Tabulação em Programação Lógica

Ricardo Rocha

DCC-FCUP, Universidade do Porto
ricroc@dcc.fc.up.pt

Resolução SLD

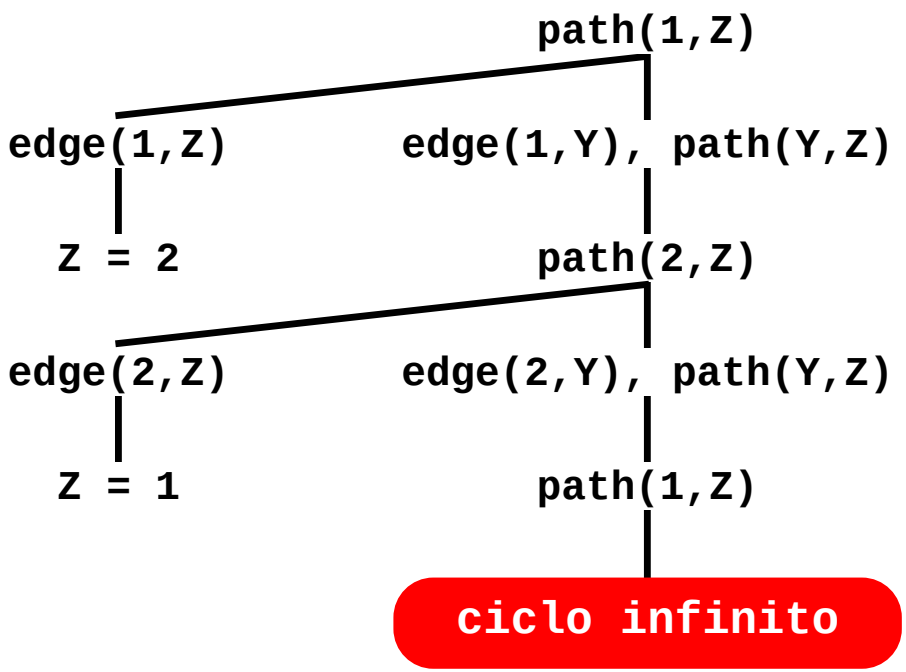
Limitações

- Apesar do poder, da flexibilidade e dos bons resultados que o Prolog tem demonstrado desde o aparecimento da WAM, a estratégia de resolução SLD na qual o Prolog se baseia é limitadora do potencial inerente ao paradigma da Programação Lógica.
- A resolução SLD não consegue tratar devidamente:
  - **Ciclos positivos infinitos** (expressividade insuficiente)
  - **Ciclos negativos infinitos** (inconsistência)
  - **Computações redundantes** (ineficiência)

Resolução SLD

Ciclos Infinitos

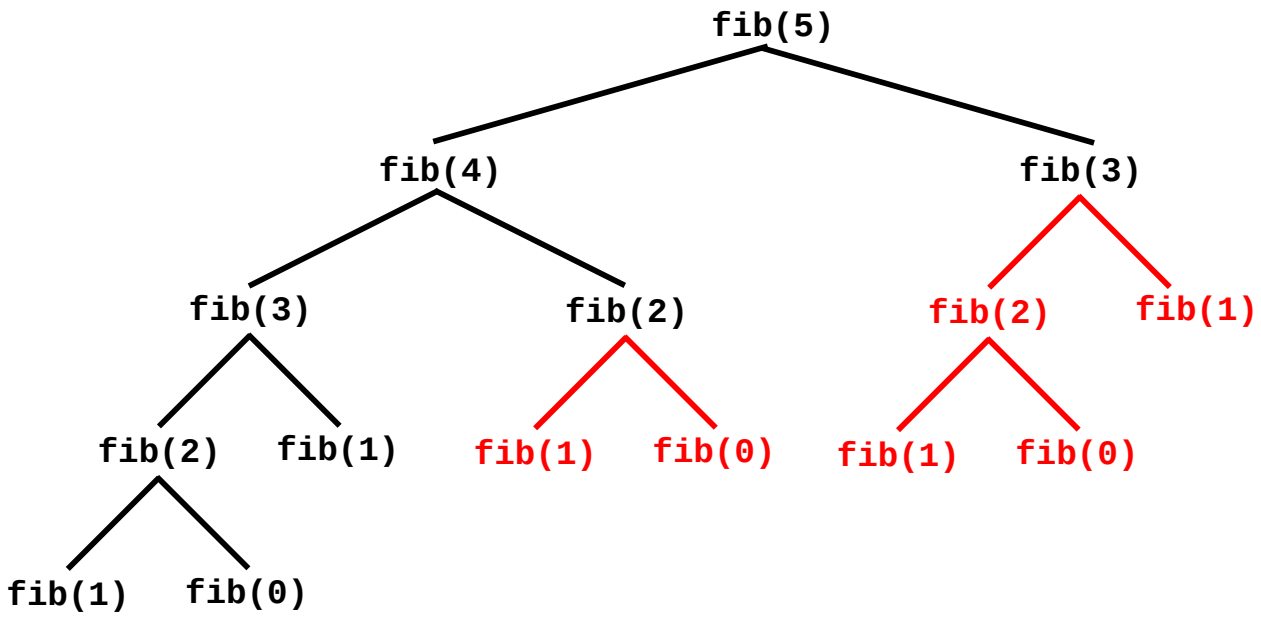
```
path(X,Z) :- edge(X,Z)
path(X,Z) :- edge(X,Y), path(Y,Z).

edge(1,2).
edge(2,1).
```



Resolução SLD

Computações Redundantes

```
fib(0,0).
fib(1,1).
fib(N,Z) :- N > 1,
            P is N - 1,
            Q is N - 2,
            fib(P,X),
            fib(Q,Y),
            Z is X + Y.
```



Resolução SLD

É necessária uma estratégia de resolução diferente!

Tabulação

- Uma das mais bem sucedidas técnicas para solucionar a incapacidade da resolução SLD no que respeita a ciclos infinitos e computações redundantes é a **Tabulação**.
- A ideia básica da tabulação consiste em ir guardando numa área auxiliar, a **tabela**, as soluções intermédias encontradas para determinados objetivos, para que estas possam ser reutilizadas assim que surgirem chamadas repetidas a esses mesmos objetivos.

Tabulação

Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objetivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objetivo, e a execução prossegue aplicando resolução SLD.
- Sempre que se obtém uma **nova solução** para um objetivo tabelado, esta é guardada na tabela.
- Chamadas repetidas de objetivos tabelados são resolvidas **consumindo** as soluções previamente encontradas e guardadas na tabela.

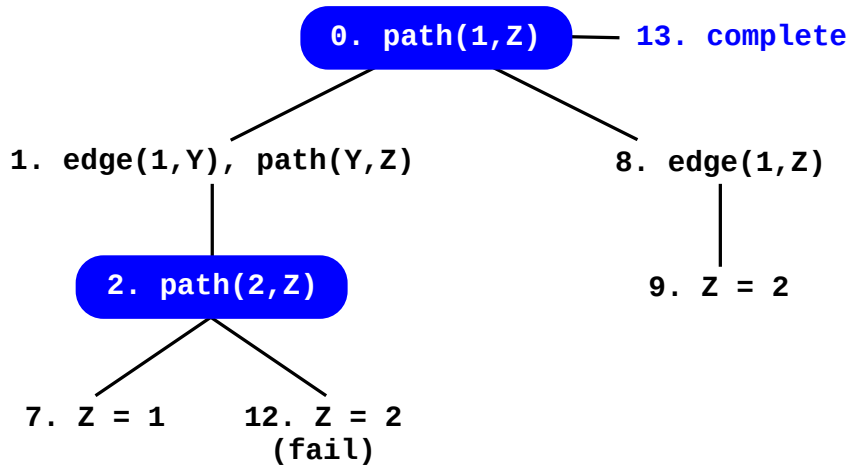
Tabulação

Modelo Básico de Execução

- Assim que todas as soluções forem consumidas, o estado da computação é **suspenso** e a execução falha. Entretanto, se durante a exploração do objetivo em causa surgirem novas soluções, então a computação suspensa é **retomada** para consumir as novas soluções. Caso contrário, é considerada **completa**.
- A computação **termina** assim que todas as soluções tiverem sido consumidas e não for possível continuar a execução por aplicação da resolução SLD.

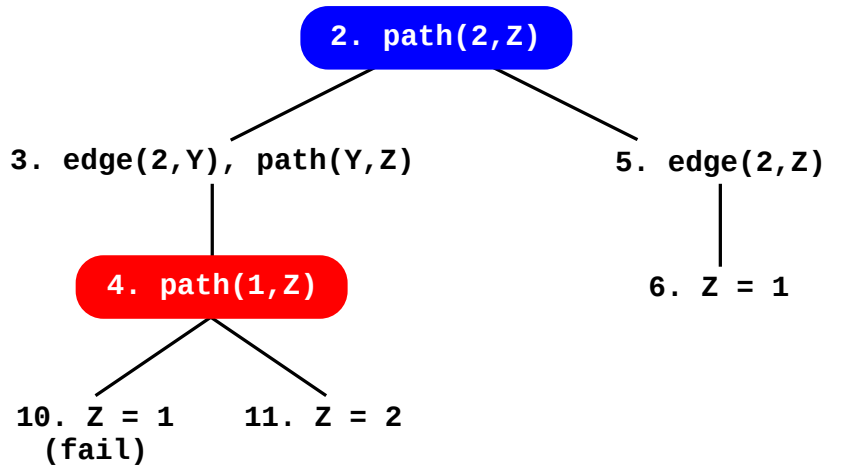
Tabulação

Ciclos Infinitos



Programa

```
:- table path/2.

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

edge(1,2).
edge(2,1).
```



