SICS/T-89/0004

The Instruction Set for the GCLA Abstract Machine

by

Martin Aronsson

SICS technical report
T89004
ISSN 1100-3154

The Instruction Set for the GCLA Abstract Machine

by

Martin Aronsson
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
email:martin@sics.se

Tel: +46 8 752 15 00

Abstract

GCLA is a new programming language, which increases expressiveness compared with traditional logic programming languages and functional programming languages. The basis for the language is a generalization of the concept *inductive definitions*, called *partial inductive definitions*. The program defines a logic, which is used to make inferences to prove if a query holds or not. This report first presents a short introduction to these ideas. Then, an abstract machine, called GAM, for GCLA is presented; the instructions as well as an introduction to the compiling schema is given together with some examples. The main idea is to extend the Warren Abstract Machine (WAM), which is an abstract machine for the language Prolog.

Keywords: Logic programming, Functional programming, Inductive definitions, Abstract machine

Contents:

1. Introduction
2. GCLA the Language
2.1 Syntax
2.2 Operational Semantics
2.3 Procedural Semantics
3. An Example Deduction
4. Some Programming Examples
5. The Ideas Behind the Abstract Machine (GAM)
6. An Introductory Example of Compiled Code
7. Data Areas and Registers
8. The Clause Orders
9. Declarations
10. The Procedural Instructions
10.1 Calling Instructions
10.2 Premise Instructions
10.3 Indexing Instructions
10.4 Choicepoint Instructions
10.5 Guard Instructions
10.6 Fail Instruction
11. The Unification Instructions
11.1 Get Instructions
11.2 Put Instructions
11.3 Unify Instructions
12. Realization of the Procedural Semantics
13. Compiling Schema
14. Some Examples
15. Conclusions and Future Work
References

Appendices:

A Formats of Different Frames:
A.1 Choicepoints
A.2 Environments
A.3 Continuations
A.4 Premises
A.5 Different trail-formats

B Transitions Describing GAM
B.1 Calling Instructions
B.2 Premise Instructions
B.3 Indexing Instructions
B.4 Choicepoint Instructions
B.5 Guard Instructions
B.6 Fail Instruction
B.7 Get Instructions
B.8 Put Instructions
B.9 Unify Instructions
B.10 Rule Transitions

1. Introduction

During the past years logic programming has proved to be a good framework for writing programs with high expressive power together with the ability to execute programs efficiently. The key to this combination is to use definite horn clauses in programs, and to use resolution as the computation rule. Generalized horn clauses (GCLA) [2,7] is a generalization of definite horn clauses. It is not a generalization within the traditional logical framework, but rather a more proof-theoretic view of horn clause programming. The program is not looked upon as a set of true facts and implications, but as a definition determining the possible inferences. One could say that in GCLA the program forms the system´s mind, meaning that the inference rules are given by the program. In Prolog the inference rule is given a priori and the program is executed within this given framework. The inductive definitions [6] which form the basis of GCLA are in a sense more primitive than ordinary logic. Primitive in the sense that when writing down the definition of any kind of logic one is actually writing down an inductive definition. Several experimental interpreters for GCLA have been implemented in Prolog, and they have shown some of the expressive power of GCLA. But they have also suffered from inefficient treatment of the rules introduced by the extension of the definite hornclauses. Therefore the development of an abstract machine was interesting, to see if a suitable instruction set for GCLA could be found. This would mean increasing the expressive power without losing efficiency compared to for example Prolog. The idea is to extend the Warren Abstract Machine (WAM) [9], which is an abstract machine for Prolog.
This report does not describe a garbage collection routine, although one is needed. This will not differ from already existing ones, and in fact, the garbage collecting routine for the heap (which is the only area that should be garbage collected) is the same as for the WAM (for example see [1] and [3]).
When reading this report, be aware that there is a difference between the P´s. When we refer to P, we refer to a program, but when we refer to P⊢ or ⊢P we refer to a specific rule, and when we refer to $\vdash_P$, we refer to the logic "created" or "generated" by the program P. We often omit the index P in $\vdash_P$ although it should actually be written out to be correct.
This work is a part of a larger project, which also contains a programming calculus based on partial inductive definitions [5], theoretical foundations for Generalized Horn clauses as a programming language [7] and the definition of a new programming language [2].

2. GCLA the Language

2.1 Syntax

The syntax for a GCLA program is extended "ordinary Prolog" syntax (i.e. DEC-10 syntax). The clauses in the program look like ordinary Prolog clauses, except that they can have a list of unifying guards as a first element in the body, and a new primitive operator, ->, can occur in the body of a clause. Another new primitive operator, |-, is introduced for the goals, which are sequents consisting of assumptions and a conclusion, instead of just a conclusion as in Prolog.
A difference in reading the clauses is that the operator ":-" should be read as "is defined by" rather than "if" as in Prolog in order to grasp the intuition of inductive definitions.
The syntax is the same as the one given in [2].

2.1.1 Variables

- A variable is a string beginning with an uppercase letter.

2.1.2 Functors

- A functor is a string beginning with a lower case letter. Each functor has a certain arity. A constant is a functor of arity 0.

2.1.3 Terms

- Each variable X is a term.
- If $t_1 ... t_n$ are terms and f is an n-ary functor, then $f(t_1 ... t_n)$ is a term.

