AMBIENTE DE EXECUÇÃO DO PRONUS PROLOG

Autor
Luciano Conrado Machado Dallolio
Orientador
Luiz Fernando Pereira de Souza

NCE-10/91

Grupo de Inteligência Artificial/NCE

julho/91

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL
E-mail : PROLOG@VAX1.UFRJ.RNP.BR

Ambiente de execução do Pronus Prolog.

Sinopse

Este é o primeiro de uma série de relatórios que descrevem o trabalho realizado pelo Grupo de Inteligência Artificial do NCE visando desenvolver um interpretador completo para a linguagem Prolog denominado Pronus Prolog.

Este primeiro relatório descreve o ambiente de execução do interpretador e sua organização interna. Aqui são abordadas a gerência de memória, as estruturas de dados internas do interpretador, a implementação das estruturas de dados da linguagem Prolog, problemas relacionados com a representação interna de variáveis e como o módulo Mestre integra os demais módulos do interpretador.

Pronus Prolog execution environment.

Abstract.

This is the first report of a series that describes the work performed by the Artificial Intelligence Group of NCE. The main purpose of this work is to develop a complete interpreter for Prolog named Pronus Prolog.

This report describes the execution environment and the internal organization of the interpreter. It is discussed the memory management, the internal data structures, the implementation of Prolog's data structures, problems related with representation of variables, and the Master module that controls all the interpreter.

Índice.

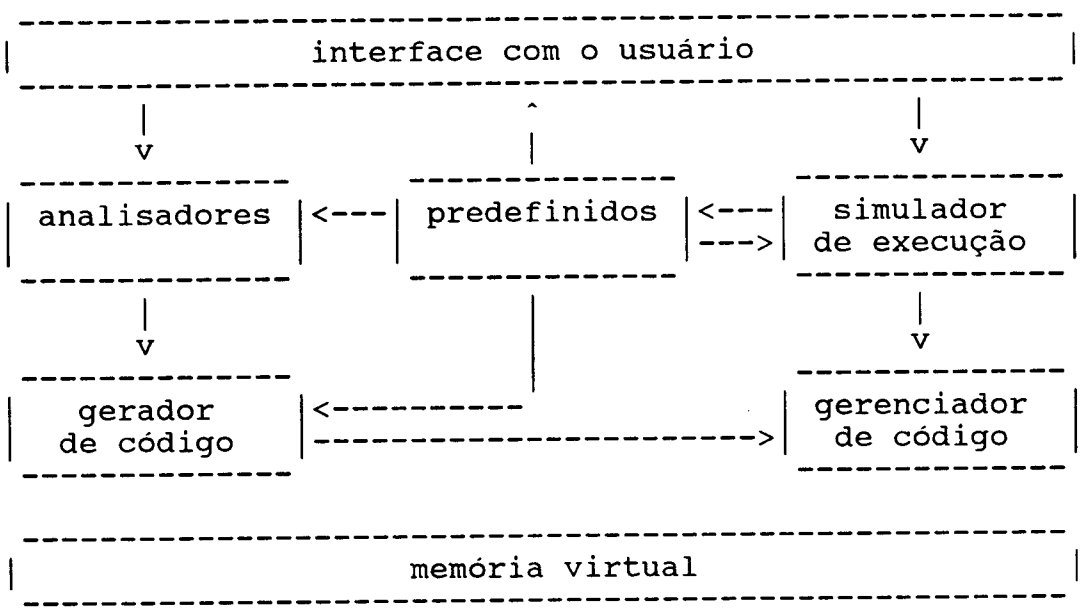
1. Introdução 1
2. Descrição do ambiente de execução 3
   2.1. O módulo de memória virtual 3
   2.2. Estruturas de dados internas do Pronus Prolog 4
      2.2.1. A tabela de símbolos 5
      2.2.2. A tabela de operadores 6
      2.2.3. A tabela de arquivos 6
      2.2.4. A tabela de variáveis 8
      2.2.5. A tabela de cláusulas 8
      2.2.6. A tabela de predicados 9
      2.2.7. A tabela de estruturas 11
      2.2.8. A tabela de predefinidos 11
   2.3. Termos da linguagem Prolog 12
      2.3.1. Objetos 12
      2.3.2. Inteiros 13
      2.3.3. Reais 13
      2.3.4. Átomos 14
      2.3.5. Cadeias 15
      2.3.6. Estruturas 15
      2.3.7. Listas 16
      2.3.8. Referências 17
      2.3.9. Variáveis 17
   2.4. O processo de unificação 17
   2.5. Representação interna de variáveis no Pronus Prolog ... 19
      2.5.1. Variáveis a tempo de execução 19
      2.5.2. Variáveis associadas a símbolos 20
      2.5.3. Variáveis durante a geração de código 20
   2.6 Conversão da representação de variáveis 21
      2.6.1. Conversão referências -> variáveis-código 21
      2.6.2. Conversão variáveis-código -> referências 26
      2.6.3. Conversão variáveis-código -> variáveis-símbolo 28
      2.6.4. Conversão variáveis-símbolo -> variáveis-código 29

2.7. O módulo Mestre 31
2.7.1. Modos de funcionamento e níveis de recursão 32
2.7.2. Leitura de um termo 33
2.7.3. Compatibilização do termo 33
2.7.4. Representação da cláusula e geração de código .. 34
2.7.5. Disparando uma consulta 35
2.7.6. Instalando uma cláusula 37
3. Conclusão 39
4. Referências bibliográficas 40

1. Introdução.

O Grupo de Inteligência Artificial do NCE/UFRJ vem pesquisando a linguagem de programação Prolog e aspectos relacionados com a sua implementação desde o ano de 1987. Como fruto deste trabalho, foi desenvolvido o interpretador Pronus Prolog, baseado no modelo de execução proposto por Warren [Warren 83].

Em uma primeira fase da pesquisa, foi implementado um protótipo que visava validar as pseudo-instruções e os aspectos básicos da execução Prolog. Este protótipo era composto por três diferentes programas que desempenhavam as tarefas de geração de código, montagem do código binário e simulação de uma máquina Prolog [Souza 88]. A partir dos conhecimentos adquiridos nesta fase inicial é que foi possível implementar o interpretador Pronus Prolog. O esquema abaixo descreve a arquitetura interna do interpretador que engloba em um só programa todas as três tarefas antes desempenhadas pelo protótipo.



À medida que o programa Prolog é lido pela interface, ele é analisado, e é gerada uma representação interna. A partir desta representação, o módulo de geração de código constrói uma seqüência de pseudo-instruções que é tratada pelo módulo de gerenciamento de código. Em tempo de execução, o módulo de simulação executa a semântica associada a cada pseudo-instrução [Souza 88].

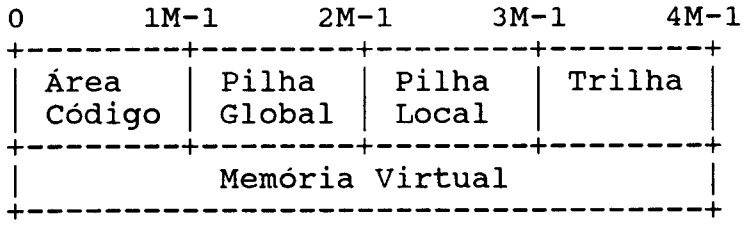
O presente trabalho apresenta uma visão dos principais módulos do interpretador procurando dar uma idéia das soluções adotadas para representar as estruturas de dados da linguagem Prolog, do mecanismo de execução e das estruturas de dados internas ao interpretador.

2. Descrição do ambiente de execução.

Ao longo deste capítulo serão abordados diversos aspectos da implementação do interpretador procurando dar uma visão geral do seu funcionamento. O objetivo desta seção é tornar mais fácil o entendimento dos relatórios que descrevem a biblioteca de predefinidos [Dallolio 91a] [Dallolio91b] uma vez que a maioria dos predefinidos depende de mecanismos aqui descritos para o seu funcionamento.

2.1 O módulo de memória virtual.

A memória do interpretador Pronus Prolog abrange quatro Mega palavras e está dividida em quatro áreas denominadas: área de código, pilha global, pilha local e trilha (necessariamente nesta ordem). O esquema abaixo representa o modo como está dividida a memória do interpretador. Os endereços estão colocados em relação a palavras.



O memória do interpretador está dividida em páginas de 512 palavras, gerenciadas por um módulo de memória virtual. Deste modo, parte destas páginas está alocada na área de dados do processo, enquanto que o restante está alocado em disco.

O número máximo de páginas contidas na memória do processo varia de acordo com o porte do sistema em que o interpretador está instalado. Quando instalado no sistema EBC-32010, que possui 3 Mega octetos de memória, o número máximo de páginas fica em torno de 400, enquanto que no sistema VAX-8810, que possui gerência de memória virtual por hardware, este total chega a 8.0000 páginas.

A gerência de memória virtual implica no uso de funções especiais para as operações de escrita e leitura em memória. Estas funções recebem como argumento um endereço virtual (M_Endereço), devendo buscar a página em que se encontra este endereço. Se a página de memória referenciada estiver em disco, o seu conteúdo deve ser carregado para a memória do processo a fim de realizar uma leitura ou escrita no endereço dado. Vamos descrever de modo superficial as funções que implementam operações de leitura e escrita em memória virtual de modo a facilitar o entendimento de alguns exemplos apresentados nas próximas seções que fazem referência a estas funções.

Função m_le_p - Recebe como argumento um endereço virtual retornando a palavra contida neste endereço.

Função m_le_o - Recebe como argumento um endereço virtual retornando o objeto que inicia neste endereço (Os objetos são compostos por duas palavras e serão definidos em detalhes na seção 2.3.1).

Função m_esc_p - Recebe como argumentos um endereço virtual e uma palavra. A palavra é escrita no endereço virtual dado.

Função m_esc_o - Recebe como argumentos um endereço virtual e um objeto. O objeto é escrito no endereço virtual dado.

2.2 Estruturas de dados internas do Pronus Prolog.

Nesta seção, vamos descrever as principais estruturas de dados utilizadas pelo interpretador para armazenar informações relativas à base de dados, código gerado, símbolos definidos pelo usuário, definições de operadores etc. É importante notar que as estruturas de dados aqui descritas são utilizadas como suporte na implementação da linguagem Prolog, ficando invisíveis para o usuário.