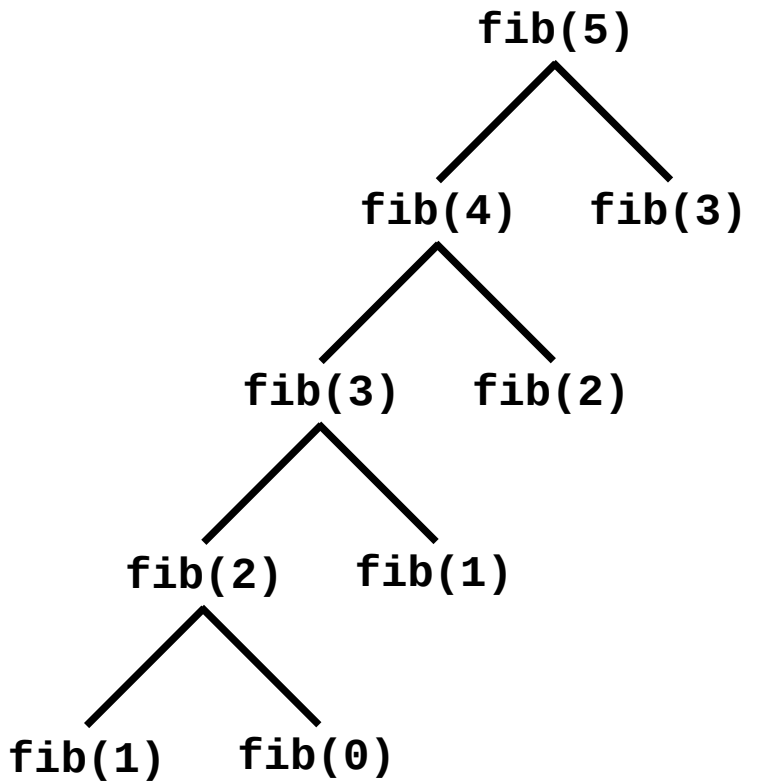
Tabela

Objetivo	Soluções
0. path(1,Z)	7. Z = 1 9. Z = 2 13. complete
2. path(2,Z)	6. Z = 1 11. Z = 2 13. complete

Tabulação

Computações Redundantes

```
:- table fib/2.

fib(0,0).
fib(1,1).
fib(N,Z) :- N > 1,
            P is N - 1,
            Q is N - 2,
            fib(P,X),
            fib(Q,Y),
            Z is X + Y.
```



Tabulação

Completude e Líderes

- Classificação dos nós da árvore de procura:
  - **Interiores:** representam os objetivos não tabelados.
  - **Geradores:** representam as primeiras chamadas a objetivos tabelados.
  - **Consumidores:** representam as chamadas repetidas a objetivos tabelados.
- Um objetivo tabelado está **completamente avaliado** quando:
  - Não podem ser geradas mais soluções por resolução SLD.
  - Todas as chamadas ao objetivo consumiram todas as soluções.
- Um conjunto de objetivos pode ser mutuamente dependente, **strongly connected component (SCC)**, e nesse caso só pode ser completo em simultâneo.
- Um SCC está completamente avaliado quando cada objetivo do SCC está completamente avaliado. A completude de um SCC deve ser realizada pelo **líder** do SCC, isto é, o gerador associado ao objetivo mais antigo do SCC.

Tabulação

Principais Estratégias de Escalonamento

- Em vários pontos da computação é possível escolher entre continuar a execução (forward execution), retroceder para nós interiores, consumir soluções em nós consumidores ou completar nós geradores.
- A decisão a tomar é definida pela **estratégia de escalonamento**. A escolha da estratégia de escalonamento pode dar origem a diferentes ordenações da sequência de soluções e pode ter um impacto significativo no desempenho.
- As duas estratégias de escalonamento mais conhecidas são:
  - **Batched Scheduling**
  - **Local Scheduling**

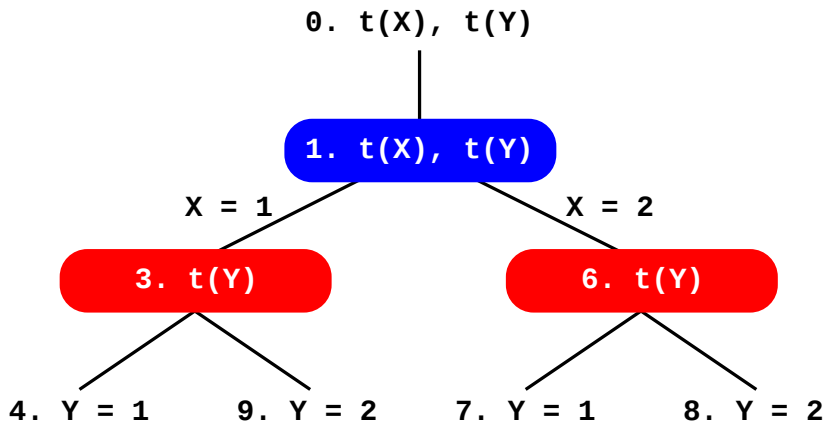
Tabulação

Batched Scheduling

- A estratégia de batched scheduling tem como principal objetivo **reduzir a movimentação da computação pela árvore de procura**. Esta estratégia favorece forward execution primeiro, retroceder para nós interiores depois, e consumir soluções ou completar por último.
- A execução é muito semelhante à da WAM, quando surgem novas soluções, estas são guardadas na tabela e a **execução continua** como na WAM. A computação é retomada nos nós consumidores (para estes consumirem as novas soluções) apenas quando todas as alternativas do SCC tiverem sido exploradas.
- Em certas situações, isto leva a que **surjam mais dependências entre objetivos**, o que aumenta o tamanho do SCC corrente e consequentemente atrasa o completar desses objetivos.

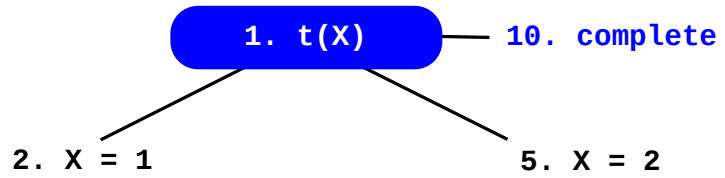
Tabulação

Batched Scheduling



Programa

```
:- table t/1.

t(1).
t(2).
```



Tabela

Objetivo	Soluções
1. t(X)	2. X = 1 5. X = 2 10. complete

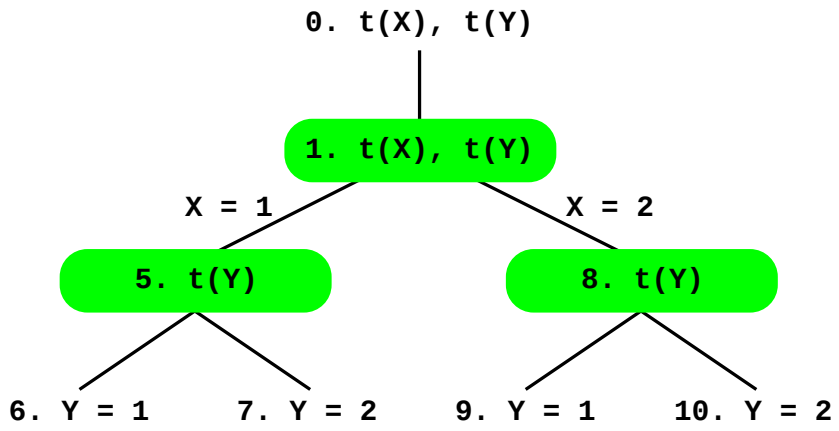
Tabulação

Local Scheduling

- A estratégia de local scheduling tem como principal objetivo **completar SCCs o mais cedo possível**. Esta estratégia favorece retroceder para nós interiores e completar primeiro, consumir soluções depois, e forward execution por último.
- Quando surgem novas soluções, estas são guardadas na tabela e a **execução falha**. As soluções só são propagadas para fora do ambiente do SCC quando este estiver completo.
- Como esta estratégia completa SCCs mais cedo, em princípio, são **esperadas menos dependências** entre os objetivos e consequentemente SCCs menores.

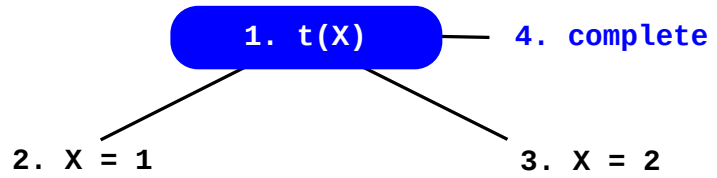
Tabulação

Local Scheduling



Programa

```
:- table t/1.

t(1).
t(2).
```



Tabela

Objetivo	Soluções
1. t(X)	2. X = 1 3. X = 2 4. complete

Tabulação

Batched x Local Scheduling

- **Principais Diferenças**
  - Em batched, quando surgem novas soluções, a execução continua. Em local, a execução falha.
  - Em batched, quando um SCC é completo, a execução falha. Em local, o líder do SCC age como um nó consumidor e começa a consumir as soluções encontradas.
- **Vantagens e Desvantagens**
  - Em batched, quando surgem novas soluções, as instanciações das variáveis são automaticamente propagadas para o ambiente que fez a chamada. No entanto, como já foi referido, este comportamento pode levar a um maior número de dependências entre os objetivos em execução.
  - Por outro lado, a estratégia de local parece mais atractiva porque aparentemente diminui as dependências entre objetivos. No entanto, isso nem sempre se verifica. Para além disso, como local não benificia da propagação automática da instanciação de variáveis, paga o custo extra de nos nós geradores consumir soluções a partir da tabela.

Tabulação

Principais Modelos de Execução

- **Tabulação por Suspensão da Computação**
  - A execução é vista como uma sequência de sub-computações que podem ser suspensas e mais tarde recuperadas até se atingir um ponto-fixo.
  - A suspensão das sub-computações é conseguida por congelamento das pilhas de execução, por cópia das pilhas de execução para áreas auxiliares, ou por combinação de ambas as técnicas.
- **Tabulação Linear**
  - A execução é vista como uma única computação onde os objetivos tabelados são recursivamente reavaliados até se atingir um ponto-fixo sem que para isso seja necessário suspender/recuperar sub-computações.
  - O ponto fraco destes modelos é a necessidade de utilizar re-computação para calcular pontos-fixos.