2.1.4 Guards

If t is a term, then:
- *var(t)* is a guard.
- *nonvar(t)* is a guard.
- *number(t)* is guard.
- *atom(t)* is a guard.
- *atomic(t)* is a guard.
- *ground(t)* is a guard.

The only guards that are allowed are these predefined guards.
They are not of the same kind as in the parallel logic languages (GHC and others). These guards are used to constrain variables in the head of clauses, and not to rule out other possible clauses.

2.1.5 Conditions and Clauses

- Each term is a condition.
- If $C_1, ..., C_n, C$ are conditions, then $(C_1, ..., C_n) \rightarrow C$ is a condition.

- If C is a condition and n a number, then *con(n,C)* is a condition (*con* stands for contraction, see 2.2.7).

- If $C_1, \ldots, C_n$ are conditions, $g_1, \ldots, g_m$ guards and t is a term, then
t :- $[g_1, \ldots, g_m]$, $C_1, \ldots, C_n$. is a clause.
We will refer to t as the *head* of the clause and $C_1, \ldots, C_n$ as the *body* of the clause. We will use the word *predicate* for the set of all clauses in the program that have the same principal functor in the head.

t. will be short for the clause t :- [ ].
t :- $C_1, \ldots, C_n$. will be short for the clause t :- [ ], $C_1, \ldots, C_n$.

2.1.6 Sequents

A sequent S is an expression of the form $C_1, \ldots, C_n$ |- C where $C_1, \ldots, C_n$, C are conditions. We call $C_1, \ldots, C_n$ *assumptions* and C a *conclusion.*
"C." will be short for |- C.

We will also sometimes refer to the assumptions as the *premises.*

2.1.7 Programs

A program P is a finite list of clauses.

Below is a small programming example, which we will see more of later on. The program is used to determine which of the "objects" that are grey.

```
elephant(clyde).
elephant(fido).
elephant(P) :-
  albino_elephant(P).

albino_elephant(karo).

grey(P) :-
  elephant(P),
  (albino_elephant(P) -> false).
```

and some possible questions

|- grey(P) => P = clyde
=> P = fido
grey(P) |- false => P = karo

2.2 Formal Semantics

In the following two sections we will use the term *assumption* to refer to a condition to the left of the symbol |-, and the term *consequent* to refer to the condition to the right of |-. The word *premise* will not be used because it could ambiguously refer to both the top row of a rule and to a condition to the left of the turnstile. For the same reason the word *conclusion* will not be used in these two sections. A *goal* is a sequent consisting of a consequent and a possible empty list of assumptions.
There is also a difference between GCLA and partial inductive definitions in the operational semantics. In GCLA the deductions are linear, concatenating new goals in front of the old not yet solved ones, thereby defining an order in which the goals are solved. In partial inductive definitions the inferences form an and-tree where the goals are not ordered in any way.
We are interested in finding substitutions σ such that certain sequents
$C_1\sigma, \ldots, C_n\sigma \vdash C\sigma$ hold according to a program P. The sequents
$C_1\sigma, \ldots, C_n\sigma \vdash C\sigma$ should intuitively be read as "$C\sigma$ follows from the assumptions $C_1\sigma, \ldots, C_n\sigma$ in the program P (according to P)". To solve these sequents we have a number of rules, going from one state to another. The states are given as triples $\langle L; \sigma, P \rangle$ where L is a list of sequents, σ is an answer substitution and P is a program.
The following abbreviations will be used:
- "nil" to denote the empty list,
- C to denote a condition,
- L,M to denote (possibly empty) lists of conditions,
- C.M means that C is concatenated to M,
- L+M means appending L and M together, and
- $M_1 + t + M_2$ will indicate that the condition t occurs in a list of conditions.

L |- M is recursively defined as follows:

L |- nil = nil

L |- C.M = (L |- C).(L |- M)

D(t).L |- C is similarily defined by recursion on D(t), i.e. D(t).L |- C $=_{def}$ {M+L |- C such that M$\in$ D(t)}.

We have here slightly changed the accumulation of the answer substitution compared with the one given in [2]. This change is made because we are here working from the bottom to the top, going backwards trying to reach the Initial List (IL). Therefore, if IL is reached, the resulting answer substitution should be presented.
It should also be mentioned that there are some other rules which will not be mentioned here. Among these additional rules are the ones for asserting and retracting clauses [2]. These concern the interpreter and not the abstract machine´s instruction set, and therefore we have left them out.

2.2.1 Initial List (IL)

$\langle nil ; \theta, P \rangle$

where the answer substitution could be constructed from θ.

