# Just Enough Tabling

Konstantinos Sagonas
Computing Science
Dept. of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Peter J. Stuckey
National ICT Australia Victorian Laboratories
Dept. of Comp. Sci. & Soft. Eng.
University of Melbourne, Australia
pjs@cs.mu.oz.au

## ABSTRACT

We introduce *just enough tabling* (JET), a mechanism to suspend and resume the tabled execution of logic programs at an arbitrary point. In particular, JET allows pruning of tabled logic programs to be performed without resorting to any recomputation. We discuss issues that are involved in supporting pruning in tabled resolution, how re-execution of tabled computations which were previously pruned can be avoided, and we describe the implementation of such a scheme based on an abstract machine like CHAT, which implements the suspension/resumption support that tabling requires through a combination of freezing and copying of execution states of suspended computations. Properties of just enough tabling and possible uses of the JET mechanism in a tabling system are also briefly discussed.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*

## General Terms

Languages, Algorithms

## Keywords

Logic programming, tabling, suspension/resumption in the WAM, pruning

## 1. INTRODUCTION

Resolution strategies based on tabling, such as OLDT [13] or SLG resolution [3], provide execution mechanisms for logic programs which are often more efficient and flexible than SLD resolution. Their efficiency stems from avoiding repeated subcomputations and the added flexibility is due to allowing more programs to terminate. Uses of the XSB system [11] in particular have demonstrated that tabling can provide elegant and efficient solutions to complex problem areas such as symbolic model checking and semantics-based program analysis, and be the basis for efficient non-monotonic reasoning and object-oriented database systems based on logic. As a result, besides XSB, many logic programming systems such as YAP, B-Prolog, ALS-Prolog, and Mercury currently incorporate some form of tabled execution; see [9, 15, 6].

Despite the added flexibility provided by tabling, it is currently the case that implementations of proper tabling[1] do not allow pruning of tabled computations. For example, the XSB compiler (up to version 2.4) rejected programs in which a call to a tabled-dependent predicate statically occurs in the scope of a Prolog cut; partly due to the inadequacy of such a static test, these programs are rejected at runtime in the current XSB version (version 2.6). Although not all uses of pruning operators in the presence of tabling are semantically valid, cases where for example pruning is used to enhance performance in a don't-care `once/1`-like context are. As a result, many semantically valid programs are rejected unnecessarily and effective pruning cannot be employed in tabled programs. The first situation is unfortunate. The second, subtly undermines the *relevance* and goal-directedness properties of the evaluation: due to lack of pruning, many redundant computations may be performed and unwanted answers may be produced unnecessarily.

Pruning tabled computations has been a challenging issue. For quite some time, this line of research has been hopelessly aiming at an integration of Prolog's cut into the SLG resolution framework. More specifically, the focus of attention was how to provide as faithfully as possible a cut-like 'semantics' in the context of a proper tabling resolution strategy (i.e., without requiring recomputation). A satisfactory solution to this problem is quite tricky for the following reasons:

1. Tabled resolution strategies have different operational semantics than Prolog's. In particular, SLG resolution requires a suspension/resumption mechanism to handle dependencies between calls to tabled predicates. Resolution strategies such as linear tabling and DRA also significantly depart from Prolog's clause selection strategy by reordering clauses dynamically.

2. The semantics of tabled resolution strategies is based on sets rather than sequences of answer substitutions. Stated differently, tabling eliminates duplicate answers by failing computations that produce them; under SLD resolution, these computations succeed.

3. In proper tabling resolution strategies, there is an additional


*PPDP'04,* August 24–26, 2004, Verona, Italy.


[1]To avoid confusion, we use the term *proper tabling* to refer to a tabled resolution strategy that does not perform recomputation. OLDT and SLG resolution are such strategies, while the linear tabling strategy SLDT [15] and the dynamic reordering of alternatives (DRA) technique [6] are not.

degree of freedom in the form of a *scheduling strategy*, i.e., the strategy that governs the order of resuming suspended computations by returning answers to them. A scheduling strategy can in turn directly influence the order in which computed answer substitutions are produced.

From the above, it should be clear that uses of cuts in a tabled program cannot always have their familiar Prolog operational semantics. As a result, it should not come as a surprise that we find existing attempts to incorporate Prolog's cut in improper tabling strategies to be quite disappointing. For example, in [15, Section 3.4], after a claim that "*the cut operator in SLDT behaves in strictly the same way as that in SLD resolution*," it is acknowledged that not all programs can be handled. A recent attempt to handle Prolog cuts in DRA ([7]) is also unsatisfactory: because of the dynamic reordering of alternatives, a cut in some clause might prune execution of clauses which textually appear before the one containing the cut, a situation which of course can be quite confusing to a Prolog programmer.

In this paper we take a radically different approach to providing pruning in tabling: Rather than attempting to include the Prolog cut operator in a tabling environment — thus giving programmers a weapon to shoot themselves in the foot if they are not careful — we describe a scheme that, when guided by a static analysis which discovers program points where pruning can occur, enables pruning of tabled computations by the underlying runtime system while at the same time avoids recomputation. This work was partly motivated by our attempt to include tabling in Mercury's execution model [12] where the ability to perform pruning is actually an implementation necessity rather than a luxury. However, even in proper tabling implementations such as XSB that do not perform recomputation, the scheme we describe, when guided by a compiler which performs even simple analyses, has the potential of significantly improving time performance by avoiding unnecessary computations.

For example, consider the execution of the two top-level queries below against the following program.

```
?- query1.
?- query2.

query1 :- t(100000,X), X =< 50000.
query2 :- t(95000,Y), Y =< 55000.

:- table t/2.
t(N,M) :- N >= 0, p(N,N,M).

p(0,N,M) :- long(N,M).  %% computation which eventually
                        %% succeeds with N = M
p(K,N,M) :- K > 0, N1 is N - 1, t(N1,M).
p(K,N,M) :- K > 0, K1 is K - 1, p(K1,N,M).
```

In the top-level of a Prolog interpreter, since none of the two queries contains any variables, there are implicit `once/1` constructs around them.[2] Execution of `query1` starts by the tabled goal `t(100000,X)` and after a series of long computations succeeds, with `X = 50000`. Because of the single solution context, at this point we are not interested in other values of `X` so we should stop execution paths that will generate answers for tables `t(100000,X)`, ..., `t(50000,X)` that the evaluation has encountered. If we do so, we avoid generating all possible answers for these tables, a process which involves a series of long computations. But if we throw away the computations for these partially filled tables, when execution of the second query begins, we will have to redo most of the work that we have done so far. This is clearly wasteful because in our example, if we keep the tables, the second query can be answered without performing any program clause resolution or recomputation.

*Organization of the paper.* The next section overviews tabling and aspects of its implementation. Section 3 contains definitions which are necessary to characterize the concept of just enough tabling in terms of the frontier of a derivation forest. A detailed example of JET is presented in Section 4 followed by a description of its implementation based on the CHAT engine (Sect. 5). Properties of just enough tabling and possible uses of the JET mechanism in a tabling system are discussed in Section 6 and the paper ends by contrasting JET with existing approaches to pruning tabled logic programs and some concluding remarks.

## 2. PRELIMINARIES

We assume familiarity with the terminology of logic programming and with the WAM [14], and limit our attention to definite logic programs.

### 2.1 Tabled resolution and its terminology

In a tabled logic program, some predicates are designated as *tabled* by means of a declaration; all other predicates are *non-tabled* and are evaluated as in Prolog. In this paper, we adopt a convention of denoting all tabled predicates starting with a `t`. Following SLG resolution [3], we will consider two tabled subgoals as identical if they are *variants* of each other (i.e., identical up to variable renaming). However, this is an issue which is orthogonal to the issues of this paper as our results also hold for tabling based on subsumption. Tabled subgoals which are encountered in the evaluation of a query against a program are stored in a global memory area called the *table space*.

When a tabled subgoal is encountered, a check is made to see whether this is its first occurrence or not. This is the purpose of the NEW SUBGOAL operation of SLG resolution. If it is new, the subgoal is termed a *generator*, a table for it is created in the table space, and evaluation uses PROGRAM CLAUSE RESOLUTION to derive answers for the subgoal. Through the NEW ANSWER operation, these answers are also recorded in the table created for this subgoal. All other occurrences of identical subgoals are called *consumers* as they do not use the program clauses for deriving answers but they instead consume answers from this table using ANSWER RETURN operations. Note that a generator also acts as a consumer of its table; we return to this point later.