Tabulação

Sistemas Mais Conhecidos

- **Suspensão por Congelamento das Pilhas de Execução**
  - XSB Prolog (modelo SLG-WAM)
  - Yap Prolog (modelo SLG-WAM)
- **Suspensão por Cópia das Pilhas de Execução**
  - XSB Prolog (modelo CAT)
  - Mercury
- **Suspensão por Congelamento e Cópia das Pilhas de Execução**
  - XSB Prolog (modelo CHAT)
- **Tabulação Linear**
  - ALS-Prolog (modelo DRA)
  - B-Prolog (modelo SLDT)
  - Yap Prolog (modelos DRA + SLDT)

Yap Prolog

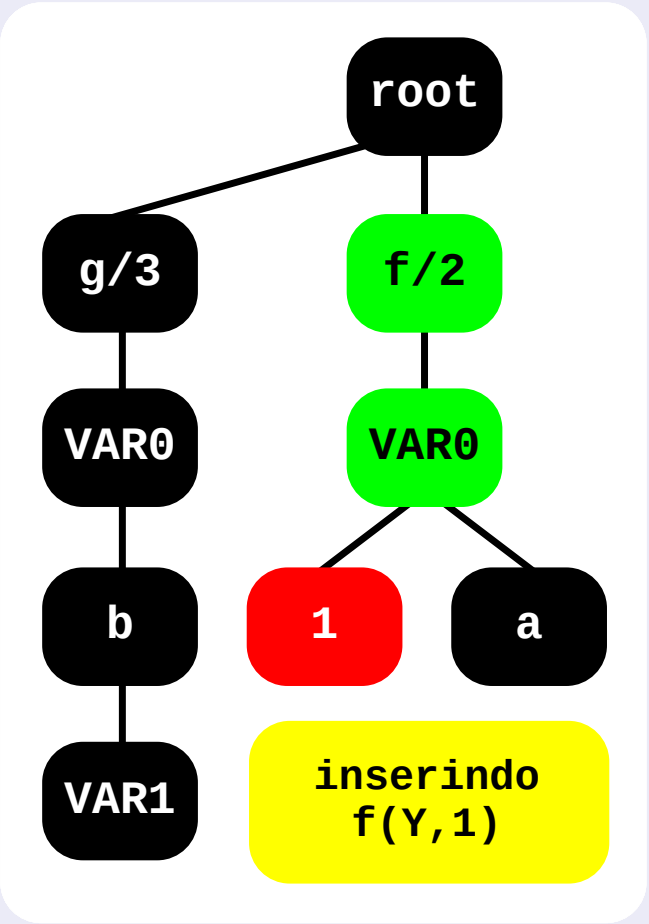
Extensões de Suporte

- **Tabela:** espaço para guardar os objetivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas Instruções:** instruções de suporte às 4 operações básicas.
  - **Tabled Subgoal Call:** verifica se um objetivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.
  - **New Answer:** verifica se uma nova solução já existe na tabela, e senão insere-a.
  - **Answer Resolution:** consume a próxima solução disponível, se alguma. Caso contrário, suspende a computação e calcula uma possível resolução para continuar a execução.
  - **Completion:** determina se um SCC está completamente avaliado, e senão calcula uma possível resolução para continuar a execução.

Yap Prolog

Representação da Tabela

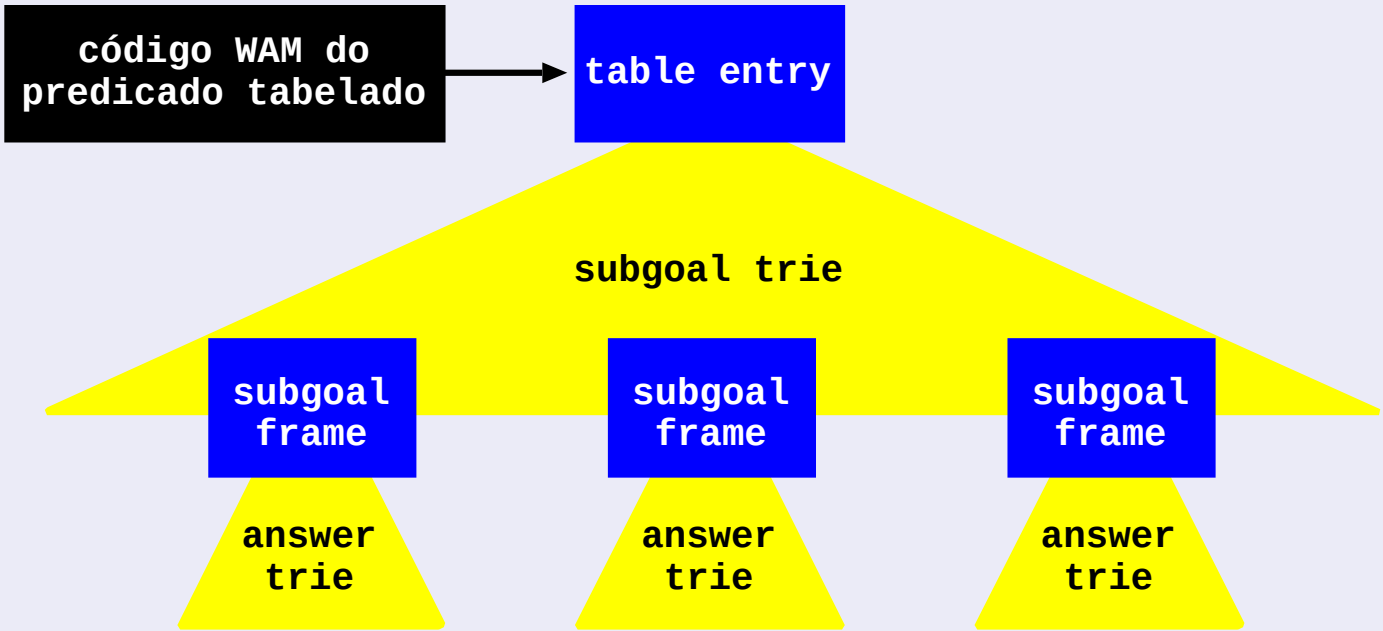
- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
  - Cada caminho através dos nós da trie corresponde a um único termo.
  - Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.



Yap Prolog

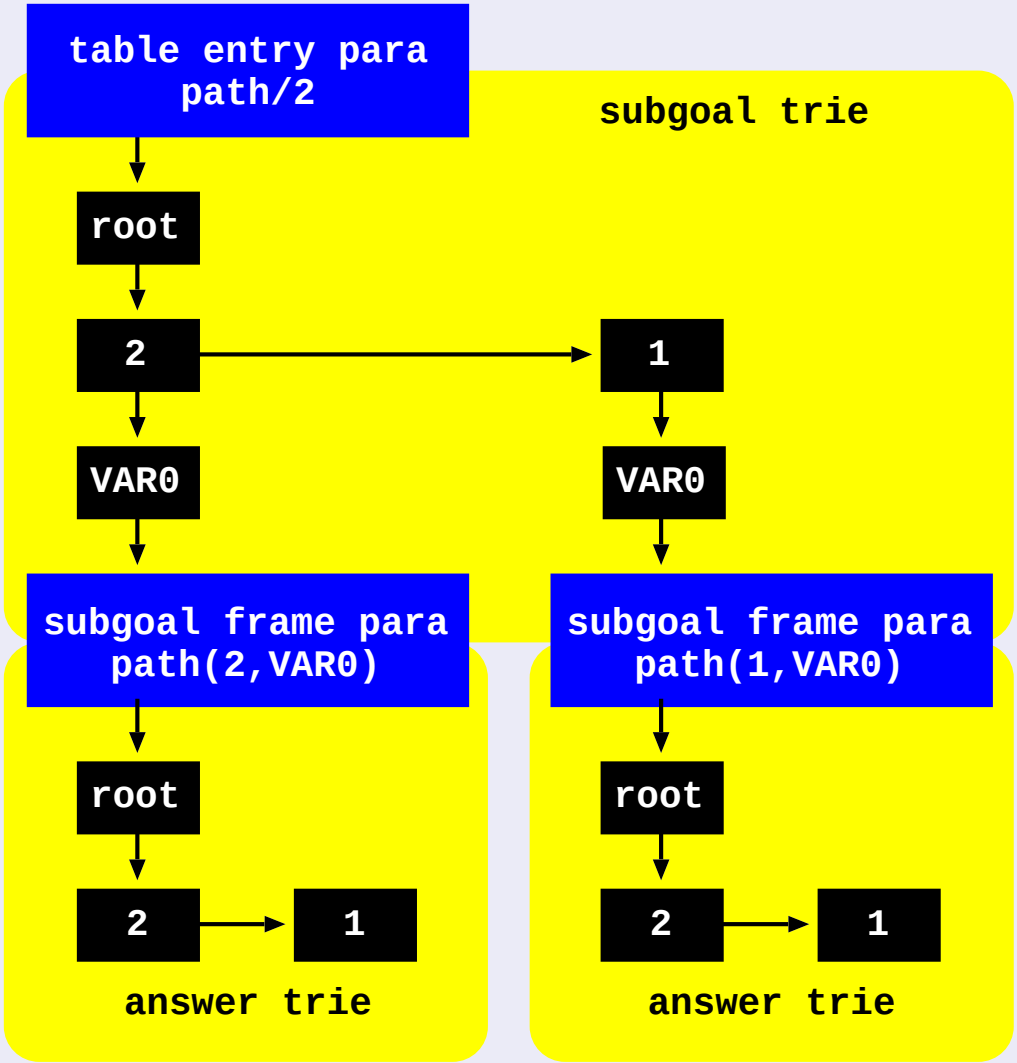
Organização da Tabela

- **Subgoal Trie**
  - Guarda os objetivos tabelados. Cada **table entry** representa o ponto de entrada dos objetivos de um predicado.
- **Answer Trie**
  - Guarda as soluções dos objetivos tabelados. Cada **subgoal frame** representa o ponto de entrada das soluções de um objetivo.