2.2.1 A tabela de símbolos.

A tabela de símbolos é o lugar onde são armazenados os àtomos predefinidos [Clocksin 81] e aqueles criados pelo usuário durante a execução do interpretador. Deste modo, no interpretador Pronus Prolog, um átomo é identificado pelo índice da posição que ele ocupa dentro desta tabela. A definição da tabela de símbolos está descrita abaixo :

```
typedef struct ts_símbolo Ts_Símbolo;

struct ts_simbolo
{
    char         tsi_quotado;   /* Indica se deve ser quotado   */
    char         tsi_indice;    /* Índice dentro do bloco       */
    char         *tsi_nome;     /* Nome do símbolo              */
    Ts_Simbolo   *tsi_sinonimo; /* Lista de colisão             */
    M_Endereco   tsi_tbestr;    /* Tabela de estruturas (f/a)   */
    To_Operador *tsi_tbopr;     /* Tabela de operadores         */
};
```

O campo tsi_quotado indica se o átomo é quotado ou não. Um átomo é não quotado se ele inicia por letra minúscula seguida de minúsculas, maiúsculas ou dígitos ou se ele contém apenas símbolos; todos os demais átomos são quotados. A tabela abaixo define os caracteres reconhecidos como símbolos pelo interpretador :

Símbolos
& * + - . : < = > ? @ \ ^ ~ / .

No caso em que o átomo é funcional de alguma estrutura, o campo tsi_tbestr aponta para o campo da tabela de estruturas onde estão listadas todas as estruturas que têm o mesmo funcional. Do mesmo modo, se o átomo está definido como um operador, o campo tsi_tbopr aponta para a posição na tabela de operadores onde estão armazenadas as informações relativas a este operador. O campo tsi_nome contém a cadeia de caracteres que representa o átomo. Os demais campos são utilizados no gerenciamento interno da tabela de símbolos.

2.2.2 A tabela de operadores.

A tabela de operadores contém informações relativas à definição de operadores. O seu conteúdo não é fixo, podendo ser alterado pelo usuário durante a execução de um Programa Prolog. A definição da tabela de operadores está descrita abaixo :

```
typedef struct to_operador To_Operador;

struct to_operador
{
    Ts_Simbolo   *top_tbsimb;  /* Ent. na tabela de símbolos   */
    short        top_assoc;    /* Associatividades válidas     */
    short        top_faprec;   /* Precedência de fx ou fy      */
    short        top_afprec;   /* Precedência de xf ou yf      */
    short        top_afaprec;  /* Precedência xfx, xfy ou yfx  */
};
```

O campo top_tbsimb aponta para a posição da tabela de símbolos que contém o nome do operador. As demais posições caracterizam o operador definindo as sua associatividade e a precedência de cada associatividade. Estes campos estão analisados com maior detalhe no relatório que trata dos predefinidos que manipulam a tabela de operadores [Dallolio 91a].

2.2.3 A tabela de arquivos.

As informações relativas a arquivos acessíveis pelo interpretador estão distribuidas em duas tabelas denominadas tabela de nomes e tabela de descritores. A tabela de nomes contém informações a respeito dos arquivos associados a átomos, enquanto que a tabela de descritores armazena informações a respeito de todos os arquivos acessíveis pelo interpretador, incluindo os arquivos padrões e aqueles associados a átomos. A definição das estruturas que compõem estas tabelas está descrita abaixo :

```
typedef struct ta_desc Ta_Desc;

struct ta_desc
{
     FILE    *tad_pa;           /* Ponteiro para o arquivo        */
     int     tad_modo;          /* Modo de acesso ao arquivo      */
     long    tad_pos;           /* Posição do arquivo             */
     int     tad_nlinha;        /* Número da linha lida (léxico) */
     char    tad_linha[TA_TAMLINHA];
                                /* Linha lida (léxico)            */
     char    *tad_pc;           /* Ponteiro para a última posição*/
                                /* analisada na linha (léxico)    */
};


typedef struct ta_nome Ta_Nome;

struct ta_nome
{
    int    taa_desc;           /* Índice da tabela descritores  */
    int    taa_simb;           /* Índice da tabela de símbolos  */
    int    taa_estado;         /* Estado do arquivo p/ tell,see */
    int    taa_pos;            /* Última posição de leitura     */
};
```

O campo tad_pa é utilizado para acessar o arquivo através da biblioteca da linguagem C, o campo tad_modo permite verificar se o usuário está usando o arquivo de modo consistente e o campo tad_pos armazena a última posição lida de um arquivo para o caso de ser necessário um novo posicionamento. Os demais campos da tabela de descritores são utilizados pelo analisador léxico nos casos em que a entrada padrão do Pronus é redirecionada para o arquivo contido naquela posição da tabela de descritores. Inicialmente, a tabela de descritores contém entradas para os arquivos padrões stdin, stdout e stderr da linguagem C.

A tabela de nomes realiza a associação entre um átomo e uma posição da tabela de descritores. A finalidade desta tabela é dar suporte à implementação dos predefinidos de entrada e saída do padrão de Edimburgo (see, seeing, seen, tell, telling, told). Inicialmente, a tabela de nomes contém duas entradas associadas ao símbolo "user". Este símbolo designa um arquivo especial que contém os arquivos padrões stdin e stdout. O funcionamento destes predefinidos é descrito no relatório que trata dos predefinidos de entrada e saída [Dallolio 91a].

2.2.4 A tabela de variáveis.

A tabela de variáveis armazena informações relativas às variáveis de um termo lido pelo interpretador. A definição da tabela de variáveis está descrita abaixo :

```
typedef struct tv_variavel Tv_Variável;

struct tv_variável
{
    Ts_Simbolo  *tva_ptsimb;   /* Ponteiro para tabela de simb.*/
    int          tva_esq;      /* Ramo esquerdo da árvore      */
    int          tva_dir;      /* Ramo direiro da árvore       */
    int          tva_numero;   /* Número da variável           */
    int          tva_tipo;     /* Tipo da variável             */
    Tv_Estado    tva_estado;   /* Estado da variável           */
    int          tva_nivel;    /* Nível da variável            */
    int          tva_achou;    /* Nível em que achou a variável*/
    M_Endereco   tva_insegura; /* Ocorrência insegura          */
    M_Endereco   tva_ultima;   /* Última ocorrência da variável*/
    M_Endereco   tva_ref;      /* Variável/Referência          */
    int          tva_alocacao; /* Alocação da variável         */
    int          tva_nasce;    /* Linha primeiro uso           */
    int          tva_morre;    /* Linha último uso             */
    int          tva_vem;      /* No. do registrador fonte     */
    long         tva_vai;      /* No. registrador destino      */
    Tv_Variavel *tva_prox;     /* Lista variáveis mesmo nível  */
};
```

O campo tva_ptsimb aponta para a posição na tabela de símbolos que contém o nome da variável. O campo tva_ref é utilizado na implementação do algoritmo que converte variáveis livres em variáveis-código (Ver seção 2.6.1). Os demais campos são de uso do gerador de código [Dallolio 89], não cabendo a sua descrição neste trabalho.

2.2.5 A tabela de cláusulas.

As cláusula são a unidade de programação Prolog. Cada cláusula é composta de uma cabeça, com ou sem argumentos, e um corpo, separados através do símbolo "se" (:-). O corpo da cláusula é composto por um ou mais objetivos separados pela conjunção "e" (,). Deste modo na cláusula "a(X) :- b(Y,Z), c", a cabeça da cláusula é a estrutura "a(X)", e o corpo da clásula é a estrutura "b(Y,Z), c". A tabela de cláusulas armazena informações relativas

às cláusulas presentes na base de dados do interpretador. A definição da tabela de cláusulas está descrita abaixo :

```
typedef struct tc_claus Tc_Claus;

struct    tc_claus
{
    M_Endereco  tcl_repres;    /* Representação da cláusula    */
    M_Endereco  tcl_codigo;    /* Código da cláusula           */
    M_Endereco  tcl_predicado; /* Predicado ao qual pertence   */
    M_Endereco  tcl_anterior;  /* Cláusula anterior            */
    M_Endereco  tcl_proxima;   /* Próxima cláusula             */
    M_Endereco  tcl_removida;  /* Lista de cláusulas removidas */
};
```

O campo tcl_repres guarda a representação da cláusula a fim de que usuário possa listar as cláusulas presentes na base de dados. O campo tcl_código contém o código associado à cláusula. O campo tcl_removida é utilizado para encadear as cláusulas que são removidas da base de dados em uma lista à parte para posterior liberação ao final de uma execução.

O campo tcl_predicado aponta para o predicado ao qual a cláusula pertence. Um predicado é definido como um conjunto de cláusulas que têm o mesmo funcional e a mesma aridade. Assim, a cláusula do nosso exemplo "a(X) :- b(Y,Z), c" pertence ao predicado que tem funcional "a" e aridade um, o que é representado pela notação a/1.

Os campos tcl_proxima e tcl_anterior são utilizados para armazenar as cláusulas sob a forma de uma lista duplamente encadeada na ordem em que elas são inseridas na base de dados.

2.2.6 A tabela de predicados.

A tabela de predicados armazena informações relativas aos predicados definidos durante a inicialização ou execução do interpretador. A definição da tabela de predicados está descrita a seguir:

```
typedef struct tp_pred Tp_Pred;

struct tp_pred
{
    M_Palavra    pr_nclaus;     /* No. cláusulas do predicado  */
    M_Endereco   pr_inicio;     /* Primeira cláusula           */
    M_Endereco   pr_fim;        /* Última cláusula             */
    M_Endereco   pr_codigo;     /* Código do predicado         */
    M_Palavra    pr_alterado;   /* Indica estado do predicado  */
    M_Endereco   pr_anterior;   /* Predicado alterado anterior */
    M_Endereco   pr_proximo;    /* Próximo predicado alterado  */
    M_Endereco   pr_pred;       /* Lista de predicados         */
    M_Endereco   pr_ord;        /* Lista predicados ordenados  */
    M_Endereco   pr_estr;       /* Entrada na tb de estruturas */
    M_Palavra    pr_tipo;       /* Indica se o predicado é     */
                                /* predefinido ou do usuário   */
    M_Palavra    pr_recons;     /* Predicado foi alterado pela */
                                /* reconsult                   */
};
```

Os campos pr_nclaus, pr_inicio e pr_fim são utilizados para armazenar as cláusulas do predicado. Os ponteiros pr_inicio e pr_fim apontam respectivamente para o início e o fim da lista de cláusulas do predicado, enquanto que o contador pr_nclaus indica quantas cláusulas compõem o predicado.