2.2.2 Initial sequent or axiom (I)

$$\frac{\langle Rest\sigma ; \sigma\theta, P \rangle}{\langle (L_1 + t + L_2 \vdash r) . Rest ; \theta, P \rangle}$$

where σ is a mgu of t and r

2.2.3 Arrow right ($\vdash\rightarrow$)

$$\frac{\langle (M + L \vdash C) . Rest ; \theta, P \rangle}{\langle (L \vdash M \rightarrow C) . Rest ; \theta, P \rangle}$$

2.2.4 Arrow left ($\rightarrow\vdash$)

$$\frac{\langle (L_1 + L_2 \vdash M) + (C.L_1 + L_2 \vdash C') . Rest ; \theta, P \rangle}{\langle (L_1 + (M \rightarrow C) + L_2 \vdash C') . Rest ; \theta, P \rangle}$$

2.2.5 Program clauses right ($\vdash$P)

$$\frac{\langle (L\sigma \vdash M\sigma) + Rest\sigma ; \sigma\theta, P \rangle}{\langle (L \vdash t) . Rest ; \theta, P \rangle}$$

where $r :- [g_1, \ldots, g_n]$, M. is a clause in P, σ is a mgu of t and r and $g_1\sigma \ldots g_n\sigma$ are all true.

2.2.6 Program clauses left (P$\vdash$)

$$\frac{\langle (D(t\sigma).M_1\sigma + M_2\sigma) \vdash r\sigma) + Rest\sigma ; \sigma\theta, P \rangle}{\langle (M_1 + t + M_2 \vdash r) . Rest ; \theta, P \rangle}$$

where σ is a t-sufficient substitution computed as:

Let $r_1 \ldots r_n$ be a permutation of the heads of the program clauses in P and
mgu(r,t) be the most general unifier of r and t if there is one, nil otherwise

Then define:

$\sigma_0 = nil$

$\sigma_{m+1} = mgu(t\sigma, r_{m+1})$

$\sigma = O_{i \leq n}\sigma_i$

$D(t\sigma)$ is a list containing all $M\sigma$ such that $r\sigma = t\sigma$ where $r :- [g_1 \ldots g_n]$, M. is a clause in the given program P and all $g_1\sigma \ldots g_n\sigma$ are true.

In order for σ to be correct a variable check must also be performed, namely that $D(t\sigma)$ does not introduce any new variables, i.e. that the bodies of the clauses considered do

not contain variables that do not occur in the heads of the clauses. These variables are existentially quantified, which gives rise to problems to the left of the symbol ⊢.
The intuitive understanding of this rule is what an assumption should mean according to the given definition, the program. An assumption holds if all objects that define it also hold. If there is no clause in the program defining this assumption, the current goal-sequent succeeds immediately.

2.2.7 Contraction

$$\frac{<(L \vdash C) + Rest ; \theta, P>}{<(L \vdash con(n,C)) . Rest ; \theta, P>}$$

$$\frac{<(L_1 + L_2 \vdash C) . Rest ; \theta, P>}{<(L_1 + con(0,C') + L_2 \vdash C) . Rest ; \theta, P>}$$

$$\frac{<(C' + L_1 + con(n,C') + L_2 \vdash C) . Rest ; \theta, P>}{<(L_1 + con(n+1,C') + L_2 \vdash C) . Rest ; \theta, P>}$$

2.2.8 Semantics for Guards

- *var(t)* is true if *t* is currently not bound, otherwise false.
- *nonvar(t)* is true if *t* is currently bound to something else than a variable, otherwise false.
- *number(t)* is true if *t* is currently bound to a number, otherwise false.
- *atom(t)* is true if *t* is currently bound to an atom (i.e. a functor of arity 0), otherwise false.
- *atomic(t)* is true if *t* is currently bound to a number or an atom, otherwise false.
- *ground(t)* is true if *t* is currently not bound to a term containing a variable, otherwise false.

2.3 Procedural Semantics

Here we are concerned about in what order different possible solutions should be tried. That is, if at some point in the execution a choicepoint is reached, a choice of which of the different possible ways that should be tried first. There are five points where a choicepoint can arise, namely:

- the choice of a rule,
- the choice of a clause when the rule is ⊢P,
- the choice of an assumption when the rule is I,
- The choice of an assumption if the rule to be tried is P⊢ or →⊢

- different possible permutations of the clauses in P if the rule is P⊢, for a certain assumption

This list is the basis for another definition. The procedure in each inference is defined [2] as first choosing among three different possibilities. The possibilities are the Initial sequent (I), some rule to the right of the turnstile, or some rule to the left of the turnstile. These three possibilities are ordered as: first try the Initial sequent, secondly try the rules to the right, and thirdly try the rules to the left. This choice determines the *mode* for the machine, more about this later.

Depending on this choice, other choicepoints can arise:

- If rule I is tried, a suitable assumption must be chosen, which is a choicepoint
- If some rule to the right of the turnstile is tried, the consequent uniquely determines the choice of rule. If that rule is the rule ⊢P (i.e. the consequent is something else than the arrow), a suitable clause in the program P has to be chosen, which is a choicepoint.
- If some rule to the left of the turnstile should be tried, a suitable assumption has to be chosen first, which is a choicepoint. This choice of assumption uniquely determines the choice of rule in the same manner as in the case above to the right of the turnstile. If the rule is P⊢, a permutation of the clauses in the program P has to be chosen, which also is a choicepoint.

So, one inference could involve zero or more choicepoints depending on the sequent considered.

To form a search strategy among all these choices, we have to order both the program´s clauses and the assumptions in a sequent. The clauses are ordered from top to bottom, that is, the program is searched from top to bottom. The assumptions are ordered from left to right, that is, the assumptions are searched from left to right.