Yap Prolog

Organização da Tabela



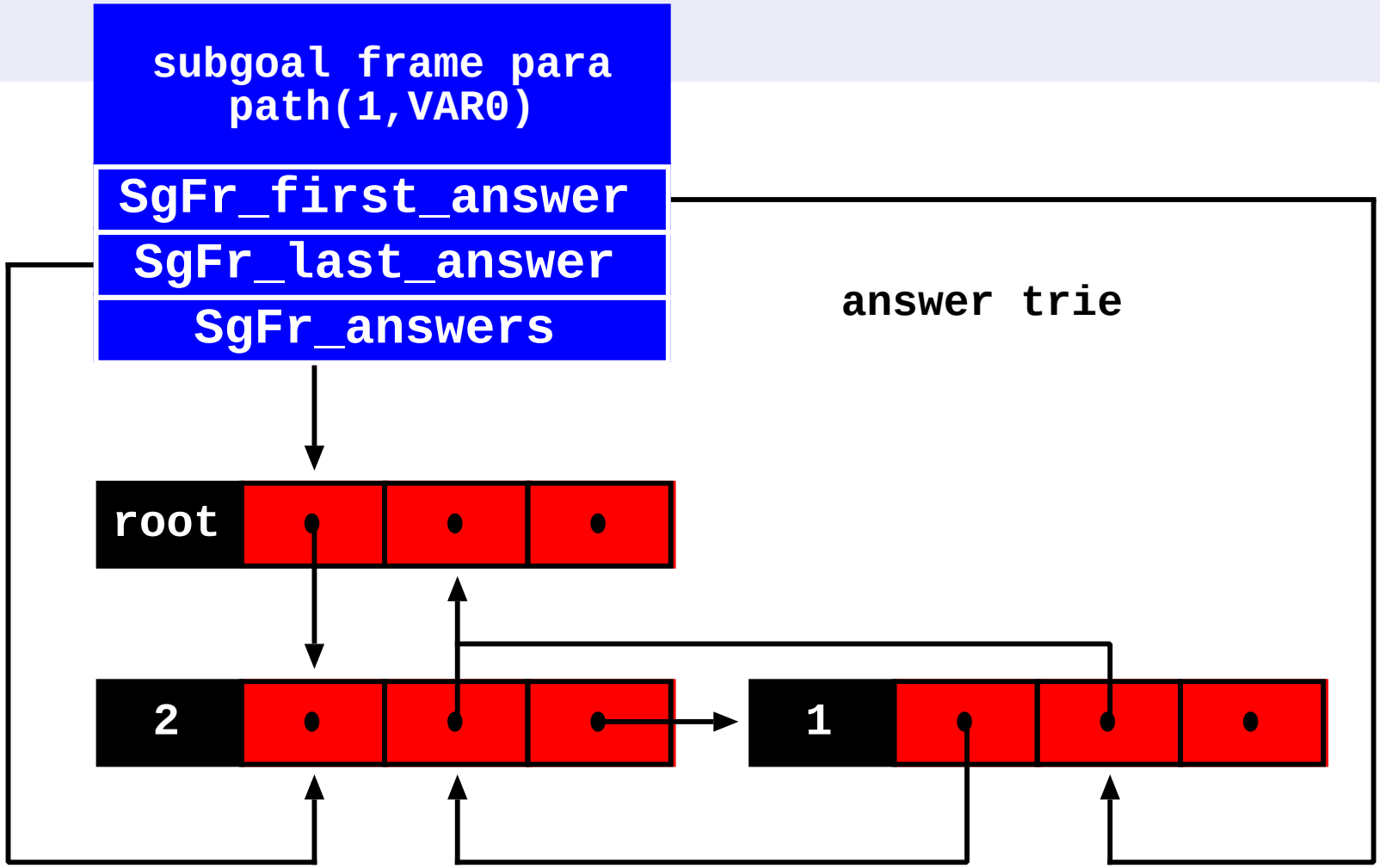
Yap Prolog

Organização da Tabela

- Cada nó da trie é constituído por 4 campos:
  - **TrNode_symbol** guarda o símbolo do nó.
  - **TrNode_child** aponta para o primeiro nó filho. Nos nós folha da answer trie este campo é utilizado para manter a lista ordenada das soluções encontradas.
  - **TrNode_parent** aponta para o nó pai.
  - **TrNode_sibling** aponta para o próximo nó irmão.
- Cada subgoal frame guarda os seguintes apontadores para a answer trie:
  - **SgFr_answers** aponta para o nó raiz.
  - **SgFr_first_answer** aponta para o nó folha da primeira solução.
  - **SgFr_last_answer** aponta para o nó folha da última solução.

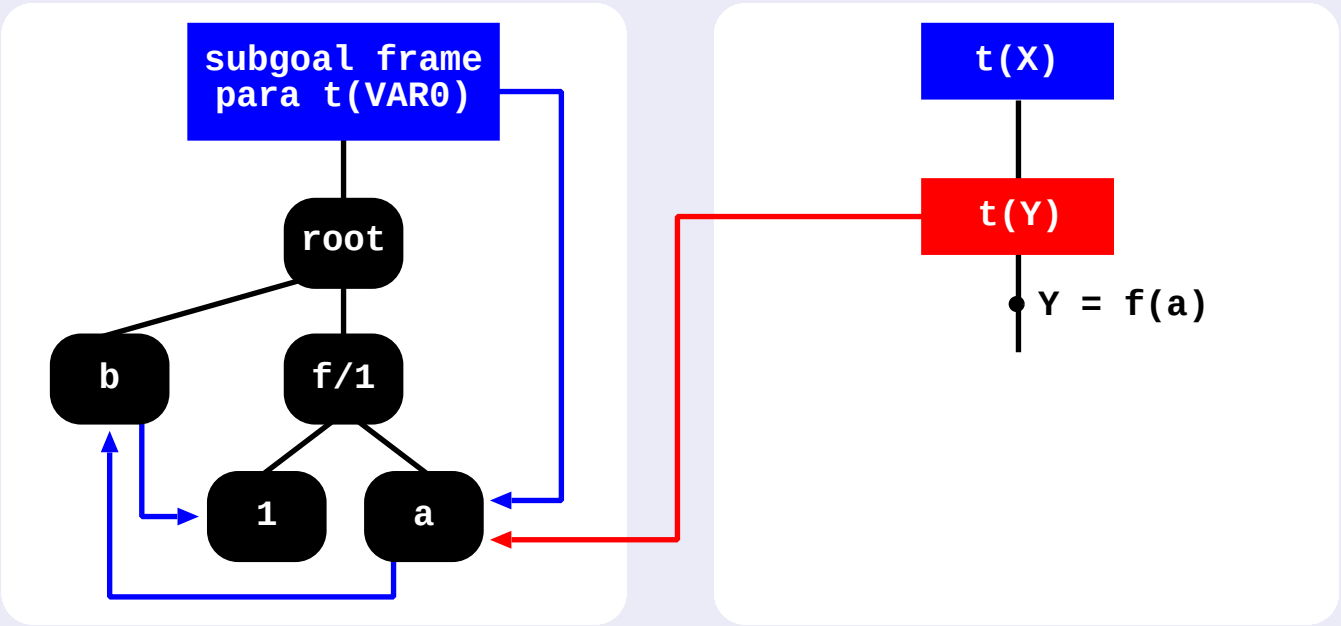
Yap Prolog

Organização da Tabela



Organização da Tabela

- **Answer Trie**
  - Para cada solução, guarda apenas as substituições das variáveis existentes na chamada do objetivo (**substitution factoring**).
  - As soluções são mantidas numa lista pela ordem que foram encontradas. Os nós consumidores guardam a referência da última solução consumida, e consumem mais soluções seguindo a lista.