O ponteiro pr_código referencia o código do predicado que é responsável pela seleção da cláusula a ser executada no momento. O campo pr_alterado indica se o número de cláusulas do predicado foi alterado. Os ponteiros pr_próximo e pr_anterior encadeiam a lista de predicados alterados. Todos os predicados alterados devem ter o seu código renovado antes do início de uma nova execução.

Os ponteiros pr_pred e pr_ord implementam duas listas de predicados. A primeira está organizada de acordo com a ordem de inserção na base de dados, enquanto que a segunda se apresenta em ordem de funcional/aridade.

O campo tr_tipo distingue os predicados definidos pelo usuário dos predicados predefinidos. O ponteiro pr_estr aponta para a posição da tabela de estruturas relacionada com a estrutura que tem o mesmo funcional e aridade do predicado. O campo pr_recons é utilizado na implementação do predefinido reconsult, descrito no relatório que trata dos predefinidos de manipulação de bases de dados [Dallolio 91b].

2.2.7 A tabela de estruturas.

A tabela de estruturas contém uma entrada para cada estrutura definida durante a inicialização ou execução do interpretador. A definição da tabela de estruturas está descrita abaixo :

```
typedef struct te_estr Te_Estr;

struct te_estr
{
    M_Palavra   tes_aridade;   /* Aridade                          */
    M_Objeto    tes_tbsimb;    /* Índice da tabela de símbolos     */
    M_Endereco  tes_pred;      /* Predicado associado              */
    M_Endereco  tes_sinonimo;  /* Lista predicados de mesmo        */
                               /* funcional                        */
};
```

O campo tes_aridade guarda a aridade da estutura, isto é, o número de argumentos que a estrutura possui. O campo tes_tbsimb guarda o índice da posição na tabela de símbolos onde está armazenado o nome do funcional da estrutura. No caso em que a estrutura tem um predicado associado, o ponteiro tes_pred aponta para o predicado a que a estrutura pertence, enquanto que o ponteiro tes_sinonimo aponta para uma lista de todos os predicados que têm o mesmo funcional.

2.2.8 A tabela de predefinidos.

A tabela de predefinidos guarda informações a respeito da biblioteca de predefinidos. As informações contidas nesta tabela são utilizadas na inicialização do interpretador e durante uma execução para ativar os predefinidos. A definição da tabela de predefinidos está descrita abaixo :

```
typedef struct tra_trap Tra_Trap;

struct    tra_trap
{
    long   tra_tbsimb;          /* Entrada na tabela símbolos   */
    short  tra_aridade;         /* Aridade do predicado         */
    int    (*tra_funcao) ();    /* Função que implementa o      */
                                /* predefinido                  */
};
```

Os campos tra_tbsimb e tra_aridade são utilizados na inicialização das tabelas de estruturas e de predicados. Ao final desta inicialização estas tabelas deverão conter as estruturas e predicados associados aos predefinidos da biblioteca.

O campo tra_função implementa a pseudo-instrução "trap" que dispara um predefinido durante uma execução Prolog. O pseudo código abaixo descreve o funcionamento da pseudo-instrução "trap".

```
i_trap (n)
int     n;                          /* Índice da Trap                */
{
      if (!tra_tab[n].tra_funcao ())
          i_fail ();
}
```

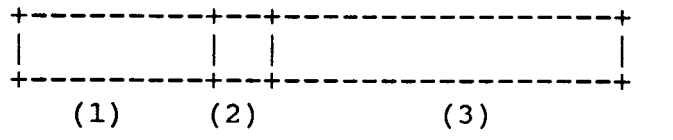
O índice "n" é utilizado para indexar a tabela de predefinidos onde está guardado o ponteiro para a função que implementa o predefinido desejado. Maiores detalhes quanto a ativação de predefinidos são encontrados no relatório "Ativação dos predefinidos, predefinidos de E/S e controle de fluxo" [Dallolio 91a].

2.3 Termos da linguagem Prolog.

A denominação genérica de termo é utilizada para designar qualquer uma das estruturas de dados suportadas pelo Prolog (inteiros, reais, átomos, cadeias, estruturas, listas, referências e variáveis). Esta seção descreve o modo pelo qual estas estruturas de dados foram implementados no interpretador Pronus Prolog.

2.3.1 Objetos.

Um objeto é a menor unidade de informação que o interpretador reconhece. Cada objeto abrange duas palavras e está dividido em três campos denominados etiqueta, bites de coleta de lixo e campo valor. O esquema abaixo descreve a organização interna de um objeto.

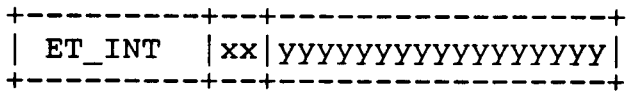


(1) - Etiqueta (3 bites).
(2) - Bites de coleta (2 bites).
(3) - Valor (27 bites).

O campo etiqueta designa o tipo de informação que está armazenada no campo valor. Os bites de coleta estão reservados para a futura implementação de coleta de lixo. O campo valor contém 27 bites que são interpretados de acordo com a etiqueta do objeto.

2.3.2 Inteiros.

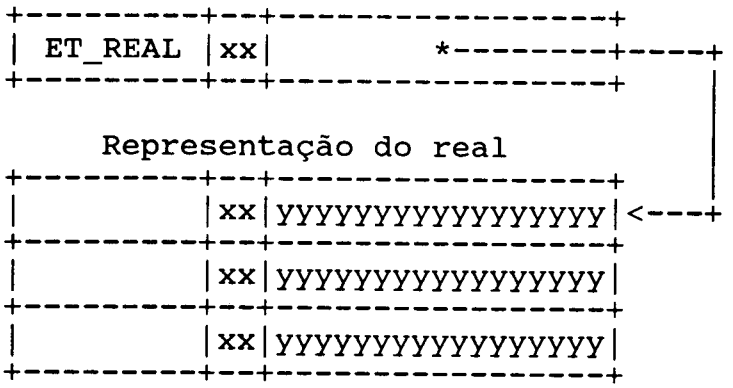
Um inteiro no Pronus é composto por um único objeto com etiqueta ET_INT. O seu valor é mapeado nos 27 bites do campo valor e varia de -67108864 até 67108863. O esquema abaixo descreve a organização de um inteiro.



x - Bites de coleta de lixo.
y - Valor do inteiro.

2.3.3 Reais.

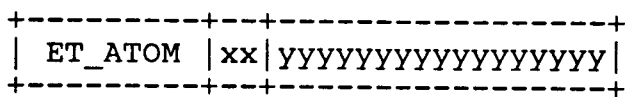
Um real no Pronus é mapeado em quatro objetos. O interpretador identifica um real através de um objeto com etiqueta ET_REAL que aponta para a área onde está mapeado o real na pilha global. O esquema a seguir descreve a organização de um real.



x - Bites de coleta de lixo.
y - Expoente e mantissa do real.

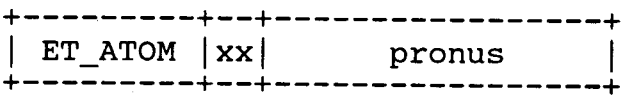
2.3.4 Átomos.

Os átomos ocupam um único objeto com etiqueta ET_ATOM e cujo campo valor contém o índice da posição na tabela de símbolos onde o átomo está armazenado. O esquema abaixo descreve a organização de um átomo.



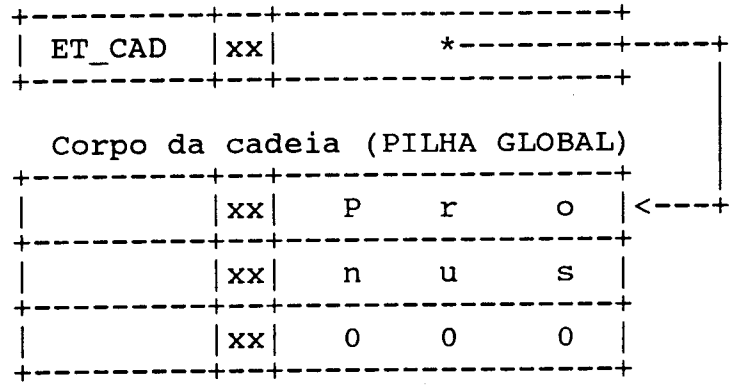
x - Bites de coleta de lixo.
y - Índice da tabela de símbolos.

Ao longo deste texto, por questão de simplicidade e clareza, os átomos serão representados esquemáticamente como objetos cujo campo valor contém o próprio nome do átomo, e não um índice da tabela de símbolos. Deste modo, a representação esquemática do átomo "pronus" seria :



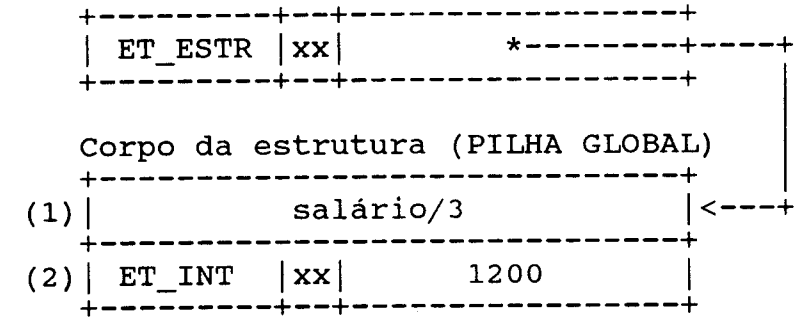
2.3.5 Cadeias.