In evaluating tabled logic programs, one final operation enters the picture: COMPLETION. Its purpose is to determine whether a set of subgoals is *completely evaluated*, meaning that all answers for these subgoals already exist in the tables. If so, then we say that each of these subgoals is *complete* and evaluation is finished as far as these subgoals are concerned. Otherwise, at least one of the subgoals is *incomplete* and evaluation continues by returning answers to subgoals which are checked for completion. How this is done is described in the next section.

### 2.2 Implementation aspects of tabling

*Table space.* An important ingredient for efficiently implementing tabling is the table space. The organization of this area is by now well-understood and efficient data structures for it have been proposed; see [8]. Before a table's status changes from incomplete to complete, the table knows about its generator and its list of consumers, maintains a list of answers in the table (in chronological order of insertion), and keeps information about the points in this

[2] The queries are not explicitly shown wrapped with `once/1` constructs, since we would very much like to decouple the points where pruning of tabled computations takes place from the implicit or explicit presence of Prolog-style pruning operators.

list up to which answers have been returned to each consumer. On the other hand, automatic memory management aspects of the table space are less explored. Typically, after completion, space for generators and consumers is reclaimed but information about calls and answers to tabled predicates is persistent and only deallocated through explicit programmer intervention, regardless of whether there is anticipated demand for these tables. In a tabled system that supports some form of pruning, either via Prolog's cut ([15, 7]) or as suggested by Castro and Warren [2], tables that are still incomplete and lie in the scope of a pruning operator are typically abolished when pruning occurs. This is because tabling engines do not maintain enough information to allow them to continue the process of tabled resolution from an arbitrary execution point.

*Suspension and resumption.* In the context of the WAM, whose search strategy is pure depth-first search, implementation of proper tabling is complicated by the fact that the generation and consumption of answers are asynchronous and interleaved events. As a result, to avoid recomputation, a tabled engine needs to retain or reconstruct execution environments of calls to a tabled predicate until the table's completion. Likewise, newly derived answers must be queued to resolve against subgoals which do not necessarily correspond to the current execution environment. In fact, a tabled engine may need to switch back and forth between different consumers multiple times.

In general, a consumer needs to be *suspended* when it has exhausted all answers currently in the table and *resumed* when new answers have been derived for it. Suspension is performed in the SLG-WAM [10] by creating a *consumer choice point* to represent the suspended environment, freezing all stacks by setting the freeze registers to point to the current top, and then failing to a previous choice point without reclaiming any stack space; in particular, the freeze registers are not reset. Frozen space is not reclaimed until completion of the corresponding table. Resuming, besides restoring the WAM registers to the values saved in the consumer choice point, uses the addresses and the values saved in a *forward trail* to restore variable bindings along the path to the suspended consumer. The next unconsumed answer is then returned to the restored consumer and execution continues by taking the forward continuation of the restored computation.

Instead of maintaining execution states of suspended computations through freezing the stacks and using an extended trail to reconstruct them, one can also preserve environments of tabled subgoals by copying all the relevant information about them in a separate memory area, let execution proceed as in the WAM, and reinstall these copies whenever the corresponding computations need to be resumed. In fact, the suspension/resumption needed for tabling can also be implemented through a combination of freezing and copying as done in CHAT [5]. CHAT leaves the WAM unchanged for Prolog execution. It implements suspension by freezing the heap and local stack and copying the choice points of tabled calls. CHAT selectively and incrementally copies the trail entries (together with values for bindings) between the consumer and the generator which leads this consumer to a memory area separate from the WAM stacks called the *CHAT area*. Resumption happens by copying the saved choice point from the CHAT area back on the CP stack, copying the saved trail segments back on the trail stack and reinstalling the saved bindings.

*Completion and leaders.* To allow space reclamation during tabled execution, implementations of tabling try to determine *completion* of tables: i.e. when the evaluation has produced all their answers. Doing so, involves examining dependencies between tables and interacts with consumption of answers by consumers. The SLG-WAM has a particular stack-based way of determining completion which is based on maintaining *scheduling components*; that is, sets of subgoals which are possibly inter-dependent. A scheduling component is uniquely determined by its *leader*: a (generator) subgoal $G_L$ with the property that subgoals younger than $G_L$ may depend on $G_L$, but $G_L$ depends on no subgoal older than itself. Besides determining completion, the leader of a scheduling component is also responsible for scheduling consumers of all tables that it leads to consume their answers. Obviously, leaders are generally not statically known and might change in the course of tabled evaluation. Leaders can be maintained either approximately or precisely by having each choice point maintain information about its *root tabled subgoal* (as described in [10, Section 6.4]). The root tabled subgoal for any choice point is the table of the youngest generator on the CP stack (the nearest generator which lead to the creation of this choice point through tabled resolution). We elaborate in Section 3. In fact, in this paper we assume that information about root tabled subgoals is maintained and present in all choice points, regardless of whether it is used for determining completion or not.

*Double-duty of generators.* As mentioned, a generator also needs to act as a consumer of its table. In fact, depending on the scheduling strategy, a generator can either consume answers immediately when these are produced, or can postpone their consumption until all of them have been generated. The former is what happens in *batched scheduling*; the latter is the action when *local scheduling* is employed. In order to make this double-duty of generators explicit, we will show generator choice points as having an associated consumer. This consumer can be thought as a choice point which is tightly coupled with the generator and eagerly consumes the answers as these are produced (in batched) or is a separate choice point placed immediately before the generator choice point (in local) so that it is picked up after the generator is exhausted.

## 3. DERIVATION FORESTS

Tabling implicitly involves the concurrent traversal of multiple derivation trees, forming a derivation forest. In this section we give theoretical definitions to explain the tabling searches we will undertake, what we consider proper tabling, and set the foundation to characterize the amount of recomputation that just enough tabling can avoid.

As usual, an *atom* is of the form $p(\bar{t})$ where $p$ is a predicate symbol and $\bar{t}$ is a sequence of terms. A *goal* $G_i$ is a sequence of atoms. We use $\Box$ to denote the empty sequence of atoms. We also add a special goal *fail* to indicate failure. A set of tables $T$ consists of triples of the form $\langle Call, Answers, Status \rangle$ where $Call$ is an atom of a tabled predicate, $Answers$ is a set of instances of the $Call$, and $Status$ is an indication of the completion status of the table. We require — and the definitions below ensure — that in $T$ there are no two tables where the corresponding calls are variants of each other.

Tabling systems start from an initial goal $G_0$ and a set of tables $\{\langle G_0, \emptyset, incomplete \rangle\}$ and construct and explore a *derivation forest* $\mathcal{F}$ which is a set of derivation trees $F$ and a set of tables $T$. To simplify the presentation we require that the initial goal $G_0$ is an atom of a tabled predicate.

The construction of the forest $\mathcal{F}$ proceeds as a sequence of operations defined below. There are operations that affect the set of tables $T$ only: creation of a new table, addition of a new answer in some table, and change of a table's status from *incomplete* to *complete*. Likewise, there are operations that only affect derivation trees: ordinary program clause resolution and answer return via table lookup. Given a goal $G_0$ a *derivation* $\langle G_0 \rangle \Rightarrow_{\theta_1} \langle G_1 \rangle \Rightarrow \cdots \Rightarrow_{\theta_n} \langle G_n \rangle$ is a sequence of derivations steps defined below. Furthermore, we will abbreviate a derivation as $G_0 \Rightarrow^*_\theta G_n$ where $\theta = \theta_1 \circ \cdots \circ \theta_n$.

In operations affecting goals, the corresponding *derivation step* $G_i \Rightarrow_\theta G_{i+1}$ is as follows. Let $G_i \equiv p(\bar{s}), G$. (In the derivation tree, the new goal $G_{i+1}$ is added as a child of node $G_i$.)