Yap Prolog

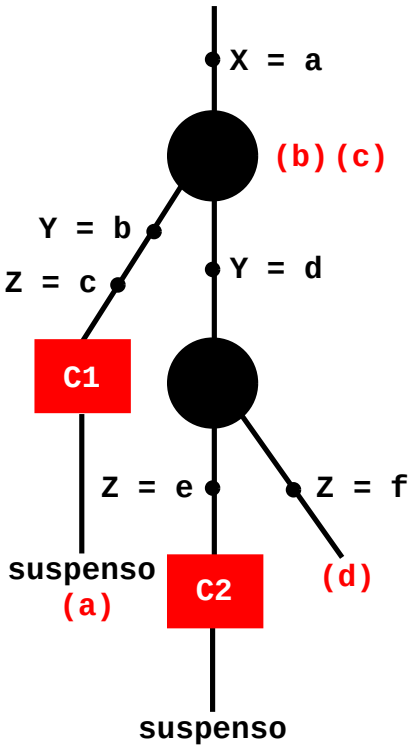
Suspensão/Recuperação

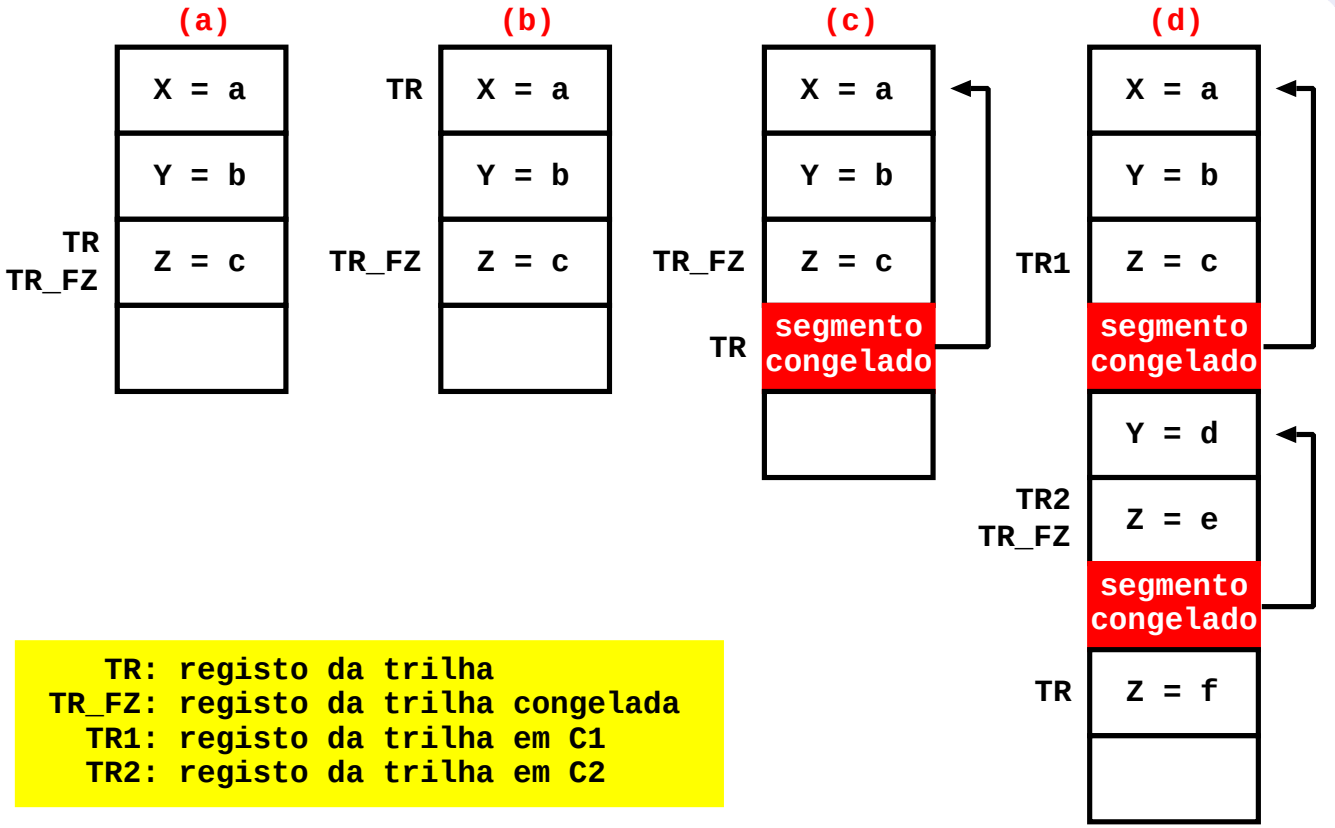
- No Yap, o ambiente de uma computação suspensa é preservado através do **congelamento das pilhas de execução**. Um conjunto de **freeze registers**, um por pilha de execução, marcam o ponto até onde as pilhas se encontram congeladas:
  - **B_FZ:** é o freeze register da STACK.
  - **H_FZ:** é o freeze register da HEAP.
  - **TR_FZ:** é o freeze register da TRAIL.
- Os freeze registers são atualizados em duas situações:
  - **Quando uma computação é suspensa**. Esta operação pode ser simplificada se atualizarmos os freeze registers quando se aloca um novo nó consumidor (um nó consumidor pode ser suspenso várias vezes mas só é alocado uma única vez).
  - **Quando um SCC é completo**. Neste caso, os freeze registers passam a referenciar as pilhas de execução associadas ao novo nó consumidor mais recente (fora do SCC entretanto completo).

Yap Prolog

Suspensão/Recuperação

- **Forward Trail:** extensão à trilha da WAM que permite restaurar as atribuições condicionais de uma computação suspensa.





Yap Prolog

Suspensão/Recuperação

```
restore_bindings(trail entry UNBIND, trail entry REBIND)
   unbind_entry = UNBIND
   common_entry = REBIND
   while (unbind_entry != common_entry)    // find common entry
      while (unbind_entry > common_entry)  // rewind loop
         ref = trail_addr(unbind_entry--)
         if (is_variable(ref))
            unbind_variable(ref)
         else if (is_frozen_segment(ref))
            unbind_entry = ref
      while (unbind_entry < common_entry)
         ref = trail_addr(common_entry--)
         if (is_frozen_segment(ref))
            common_entry = ref
   rebind_entry = REBIND
   while (rebind_entry != common_entry)  // rebind loop
      ref = trail_addr(rebind_entry)
      if (is_variable(ref))
         bind_variable(ref, trail_value(rebind_entry--))
      else if (is_frozen_segment(ref))
         rebind_entry = ref
```

Yap Prolog

Compilação de Predicados Tabelados

- Os predicados tabelados são compilados utilizando um conjunto de instruções semelhantes às instruções de indexação da WAM:
  - As instruções **table_try_me** e **table_try_single** (para predicados definidos por uma única cláusula) estendem a instrução **try_me** da WAM para implementar a operação de **Tabled Subgoal Call**.
  - As instruções **table_retry_me** e **table_trust_me** são semelhantes às instruções **retry_me** e **trust_me** da WAM só que operam sobre nós geradores.
- O código compilado de cada cláusula é ainda estendido para incluir no final uma nova instrução **new_answer** responsável por implementar a operação de **New Answer**.

Yap Prolog

Compilação de Predicados Tabelados

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

// código compilado do predicado t/1
t/1:  table_try_single
      ... // código WAM de 't(X) :- ...'
      new_answer
// código compilado do predicado t/2
t/2:  table_try_me t2_2
      ... // código WAM de 't(X,1) :- ...'
      new_answer
t2_2: table_retry_me t2_3
      ... // código WAM de 't(X,2) :- ...'
      new_answer
t2_3: table_trust_me
      ... // código WAM de 't(X,3) :- ...'
      new_answer
```

Tabled Subgoal Call

- Esta operação verifica se um objetivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.
- A informação do estado dos objetivos é guardada na subgoal frame respectiva no campo **SgFr_state**, que pode ter o valor **ready**, **evaluating** ou **complete**.

Yap Prolog

Tabled Subgoal Call

```
tabled_subgoal_call(subgoal SG)
    sg_fr = subgoal_trie_check_insert(SG)  // get subgoal frame for SG
    if (SgFr_state(sg_fr) == ready)
        gen_cp = store_generator_node(sg_fr)
        CP_BP(gen_cp) = failure_continuation_instruction()
        SgFr_state(sg_fr) = evaluating
        goto next_instruction()
    else if (SgFr_state(sg_fr) == evaluating)
        cons_cp = store_consumer_node(sg_fr)
        CP_BP(gen_cp) = answer_resolution
        goto answer_resolution(cons_cp)  // start consuming answers
    else if (SgFr_state(sg_fr) == complete)
        goto completed_table(sg_fr)
```

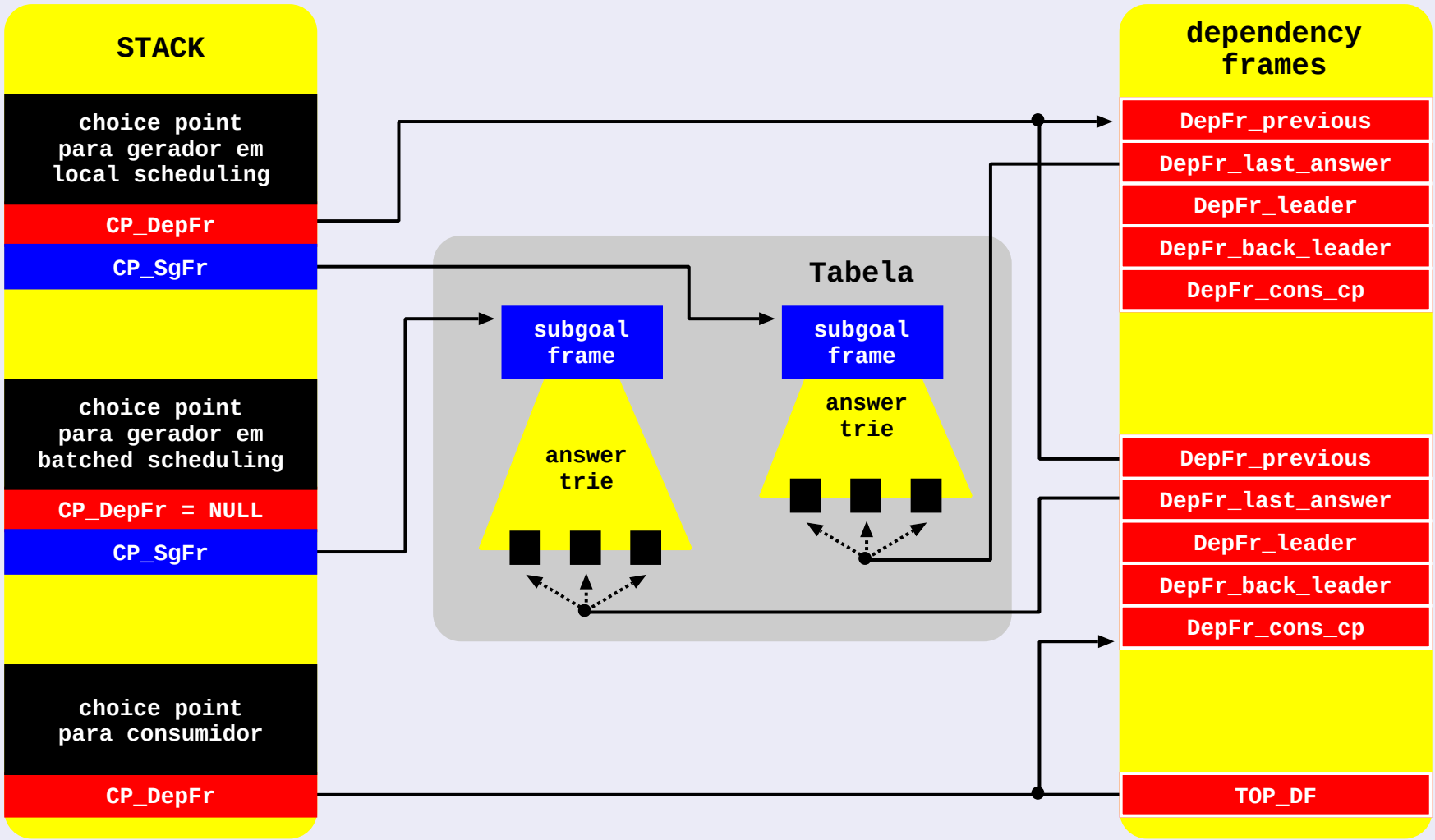
Yap Prolog

Pontos de Escolha

- No Yap, os nós geradores e os nós consumidores são implementados como pontos de escolha WAM estendidos com campos extra:
  - **CP_DepFr** aponta para uma estrutura de dados (**dependency frame**) onde se guarda informação relativa ao nó consumidor.
  - **CP_SgFr** aponta para a subgoal frame onde as soluções devem ser guardadas (apenas em nós geradores).
- Em local scheduling, quando um SCC é completo, o líder do SCC age como um nó consumidor e começa a consumir as soluções encontradas. No Yap, isto é conseguido através da colocação do campo **CP_DepFr** na mesma posição nos nós geradores e nos nós consumidores, o que permite que um nó gerador, em local scheduling, facilmente se transforme num nó consumidor.
- O campo **CP_DepFr** é ainda utilizado para distinguir se um nó gerador está a ser avaliado usando batched (casos em que é `NULL`) ou local scheduling.

Yap Prolog

Pontos de Escolha



Yap Prolog

Dependency Frames

- O registo global **TOP_DF** aponta para a dependency frame mais recente e o campo **DepFr_previous** permite seguir a lista ordenada de dependency frames.
- O campo **DepFr_last_answer** guarda a referência para o nó folha da última solução consumida.
- Os campos **DepFr_leader** e **DepFr_back_leader** são utilizados pelo algoritmo de detecção do ponto-fixo da computação.
- O campo **DepFr_cons_cp** aponta de volta para o choice point ao qual a dependency frame diz respeito.

Yap Prolog

New Answer

- Esta operação verifica se uma nova solução já existe na tabela, e senão insere-a. Caso contrário, a execução falha.

Yap Prolog

New Answer