The contraction rule is handled through something called *annotation*. When the primitive *con(n,C)* occurs as an assumption, the assumption *C* is annotated with the number *n*. This means that together with the assumption *C* the number *n* is stored. Each time this assumption is used by another rule than the rule I, its annotation number will be decreased by one. The annotation value is restored either by backtracking or when the current consequent is proved.

3. An Example Deduction

Here are two simple examples illustrating the basic behavior of GCLA´s basic inference rules. Let the program P be

```
a.
b(1) :- a.
b(2) :- c.
```

The goal a -> b(1) is then given by the following derivation:

```
          <nil ; Ø , P>                    (Initial sequent)
     <(a |- a) . nil ; Ø , P>              (⊢P)
   <(a |- b(1)) . nil ; Ø , P>             (⊢→)
<(|- a -> b(1)) . nil ; Ø , P>
```

The goal b(x) |- a may be solved in the following manner with answer substitution {X/1}:

```
          <nil ; {X/1,Z/1} , P>                  (P⊢)
     <(c |- a) . nil ; {X/1,Z/1} , P>            (Initial sequent)
<(a |- a) . (c |- a) . nil ; {X/1,Z/1} , P>      (P⊢)
      <(b(X) |- a) . nil ; Ø , P>
```

where D(`b(X/1,Z/1)`) = (`a, c`) given the stated order of the program clauses and D(`c`) = `nil`. (D(x) is defined by the rule P⊢).

4. Some Programming Examples

The examples below are just small examples of code, to get some feeling of what a GCLA program could look like, and what queries could be put to the system. Remember that the program defines the system $\vdash_P$, so $\vdash_P$ differs from one program to another.
We start with a small example showing negation. As a false symbol we can choose any symbol that does not have a definition in the program. In the program below we have chosen the symbol `false`. This symbol corresponds to the empty set, i.e. there is no definition of what `false` should mean in terms of the other objects in the universe defined by the program.
First we define the object `clyde` as an elephant. `fido` is also defined as an elephant. Then we also define all objects that are albino elephants to also be elephants. The object `karo` is defined to be an albino elephant. Now, what constitutes a grey object? Well, in our universe it should be an elephant that is not an albino elephant.

```
elephant(clyde).
elephant(fido).
elephant(P) :-
  albino_elephant(P).

albino_elephant(karo).
```

```
grey(P) :-
  elephant(P),
  (albino_elephant(P) -> false).
```

Now, what possible questions could be put to this small program? The first question is "Which objects are elephants?", in GCLA syntax `|- elephant(Object)`, and the system will respond with the variable `Object` bound to `clyde`, and with `fido` and `karo` upon backtracking. If the query `|- grey(Object)` is put to the program, that is, asks which objects are grey, `Object` will get bound to `clyde`, and to `fido` upon backtracking, but not to `karo` because `karo` is an albino elephant.

The next example is a small toy expert system. The universe here is the set of diseases, symptoms and temperatures. Diseases are defined in terms of temperature and symptoms. The temperature is defined to be `high`, `normal` or `low`. All symptoms are true, i.e. there are no false symptoms, which is accomplished by the circular definition symptom(X) :- symptom(X) (these circular definitions are sometimes referred to as *opening* the predicate, also see section 9). The circular definitions should be read as "if X is a symptom, then it is a symptom", i.e. if X is assumed to be a symptom, then the program cannot inform us what it should mean that X is a symptom. The same is applicable for the definition of temperatures; if we assume `temp(high)`, the meaning of that is that the temperature is high, nothing more. But to assume `temp(red)` or something like that is absurd, since there is no definition of what `temp(red)` should mean (in this program).

```
disease(plague) :-
  temp(high),
  symptom(perspire).
disease(cold) :-
  temp(normal),
  symptom(cough).
disease(pneumonia) :-
  temp(high),
  symptom(multiple_chills),
  symptom(persistent_cough).

symptom(X) :- symptom(X).
temp(high) :- temp(high).
temp(normal) :- temp(normal).
temp(low) :- temp(low).
```

Then some of the possible questions are:

i) If the person has a normal temperature and a cough, what disease does he suffer from? The query put to the system is:
`temp(normal),symptom(cough)|-disease(K),`
resulting in the answer substitution `K = cold`

ii) We know that a person has a high temperature and multiple chills, and we are interested in additional symptoms for the possible diseases. The query is
`temp(high),symptom(multiple_chills),symptom(Symp) |-disease(K)`
resulting in the answer substitution `Symp = persistent_cough` and `K = pneumonia`

iii) If we assume the plague, what temperature is the patient supposed to have? In the query this time the consequent and the assumption are reordered `disease(plague) |- temp(G)`, and the system comes up with the answer substitution `G = high`.

In the real program the circular definitions of symptoms and temperatures are handled through declarations to prevent looping programs. The different possible declarations are given later.