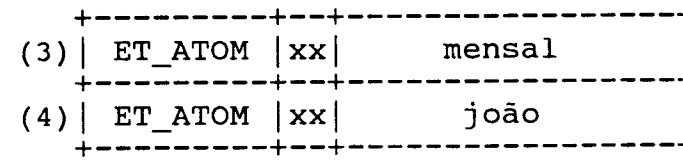
No Pronus Prolog, as cadeias são armazenadas mapeando-se os caracteres em objetos consecutivos na pilha global. Apesar de abranger quatro bytes, cada objeto contém apenas três caracteres, pois o octeto que está reservado à etiqueta e aos bites de coleta de lixo não pode ser utilizado. O último objeto deve ser completado com caracteres nulos de modo a marcar o fim da cadeia. Uma cadeia é reconhecida através de um objeto com etiqueta ET_CAD que aponta para a área na pilha global onde a cadeia está armazenada. O exemplo abaixo mostra como ficaria armazenada a cadeia $Pronus$ na pilha global :



2.3.6 Estruturas.

Uma estrutura em Prolog é composta de um funcional e seus argumentos. Os argumentos de uma estrutura podem ser quaisquer termos válidos no Prolog e o seu número determina a aridade da estrutura. Assim, a estrutura "salário(1200, mensal, joão)" tem funcional "salário" e aridade três ou seja salário/3. O esquema abaixo descreve como a estrutura salário/3 é armazenada no Pronus.



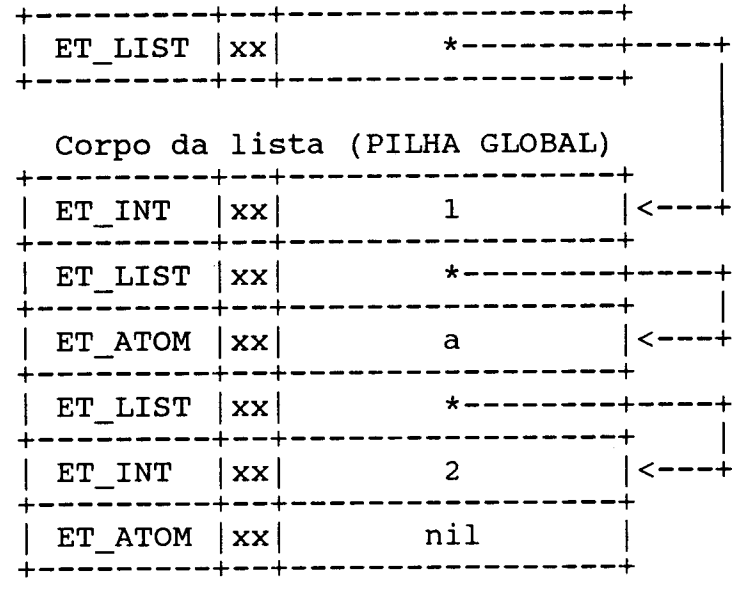


Uma estrutura é reconhecida por um objeto com etiqueta ET_ESTR que aponta para a área na pilha global onde a estrutura está armazenada. O primeiro objeto de uma estrutura (Objeto (1) do esquema) é na realidade um ponteiro para uma posição da tabela de estruturas, enquanto que os demais objetos são os argumentos da estrutura (Objetos (2), (3) e (4) do esquema).

2.3.7 Listas.

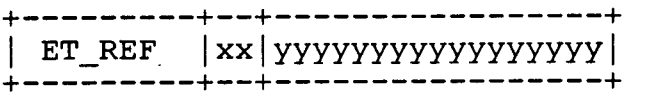
As listas são um tipo especial de estrutura. Uma lista é composta de uma cabeça, o primeiro elemento, e um corpo, que compreende o resto da lista. A cabeça e o corpo são argumentos do funcional ponto (.) e a lista vazia é representada pelo átomo "nil" ([]).

No Pronus, uma lista é identificada por um objeto com etiqueta ET_LIST que aponta para a pilha global, onde estão a cabeça e o corpo da lista. O esquema abaixo descreve a organização da lista "[1, a, 2]".



2.3.8 Referências.

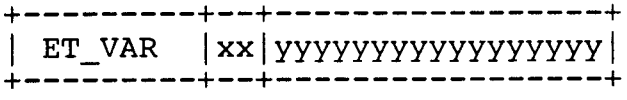
As referências são objetos com etiqueta ET_REF cujo campo valor contém um endereço virtual. Referências podem apontar para outras referências formando cadeias de referências. O ato de seguir uma cadeia de referências até o seu fim denomina-se desreferência. Uma referência que aponta para si mesma representa uma variável livre a tempo de execução. O esquema abaixo descreve a organização de uma referência.



x - Bites de coleta de lixo.
y - Endereço virtual (M_Endereço).

2.3.9 Variáveis.

As variáveis, exceto a tempo de execução, são objetos com etiqueta ET_VAR cujo campo valor deverá ser interpretado como um índice da tabela de símbolos ou um índice da tabela de variáveis, dependendo do contexto em que esta variável está sendo utilizada. A seção 2.5 descreve em detalhes os problemas relacionados à representação de variàveis no interpretador Pronus Prolog. O esquema abaixo apresenta a organização de uma variável :



x - Bites de coleta de lixo.
y - Índice.

2.4 O processo de unificação.

A unificação é um conjugado de comparação e atribuição que, em Prolog, é o principal tipo de tratamento dispensado aos dados. A unificação pode suceder ou falhar de acordo com as seguintes regras :

- A unificação de duas constantes será o resultado de sua comparação. Se as duas constantes são iguais, então a unificação sucede, caso contrário, falha.

- A unificação de uma variável livre com qualquer outro objeto se traduz na atribuição do objeto à variável, de modo que qualquer outra referência à variável em questão será na realidade uma referência ao objeto à ela instanciado. A unificação de um objeto com uma variável livre sempre sucede.

- A unificação de duas variáveis livres sempre sucede e fará com que a variável mais nova passe a referenciar a variável mais antiga.

- A unificação de uma variável instanciada com um objeto é na realidade a unificação de dois objetos. Deste modo recaímos em um dos outros casos descritos.

- A unificação de duas estruturas sucede se seus funcionais e aridades são idênticos e se os seus argumentos unificam, isto é, se o i-ésimo argumento da primeira estrutura unifica com o i-ésimo argumento da segunda estrutura.

A tabela abaixo descreve resumidamente todos os casos possíveis em uma unificação :

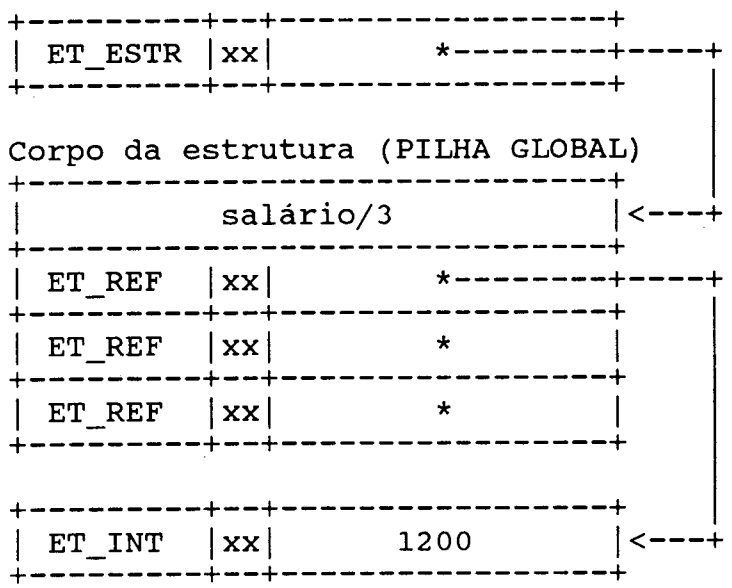
OBJETOS	Constante C1	Variável X1	Estrutura E1
Constante C2	Sucede se C1 == C2	Sucede faz X1 = C2	Falha
Variável X2	Sucede faz X2 = C1	Sucede faz X1 = X2	Sucede faz X2 = E1
Estrutura E2	Falha	Sucede faz X1 = E2	Sucede se f(E1) == f(E2), a(E1) == a(E2) e unifica args

2.5 Representação interna de variáveis no Pronus Prolog.

Na sintaxe Prolog, as variáveis são reconhecidas por iniciarem com letra maiúscula. A representação interna de variáveis no interpretador Pronus Prolog varia de acordo com o contexto em que a variável está sendo utilizada.

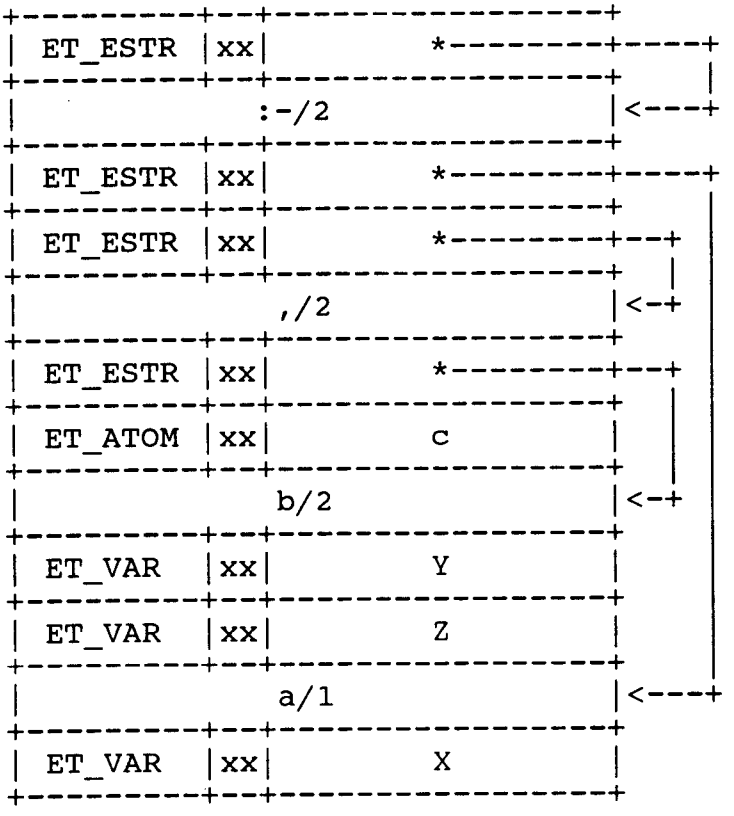
2.5.1 Variáveis a tempo de execução.