```
new_answer(answer AW, generator node GN)
    sg_fr = CP_SgFr(GN)
    ans = answer_trie_check_insert(AW, sg_fr)  // get leaf node for AW
    if (isa_repeated_answer(ans))
        fail()
    // update subgoal frame pointers
    if (SgFr_first_answer(sg_fr) == NULL)
        SgFr_first_answer(sg_fr) = ans
    else
        TrNode_child(SgFr_last_answer(sg_fr)) = ans
    SgFr_last_answer(sg_fr) = ans
    // scheduling strategy
    if (CP_DepFr(GN) != NULL)  // local scheduling
        fail()
    else  // batched scheduling
        goto next_instruction()
```

Yap Prolog

Completion

- Esta operação determina se um SCC está completamente avaliado, e senão calcula uma possível resolução para continuar a execução.
- A operação de completion é executada sempre que a computação atinge um **nó gerador e este já explorou todas as suas alternativas**.

O nó gerador é líder?

- **Não** → Falha (backtracking).
- **Sim** → Existe algum nó consumidor CN no SCC com soluções por consumir?
  - **Sim** → Retoma a computação em CN.
  - **Não** → Completa o SCC corrente.

Yap Prolog

Answer Resolution

- Esta operação consume a próxima solução disponível, se alguma. Caso contrário, suspende a computação e calcula uma possível resolução para continuar a execução.
- A operação de answer resolution é executada sempre que a computação atinge um **nó consumidor**.

O nó consumidor tem soluções por consumir?

- **Sim** → Carrega a próxima solução e prossegue a execução.
- **Não** → Retoma a computação no mais novo dos seguintes nós:
  - Nós consumidores mais antigos com soluções por consumir.
  - Líder onde foi executada a última operação de completion sem sucesso.

Yap Prolog

Como Calcular o Líder

- **Ideia Chave:** para cada novo consumidor C, calcular antecipadamente o líder do SCC que inclui C.
  - Calcular o gerador G associado ao consumidor C.
  - Para todos os consumidores entre C e G, calcular o líder L mais antigo.
  - O mais antigo entre G e L é o líder do SCC que inclui C.

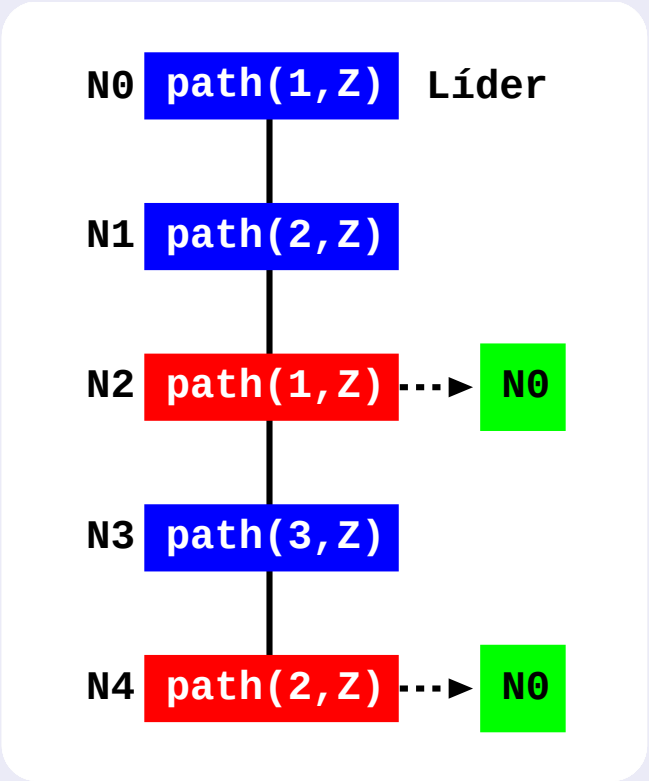
Cálculo do Líder

```
compute_leader(consumer node CN)
   leader = generator_for(CN)  // default leader
   df = TOP_DF
   while (DepFr_cons_cp(df) is younger than leader)
      if (DepFr_leader(df) is older than leader)  // older dependency
         leader = DepFr_leader(df)
         break
      df = DepFr_previous(df)
   return leader
```

Yap Prolog

Como Calcular o Líder

- Em qualquer momento da computação, o líder é sempre o mais novo de entre:
  - O líder calculado pelo consumidor mais novo.
  - O gerador mais novo.



Yap Prolog

Completion

```
completion(generator node GN)
   if (is_leader(GN))
      // check for unconsumed answers in the SCC
      df = TOP_DF
      while (DepFr_cons_cp(df) is younger than GN)
         if (TrNode_child(DepFr_last_answer(df)) != NULL)
            back = DepFr_cons_cp(df)
            restore_bindings(CP_TR(GN),CP_TR(back))
            DepFr_back_leader(df) = GN   // leader to return to
            goto answer_resolution(back)   // try next unconsumed answer
         df = DepFr_previous(df)
      // no unconsumed answers left in the SCC
      complete_SCC_and_adjust_freeze_registers()
      if (CP_DepFr(GN) != NULL)   // local scheduling
         goto completed_table(CP_SgFr(GN))
   ...
```

Yap Prolog

Completion

```
completion(generator node GN)
    ...
    if (CP_DepFr(GN) != NULL)  // local scheduling
       CP_BP(GN) = answer_resolution  // act like a consumer
       ans = SgFr_first_answer(CP_SgFr(GN))
       if (ans != NULL)  // there are unconsumed answers
          // load first answer
          df = CP_DepFr(GN)
          DepFr_last_answer(df) = ans
          load_answer(ans)
          goto continuation_instruction()
    backtrack()
```

Yap Prolog

Answer Resolution

```
answer_resolution(consumer node CN)
   df = CP_DepFr(CN)  // dependency frame for CN
   // check for unconsumed answers in CN
   ans = TrNode_child(DepFr_last_answer(df))
   if (ans != NULL)  // there are unconsumed answers
      // load next answer
      DepFr_last_answer(df) = ans
      load_answer(ans)
      goto continuation_instruction()
   ...
```

Yap Prolog

Answer Resolution