As was mentioned before, GCLA integrates logic programming and functional programming into one single framework. This is accomplished by using the assumptions to evaluate expressions and using the Initial sequent to bind variables.
Consider the functional definition of `add`, to add two numbers. Two clauses are defined, `add` and `s`, where `s` is a substitution schema saying "if the argument to `s` can be evaluated to a term denoting the same value, continue with that new term". In this case this term is known to be canonical and cannot be reduced any further. The guiding primitive `&axiom` is used to restrict the canonical term to be used just by the axiom rule.

```
add(0,N) :- N.
add(s(M),N) :- s(add(M,N)).

s(Y) :- ((Y -> X) -> &axiom(s(X))).
```

Then some of the possible questions are:

i) What is one plus one?
`add(s(0),s(0)) |- X.`
resulting in the answer substitution `X = s(s(0)).`

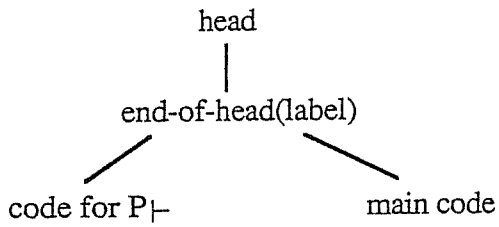
ii) What should be added to one to get two?
`add(X,s(0)) |- s(s(0)).`
resulting in the answer substitution `X = s(0).`

5. The Ideas Behind the Abstract Machine (GAM)

One of the main ideas behind the abstract machine, called GAM, is to build upon WAM [9]. The unifying primitives have not been changed much, nor have the indexing instructions. The control instructions have been replaced by an extended set of instructions, among other things to handle the creation and deletion of assumptions.
In WAM a clause is compiled into a head part and a body part. The body part consists of a number of goals to satisfy. Every clause is looked upon as a procedure, and each body goal as a procedure call. The current goal is the program counter pointing into a procedure representing this goal. In GAM there are also the assumptions which should be executable. Together with every assumption a "program counter" is stored pointing towards the code area where the execution should start if this assumption is chosen. This implies that the code must be executable for both the rules ⊢P and P⊢. This implies that the instructions must perform different things depending on the rule. Furthermore the code must be executable for the rule axiom.
For both P⊢ and ⊢P, the unification in the head of a clause is the same. What differs is how the body of a clause should be treated. So, each clause is compiled into one head part and two bodies, one for the P⊢ rule and one for the main body. A typical clause will look like:

```
        get-instructions
        end_of_head(label)
        create_premises-instructions      call this code "code for P⊢"
        take_next_clause
label   put-instructions and              call this code "main code"
        call-instructions
```

which also could be illustrated by



The purpose of the left code is to create assumptions, which point towards the main code of the clause. When one of these new assumption is executed, the code in the main body will be executed.
As we mentioned earlier the machine could be in different modes. These are `rule`, `right`, `left` and `axiom`. In mode `axiom`, the rule axiom is tried on the current goal.

In mode `right` the current consequent is examined. In mode `left` an assumption is chosen and examined. In mode `rule` the other modes are tried one after another; first try the `axiom` mode, then try the `right` mode and last try the `left` mode.

The head of a clause is always executed in `right` mode or `left` mode, the code for P|– is always executed in mode `left` while the main code can be executed in `axiom` mode, `right` mode or `left` mode .

Then the indexing instructions and the choicepoints´-instructions are used to link the clauses of a predicate together into one procedure.

There is also an area for handling the encoding of the procedural semantics, called the *rule code* or *guiding code*. As of the writing of this report it is not clear how the rule code will look like, but how it should act is described in the section "Realization of the Procedural Semantics".

6. An Introductory Example of Compiled Code

In all the examples we have used relocatable code, that is, all labels are relative to the first instruction. As a first example of code we give the ordinary definition for append compiled into GAM code. The GCLA definition looks like

```
append([],Y,Y).
append([F|X],Y,[F|Z]) :-
   append(X,Y,Z).
```

and the compiled code looks like

```
        append / 3                                  append(
head            [[[L4], [L10]]],
                switch-on-term(L3, L4, L10, fail),
        L3      try-me-else(L9)
        L4      get-nil(A1),                            [],
                get-value(A1, A2),                        Y,Y
                end-of-head(L8),                             )
left code       take-next-clause,                             .
main codeL8     end-of-clause,                                .

        L9      trust-me-else-fail,
head    L10     allocate(3),
                get-list (A0),                          [
                unify-variable(A4),                      F|
                unify-variable (Y0),                        X],
                get-variable(Y1, A1)                           Y,
                get-list (A2),                                    [
                unify-value (A4),                                  F|
```

```
                unify-variable(Y2),                                  Z]
                end-of-head (L21),                                    ):-
left code       create-premise(append / 3, 1, L21),    append(X,Y,Z)
                take-next-clause,                                     .
main code L21   put-value (Y0, A0),                    append(X,
                put-value(Y1, A1),                             Y,
                put-value(Y2, A2),                               Z]
                execute(append  / 3).                                 .
```

In GAM there is no instruction `deallocate` as there is in the WAM. The environment is deallocated, if it is possible, by the instructions `execute` and `end-of-clause` (i.e. the environment is created after the topmost choicepoint and there is no assumption referring to this environment).

7. Data Areas and Registers

A lot of the areas and registers in GAM are roughly the same as in WAM. The new areas are the area holding the assumptions and the area holding the registers for an assumption (which corresponds to the the consequent´s argument registers). Also the terms environment, continuation and choicepoint have the same meaning as in WAM.