A tempo de execução, uma variável é representada por uma referência e pode estar livre ou instanciada. Uma variável está livre quando ainda não foi associado nenhum valor a ela; caso contrário ela é dita instanciada. A desreferência de uma variável instanciada aponta para o objeto à ela associado, enquanto que a desreferência de uma variável livre aponta para um objeto que é uma referência para si mesmo. Observe o exemplo abaixo, em que descrevemos a representação da estrutura "salário(X, Y, Z)", onde a variável X está instanciada com o inteiro 1200 e as variáveis Y e Z estão livres (Referências que apontam para si mesmas são representadas por um "*" sem seta).



2.5.2 Variáveis associadas a símbolos.

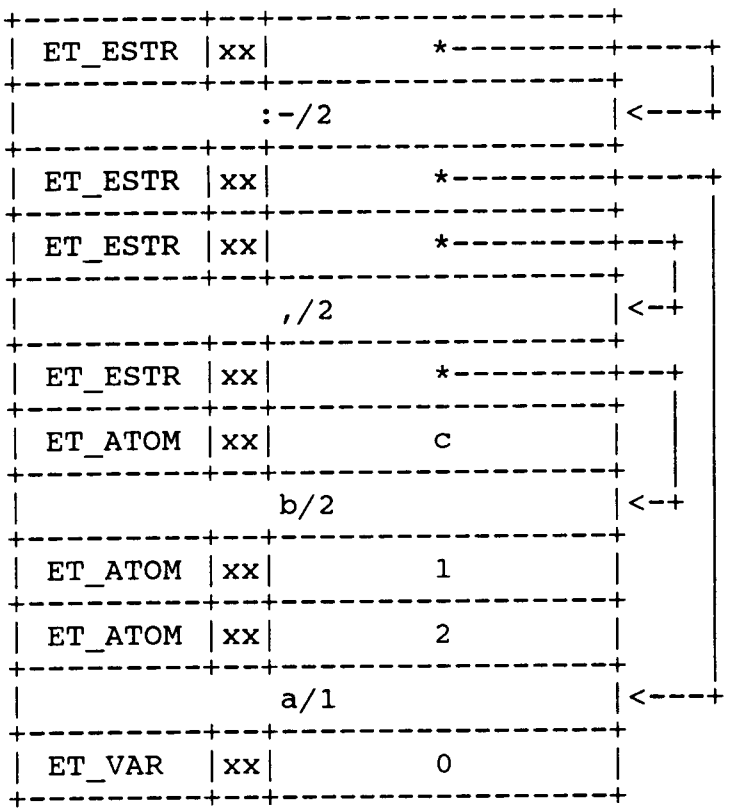
Como já vimos, o campo tcl_repres da tabela de cláusulas aponta para a representação da cláusula armazenada na área de código. Neste tipo de representação, as variáveis são denominadas variáveis-símbolo e representadas por objetos com etiqueta ET_VAR, cujo campo valor contém o índice da posição na tabela de símbolos onde está armazenado o nome da variável. Observe o exemplo abaixo onde descrevemos a representação da cláusula "a(X) :- b(Y,Z), c.". Ao longo deste texto, por questão de simplicidade, as variáveis-símbolo serão representadas nos esquemas por objetos com etiqueta ET_VAR, contendo em seu campo valor o próprio nome da variável ao invés do índice da tabela de símbolos.



2.5.3 Variáveis durante a geração de código.

Durante a geração de código, as variáveis são denominadas variáveis-código e representadas por objetos com etiqueta ET_VAR cujo campo valor aponta para a posição da tabela de variáveis associada àquela variável. Na tabela de variáveis estão contidas in-

formações importantes para a geração do código. Observe o esquema abaixo, onde mostramos a representação para geração de código da cláusula "a(X) :- b(Y,Z), c", onde consideramos as variáveis "X", "Y" e "Z" associadas às posições 0, 1 e 2 da tabela de variáveis respectivamente.



2.6 Conversão da representação de variáveis.

Como vimos na seção anterior, existem três maneiras diferentes de se representar variáveis no interpretador Pronus Prolog, dependendo do contexto em que esta variável está sendo utilizada. Na implementação de alguns predefinidos é necessária a conversão de uma forma de representação para outra. Esta seção descreve os algoritmos utilizados para este fim.

2.6.1 Conversão referências -> variáveis-código.

Este algoritmo tranforma a representação de variáveis utilizada a tempo de execução para a representação utilizada durante a geração de código. O termo a ser convertido é copiado da área de

memória origem para a área destino associando variáveis livres a posições na tabela de variáveis. O código abaixo descreve detalhadamente apenas a parte do algoritmo que converte as variáveis livres :

```
static    int    indreg;        /* Número variáveis convertidas*/

ocp_rvar (destino, origem)
M_Endereco     destino;         /* Área de memória destino     */
M_Endereco     origem;          /* Área de memória origem      */
{
       M_Endereco    dref;      /* Resultado da desreferência  */
       M_Objeto      objeto;    /* Um objeto                   */
       int           tipo;      /* Tipo do objeto              */

(1)    objeto = m_le_o (origem);
(2)    tipo = M_TIPO (objeto);

       Se tipo == ET_REF
       {
(3)        dref = desrefencia (origem);

           Se dref == origem      /* Se é uma variável livre   */
           {
                int         inds; /* Índice da tabela símbolos */
                int         indv; /* Índice da tabela variáveis*/
                Ts_Simbolo  *ts;  /* Ponteiro p/ tabela símbolo*/
                M_Objeto variável;/* Um objeto                 */

(4)             indreg = indreg + 1;
(5)             A[indreg] = M_OBJ (ET_REF, dref);
(6)             Cria um nome para a variável;
(7)             inds = ts_instale (inds);
(8)             ts = ts_desindexe (inds);
(9)             indv = tv_instale (ts);
(10)            variável = M_OBJ (ET_VAR, indv);
(11)            m_esc_o (dref, variável);
(12)            m_esc_o (TR, dref);
(13)            TR += 2;
(14)            m_esc_o (destino, variável);
           }
           senão
               Copia o conteúdo de dref para destino;
       }
       senão
           Copia o termo apontado por origem para destino;
}
```

Descrição das linhas.

(1) - Lê o objeto apontado por "origem" na memória virtual.

(2) - Pega o tipo do objeto lido.

(3) - Desreferencia a posição apontada por "origem", colocando o resultado em "dref". Note que se "dref" contiver o mesmo valor que "origem" então o objeto é uma variável livre, uma vez que uma variável livre é um objeto que aponta para si mesmo.

(4) e (5) - Incrementa o número de variáveis livres convertidas e associa a i_ésima variável livre convertida ao i_ésimo registro A.

(6) - A tempo de execução, as variáveis são apenas referências, não podendo ser associadas a um nome. Devemos então criar um nome para a nova variável a ser instalada na tabela de variáveis.

(7) e (8) - Aqui o nome criado é instalado na tabela de símbolos. O índice da nova posição da tabela de símbolos é armazenado em "inds" e utilizado para calcular o ponteiro "ts".

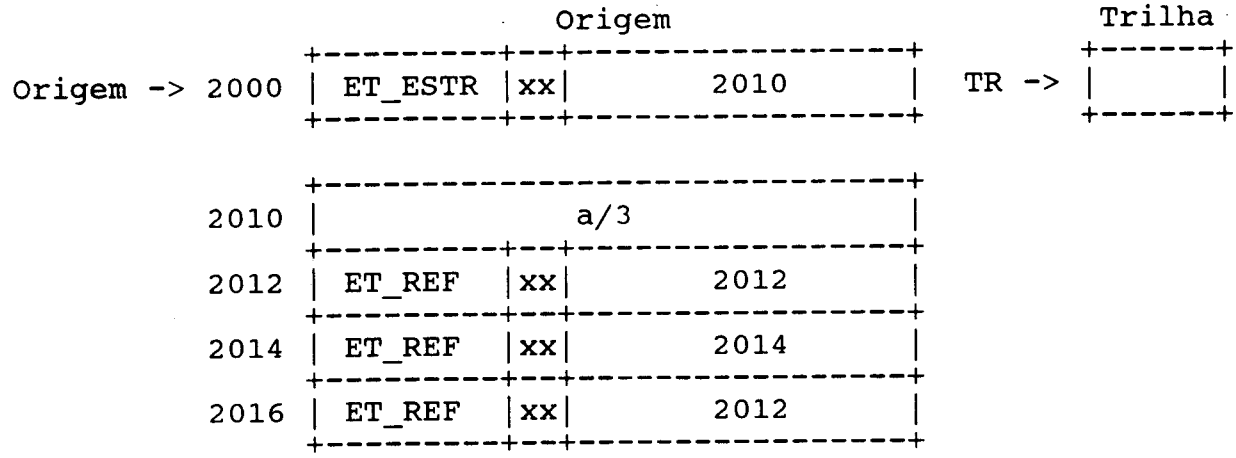
(9) - Aqui nós o criamos uma nova posição na tabela de variáveis cujo índice será colocado em "indv". O valor de "ts", fornecido como argumento, será copiado para o campo tva_ptsimb da tabela de variáveis.

(10) - Uma vez determinado o índice da posição criada na tabela de variáveis, nós podemos montar a representação da variável-código. É montado um objeto com etiqueta ET_VAR cujo campo valor contém o novo índice da tabela de variáveis.

(11), (12) e (13) - A representação da variável-código criada é copiada para o endereço onde antes estava a variável livre, alterando assim a representação do termo que está sendo convertido. Deste modo, estaremos garantindo que qualquer outra referência para este endereço será traduzida para a nova representação. Os endereços onde estavam colocadas referências livres e que foram alterados devem ser anotados na trilha. Ao final da conversão, as informações salvas na trilha serão utilizadas para recuperar a antiga representação do termo que está sendo convertido.