```
answer_resolution(consumer node CN)
    ...
    back_leader = DepFr_back_leader(df)
    if (back_leader == NULL)  // first time here
        backtrack()
    // check for unconsumed answers in the SCC
    df = DepFr_previous(df)
    while (DepFr_cons_cp(df) is younger than back_leader)
        if (TrNode_child(DepFr_last_answer(df)) != NULL)
            back = DepFr_cons_cp(df)
            restore_bindings(CP_TR(CN),CP_TR(back))
            DepFr_back_leader(df) = back_leader  // leader to return to
            goto answer_resolution(back)  // try next unconsumed answer
        df = DepFr_previous(df)
    // no unconsumed answers left in the SCC
    restore_bindings(CP_TR(CN),CP_TR(back_leader))
    goto completion(back_leader)  // move to last leader node
```

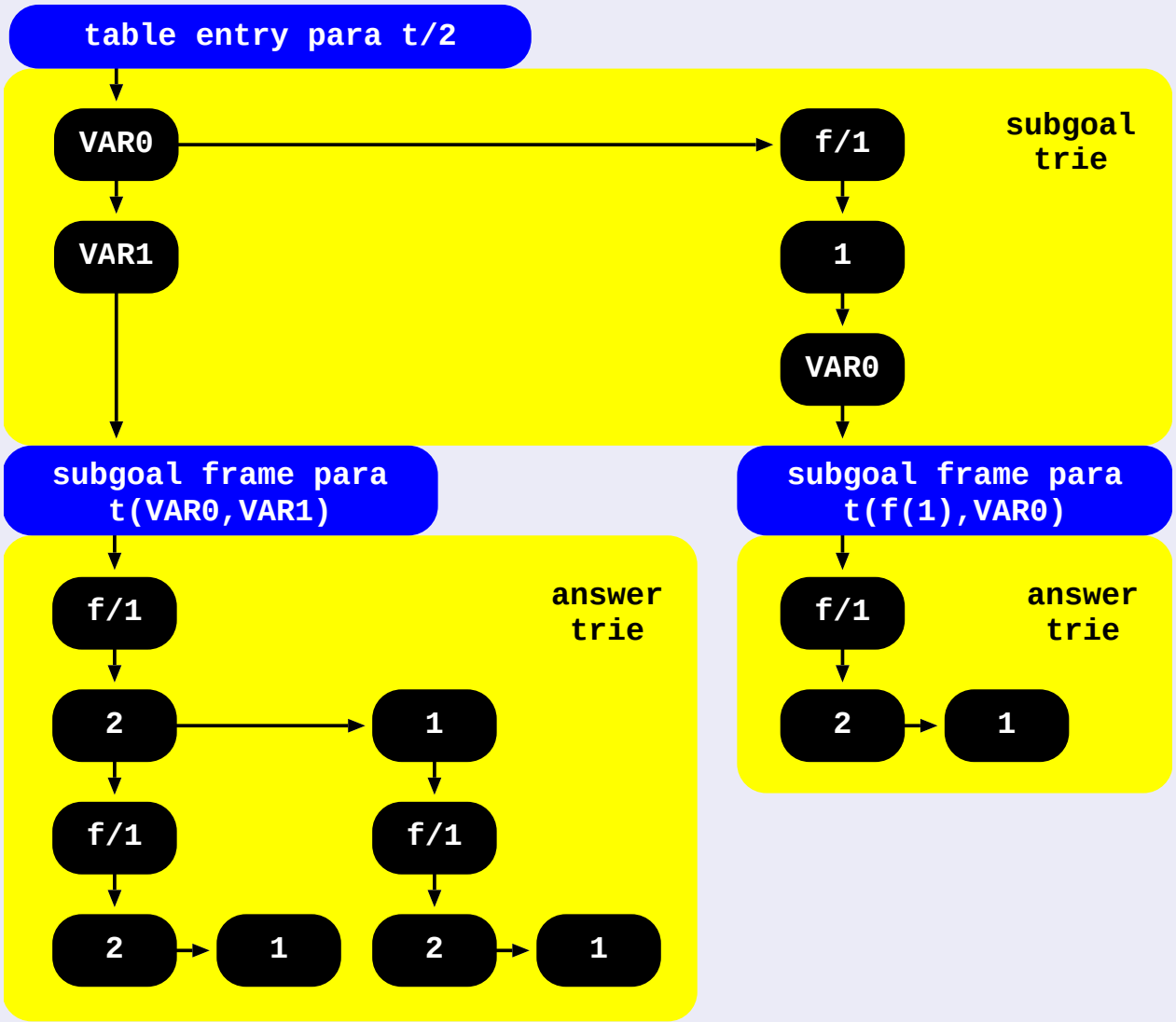
Global Trie

Motivação

- Como já vimos, o desempenho de um sistema de tabulação depende em grande parte da implementação do espaço de tabelas.
- Apesar da eficiência da estrutura de dados baseada em tries, esta apresenta algumas limitações no que diz respeito à capacidade de reconhecer e representar **termos repetidos**.
- A **Global Trie (GT)** é uma extensão ao desenho original do espaço de tabelas onde termos repetidos são representados de forma única numa **estrutura comum**, a global trie, em lugar de serem representados de forma dispersa por diferentes tries.

Global Trie

Motivação



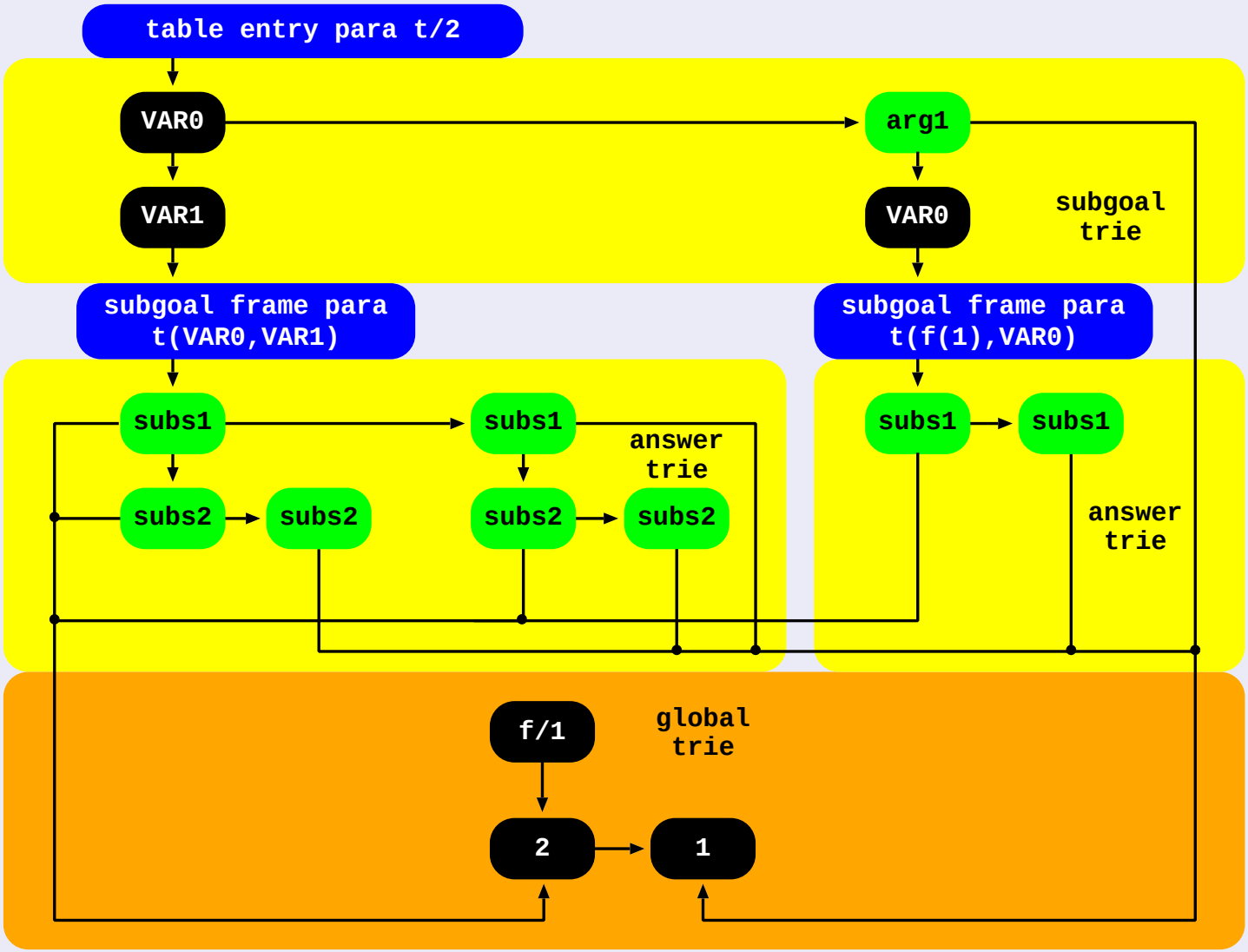
Global Trie

Global Trie for Terms (GT-T)

- Com a GT-T, todos os **argumentos e termos (compostos) de substituição** existentes nas chamadas e/ou soluções tabeladas passam a ter uma **representação única na global trie**, o que evita a sua representação múltipla por tries diferentes.
- Os caminhos nas subgoal e answer tries originais passam a ser representados por um **número fixo de nós** correspondente ao número de argumentos ou termos de substituição na subgoal ou answer trie respetiva.

Global Trie

Global Trie for Terms (GT-T)



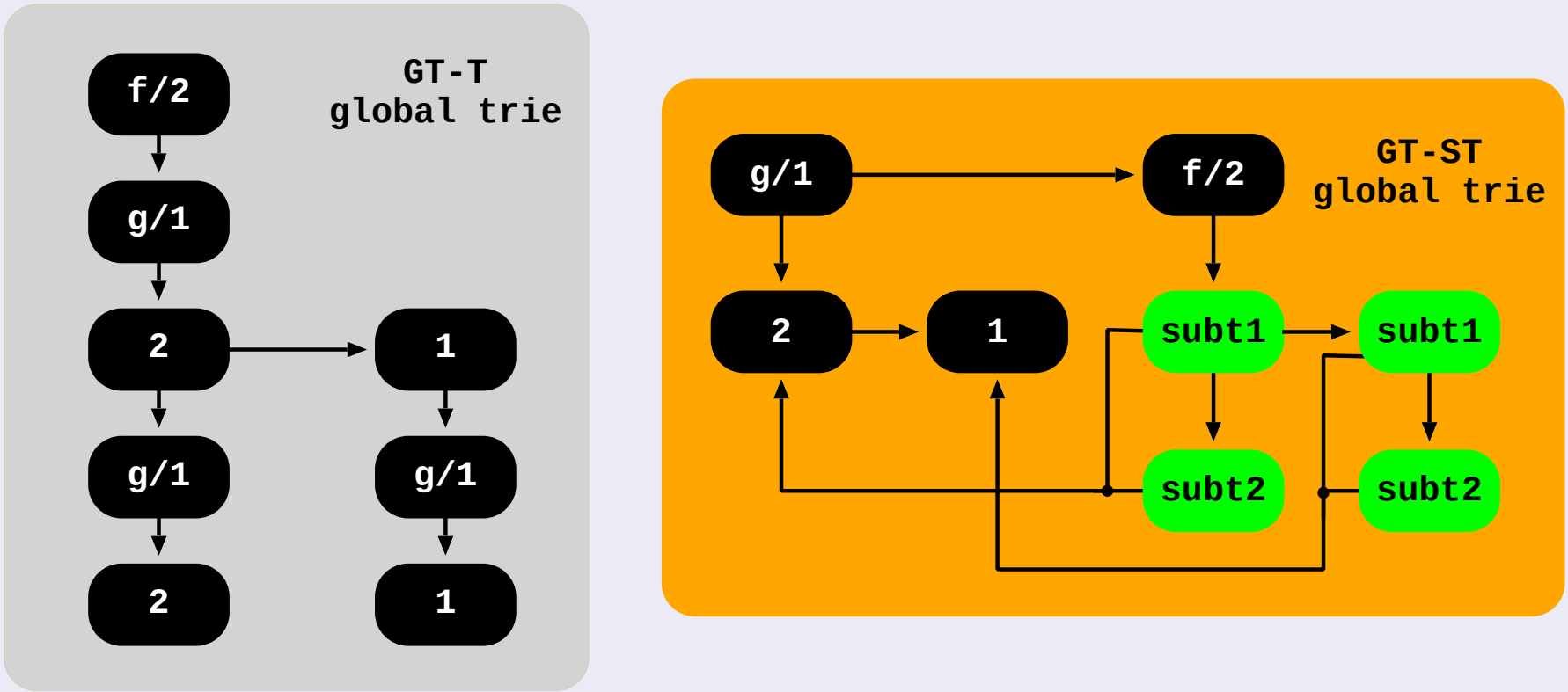
Global Trie

Global Trie for Subterms (GT-ST)

- Com a GT-ST, todos os **termos e subtermos tabelados** que são **estruturalmente iguais** passam a ter uma **representação única na global trie**, o que evita a representação múltipla de subtermos na global trie.
- Apesar da GT-ST usar a mesma estrutura da GT-T para implementar a global trie, cada caminho na GT-ST pode não só representar um termo completo mas também um **subtermo de outro termo**. Em ambos os casos, todos os caminhos representam (sub)termos únicos.

Global Trie

Global Trie for Subterms (GT-ST)



Global Trie

Resultados Experimentais (sem subtermos)

Termos	GT-T/Yap			GT-ST/Yap		
	Mem	Store	Load	Mem	Store	Load
1K ints	1.00	1.05	1.00	1.00	1.09	1.11
1K atoms	1.00	1.04	1.01	1.00	1.04	1.03
1K f/1	1.00	1.32	1.16	1.00	1.34	1.17
1K f/2	**0.50**	1.10	1.14	**0.50**	1.06	1.11
1K f/4	**0.25**	**0.81**	**0.98**	**0.25**	**0.78**	1.04
1K f/6	**0.17**	**0.72**	**0.72**	**0.17**	**0.66**	**0.71**
1K [ ]/1	**0.50**	1.08	1.05	**0.50**	1.10	1.02
1K [ ]/2	**0.25**	**0.80**	**0.94**	**0.25**	1.00	1.05
1K [ ]/4	**0.13**	**0.63**	**0.54**	**0.13**	**0.89**	**0.66**

Utilização de memória e tempos de store/load para um predicado t/5, chamado com todas as combinações de uma e duas variavéis livres nos argumentos, que guarda na tabela termos definidos por fatos `term/1`

Global Trie

Resultados Experimentais (com subtermos)

Termos		GT-T Mem Total/GT	GT-T Tempos Str/Load	GT-ST/GT-T Mem Total/GT	GT-ST/GT-T Tempos Str/Load
500K f/1	**g/1**	17.17/7.63	126/28	1.44/2.00	1.55/1.14
	g/3	32.43/22.89	198/34	1.24/1.33	3.29/1.12
	g/5	47.68/38.15	293/47	1.16/1.20	1.46/1.00
500K f/2	**g/1**	32.43/22.89	203/38	1.00/1.00	1.28/1.13
	g/3	62.94/53.41	45/60	**0.76**/**0.71**	1.18/**0.84**
	g/5	93.46/83.92	438/111	**0.67**/**0.64**	1.10/**0.67**
500K f/3	**g/1**	47.68/38.15	296/50	**0.84**/**0.80**	2.87/1.02
	g/3	93.46/83.92	616/142	**0.59**/**0.55**	1.25/**0.80**
	g/5	139.24/129.7	832/197	**0.51**/**0.47**	**0.96**/**0.67**

Utilização de memória (MBytes) e tempos de store/load (ms) para um predicado t/1, que guarda na tabela termos definidos por fatos `term/1`

Global Trie

Recursão com Listas