The different data areas in GAM are:

- S — The stack, contains environments and continuations. This area should take care of the procedural information during the execution. It corresponds roughly to the stack in PASCAL or the dump in the SECD machine [8].
- H — The heap, holds global values, structures, lists, etc (The same as in the WAM). This area must be garbage collected. All objects that should be visible when leaving the current procedure should reside in this area , and therefore the answer substitutions are built from the heap.
- P — The assumptions (also called *premises*) hold the defined assumptions, which form a tree. The tree arises when the P|– rule is used and there is more than one definition that is unifiable with the assumption chosen. Each assumption has a pointer to its father. The register CA always points to the leaf of the current list of defined assumptions.
- B — Backtrack, a stack containing the choicepoints.
- T — Trail, a stack containing things to be undone upon backtracking. The cell should be one of the following: a reference pointer to a bound variable, a pointer to an assumption whose annotation value should be increased, or a pointer to an assumption whose annotation value should be decreased.
- R — The argument registers for the consequent, numbered A_1, ..., A_n (the same as in WAM)

L The argument registers for an assumption, numbered B_1, ..., B_n

Of course there is also the code area, split into two parts; one containing the code for predicate definitions, and one part containing the code for guiding the execution.

The *argument registers* are used to pass arguments between a calling procedure and the called procedure. They are also used as temporary registers inside a procedure when a variable does not need to be permanently stored. There are two different sets of argument registers, one for the consequents and one for the assumptions.

An *environment* is a frame holding the so called permanent variables, which are such variables that must be saved when a procedure is called. An environment also contains the value of CA (Current-Assumption, see below) to be able to "pop off" assumptions when they should not be defined any longer. It is used to restore the state as it was when this procedure was entered. The environment is created by the instruction allocate in the called procedure, if it needs an environment. All procedures that are not representing an atomic clause need an environment. The environment is deallocated by one of the instructions execute or end-of-clause.

A *continuation* contains information for continuing the execution in a procedure when a subgoal is successfully finished. It contains such information as the mode, the value of top of stack when the subgoal was entered, a pointer to the previous continuation, a pointer to the list of assumptions when the subgoal was entered, the environment pointer when the subgoal was entered and the continuation value of the program counter (see appendix A.3).

A continuation is created:

- every time a procedure is called by the instruction call in mode `right`,
- by the instruction take-next-clause when there is more than one clause unifiable with an assumption if the mode is `left`,
- by the instruction push-premise when the mode is `left`.

A *choicepoint* consists of enough information to restore a prevailing state. In GAM there are 5 different choicepoints, called rule, right, axiom, left1 and left2. These store different kinds of information. (see appendix A.1).

An unbound variable is represented by a cell containing a reference pointer to itself. Other types of pointers are structure pointers and list pointers. There are two data types, integers and symbols, where a symbol consists of a functor and the arity of that functor, for example f/1 and f/2 are different symbols. Constants are functors of arity 0.

In WAM the stack contains variables that could be popped off when a procedural call is ended, while the heap contains bindings that could not be forgotten (i.e. the bindings that are answers/instantiations of variables in previous goals). To prevent dangling references when an environment is popped off the stack there should not be any pointers from the heap to the stack. The pointers in the stack should point from younger variables to older ones. On the heap the pointers could be in either direction, but the order of the objects on the heap is still significant for efficient treatment of backtracking.

All the data areas are treated as stacks when a new object should be created. Therefore all

the areas have a pointer to the top of the area. The areas S, B and T are "true" stacks, while P is merely treated as a stack when creating new objects and when popping off objects during backtracking. The objects in P are then linked together forming a tree. H is garbage collected when the upper bound of H is (almost) reached. The names of each "top of the area" register are:

ToS	Top of stack
ToH	Top of heap
ToP	Top of premises
ToB	Top of backtrack
ToT	Top of trail

The other registers are:

Me	Mode register, holding the mode. One of: `rule`, `right`, `axiom`, `left`. For convenience two additional modes are added, `fail` for an execution that fails, and `finish` for an execution that is successfully executed.
CA	Current Assumption, points to the leftmost assumption in the list of assumptions (the last created). This assumption is one of the leaves in the assumption tree in the area P.
CAP	Current Assumption Pointer, a help register for pointing out a specific assumption, used when a premise is to be chosen by the rules Initial sequent and P⊢.
TA	Treat as Absurd, a flag which is set to true every time the rule P⊢ is entered. As soon as one head in the program is unifiable with the chosen assumption, this flag is set to false, indicating that there was at least one head defining this assumption. The purpose of this register is to determine when an assumption does not have a definition in the program, in which case the current sequent trivially holds (see the rule P⊢, page 7).
CL	Clauses Left, a register holding the clauses left to unify with under the mode `left`. Actually CL points to the first address not tried in the clause order associated with each predicate.
ST	Succeed To, points to the next continuation on the stack, which is where the execution should proceed to when this clause is successfully finished. (This register differs slightly from the CP register in the WAM in that it points towards the stack where the next continuation is stored instead of holding the next value for the program counter.)
PC	Program Counter.
Sp	Structure pointer, a help register for unifying structures. This register always points towards the heap. It is the same as the S-register in the WAM.
Um	Unification mode, read or write. The same as mode in the WAM, and

affects only the unify instructions.

E Environment pointer, points to the current environment in the stack.