(PROGRAM CLAUSE RESOLUTION) If $p$ is a non-tabled predicate or $i = 0$ (this is the root goal so $G = \Box$), and $p(\bar{r}) \leftarrow B$ is a (renamed-apart) rule in $P$ then, let $\theta = mgu(\bar{s} = \bar{r})$ and $G_{i+1} \equiv \theta(B), \theta(G)$. If the most general unifier does not exist, then $\theta = \emptyset$ and $G_{i+1} = \mathit{fail}$.

(ANSWER RETURN) If $p$ is a tabled predicate and $i \neq 0$ (so this is not a root goal), and $T$ contains an entry $\langle p(\bar{r}), A, \_\rangle, A \neq \emptyset$ where there is a bijective variable renaming $\rho$ such that $\rho(\bar{r}) = \bar{s}$ (i.e. $\bar{r}$ and $\bar{s}$ are variants), and $\theta'(p(\bar{r})) \in A$ (that is $\theta'(p(\bar{r}))$ is an answer for $p(\bar{r})$) then $\theta = \{\rho(v) \mapsto \rho(\theta'(v)) \mid v \in vars(\bar{r})\}$ and $G_{i+1} \equiv \theta(G)$. If $T$ contains an entry $\langle p(\bar{r}), \emptyset, complete\rangle$ where there is a bijective variable renaming $\rho$ such that $\rho(\bar{r}) = \bar{s}$ (hence the table is complete and has no answers), then $\theta = \emptyset$ and $G_{i+1} = \mathit{fail}$.

In operations affecting tables, the set of tables $T$ is changed. The operations are:

(NEW SUBGOAL) If there exists a node $G_i \equiv p(\bar{s}), G$ in $S$ where $p$ is a tabled predicate and $T$ does not contain a table for a variant of $p(\bar{s})$ (a table entry of the form $\langle p(\bar{r}), \_, \_\rangle$, where $p(\bar{s})$ and $p(\bar{r})$ are variants), then $T$ becomes $T \cup \{\langle p(\bar{s}), \emptyset, incomplete\rangle\}$. We also add a new derivation tree to $F$ rooted by $\langle G'_0 \rangle$ where $G'_0 \equiv p(\bar{r})$ where $\bar{r}$ is a variant of $\bar{s}$.

(NEW ANSWER) If there exists a leaf node $G_i \equiv \Box$ in $F$ and the derivation step is part of a derivation $G_0 \Rightarrow^*_\theta G_i$ rooted by $G_0 \equiv p(\bar{r})$, and if $t = \langle p(\bar{r}), A, incomplete\rangle$ is the table for the $p(\bar{r})$ call in $T$ and $A$ contains no answer which is a variant of $\theta(p(\bar{r}))$, then $T$ becomes $(T - \{t\}) \cup \{\langle p(\bar{r}), A \cup \{\theta(p(\bar{r}))\}, incomplete\rangle\}$ (i.e., insert the answer substitution $\theta$ to the set of answers for $t$).

A *derivation tree* for goal $G_0$ shares all the derivations for $G_0$. When a PROGRAM CLAUSE RESOLUTION operation applies at $G_i$ then this node has one child for each rule for $p$ in the program. When ANSWER RETURN operations apply at $G_i$ then this node has one child for each tabled answer of the call $p(\bar{r})$, or a single child *fail* if the corresponding table is complete without any answers. Note that it is possible for the derivation trees to have infinite branching factors.

Also, note that there is a strong relationship between trees in a derivation forest and the set of tables that a derivation starting from an initial goal creates. In particular, there is a one-to-one correspondence between tabled calls and roots of derivation trees in the forest and the table corresponding to the root of each derivation tree is the *root tabled subgoal* for each goal $G_i$ appearing in its derivation tree.

Given a forest $\mathcal{F}$ for a goal $G_0$, another operation comes to play:

(COMPLETION) Let $C$ be a set of calls in $\mathcal{F}$ for which all preceding operations have been exhaustively performed. Moreover, no call in $C$ has a corresponding tree which contains a selected atom for a call whose table has status incomplete and is not in $C$. Then change the tables' status for calls in $C$ from $incomplete$ to $complete$ and simultaneously add *fail* children to all nodes in $\mathcal{F}$ which have a selected atom in $C$ and have no children.

DEFINITION 1 (FRONTIER). *A* frontier *of a derivation forest $\mathcal{F}$ for goal $G_0$ is a partial numbering of nodes in $\mathcal{F}$ that satisfies the following conditions:*

- *The node $G_0$ is numbered* 1.
- *Each numbered child node has a number greater than its parent node.*
- *Each numbered node created by an* ANSWER RETURN *operation has a number greater than the number of $\Box$ in at least one derivation $p(\bar{r}) \Rightarrow^*_{\theta'} \Box$.*
- *Each root node $p(\bar{r})$ apart from $G_0$ has a number greater than at least one node $G_i \equiv p(\bar{s}), G$ where $p(\bar{r})$ and $p(\bar{s})$ are variants.*
- *Each numbered fail node created by a* COMPLETION *operation (where the selected atom is $p(\bar{s})$) has a number greater than all the numbers on the fail nodes in the derivations $p(\bar{r}) \Rightarrow^*_{\theta'}$ fail ($p(\bar{r})$ is a variant of $p(\bar{s})$).*
- *Each fail node added by a single* COMPLETION *operation has the same number.*

A *scheduling strategy* defines an exploration of trees in a derivation forest. In particular, during execution, a scheduling strategy $S(G_0)$ returns a frontier of the derivation forest for $G_0$. We can view this as an incremental exploration of the forest by considering all the partial frontiers defined by removing nodes which are either still unnumbered or whose number is greater than some $n$.

The above definition of the frontiers returned by a scheduling strategy is restricted to proper tabling systems. A *proper tabling* system executing a goal $G_0$ never executes a derivation step in the derivation forest for $G_0$ more than once.

# 4. A STEP-BY-STEP EXAMPLE OF JUST ENOUGH TABLING

We illustrate the concept of just enough tabling using the tabled logic program shown in Figure 1. For presentation purposes we have annotated it with an indication of the program points (❶, ❷) where pruning could take place. We refer to these points as *just enough tabling* points (or JET points). How these points can be automatically discovered is discussed in Section 6.2.

## 4.1 Derivation forest level description

Let us examine the forest of derivation trees built when executing this program under batched scheduling. In Figure 2, the nodes in the forest are numbered in the order that they are created, thus defining a frontier. The nodes explored to reach ❶ are shown shaded, while the additional nodes explored to reach ❷ are shown shaded surrounded by a bounding box.

The derivation proceeds as follows. The goal `test` finds a solution to `start(S)` and then calls `t(a)` which is a subgoal of a tabled predicate. A new derivation tree is created for this subgoal, and execution proceeds by first trying the first alternative which encounters a recursive call to `t(a)`. Since there are as yet no answers for this table, this execution path is suspended, and we try the other program clause for `t/1` leading to the call `tc(a,Y)`.

This begins a new derivation tree. Since variables across trees are not shared, we denote the root of this tree by `tc(a,Ya)`. The derivation for `tc(a,Ya)` calls `tle(a,Z)` and a new derivation tree with root `tle(a,Za)` is created. After a long computation, the first answer (`Za = a`) is found. Under batched scheduling, it is returned to the call `tle(a,Z)` setting `Z = a`. Execution proceeds in the derivation tree for `tc(a,Ya)` with the call `tc(a,Ya)`, which currently has no answers, and this path suspends.[3]

[3]Note that this derivation path will in fact never generate any new answers for `tc(a,Ya)`.

```
:- table test/0, t/1, tc/2, tle/2.
test :- start(S), t(S)❶, tle(S,G), good(G)❷.

t(X) :- p(X).                                   % (1)
t(X) :- tc(X,Y).                                % (2)

tc(X,Y) :- tle(X,Z), tc(Z,Y).                   % (3)
tc(X,Y) :- se(X,Y).                             % (4)

tle(A,C) :- long(A,B), le(B,C).                 % (5)

p(X) :- t(X).

% A long deterministic computation which sets V to U
long(U,V) :- ... V = U.
```

```
?- test.

start(a).

le(a,a).
le(a,b).
le(a,g).
le(e,f).

se(a,c).
se(b,c).
se(b,d).

good(g).
```