```
:- table is_list/1.

is_list([]).
is_list([_|L]):- is_list(L).

test(N):- new_list(N,L), is_list(L).
```

Scheme	Mem		Store	
	Q1	Q2	Q1	Q2
LT	200,380,376	800,760,376	280	1,122
GT-T	200,460,408	800,920,408	247	1,029
GT-ST	**540,408**	**1,080,408**	**140**	**536**

Utilização de memória (bytes) e tempos de store (ms) para as chamadas `test(2500)` (Q1) e `test(5000)` (Q2)

Global Trie

Junção de Listas

```
:- table append/3.

append([],L,L).
append([H|T],L,[H|L1]):- append(T,L,L1).

test(N):- new_list(N,L1), new_list(N,L2), append(L1,L2,_).
```

Scheme	Mem		Store	
	Q1	Q2	Q1	Q2
LT	216,416,576	864,807,808	317	1,125
GT-T	96,360,576	384,695,744	**149**	**589**
GT-ST	**381,664**	**745,056**	239	1,080

Utilização de memória (bytes) e tempos de store (ms) para as chamadas `test(1000)` (Q1) e `test(2000)` (Q2)

Agregação de Soluções

Mode-directed tabling

- A tabulação pode ser vista como o resultado da composição das seguintes duas operações:
  - **Generate():** corresponde a executar a resolução com tabulação a partir da qual é gerada uma coleção de soluções, i.e., podemos ter soluções duplicadas (e infinitas) tal como em Prolog tradicional.
  - **Aggregate():** define o critério que especifica como as soluções são tabeladas, o que para tabulação tradicional, corresponde a eliminar soluções repetidas.
- **Mode-directed tabling** é uma extensão da técnica da tabulação que permite definir diferentes critérios para a agregação de soluções.
  - Esta extensão pode ser vista como uma especialização da operação de Aggregate(), permitindo a definição de critérios alternativos para especificar o modo como soluções variantes (**nos argumentos de indexação**) devem ser tabeladas (**nos argumentos de output**).

Agregação de Soluções

Operadores de Modo

- Com mode-directed tabling, a declaração dos predicados tabelados passa a ser feita da seguinte forma:

  table p$(m_1, ..., m_n)$

  onde os m_i's são **operadores de modo** para os argumentos do predicado.
- No Yap Prolog, existem 7 operadores de modo diferentes:
  - **index:** define os argumentos de indexação (para teste de variância)
  - **first:** considera apenas a primeira solução
  - **last:** considera apenas a última solução
  - **min:** considera apenas a solução mínima
  - **max:** considera apenas a solução máxima
  - **all:** considera todas as soluções
  - **sum:** considera a soma de todas as soluções

Agregação de Soluções

Definição

- Dada a declaração genérica $p(m_1, ..., m_j, m_{j+1}, ..., m_n)$, onde para $1 <= i <= j$, m_i é um argumento de indexação e para $j+1 <= i <= n$, m_i é um argumento de output, a operação de $Aggregate(p/n)$ é definida como:

$$
\begin{aligned}
Aggregate(p/n) = \{ & p(x_1, ..., x_n) \mid \\
& \exists (z_{j+1}, ..., z_n) : p(x_1, ..., x_j, z_{j+1} ..., z_n) \in Generate(p/n) \\
& \wedge\ x_{j+1} \in m_{j+1}(Out_{j+1}(x_1, ..., x_j)) \\
& \wedge \ldots \\
& \wedge\ x_n \in m_n(Out_n(x_1, ..., x_{n-1}))\}
\end{aligned}
$$

onde

$$
\begin{aligned}
& Out_j(x_1, ..., x_{j-1}) = \\
& \quad \{y \mid \exists (z_{j+1}, ..., z_n) : p(x_1, ..., x_{j-1}, y, z_{j+1} ..., z_n) \in Generate(p/n)\}
\end{aligned}
$$

Agregação de Soluções

Definição

- Considere a declaração $p(index, min, all)$ e o seguinte conjunto de soluções:

$$Generate(p/3) = \{p(a,2,2), p(a,3,1), p(b,2,1), p(b,1,2), p(b,1,1)\}$$

$$\begin{aligned} Aggregate(p/3) = \{p(x_1,x_2,x_3) \mid \\ \exists(z_2,z_3) : p(x_1,z_2,z_3) \in Generate(p/3) \\ \wedge\ x_2 \in min(Out_2(x_1)) \wedge\ \ x_3 \in all(Out_3(x_1,x_2))\} \end{aligned}$$

$$x_1 \in \{a,b\}$$
$$\mathbf{x_1 = a} \Rightarrow \mathbf{x_2} \in min(Out_2(a)) = min(\{2,3\}) = \mathbf{\{2\}}$$
$$\mathbf{x_1 = a \wedge x_2 = 2} \Rightarrow \mathbf{x_3} \in all(Out_3(a,2)) = all(\{2\}) = \mathbf{\{2\}}$$
$$\mathbf{x_1 = b} \Rightarrow \mathbf{x_2} \in min(Out_2(b)) = min(\{2,1\}) = \mathbf{\{1\}}$$
$$\mathbf{x_1 = b \wedge x_2 = 1} \Rightarrow \mathbf{x_3} \in all(Out_3(b,1)) = all(\{2,1\}) = \mathbf{\{2,1\}}$$

$$Aggregate(p/3) = \mathbf{\{p(a,2,2), p(b,1,2), p(b,1,1)\}}$$

Agregação de Soluções

Exemplo I

table p(index,first)

Soluções		Tabela
1. **p(a,1)**	nova	1
2. **p(a,2)**	variante da 1	1
3. **p(b,3)**	nova	1 , 3
4. **p(b,4)**	variante da 3	1 , 3
5. **p(a,5)**	variante da 1	1 , 3

Agregação de Soluções

Exemplo II

table p(index,min,all)

	Soluções	Tabela
1. **p(a,2,f(x))**	nova	1
2. **p(a,1,f(y))**	variante melhor do que 1	2
3. **p(b,1,g(x))**	nova	2 , 3
4. **p(b,2,g(y))**	variante pior do que 3	2 , 3
5. **p(a,1,f(z))**	variante igual a 2	2 , 3 , 5

Agregação de Soluções

Caminho Mínimo num Grafo

```
:- table path(index,index,min).

path(X,Z,D):- edge(X,Z,D).
path(X,Z,D):- path(X,Y,D1), edge(Y,Z,D2), D is D1+D2.
```

Agregação de Soluções

Knapsack (máximo número de itens num conjunto)

```
:- table knapsack(index,index,max).

knapsack(_,0,0).
knapsack(I,K,V):-
    I > 0,
    I1 is I-1,
    knapsack(I1,K,V).
knapsack(I,K,V):-
    I > 0,
    item(I,F),
    K1 is K-F,
    K1 >= 0,
    I1 is I-1,
    knapsack(I1,K1,V1),
    V is V1+1.
```