`



suspend computation



`reset/3`



capture suspended  
computation

**other applications: Effect Handlers**

# Tabling Library with Delimited Control

prototype implementation < **600 lines of Prolog**

Tabling Logic  
60 lines of  
Prolog

data structures, glue code, etc  
517 lines of Prolog



# Summary

# Summary

- ★ **Tabling** makes logic programs significantly more declarative
- ★ Existing systems don't support tabling:  
**hard to do with existing techniques**
- ★ With **delimited control**:  
**no reason** not to implement tabling

# Conclusion

**YAP**rolog



*xsb*



Supports Tabling



**SWI** Prolog

**SICS**<sup>4</sup>tus

BinProlog



Visual Prolog



...

Doesn't support tabling



# Conclusion

**YAP**<sub>rolog</sub>



*xsb*



Supports Tabling



**SWI Prolog**



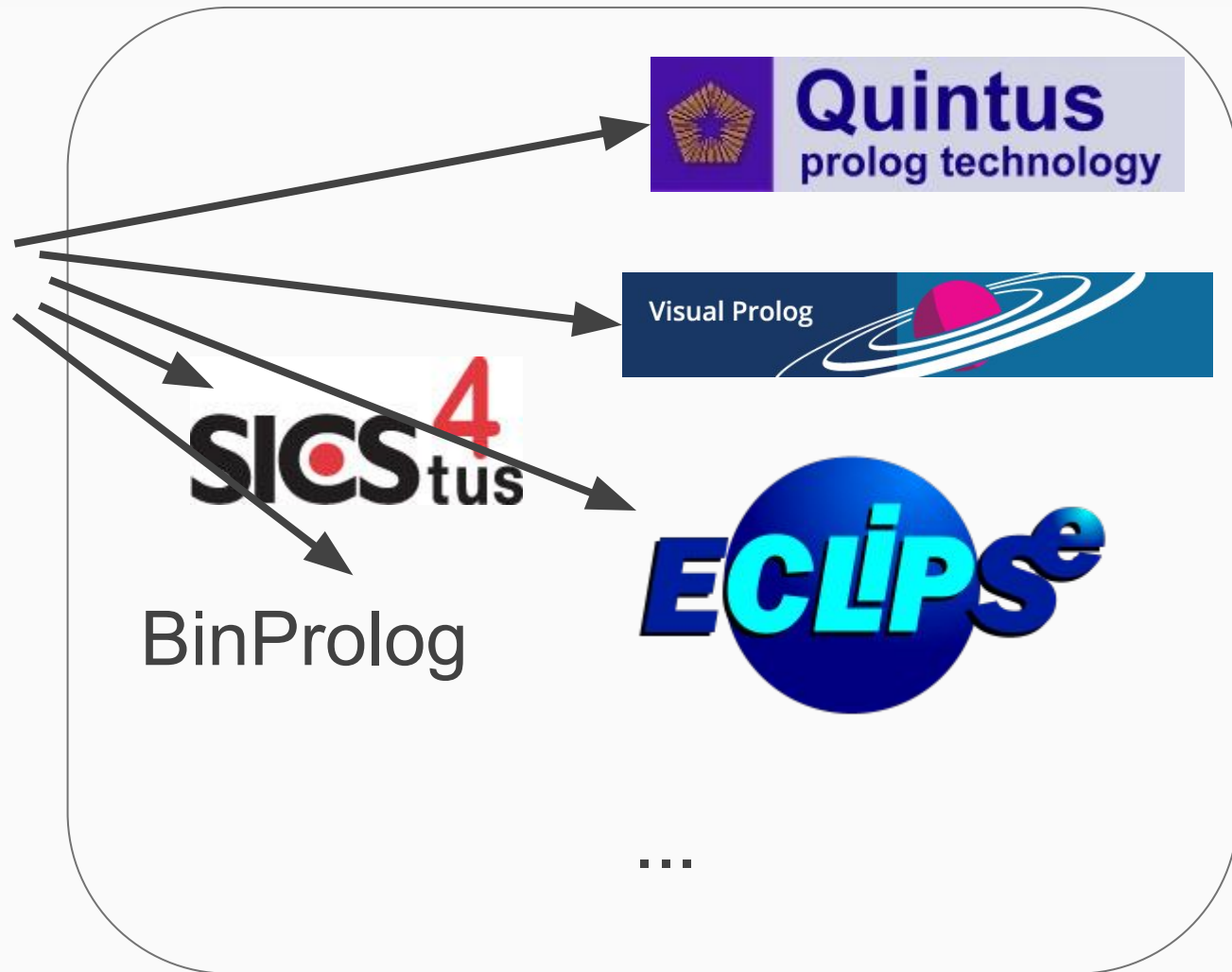
- ★ with delimited control
- ★ control logic in Prolog
- ★ data structures in C
- ★ Performance 2x-4x of native

...

Doesn't support tabling

# Conclusion

There's no  
excuse now!



Doesn't support tabling

# Questions