**Figure 1: A tabled logic program and query.**

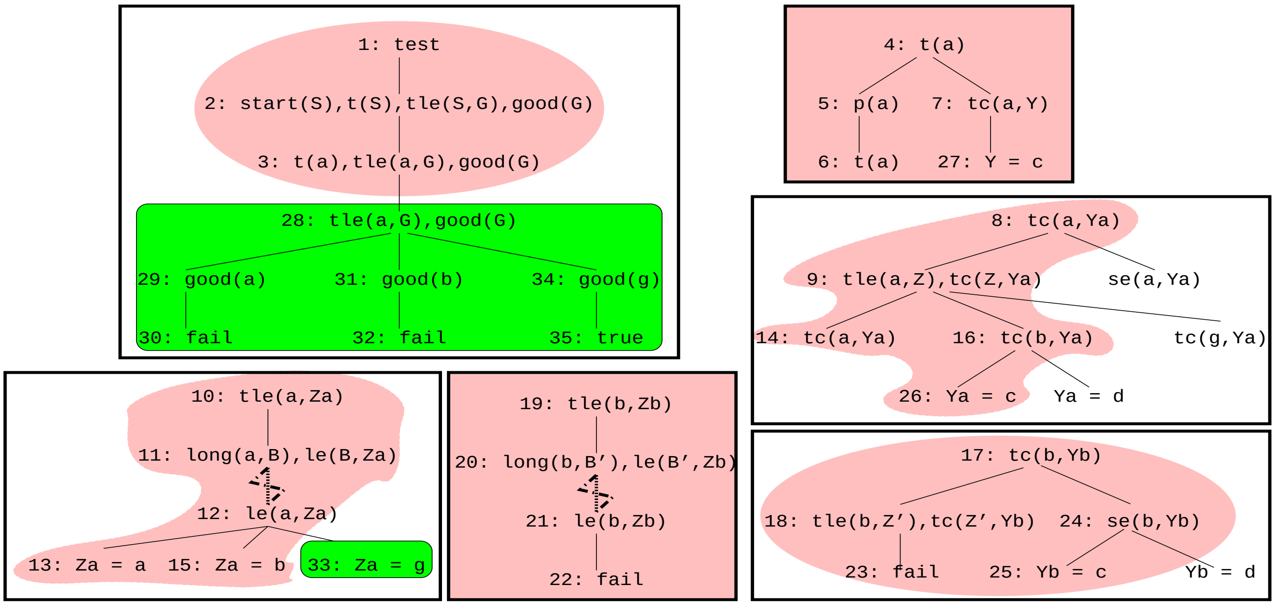



**Figure 2: Forest of SLG trees explored to get to ❶ (shaded areas) and ❷ (boxed shaded areas).**

Execution proceeds at node 15 by finding the next answer to `tle(a,Za)`, `Za = b`. This answer is returned to the call `tle(a,Z)` setting `Z = b`. This leads to a call `tc(b,Ya)` which starts a new derivation tree for `tc(b,Yb)`. This derivation tree creates a new tree for `tle(b,Zb)` which after another long computation fails, and on returning to the tree for `tc(b,Yb)`, its first answer (`Yb = c`) is generated. This answer is returned to the call `tc(b,Ya)` setting `Ya = c`. This produces the first answer for the derivation tree for `tc(a,Ya)`, which in turn produces the first answer for `t(a)`. We are now at node 27 in the forest (in the tree rooted by `t(a)`) and point ❶ in the program where pruning is supposed to take place.

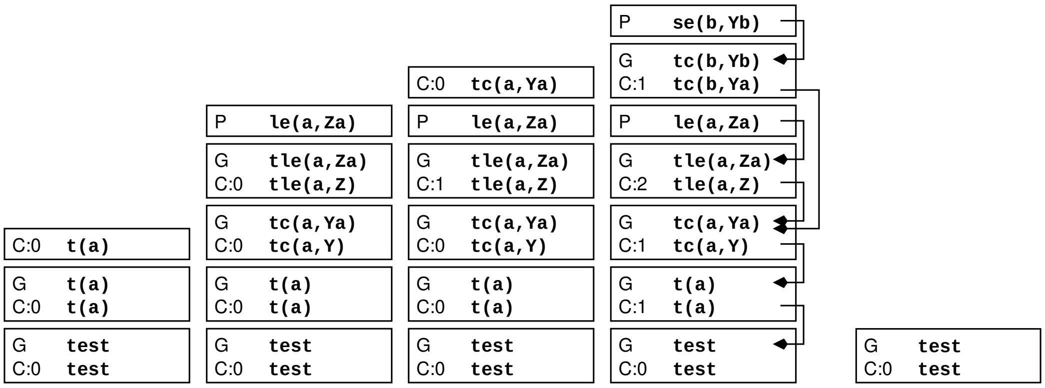
If pruning performs an action like Prolog's cut, then all unexplored alternatives are cut away and all incomplete tables (those of calls `tc(a,Ya)`, `tle(a,Za)`, and `tc(b,Yb)` in our example) are abolished at this point. If pruning takes place as proposed by Castro and Warren [2] the action is similar, because none of the tables is demanded at this point. In either case, if a call to one of these tables is encountered later in the computation, all execution performed so far for these tables will need to be repeated. This is clearly wasteful and not in accordance to using tabling as a means to avoid recomputation.

In our example, some of the tabled computations performed thus far are needed. In fact, immediately after JET point ❶, execution proceeds with a call to `tle(a,G)`. If we preserve information in the tables and in the derivation trees, execution can begin by examining the derivation tree for `tle(a,Za)` and consume answers from the corresponding table. Here we begin to see why sharing variables across derivation trees is not desirable. We want to reuse the answers originally calculated for the call `tle(a,Z)` at node 9 for the call to `tle(a,G)` at node 28. Execution can proceed by returning the already calculated answer substitutions `Za = a`, `Za = b`, copying them onto the variable `G`. Both these fail, so we must further execute the derivation tree for `tle(a,Za)`. Execution proceeds finding the new answer `Za = g`. This is returned to `tle(a,G)`, leads to success, and we have reached JET point ❷. The table for `test` is *early completed* (cf. [10]) at this point and execution stops.

## 4.2 Abstract machine level description

For the execution of the query `?- test`, a WAM-based tabling abstract machine builds the choice point stacks shown in Figure 3. Choice points are denoted as G, C, or P depending on whether they correspond to generators, consumers, or calls to non-tabled predicates which are evaluated using Prolog-style execution, respectively. We explicitly associate a consumer with each generator and the two choice points are tightly coupled. In addition, all consumers are annotated with an integer denoting the number of answers from the corresponding table they have consumed.

On encountering the first true consumer (the call to `t(X)` in the




(a) At node 6 (b) At node 13 (c) At node 14 (d) At point ❶ (e) After pruning

**Figure 3: Choice point stacks (growing upwards) during the execution of the query `test`.**

body of the code for `p(X)`), the stack is as in Figure 3(a). Since at this point the table for `t(a)` contains no answers, the consumer is suspended, taken off the choice point stack, and attached to the table of its generator. Execution continues with the second clause of the `t/1` predicate which results in the choice point stack shown in Figure 3(b). All generators are shown with their variables renamed to emphasize the fact that for each generator, a new SLG tree is created in the SLG forest. By copying answers in and out of the tables, variables are disconnected. For example, notice how e.g. the call to `tc(a,Y)` from the `t(a)` tree is shown as a `tc(a,Ya)` generator on the stack. Each answer which is added to the table will be returned to the consumer associated with each generator by copying it back from the table. This answer return will bind the answer to the original variables that the generator's consumer maintains (to `Y` in this case). At this point the first answer substitution for the call `tle(a,Za)` is generated (`Za = a`). Assuming a scheduling strategy like *batched scheduling*, which ensures that generators consume answers eagerly, the answer is returned to the `tle(a,Z)` consumer associated with the generator. Execution now encounters the consumer `tc(a,Ya)`. A consumer choice point is created on the stack (see Figure 3(c)), but since there are no answers to consume at this point, the consumer gets suspended, is popped off the stack and saved in a CHAT area through copying.