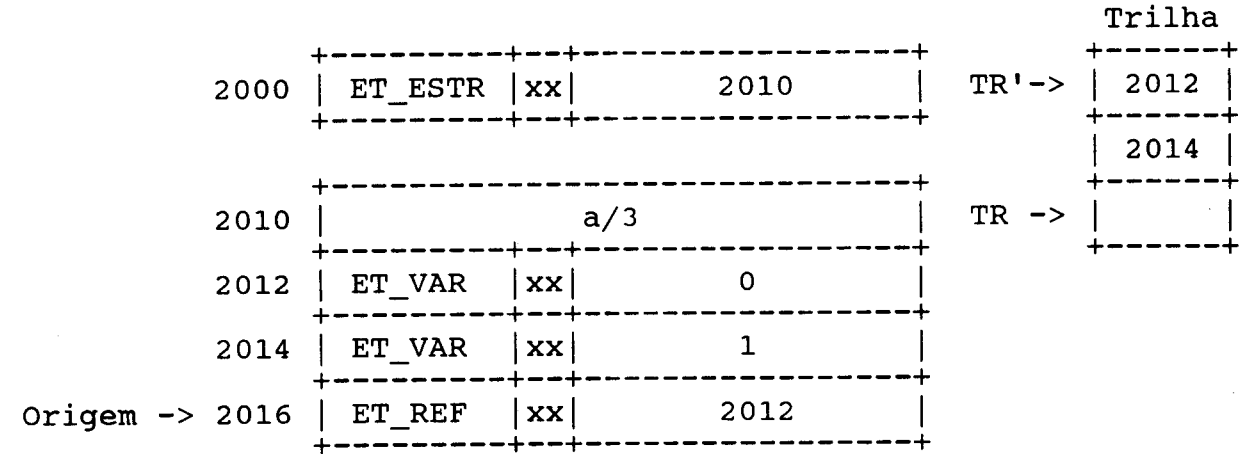
(14) - Copia a nova representação da variável para destino;

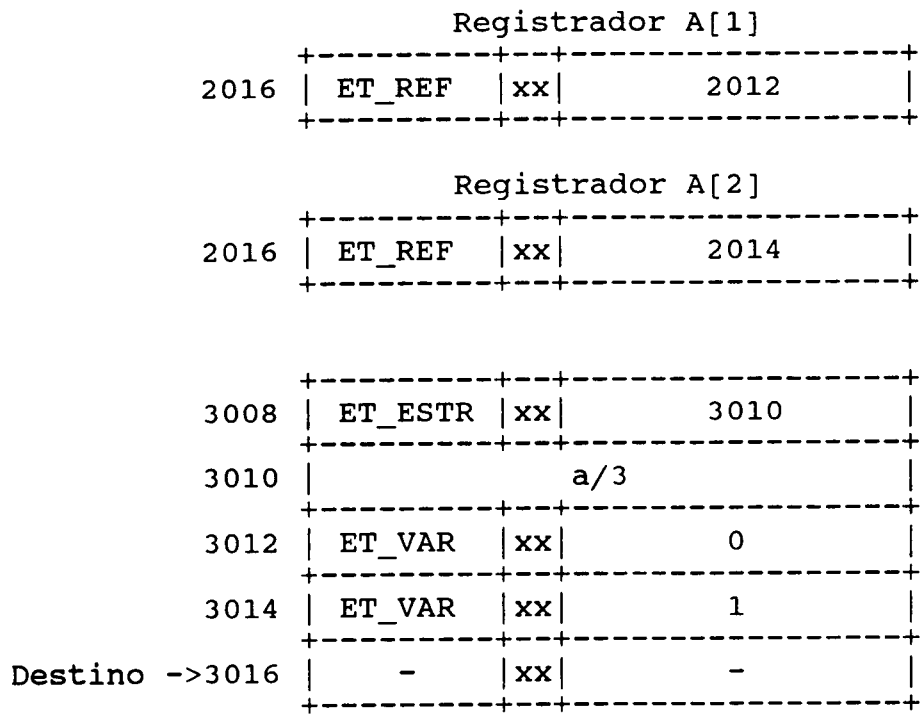
A título de exemplo, vamos converter a estrutura "a(X, Y, X)", onde as variaveis "X" e "Y" são livres. O esquema abaixo descreve a estrutura apontada por "origem" e a situação da trilha no início da conversão. Os endereços são utilizados apenas para ilustrar o exemplo :



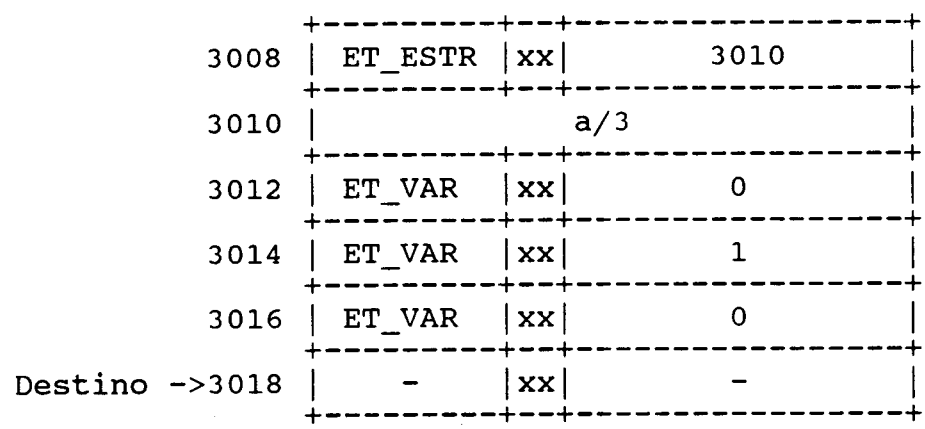
O argumento "destino" aponta para uma área de memória livre que deverá conter a nova representacão da estrutura apontada por "origem". Inicialmente o cabeçalho da estrutura é copiado e os argumentos da estrutura são analisados.

Os dois primeiros argumentos da estrutura são variáveis livres, e por isso devem ser associados às posições 0 e 1 da tabela de variáveis. O esquema abaixo representa como estão os registradores A[1] e A[2], a trilha e as áreas de memória apontadas por "origem" e "destino" após a conversão dos dois primeiros argumentos da estrutura.





O último argumento a ser analisado é uma referência para uma variável livre que já foi convertida. Deste modo, o algoritmo simplesmente copiará a nova representação da variável. O esquema abaixo mostra como fica a estrutura ao final da conversão :



Ao final da conversão, devemos restaurar a estrutura apontada por "origem". Todas as variáveis livres foram substituídas por objetos com etiqueta ET_VAR apontando para a tabela de variáveis. Os endereços dos objetos onde estavam colocadas estas variáveis livres estão armazenados na trilha, entre os apontadores TR' e TR de modo que, para restaurá-las, basta escrever em cada um destes endereços uma referência que aponta para si mesma.

2.6.2 Conversão variáveis-código -> referências.

Este algoritmo converte a representação de variáveis utilizada a tempo de geração de código para a representação de variáveis utilizada a tempo de execução. O termo a ser convertido é percorrido substituindo-se variáveis código por referências que apontam para si mesmas. O código abaixo descreve a implementação deste algoritmo :

```
ocp_rvar (termo)
M_Endereco     termo;          /* Percorre o termo              */
{
       M_Objeto      objeto;   /* Um objeto                     */
       int           tipo;     /* Tipo do objeto                */
       int           indv;     /* Índice tabela de variáveis    */

(1)    objeto = m_le_o (termo);
(2)    tipo = M_TIPO (objeto);

       Se tipo == ET_VAR
       {
(3)        indv = M_VALOR (termo);

   +--     Se (tv_pvar[indv].tva_ref == M_NULO)
   |       {
(4)|            tv_pvar[indv].tva_ref = termo;
   |            m_esc_o (termo, M_OBJ (ET_REF, termo));
   +--     }
           senão
(5)            m_esc_o(termo,M_OBJ(ET_REF,tv_pvar[indv].tva_ref));
       }
       senão
           Percorre o termo;
}
```

Descrição das linhas.

(1) - Lê o objeto apontado por "termo".

(2) - Pega o tipo do objeto lido.

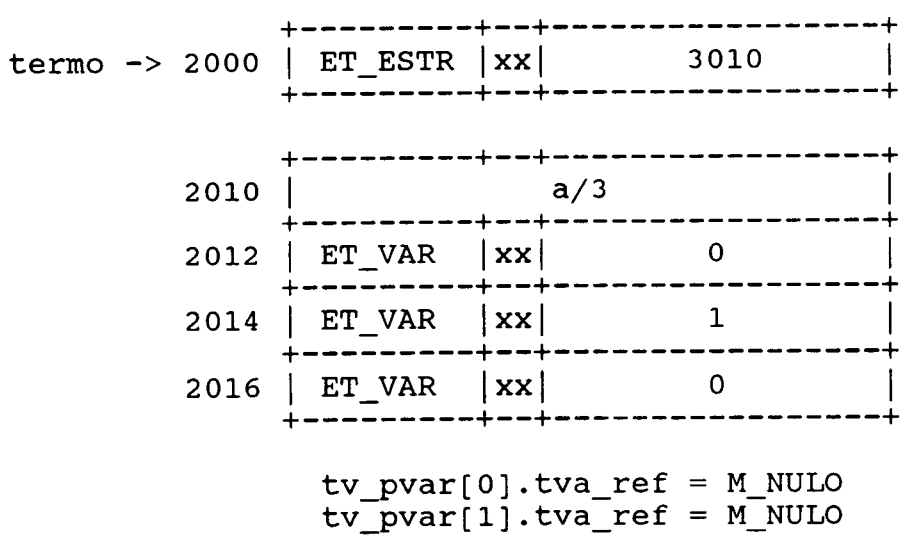
(3) - Se a etiqueta do objeto for ET_VAR então seu campo valor conterá um índice da tabela de variáveis. Este índice é colocado na variável "indv".