HB Heap Backtrack pointer, points towards the heap where the top of the heap was when the last choicepoint was created. (This register is actually not necessary because the value of HB can be found in the most recent choicepoint's ToH-cell).

RC Right Caller, holds the functor/arity of the current consequent.

8. The Clause Orders

For the rule P⊢ a suitable permutation of the clauses in the program should be chosen, where the resulting unifier of the ordered clauses should be unified with the current assumption (see 2.2.6). This choice of permutation is a choicepoint. However, a lot of these permutations result in the same unifier, so a lot of execution time could be saved if just the orders of the clauses that result in different unifiers are tried (The number of possible unifiers for a set with n terms does actually not exceed n, which is far less than (n!), but we will not prove this in the paper). This is possible to do in the compiler.
The clause orders is a list, consisting of all the possible trials that could be unifiable with an assumption. These orders represents two "kinds" of failure; failure in the unification with the current chosen assumption, and failure later in some other inference step. The latter kind of failure could be due to wrong unifier, and therefore another clause order should be tried, so this is a choicepoint, while the first kind of failure does not give rise to a choicepoint.
Lets illustrate this by an example. Consider the program

```
1:    p(1,X) :- b1.
2:    p(2,X) :- b2.
3:    p(Y,3) :- b3.
4:    p(Y,4) :- b4.
```

where `b1` to `b4` could be arbitrary bodies. With this set of clauses the possible different unifiers are

[1,3], [1,4], [2,3], [2,4]

representing

```
p(1,3), p(1,4), p(2,3), p(2,4)
```

respectively.

The orders are actually representing the permutations which the clauses should be tried. For example, the order [1,3] is representing all the permutations (1,3,2,4), (1,3,4,2), (3,1,2,4) and (3,1,4,2) in the algorithm for P⊢, giving in section 2.2.6.

If now the assumption is unifiable with at least one of these orders (all the heads in a list must be unifiable with the assumption), a choicepoint is created for the next possible order. For example, in the goal

```
p(Z,3) |- whatever
```

`p(Z,3)` is unified with the order [1,3], where `Z` is bound to `1`, and if backtracking occurs, `p(Z,3)` is unified with the order [2,3] giving as result `Z` bound to `2`.
But if the assumption is `p(Z,5)` no one of the orders above is applicable, yet there is two possibilities, namely one of the heads `p(1,X)` or `p(2,X)`. Therefore another layer is introduced, giving

```
  [1,3], [1,4], [2,3], [2,4]
      [1], [2], [3], [4]
```

representing

```
    p(1,3), p(1,4), p(2,3), p(2,4)
    p(1,Z), p(2,Z), p(Z,3), p(Z,4)
```

The lower layer is used if no order in the layer above is unifiable with the current assumption. Now the assumption `p(Z,5)` will be replaced by the body `b1`, and with `b2` upon backtracking. So, in the horizontal plane, a choicepoint is created upon successful unification, and in the vertical plane the next plane is just tried if no order in the plan above were unifiable with the current assumption.
The orders are in this paper represented as lists, so the orders above will look like

[[[1,3], [2,3], [1,4], [2,4]], [[1], [2], [3], [4]]].

9. Declarations

The declarations are used to get certain effects when the rule P⊢ is tried. As shown in the examples before there are situations where an assumption should not be treated as not defined, which is the same as always treating the assumption as defined, although there is no actual definition of it. For example, in the toy expert system (see page 10) symptoms are always true, that is, a person could not suffer from a non-existing symptom. To get such behaviour declarations are used. The declarations do not affect

any rule except P⊢. They are introduced instead of writing circular clauses, to avoid having the execution going into a loop.
If subsumption were introduced on the goals these declarations would not be necessary, but subsumption is a very expensive method. The approach with subsumption was tested on an earlier version of the experimental interpreters.
A definition could be declared in three ways:
- `normal`, which does not affect anything and is the default
- `total`, which means that this definition is not applicable to the rule P⊢. Semantically this means: suppose that a clause is named t, and it is declared `total`. This is the same as adding the clause `t :- t` to the program.
- `otherwise`, which theoretically means add the clause $t :- [g_1 \ldots g_n], t.$ to the program. This clause's guards are such that the head is not unifiable with any other clause's head. For example assume the clauses defining t/1 are t(1) and t(2). Then the `otherwise` declaration has the same effect (theoretically) as adding t(X) :- t(X) where X is constrained not to be unified with 1 or 2. If now an assumption is not unifiable with t(1) or t(2), the sequent will not trivially hold (see section 2). Instead backtracking occurs. This is sometimes referred to as "open the predicate t".

10. The Control Instructions

The control instructions can be divided into 6 groups: calling instructions, premise instructions, indexing instructions and choicepoint instructions. The guard instructions will also be treated here, as well as the instruction fail.
In the following, "WAM" means that the instruction is roughly the same as in the WAM (at least in mode `right`), "new" means that the instruction is new and "changed" means that the instruction has the same name as in the WAM, but does not perform the same actions.