Execution continues with the topmost choice point which is that of the `le(a,Za)` call. By picking up the `le(a,b)` clause, one more tabled answer (`tle(a,b)`) is produced and added in the table. This answer is returned to `tle(a,Z)`, and forward execution results in calls to `tc(b,Ya)` (which becomes a canonical form call to `tc(b,Yb)`) and `tle(b,Zb)`. The latter calls `le(b,Zb)` which fails and consequently the `tle(b,Zb)` table gets completed without any answers. Execution backtracks to the generator choice point for the `tc(b,Yb)` call which proceeds executing the second clause of `tc/2` resulting in a call to `se(b,Yb)`. This sets up a choice point and executes the first clause of `se/2` producing the answer `tc(b,c)` which gets recorded in the appropriate table. By returning this answer to the associated consumer, the answer substitution `Ya = c` is produced for `tc(a,Ya)`, which in turn produces an answer for `t(a)`. Note that this can cause the *early completion* of `t(a)` as there cannot be any other (different) answers for this table. By returning this answer, point ❶ is reached. At this time the choice point stack is as shown in Figure 3(d). The same figure shows the root tabled subgoal of each consumer and Prolog choice point in the stack; for example, the root tabled subgoal of the `tc(b,Ya)` consumer is `tc(a,Ya)`. At this point, the tables are as shown in Figure 4; two tables are complete (✔), and the remaining four are *active* (✈).

| Call | Answers | Status | Consumers |
|---|---|---|---|
| test | ∅ | ✈ | |
| t(a) | {t(a)} | ✔ | |
| tc(a,Ya) | {tc(a,c)} | ✈ | {C:1 tc(a,Y), C:0 tc(a,Ya)} |
| tle(a,Za) | {tle(a,a), tle(a,b)} | ✈ | {C:2 tle(a,Z)} |
| tc(b,Yb) | {tc(b,c)} | ✈ | {C:1 tc(b,Ya)} |
| tle(b,Zb) | ∅ | ✔ | |

**Figure 4: State of the tables before JET point ❶.**

Now JET pruning happens. To be able to pick up the tabled computations from the point that has been reached so far and avoid recomputation, execution states of incomplete tables must be preserved at this point.[4] The JET action for the heap and local stack is to freeze them. Recall that the choice point stack acts like a scheduling stack by keeping information about alternatives which are yet to be explored for tables to get all their answers. Since we are at a JET point and no more answers for the pruned tables are currently needed, choice points in the scope of the pruning operator must be taken out of the CP stack so that we avoid exploring these alternatives unnecessarily. Thus, the JET action is to run through the choice point stack from its top to the choice point associated with the JET point (the `test` generator in this case), and preserve through copying choice points of computations originating from incomplete tables that are JET pruned. We can copy the choice points as a single chunk, but since we want the ability to re-activate the computation of an arbitrary table, we will associate each choice point with the table that needs it. In effect, at this moment choice points on the stack are partitioned according to their *root tabled subgoal*.

The copied information, per table, is shown in Figure 5. Notice how the consumer parts of generators are separated from them at this point and attached to the table of their root table subgoal. No information is preserved for `t(a)` since that table is complete. This explains why the `t(a)` and `tc(a,Y)` consumers are discarded. Similarly, if there were any Prolog or consumer choice points which

[4]For complete tables, no execution state needs to be preserved since all answers are in the tables already.

| | |
|---|---|
| P | `se(b,Yb)` |
| G | `tc(b,Yb)` |

(a) For `tc(b,Yb)`

| | |
|---|---|
| P | `le(a,Za)` |
| G | `tle(a,Za)` |

(b) For `tle(a,Za)`

| | |
|---|---|
| C:1 | `tc(b,Ya)` |
| C:2 | `tle(a,Z)` |
| G | `tc(a,Ya)` |

(c) For `tc(a,Ya)`

**Figure 5: Information from the CP stack that is preserved at JET point ❶ through copying.**

| | |
|---|---|
| G | **`test`** |
| C:0 | **`test`** |

(a) $1^{st}$ pruning

| | |
|---|---|
| C:0 | **`tle(a,G)`** |
| G | **`test`** |
| C:0 | **`test`** |

(b) Consumer call

| | |
|---|---|
| P | **`le(a,Za)`** |
| G | **`tle(a,Za)`** |
| C:2 | **`tle(a,G)`** |
| G | **`test`** |
| C:0 | **`test`** |

(c) Re-activation

| | |
|---|---|
| G | **`tle(a,Za)`** |
| C:3 | **`tle(a,G)`** |
| G | **`test`** |
| C:0 | **`test`** |

(d) At point ❷

**Figure 6: Choice point stacks between the two JET pruning actions.**

are younger than the `test` generator but older than the last generator which needs to saved (`tc(a,Ya)` in this case) they also would simply be pruned away. Finally, notice how the partitioned choice point chains correspond to the trees of Figure 2.

After JET pruning, execution calls `tle(a,G)`. If pruning had removed all information for this table, this call would have been a generator and a new table would have been created at this point. However, in our case, a table for this call exists, namely a table that has been made inactive by pruning and is not complete, so this call is a consumer. Therefore, a consumer choice point is put on the choice point stack (cf. Figure 6(b)) with its next alternative pointing to an instruction that indicates that this is a consumer of a table which has been put on hold by JET pruning. This consumer will first backtrack through the existing answers in the table (`G = a` and `G = b`) but forward execution will in both cases fail when calling `good(G)`. As in any other case where failure occurs, execution will pick up the next alternative which as mentioned indicates this is a consumer of a JET-pruned table. The JET action at this point will be to reload the saved choice points of Figure 5(b) into the stack in order to generate more answers for this table. The stack will now be as shown in Figure 6(c). As we re-activate the computation for the `tle(a,Za)` table, we change the table's status to reflect the fact that this table is active again. Execution continues with the next (and last) alternative of the top-most choice point which generates the answer `tle(a,g)`. The answer is returned to the `tle(a,G)` consumer, the call to `good(G)` now succeeds and point ❷ is reached.

Another JET pruning operation will take place at this point. Although table `tle(a,Za)` actually has all its answers, it is not yet marked complete, as JET pruning has prevented the completion check from occurring. Since in general it is not known whether this table will ever need to be re-activated, information about the point that its execution has reached needs to be preserved. Running down the CP stack from its top to the choice point associated with the JET point (the re-activated `tle(a,Za)` generator in this case) will update the CP information saved for the `tle(a,Za)` table to be just the generator choice point for this call. Execution finishes at this point and the tables are as shown in Figure 7; now three of them are complete and three have been *put on hold* (⇝) by JET pruning.

| Call | Answers | Status | Consumers |
|---|---|---|---|
| `test` | {`test`} | ✔ | |
| `t(a)` | {`t(a)`} | ✔ | |
| `tc(a,Ya)` | {`tc(a,c)`} | ⇝ | {C:1 `tc(a,Ya)`} |
| `tle(a,Za)` | {`tle(a,a)`, `tle(a,b)`, `tle(a,g)`} | ⇝ | {C:2 `tle(a,Z)`} |
| `tc(b,Yb)` | {`tc(b,c)`} | ⇝ | {C:1 `tc(b,Ya)`} |
| `tle(b,Za)` | $\emptyset$ | ✔ | |

**Figure 7: State of tables after JET point ❷.**

# 5. IMPLEMENTATION ASPECTS OF JUST ENOUGH TABLING

In this section we describe the implementation of just enough tabling in the context of a WAM-based tabled abstract machine. We choose CHAT [5] as a basis for our presentation because in CHAT choice points are organized in a stack rather than a tree and, as we will see, CHAT already contains most of the machinery which is necessary to implement JET. We describe the actions performed by the abstract machine when JET pruning occurs (Sect. 5.1), and how tables which are put on hold by JET pruning are re-activated upon encountering a subsequent call for them (Sect. 5.2).

## 5.1 JET pruning

First of all, tables which are already complete or put on hold as the result of a prior JET pruning operation are not affected by the pruning performed by just enough tabling; in Section 6.3 we elaborate more on this. Secondly, although the example of Section 4 does not explicitly show this, we prefer that JET pruning is *local to the clause* that contains a JET point (i.e., unlike Prolog's cut, JET pruning does not prevent execution of clauses which follow the clause with the JET annotation).