(4) - O campo "tva_ref" da tabela de variáveis é utilizado para controlar a conversão de variáveis em referências. Se o seu conteúdo for o endereço virtual M_NULO, então a variável-código

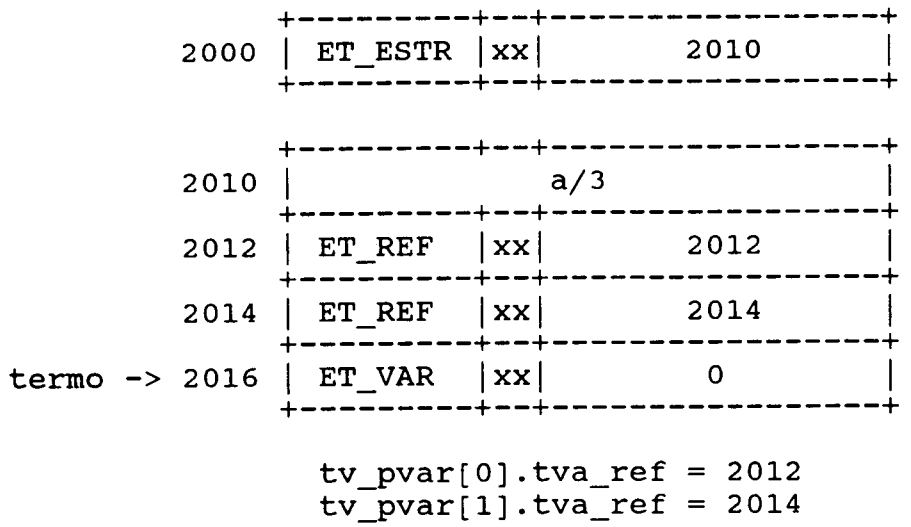
associada a esta posição da tabela de variáveis ainda não foi convertida em variável livre. Deste modo, o objeto apontado por "termo" é alterado, passando a conter uma referência que aponta para si mesma. O valor de "termo" é anotado no campo "tva_ref".

(5) - Se o valor do campo "tva_ref" é diferente do endereço virtual M_NULO, então a variável-código encontrada já está associada a uma variável livre. Esta variável livre está localizada no endereço apontado por "tva_ref". Deste modo, basta substituir a variável-código por uma referência para o objeto apontado por "tva_ref".

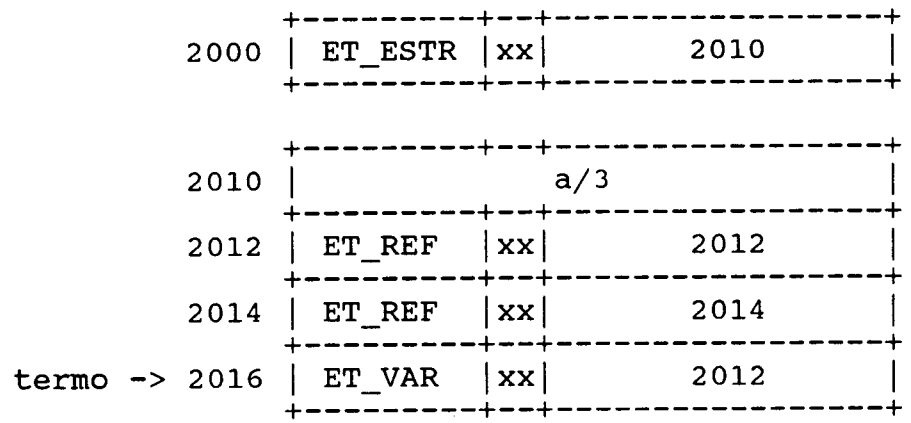
Vamos utilizar como exemplo novamente a estrutura "a(X ,Y , X)", onde as variáveis "X" e "Y" estão associadas às posições 0 e 1 da tabela de variáveis respectivamente. O esquema abaixo descreve a estrutura apontada por "termo" e o conteúdo dos campos "tva_ref" nas posições 0 e 1 da tabela de variáveis antes de ser aplicado o algoritmo de conversão :



Os dois primeiros argumentos da estrutura são convertidos em variáveis livres e seus endereços são anotados na tabela de variáveis. O esquema abaixo mostra a estrutura e os campos "tva_ref" após a conversão dos dois primeiros argumentos da estrutura :



O último argumento da estrutura é uma variável-código que já foi associada a uma variável livre localizada no endereço 2012. Assim, o terceiro argumento é substituído por uma referência para esta posição. O esquema abaixo mostra a estrutura ao final da conversão :



2.6.3 Conversão variáveis-código -> variáveis-símbolo.

Este algoritmo transforma variáveis-código em variáveis-símbolo. O algoritmo copia o termo a ser convertido da área de memória "origem" para a área de memória "destino", substituindo os índices da tabela de variáveis por índices da tabela de símbolos calculados a partir do campo "tva_ptsimb" da tabela de variáveis. O código abaixo descreve a implementação deste algoritmo:

```
ocp_vsimb (destino, origem)
M_Endereco      destino;        /* Área de memória destino      */
M_Endereco      origem;         /* Área de memória origem       */
{
        M_Objeto        objeto;     /* Um objeto                    */
        int             tipo;       /* Tipo do objeto               */
        int             inds;       /* Índice tabela de símbolos    */
        int             indv;       /* Índice tabela de variáveis   */

(1)     objeto = m_le_o (origem);
(2)     tipo = M_TIPO (objeto);

        Se tipo for ET_VAR
        {
(3)         indv = M_VALOR (objeto);
(4)         inds = ts_indexe (tv_pvar[indv].tva_ptsimb);
(5)         m_esc_o (destino, M_OBJ (ET_VAR, inds));
        }
        senão
            Copia o termo apontado por origem para destino;

}
```

Descrição das linhas.

(1) - Lê o objeto apontado por "origem".

(2) - Pega o tipo do objeto lido.

(3) - Se o tipo do objeto é ET_VAR, então o campo valor do objeto é um índice da tabela de variáveis. Este índice é colocado em "indv".

(4) - O campo "tva_ptsimb" da posição da tabela de variáveis referenciada pelo índice "indv" é utilizado para calcular o índice do campo da tabela de símbolos onde está contido o nome da variável.

(5) - A nova representação da variável é copiada em "destino".

2.6.4 Conversão variáveis-símbolo -> variáveis-código.

Este algoritmo copia o termo a ser convertido para outra área de memória, convertendo variáveis-código em variáveis-símbolo. O código abaixo descreve a implementação deste algoritmo :

```
ocp_simbvar (destino, origem)
M_Endereco     destino;          /* Área de memória destino     */
M_Endereco     origem;           /* Área de memória origem      */
{
       M_Objeto       objeto;    /* Um objeto                   */
       int            tipo;      /* Tipo do objeto              */
       int            inds;      /* Índice tabela de símbolos   */
       int            indv;      /* Índice tabela de variáveis  */
       Ts_Simbolo     *ts;       /* Ponteiro tabela símbolos    */

(1)    objeto = m_le_o (origem);
(2)    tipo = M_TIPO (objeto);

       Se tipo for ET_VAR
       {
(3)        inds = M_VALOR (objeto);
(4)        ts = ts_desindexe (inds);
(5)        indv = tv_inclua (ts);
(6)        m_esc_o (destino, M_OBJ (ET_VAR, indv));
       }
       senão
           Copia o termo apontado por origem para destino;
}
```

Descrição das linhas.

(1) - Lê o objeto apontado por "origem".

(2) - Pega o tipo do objeto lido.

(3) - Se o tipo do objeto é ET_VAR, então o campo valor do objeto é um índice da tabela de símbolos. Este índice é armazenado em "inds".

(4) - O índice da tabela de símbolos é utilizado para calcular o ponteiro "ts".

(5) - Uma nova entrada na tabela de variáveis é criada e seu índice colocado na variável "indv". O ponteiro "ts" será armazenado no campo "tva_ptsimb" da nova entrada da tabela de variáveis.

(6) - A nova representação da variável é copiada em "destino".

2.7 O módulo Mestre.

O módulo Mestre é responsável pela coordenação das atividades do interpretador. Através deste módulo é que se iniciam tarefas como geração de código, consultas e inserção de cláusulas na base de dados. O código abaixo descreve o funcionamento do módulo Mestre. Os detalhes serão descritos nas seções abaixo.

```
me_mestre (modo)
int       modo;                  /* Modo funcionamento do Mestre */
{
    Salve os registros;

    Enquanto não é fim de arquivo ou não é uma chamada recursiva
    {
        Se é fim de arquivo
            sinalize fim de cláusula.

        Reinicialize a tabela de variáveis;
        Leia um termo;

        Se leu o átomo fim de arquivo
        {
            Se é uma chamada recursiva
            {
                Recupera os registros;
                Retorne;
            }
            senão
                Sinalize termo lido como nulo;
        }

        Se o modo é ME_INTERATIVO
            Salva as variáveis explícitas;

        Se o termo lido não é nulo
        {
            Compatibilize a entrada;
            Crie a representação da cláusula;
            Gere o código da cláusula;
```

```
        Se o predicado da cabeça do termo
              compatibilizado é o predicado gol
           {
               Se não é uma chamada recursiva
                    Reinicia a tabela de predicados;

                  Se o modo é ME_INTERATIVO
                    Chama uma consulta interativa;
               senão
               {
                    Guarde o arquivo de leitura corrente;
                    Redirecione o analisador léxico para
                    o teclado;
                    Chame uma consulta não interativa;
                    Redirecione o léxico para o arquivo
                    de leitura guardado;
               }
           }
           senão
           {
               Se o modo é ME_RECONSULT e o predicado do
                  termo lido não está marcado
               {
                    Retira todas as cláusulas do predicado;
                    Marca o campo pre_reconsult do predicado;
               }

               Instale a cláusula no predicado
               a que ela pertence;
           }
    }

    Recupere os registros;

    Se é fim de arquivo e o modo é ME_INTERATIVO e
       o arquivo de leitura padrão não está direcionado
       para o teclado
       Redirecione o arquivo de leitura padrão
       para o teclado;

}
```

2.7.1 Modos de funcionamento e níveis de recursão.

O módulo Mestre pode ser chamado recursivamente durante uma execução por alguns predefinidos. O contador me_nivel indica se estamos na instância principal do Mestre (me_nivel = 1) ou em uma chamada recursiva (me_nivel > 1). Uma instância do Mestre é desativada ao ser encontrado um fim de arquivo, com exceção da instância principal que só é desativada através do predefinido halt/0. O parâmetro modo seleciona um dos seguintes modos de funcionamento do módulo Mestre :

modo ME_INTERATIVO : Neste modo de funcionamento, o Mestre dispara execuções interativas (que imprimem mensagens ao final das execuções) e imprime a mensagem "registrado" quando uma cláusula é instalada na base de dados. As execuções serão analisadas em detalhes na seção 2.7.5.

modo ME_NITERAT : Neste modo de funcionamento, o Mestre dispara execuções não interativas (que não imprimem mensagens) e não imprime qualquer tipo de mensagem após a inserção de uma cláusula na base de dados.

modo ME_RECONSULT : Este modo de funcionamento só é utilizado pelo predefinido reconsult/1. Este modo é semelhante ao modo ME_NITERAT e será tratado em detalhes na seção 3.8.2.