At the moment when a JET point is reached, the current CP stack top is pointed to by the WAM's **B** register. Let $\mathbf{B_0}$ denote the topmost CP when execution reaches the point in the body of the clause containing the JET annotation that denotes the beginning of the scope of the JET pruning action. The fact that JET pruning is clause-local means that the choice point pointed by $\mathbf{B_0}$ will not be affected by JET pruning.

In CHAT, each table contains a pointer to a CHAT area where

```
for ( ; B != B0 ; B = cp_prev(B)) {
  Tr = B->RS; /* the root table of the choice point */
  if (cp_type(B) == generator) {
    if (! complete(Tr))
      copy the consumer part of the CP in CP(Tr);
    Tg = the table of this generator CP;
    if (! complete(Tg))
      copy the generator part of the CP in CP(Tg);
  }
  else /* consumer or Prolog choice point */
    if (! complete(Tr)) copy the CP on CP(Tr));
}
```

**Figure 8: JET pruning action for the CP stack.**

suspended consumer choice points, organized in a linked list, and the associated trail chunks are stored. In order to support just enough tabling, we simply extend CHAT areas with the ability to also store pruned portions of the CP stack. For a table $\mathbf{T}$, this area is denoted $\mathbf{CP}(\mathbf{T})$.

We also assume the presence of auxiliary functions (or macros) cp_prev() and cp_type() which return the previous choice point and the choice point type of its argument, and complete() which returns true or false depending on whether its argument is a complete or incomplete table, respectively.

When JET pruning occurs, the action for the choice point stack is as shown in Figure 8. In words: Run down the CP stack from its current topmost choice point to, but not including, the choice point pointed by $\mathbf{B_0}$ and copy each choice point to the $\mathbf{CP}(\mathbf{T})$ area, where $\mathbf{T}$ is the table determined by the root parent subgoal field of the copied choice point. Note that:

- JET pruning decouples a generator from its associated consumer. If the generator is ever re-activated, a new consumer will be associated with this generator.
- Since tables that are already complete do not have a CHAT area (if they ever had one, it was reclaimed when they got completed), no CP stack portion is saved for them.
- At the end of the for loop, the **B** register points to what will be the new stack top after JET pruning: the choice point denoted as $\mathbf{B_0}$. Choice points younger than $\mathbf{B_0}$ are removed from the stack.
- As far as the CP stack is concerned, the cost of JET pruning is linear in the number of choice points which are pruned. Note that this is the same cost as that of checking during run-time whether an incomplete table lies in the scope of a pruning operator; a cost present in the current XSB version which does not allow pruning over tabled predicates in its default configuration.

Backtracking in the WAM, besides picking up another alternative, also allows instant reclamation of heap and local stack space which was allocated after the creation of the choice point. In order for table re-activations to pick up pruned computations from the point reached before JET pruning occurred, backtracking cannot perform instant reclamation after JET pruning. Thus, the JET pruning operation has to freeze the information on the heap and local stack. In CHAT, this can be done either 1) eagerly, by running the CP stack till its oldest choice point and adapting the H and EB fields of all choice points to point to the current heap and local stack tops, or 2) lazily, by doing this adjustment in a more incremental way (e.g. by modifying these values for the topmost CP and have a special trust instruction that propagates the freezing to the previous choice point upon backtracking; see [4] for more information).

In CHAT, as in the WAM, the trail is segmented by choice points and the trail is exactly that of the WAM. The JET pruning action for the trail is analogous to that for the CP stack. Trail segments, together with their corresponding trailed values, need to be preserved through copying and attached to the corresponding area where the choice point is saved. As a matter of fact, in CHAT this means that only trail segments corresponding to generator and Prolog choice points need to be copied. This is because CHAT eagerly saves the execution environments of each consumer of an incomplete table when the consumer is encountered. (The rationale for this is that in the absence of pruning the consumer will eventually need to be suspended anyway before execution fails back to the generator to check for completion of its table). Note that copying of trail segments increases the total cost of tabled execution by only a constant factor. This cost is comparable to the cost of tidying the trail after a Prolog cut. That copying rather than freezing is required for the trail is not surprising since, in an LP engine like the WAM that uses untrailing rather than copying of trailed entries, the TR fields of choice points cannot be changed without corrupting backtracking.

Finally, note that the algorithm of Figure 8 is very simple and the machinery for JET pruning is already available in CHAT; the only detail that is missing is the ability to save different types of choice points in CHAT areas rather than just choice points of consumers.

*JET pruning in the SLG-WAM.* Although CHAT lends itself naturally to the implementation of just enough tabling, adapting the JET mechanism to a tabled abstract machine like the SLG-WAM is not difficult either. One approach is that the SLG-WAM modifies its choice point management to be like CHAT's: have a stack organization for its choice points rather than a cactus stack. Indeed this is possible and such an implementation has already been described in [1]. Then the algorithm of Figure 8 is directly applicable. For the trail the SLG-WAM can preserve execution environments of computations which are put on hold by JET pruning via its usual freezing mechanism and restore them using its forward trail mechanism. If the choice points are not organized in a stack, adapting the JET mechanism is of course still possible but becomes more involved as it also requires some sort of invalidation and cleanup of the trapped choice points and trail entries. On the other hand, this approach has the advantage that the cost of saving and restoration of choice points to and from their saved area through copying is avoided. Which implementation mechanism for JET pruning behaves best in practice remains to be seen, and a theoretical investigation of their tradeoffs along the lines of [4] is a possible avenue for future work.

## 5.2 Table re-activation

Table re-activation is initiated by a variant of the instruction that implements the ANSWER RETURN operation. More specifically, upon encountering a call which is a variant of a tabled call which has been put on hold by a previous JET pruning operation, a consumer choice point is put on the CP stack. Its next alternative contains a modification of the answer_return instruction (which is the usual instruction for consumers) that differs from answer_return in that it flags that consumption of answers happens from a table which is currently on hold. Execution then proceeds by consuming answers from the table, if any. When these are exhausted, rather than failing out from this consumer choice point, as answer_return would normally do, the action taken is to re-activate the computation of answers for the table $\mathbf{T}$ which is on hold. This happens by the engine putting the choice points in the $\mathbf{CP}(\mathbf{T})$ area back to the choice point stack — starting from the generator and naturally by also adjusting their H and EB fields to reflect the current heap and local stack tops — the corresponding trail segments on the trail stack and by re-installing the saved bindings. Tabled execution then continues as usual by picking up the next alternative of the topmost choice point.

Resuming the generator choice point of $\mathbf{T}$ and any Prolog choice points that $\mathbf{CP}(\mathbf{T})$ contains is straightforward and does not require any further action than those already presented. For consumer choice points in $\mathbf{CP}(\mathbf{T})$, an extra action is required when these choice points finish consuming the current set of answers in the table. If they are consumers of a table $\mathbf{T}'$ which is not active (i.e., is not on the stack) but is on hold, then backtracking out of the consumer choice point needs to trigger the re-activation of the $\mathbf{T}'$ table. This action is needed for proper completion: $\mathbf{T}$ cannot be completed before the completion of $\mathbf{T}'$ since $\mathbf{T}$ depends on a consumer of $\mathbf{T}'$. The actual re-activation takes place as described in the previous paragraph.

## 5.3 Miscellaneous technical issues

As mentioned, the choice point stack of the CHAT abstract machine is like that of the WAM. Suspended consumers are taken off the stack and their execution environments are preserved as described in Section 2.2. Some consumers of tables which are pruned might thus not be on the stack but already saved in CHAT areas. The JET pruning operation therefore needs to attach these already saved execution environments of suspended consumers to the $\mathbf{CP}(\mathbf{T})$ area of their root tabled subgoal $\mathbf{T}$. CHAT maintains enough information to easily find these consumers.

Readers familiar with the internals of SLG-WAM and of CHAT might wonder why we have so far not described what happens to the completion stack. Since there is a one-to-one correspondence between generator choice points in the CP stack and completion stack frames, upon JET pruning the action is the expected one: the completion stack gets pruned so that its topmost frame corresponds to the topmost generator choice point which is not pruned (i.e., the oldest generator not younger than $\mathbf{B_0}$). Upon a subsequent table re-activation, whenever a generator choice point is put back on the CP stack, a corresponding completion stack frame is placed on the completion stack.

# 6. PROPERTIES AND DISCUSSION

In this section we discuss the properties of just enough tabling, examine the analyses required to support and take advantage of it, issues related to space reclamation, and show how we can employ the JET implementation mechanism for purposes other than pruning.

## 6.1 Evaluation properties

The key feature of just enough tabling is that we perform each derivation step in the part of the derivation forest that is explored at most once. This of course is a property which also holds for proper tabling systems, *but* they are not able to support pruning without resorting to recomputation when tables are thrown away. This means they may be forced to do more computation than a non-tabled system. This is not the case for JET, because each tree, and in fact even each node, in the derivation forest is created at most once.

PROPOSITION 1. *JET is a proper tabling strategy which implements pruning.*

SLG resolution with local scheduling is a proper tabling strategy since it never re-executes any derivation steps, but its laziness in returning answers to the generator makes it inappropriate for pruning and hence local evaluation can perform arbitrarily more computation than a computation with pruning. SLG resolution with batched scheduling is a proper tabling strategy where pruning makes sense, but in its current implementations pruning throws away computation, and the result is an improper tabling strategy. Other tabling strategies like SLDT and DRA are not proper tabling strategies and can require a significant amount of recomputation even without pruning.

## 6.2 Automatic discovery of JET points

Although JET pruning can be user-controlled using a `once/1`-like predicate, we have so far presented the just enough tabling idea and its implementation decoupled from the issue of having a source-level construct for it. This is a conscious design decision. Indeed, we envision a system where just enough tabling is fully guided by a separate static analysis to determine the JET points in the program and annotate the intermediate code with the corresponding built-in that performs JET pruning. Note that such an analysis is straightforward in a language like Mercury [12] in which the compiler has precise mode and determinism information. Given such information, the JET points are simply derived from the single solution contexts.

A single solution context is a possibly non-deterministic subgoal $(A_i, \ldots, A_j)$ which occurs in a rule

$$p(\bar{s}) \texttt{ :- } A_1, \ldots, A_{i-1}, A_i, \ldots, A_j, A_{j+1}, \ldots A_n.$$

when for the subgoal $(A_i, \ldots, A_j)$ none of the outputs of this goal are used later in $A_{j+1}, \ldots, A_n$. We can replace this by:

$$\begin{array}{l} p(\bar{s}) \texttt{ :- } A_1, \ldots, A_{i-1}, c_{ij}(\bar{i}), A_{j+1}, \ldots A_n. \\ c_{ij}(\bar{i}) \texttt{ :- } A_i, \ldots, A_j \text{ ❶}. \end{array}$$

where $\bar{i}$ are the input arguments for $A_i, \ldots, A_j$ and ❶ represents a pruning point for the choice points (and possible tables) created by calls in the body of $c_{ij}(\bar{i})$. The two programs will produce the same answers in the same order.

We can move this pruning point earlier by noticing that if the goal $(A_{k+1}, \ldots, A_j)$ is guaranteed to succeed then we can effectively treat $(A_i, \ldots, A_k)$ as the single solution context. Hence we can replace the definition of $c_{ij}(\bar{i})$ by

$$c_{ij}(\bar{i}) \texttt{ :- } A_i, \ldots, A_k \text{ ❶}.$$

since we do not need to execute the goals $A_{k+1}, \ldots, A_j$ (they succeed and their outputs are not used).

The Mercury compiler already determines the earliest possible places to automatically insert such commit points (and removes useless calls). As mentioned in the introduction, our work on JET was partly motivated by the necessity to support pruning of tabled subcomputations in Mercury's single solution contexts.

Adopting a full-fledged version of Mercury's analysis for determining single solution contexts to a language like Prolog where mode and determinism information is not present is an interesting avenue for future work. However, note that even a rather simple analysis can be quite effective. For our example program of Figure 1, JET points can be discovered as follows: ❶ is a JET point since `start(S)` is a deterministic call (its argument is output and `start/1` only has one clause), while `t(S)` is a ground tabled call (and can only succeed once). The compiler can push the JET point from the code for `t/1` to immediately after the call site in this case. In our example, this is not done for `test/0` which explains the placement of JET point ❷.

## 6.3 Space reclamation

First of all, freezing the heap after JET pruning is not as bad as it sounds because even frozen heap can be reclaimed by garbage collection.

Regarding table space, a tabling system must in general keep all tables for the entire computation if it wants to avoid recomputation. With just enough tabling and batched scheduling in particular, where tables might remain incomplete for quite a while, large parts of the execution stacks may be frozen or copied into the CHAT areas without any demand for them. This increases the desirability to *garbage collect table space* too.

Given a fixed goal and logic program it is straightforward to determine for each program point the "forward reachable" predicates, that is the predicates which can possibly be called in forward execution from any point. Unfortunately predicates which are not forward reachable from a program point, may still be reached by backtracking. In that sense, *a priori* any predicate reachable from the original goal is always reachable. With determinism information we can improve upon this situation. At any program point which is guaranteed to have no choice points when reached (where all the previous computation is semi-deterministic, or committed) only forward reachable predicates are reachable (since backtracking is impossible). We can use this information to reclaim space for tables that are not forward reachable (whether complete or on hold by JET pruning).

Consider the program in Figure 1. Clearly the `start(S)` call is deterministic. When execution reaches point ❶ and performs a JET pruning action there are no current choice points, hence we cannot backtrack into `t(a)`. Thus only tables for the forward reachable tabled predicate `tle/2` need to be stored. We can immediately delete the tables for `t(a)`, `tc(a,Ya)`, and `tc(b,Yb)`. In our example program, we could incorporate this information into the builtin that implements JET pruning, so that we do not split the choice point stacks or perform copying of information which is then immediately discarded. Similarly at point ❷ we can discard the tables for predicate `tle/2`.

## 6.4 Applications

We now describe how we can go berserk[5] with such a scheme. Once we have the ability to effectively suspend a derivation tree we allow many scheduling strategies to be defined. For example, given this capability we can claim that tabled evaluation is never worse than non-tabled evaluation given the right scheduling strategy.

PROPOSITION 2. *Given a program $P$ with tabled predicates $Q$, and a scheduling strategy $S$, then for any goal $G$ there is a scheduling strategy $S'$ treating predicates $Q \cup Q'$ as tabled such that the frontier $S'(G)$ numbers no more nodes than $S(G)$.*

PROOF. (Sketch) We can devise a scheduling strategy $S'$ that mimics the scheduling strategy $S$ on goal $G$ by scheduling consumers for predicates $p \in Q'$ in the order scheduled by Prolog. In order to do so we must be able to move the generator until later in the computation from where it was initially started. We can do this by uninstalling the generator from the current stacks and reinstalling it using the mechanisms for JET pruning. □

Now while this proposition does not allow us to construct the scheduling strategy $S'$, the important point is that with the just enough tabling mechanism the scheduling of SLG operations is controllable at a very fine grain and is in fact so flexible that we can simulate Prolog's evaluation strategy. Without this capability, it is not possible to have a theoretical best case no worse than SLD resolution.

A simple example of the above proposition is that we can implement a scheduling strategy that allows us to guarantee that the answers are obtained in the same order as Prolog computation (up until Prolog loops infinitely). This has important consequences for improving the programmer's understanding of the behaviour of the code. Consider the goal and program

```
?- t(A), t(B), s(A,B) ❸.

:- table t/1.
t(1).
t(2).
t(3) :- s(2,1).

s(1,2).
s(2,1) :- s(2,1).
```

Without tabling `t/1`, the goal succeeds with `A = 1, B = 2` (and then loops). With tabling and either local or batched scheduling the goal runs forever. With local scheduling the call `t(A)` tries to generate all answers for the table, and when it reaches the third clause of `t/1` it runs forever. With batched scheduling the problem is more subtle. The call `t(A)` is the generator and returns its first answer `A = 1`, this is used by the consumer `t(B)` to give answer `B = 1`, but the call `s(1,1)` fails. The consumer `t(B)` suspends and we backtrack into the generator and find the next answer `A = 2`, this creates a new consumer `t(B)` which consumes the first answer, creating `B = 1` and the call to `s(2,1)` which loops forever.