2.7.2 Leitura de um termo.

A leitura de um termo é realizada pelos módulos de análise (análise de expressões, análise sintática e análise léxica). Ao final da leitura de um termo sintáticamente correto, o Mestre recebe um ponteiro para a posição na pilha global onde está localizada uma estrutura que representa o termo lido.

Caso o termo lido contenha variáveis, elas estão representadas por variáveis-código (representação de variáveis para geração de código). Note que a tabela de variáveis é inicializada a cada leitura, de modo que a alocação de posições na tabela de variáveis coincide sempre com a ordem em que as variáveis aparecem no termo lido, isto é, a primeira variável do termo ocupa sempre a posição zero da tabela de variáveis e assim por diante.

2.7.3 Compatibilização do termo.

A compatibilização consiste em se alterar o termo lido pelos módulo de análise, de modo que a estrutura montada seja sempre da forma "cabeça :- corpo", e esteja associada a um predicado. O processo de compatibilização varia de acordo com o modo de funcionamento do módulo Mestre. O quadro abaixo descreve os casos possí-

veis na compatibilização.

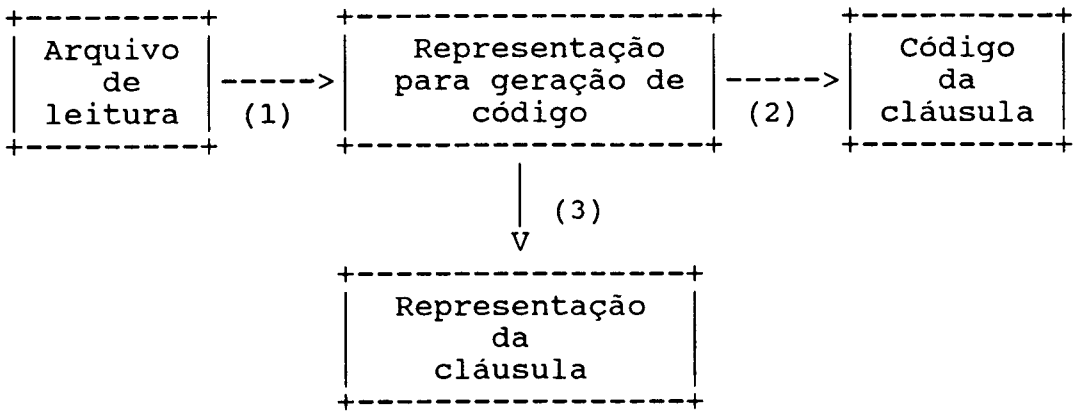
Tipo do termo	Compatibilização modo ME_INTERATIVO	Compatibilização outros modos
ET_ATOM	<2> gol :- átomo	<3> átomo :- true
ET_LIST	<2> gol :- lista	ERRO
ET_ESTR		
:-X	<2> gol :- X	<2> gol :- X
X :- Y	<1> X :- Y	<1> X :- Y
?-X	<2> gol :- X	<2> gol :- X
X	<2> gol :- X	<3> X :- true
Outros	ERRO	ERRO

<1> - Cláusula.
<2> - Consulta.
<3> - Fato.

2.7.4 Representação da cláusula e geração de código.

Uma vez compatibilizado o termo lido, temos uma cláusula associada a um predicado. Neste momento, devemos gerar código para a cláusula e criar a sua representação. O código e a representação da cláusula são obtidos a partir da estrutura montada pela compatibilização.

Na estrutura resultante da compatibilização, as variáveis estão associadas a posições da tabela de variáveis (variáveis-código), enquanto que na representação da cláusula, as variáveis devem estar associadas a posições da tabela de símbolos (variáveis-símbolo). Deste modo, para obtermos a representação da cláusula a partir da estrutura resultante da compatibilização, devemos aplicar o algoritmo que converte variáveis-código em variáveis-símbolo. O esquema abaixo descreve todas as etapas já analisadas do módulo Mestre.



(1) - Leitura do termo.
(2) - Geração de código.
(3) - Cópia e conversão.

2.7.5 Disparando uma consulta.

Após a geração de código, devemos analisar a cláusula resultante da compatibilização para determinar se devemos disparar uma consulta ou se devemos instalar a nova cláusula na base de dados. Se a cláusula gerada pela compatibilização tiver como cabeça uma estrutura com funcional "gol", então esta cláusula representa uma consulta. Ao iniciar uma consulta, devemos gerar um código de interface entre a execução Prolog e o interpretador. No caso de uma execução interativa, o código de interface está descrito abaixo :

```
(1)      try  R,0
(2)      trap 0
(3)  R : allocate
(4)      put_n_variables   NEXP
(5)      call_goal  ENDER, NEXP
(6)      trap 1
```

Início da consulta.

(1) - Cria um ponto de escolha desviando a execução para a intrução (3)

(3) - Cria um ambiente na pilha local.

(4) - Cria no ambiente NEXP variáveis livres, onde NEXP é o número de variáveis presentes na consulta. Estas variáveis são denominadas variáveis da consulta e serão utilizadas para armazenar as soluções encontradas.

(5) - Dispara a execução do código da consulta.

Término com sucesso.

(6) - Imprime a solução encontrada associando os nomes das variáveis aos valores armazenados no ambiente criado pela pseudo-instrução (3). As variáveis anônimas (representadas por "_") devem ser ignoradas. Se o usuário pedir uma nova solução através de um ponto e vírgula (;), esta trap falha causando um retrocesso. Se não for pedida uma nova solução o interpretador escreve a mensagem "Sim" na tela e a execução é encerrada . Ao longo deste texto, nós passaremos a referenciar a trap um através do nome de trap sucesso.

Término com falha.

(2) - Imprime a mensagem "Não" na tela, indicando a ocorrência de uma falha na execução do objetivo. Ao longo deste texto nós iremos referenciar a trap zero através do nome trap falha.

No caso de uma execução não interativa, o código de interface entre o interpretador e a execução Prolog difere do código utilizado na execução interativa. O conjunto de pseudo-instruções abaixo descreve este código.

```
(1)      try_me_else    R,0
(2)      continue       0
(3)      allocate
(4)      call_goal  ENDER, NEXP
(5)      cut
(6)      deallocate
(7)  R : trust_me_else_fail
(8)      trap 2
```

Início da consulta.

(1) - Cria um ponto de escolha.

(2) - Faz com que o registro de corte CR aponte para o ponto de escolha criado pela instução (1).

(3) - Cria um ambiente.

(4) - Dispara a execução do código da consulta.

Término com sucesso.

(5) - Descarta todos os pontos de escolha abaixo do registro CR.

(6) - Descarta o ambiente criado pela pseudo-instrução (3).

(7) - Descarta o ponto de escolha criado pela pseudo-instrução (1).

(8) - Chama a rotina que finaliza uma execução não interativa.

Término com falha.

A ocorrência de uma falha no objetivo da consulta, descarta todos os pontos de escolha localizados abaixo daquele criado pela pseudo-instrução (1).

(7) - Descarta o ponto de escolha criado pela pseudo-instrução (1).

(8) - Chama a rotina que finaliza uma execução não interativa.

2.7.6 Instalando uma cláusula.

Se, ao analisar a estrutura montada pela compatibilização, chegarmos à conclusão que não foi pedida uma consulta, então o termo lido é uma cláusula que deve ser instalada na base de dados.

Para instalar uma nova cláusula devemos alocar dinâmicamente uma nova estrutura Tc_Claus, onde guardaremos a representação e o código da cláusula. A nova cláusula deverá ser instalada na lista de cláusulas do seu predicado correspondente.

3. Conclusão

Os mecanismos descritos neste relatório formam, em conjunto com os módulos de análise, compilação e gerência de código, o núcleo básico do interpretador que já é capaz de traduzir e executar programas Prolog digitados via teclado. Entretanto, para que o interpretador se torne uma ferramenta realmente útil de desenvolvimento de aplicações em Prolog torna-se importante fornecer uma série de facilidades ainda não implementadas representadas por predicados predefinidos.

O próximo passo na implementação do interpretador será definir quais os predefinidos que deverão ser implementados e como deverá ser feita a interface entre uma execução Prolog e a biblioteca de predefinidos. A partir daí deverá se seguir uma longa fase em que serão implementados os diversos grupos de predefinidos.

4. Referências bibliográficas.

[Warren 83] David H. D. Warren,
An Abstract instruction set,
Tecnical note 309,
SRI International,
Menlo Park, California, Outubro de 1983.

[Souza 88] Pseudo-Instruções Prolog e seu ambiente de execução,
Luiz Fernando Pereira de Souza,
Valério Machado Dallolio,
Relatório Técnico NCE/UFRJ, Rio de Janeiro - RJ,
outubro de 1988,
código NCE-12/88, Parte I 29p. e parte II 40p.

[Clocksin 81] Programming in Prolog,
W. F. Clocksin,
C. S. Mellish,
Springer-Verlag,
Berlin Heidelberg, 1981, 281p.

[Dallolio 89] Pronus Prolog - Arquitetura e implementação,
Valério Machado Dallolio,
Luiz Fernando Pereira de Souza,
Relatório Técnico NCE/UFRJ, Rio de Janeiro - RJ,
março de 1989,
código NCE-01/89, 24p.

[Dallolio 91a] Ativação dos predefinidos,
predefinidos de E/S,
e controle de fluxo no Pronus Prolog,
Luciano Conrado Machado Dallolio,
Luiz Fernando Pereira de Souza,
Relatório técnico NCE/UFRJ, Rio de Janeiro - RJ,
código NCE-11/91, 46p.

[Dallolio 91b] Predefinidos para coleta de soluções e manipulação da base de dados no Pronus Prolog, Luciano Conrado Machado Dallolio, Luiz Fernando Pereira de Souza, Relatório técnico NCE/UFRJ, Rio de Janeiro - RJ, código NCE-12/91, 29p.