Using the JET mechanism of separating the generator from its associated consumer, we can ensure the order of answers for the goal arrive in the same order as Prolog. The execution can (using appropriate scheduling) run as follows. The call `t(A)` creates a generator `t(C)` which finds the first answer `C = 1`. This is returned to the consumer `t(A)` which then calls consumer `t(B)`. This uses the first answer to lead to call `s(1,1)`. The `t(B)` consumer now suspends and we backtrack to the generator `t(C)` which generates the new answer `C = 2`. Since `t(C)` is separated from the consumer `t(A)`, we can schedule the suspended consumer `t(B)` first to get the new answer. This calls `s(1,2)` and succeeds, the JET point ❸ is reached where the table `t(C)` is put on hold.

# 7. RELATED WORK

The approach of Castro and Warren to approximate pruning of tabled logic programs [2] is closer in spirit to that of Prolog where a pruning operator forgets computations and their alternatives which lie in its scope. Hence, it is not surprising that the pruning mechanism of [2] does not cater for re-activation of tables after pruning. Instead, the tabling engine aims to find as many as possible tables that are no longer "currently in demand" once a pruning point is reached, and simply delete those tables. The calculation of demand is approximate: when the engine is able to safely determine that no tables in the pruning scope are demanded it deletes them all, otherwise they are all kept, possibly causing significantly extra computation.

Applying this approach to the example in Section 4 will execute as shown up to point ❶. Then, because the algorithm of [2] will determine no demand on any table, the tables will all be deleted. The subsequent call to `tle(a,G)` will cause the derivation tree to be recomputed (repeating the long computation). Note that support for approximate pruning is currently *not* available by default in XSB.

Technically, the JET mechanism and the approximate pruning approach of [2] are very different. However, the differences of the two approaches are also philosophical: we hold that just enough

[5] *berserk*: An ancient Scandinavian warrior frenzied in battle and held to be invulnerable (from Webster dictionary).

| Call | Answers | Status | Alternatives |
|---|---|---|---|
| `test` | ∅ | ✈ | |
| `t(a)` | {`t(a)`} | ✔ | |
| `tc(a,Ya)` | {`tc(a,c)`} | ✂ | [(3),(4)] |
| `tle(a,Za)` | {`tle(a,a)`, `tle(a,b)`} | ✂ | [(5)] |
| `tc(b,Yb)` | {`tc(b,c)`} | ✂ | [(4)] |
| `tle(b,Zb)` | ∅ | ✔ | |

**Figure 9: Tables in DRA when reaching the first cut.**

tabling is closer to the spirit of memoization (remembering prior computations) and that the issue of space reclamation of tables, although an important one, is orthogonal to pruning unexplored alternatives. As such, pruning points should not be used as program places to perform reclamation of the table space in a rather ad hoc way.

Guo and Gupta describe a scheme for implementing cut-based pruning in their tabling system [7] which uses dynamic reordering of alternatives (DRA). Their tabling engine stores and dynamically reorders "looping alternatives" which are the clauses of the original predicate that might still generate new answers, and should be explored after clauses that can be evaluated using Prolog-style resolution. Because the tabling abstract machine for DRA is much more WAM-like than the SLG-WAM and DRA uses a fixed Prolog-like scheduling strategy, their claim is that DRA can implement cuts much like in the WAM. By keeping the "looping alternatives" in the table when a cut occurs they can still continue the execution at some later point. Of course the "looping alternatives" are a rather imprecise measure of the rest of the computation and since the reordering is dynamic, the operational semantics of cuts in DRA can be extremely confusing to a Prolog programmer.

If we replaced the JET points in the program of Figure 1 by (equivalent in this case) cuts as follows

```
test :- start(S), t(S), !, tle(s,G), good(G), !.
```

then the DRA approach would compute more or less equivalently to JET up to point where the first cut occurs, and then rather than keeping the saved CP stacks as shown in Figure 5, it could keep the list of delayed alternatives (shown as lists of clause numbers from Figure 1) for each incomplete table as shown in Figure 9. Unfortunately the alternatives are a very crude measure of unfinished computation, in particular only the first clause for the call `tc(b,Yb)` has been eliminated, all remaining clauses are possible for incomplete tables. When we call `tle(a,G)` the derivation tree will need to be recomputed (including the long computation). Note that, when compared to the approach of Castro and Warren [2], the approach of Guo and Gupta would save the recomputation of `tle(b,Zb)` if `tc(b,Yb)` was ever called again.

Another claim of [7] is that because the DRA implementation is deterministic in its scheduling strategy, the effect of a cut over tables is well-defined. It is indeed well-defined, but its operational semantics significantly departs from the semantics of cut in Prolog and requires a quite deep understanding the DRA technique from the programmer, in particular in the presence of mutually recursive calls. It is for example quite possible that in a program like:

```
:- table t/2.
t(A,B) :- p(A,C), !, ...
t(A,B) :- ...

p(A,B) :- t(B,C), ...
```

the `!/0` cuts the second clause for one call to `t/2` (when the `t(B,C)` call is not a looping alternative) but not for other calls (when the call is a looping alternative and the clause is dynamically reordered). Of course, the root of the problem is not DRA, but in that the operational semantics of the Prolog cut are not compatible with a search strategy like tabling's which is not depth-first search. We believe that *programmer-controlled Prolog-style cuts in tabled logic programs should not be allowed* just for this reason.

## 8. CONCLUSION

We developed the just enough tabling mechanism so as to support analysis-guided pruning in tabled logic programs and thereby make SLG resolution *more demand driven* and at the same time maintain its property of avoiding recomputation. Although this paper has described the JET mechanism through a prism of pruning, we would like to stress that JET is not just a fancy pruning mechanism. Instead, it offers a tabling system the capability to arbitrarily suspend the construction of some tables during tabled execution and re-activate (perhaps only a subset of) them at some later point *without any re-execution* except for the cost of reinstalling trailed bindings. It is exactly this capability that allows us to treat single solution contexts without either *throwing away incomplete tables* or *unnecessarily having to complete tables* before returning an answer to such a context. It is not difficult to give examples (such as that in the introduction) where just enough tabling prevents an arbitrary amount of unnecessary computations. Moreover, the applications of the just enough tabling mechanism may well extend well beyond pruning.

## 9. ACKNOWLEDGMENTS

The research of the first author has been supported in part by grant # 221-2000-165 from Vetenskapsrådet (the Swedish Research Council). We would like to thank Zoltan Somogyi for helpful discussions on tabling and pruning in Mercury, and our anonymous reviewers whose comments have helped improve the presentation of this work.

## 10. REFERENCES


[1] L. F. Castro, T. Swift, and D. S. Warren. Suspending and resuming computations in engines for SLG evaluation. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium*, number 2257 in LNCS, pages 332–350. Springer, Jan. 2002.

[2] L. F. Castro and D. S. Warren. Approximate pruning in tabled logic programming. In P. Degano, editor, *Programming Languages and Systems: Proceedings of the European Symposium on Programming*, number 2618 in LNCS, pages 69–83. Springer, Apr. 2003.

[3] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, Jan. 1996.

[4] B. Demoen and K. Sagonas. CHAT is Θ(SLG-WAM). In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *LPAR'99: Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, number 1705 in LNAI, pages 337–357. Springer, Sept. 1999.

[5] B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, May 2000.

[6] H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer, Nov./Dec. 2001.

[7] H.-F. Guo and G. Gupta. Cuts in tabled logic programming. In B. Demoen, editor, *Proceedings of CICLOPS'2002, the Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 62–73, July 2002.

[8] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *J. of Logic Program.*, 38(1):31–54, Jan. 1999.

[9] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A tabling engine designed to support parallelism. In *Proceedings of Tabulation in Parsing and Deduction (TAPD)*, pages 77–87, Sept. 2000.

[10] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Prog. Lang. Syst.*, 20(3):586–634, May 1998.

[11] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, May 1994.

[12] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. of Logic Program.*, 26(1–3):17–64, Oct./Dec. 1996.

[13] H. Tamaki and T. Sato. OLD resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98. Springer-Verlag, July 1986.

[14] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[15] N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Implementation of a linear tabling mechanism. *J. of Functional and Logic Program.*, 2001(10), 2001.