10.1 The Calling Instructions

The calling instructions are call, execute, end-of-premise, end-of-head, end-of-clause and take-next-clause. We have also treated the instruction allocate here.

call(F/A) *changed*
If mode equals `right`, a continuation is saved on the stack, the PC is set to the first instruction of the rule code, F/A is put into the register RC and the execution continues in mode `rule`.
If the mode equals `axiom`, call is treated like end-of-clause, that is, the next continuation is examined.
If the mode equals `left`, the program database is searched for F/A. If F/A exists the PC

is set to the first label in the clause order list. For example, if the predicate q looks like

```
q/1,
[[[4, 17], [10, 42]]],
switch-on-term....
```

then PC is set to 4 (or actually PC + 4). A choicepoint for the clause order is created, pointing at the rest of the list (in the example above pointing at [10, 42]), and the execution continues in mode `left`. If F/A does not exist the next continuation is examined, provided that F/A is not declared total or otherwise. If F/A is declared total or otherwise, the procedure fail is called.

execute(F/A) *changed*

This instruction is used to get tail recursion optimization (TRO), that is, it replaces the last call-instruction in the code of a clause when TRO could be performed. The only difference from call(F/A) is that a continuation is not pushed on the stack in mode `right`.

end-of-premise *new*

This instruction is used to mark the end of the assumption list in a body goal, which is an arrow. It is executed in mode `axiom` or `right`, and for these modes it equals the instruction end-of-clause (this instruction could actually be replaced by end-of-clause).

end-of-head(L) *new*

This instruction is used to mark the end of a head in a clause, and is used to split the execution depending on the mode. If the mode is `left` we just continue to the next instruction, if the mode equals `right`, PC is set to L.

end-of-clause *new*

This instruction marks the end of the code for ⊢P which does not end with an execute instruction. It examines the next continuation (e.g. the one pointed to by the register ST) and restores the registers according to the continuation. If the continuation is above the last choicepoint (the top of stack field in the most recent choicepoint) the continuation is popped off the stack.

allocate(N) *WAM*

This instruction occurs in the head of a clause, which has at least one body goal. It can be executed in mode `right` or mode `left`. It allocates an environment containing N+1 cells on the stack, N permanent variables and the current value of CA. The variables are initialized to unbound variables (this is necessary because it is not certain that the variable is initialized by a put-variable instruction or a get-variable instruction. This is due to the fact that the assumptions could be chosen in several different orders) .

take-next-clause *new*

This instruction marks the end of the code for P⊢. It examines the current clause order if there is another clause to check, and if so the address to that clause is put into the PC register and pushes a continuation of mode `rule` on the stack. If there are no more clauses to check, the next continuation is examined (which should be a continuation of mode `rule`, pushed by the machine when the P⊢ rule was chosen, or by a take-next-clause executed before; see the section about realization of the procedural semantics).

10.2 The Premise Instructions

Create-premise, push-premise and restore-premise belong to this group of instructions.

create-premise(L, N, F/A) *new*

This instruction is only executed in mode `left`. It creates a new assumption in the premise area. The assumption points to the address L, has annotation value N and has the name F/A. N is dereferenced, and if it is not a positive integer an error is signaled. The register CA is set to this new assumption, which in turn has a pointer to the former value of CA.

push-premise(F/A, N, L) *new*

If mode equals `right`, a new assumption is made in the premise area. The assumption has a pointer to the address L, an annotation N and the name F/A. N is dereferenced, and if it is not a positive integer an error is signaled. The register CA is set to this new assumption and the former CA value is stored in the assumption.
If mode equals `left`, a continuation of type left is stored on the stack and the mode is set to `right`. The reason for this is that the rule arrow-left is going to be executed, and the arrow´s premises should be executed in mode `right` (to check if they hold). If that succeeds, the former state must be recalled when a new assumption is created for the arrows consequent.

restore-premise *new*

This instruction restores a former state where possibly not all the currently defined assumptions were defined. The value of CA is set to the value stored in the current environment.

10.3 The Indexing Instructions

The task of these instructions is to reduce the number of choicepoints created plus increase the efficiency by minimizing the number of applicable clauses in a predicate. These instructions are always executed in mode `right`. Switch-on-term, switch-on-

constant and switch-on-structure belong to this group.

switch-on-term(Var,Const,List,Struct) *WAM*

The value of the first argument register (A_1) is dereferenced and this value is examined. If it is a reference to an unbound variable, a jump to the label Var is made. If Var is the constant fail, backtracking occurs. An Analogous action occurs if A_1 dereferences to a constant, a list pointer or a structure pointer.

switch-on-constant(Table, Default) *WAM*

The table consists of constant-label pairs of constants occurring as the first argument in the clauses of a predicate, and can be hashed for maximal efficiency. The value of the first argument register (A_1) is dereferenced and this value is examined. If the constant is in the table, the PC is set to the corresponding label, otherwise the PC is set to Default. If Default equals fail, backtracking occurs.

switch-on-structure(Table, Default) *WAM*

This instruction is analogous to the switch-on-constant instruction, but the table consist of functor/arity-label pairs, and is used to dispatch on the structures functor.

10.4 The Choicepoint Instructions

The instructions handling the choicepoints for the rule ⊢P are try, retry, trust, try-me-else, retry-me-else and trust-me-else-fail, and they are therefore always executed in mode `right`. The choicepoints for the other rules are currently handled inside the system. (The instructions try-me-else, retry-me-else and trust-me-else-fail are in fact not necessary, the same task can be fulfilled by the instructions try, retry and trust.)

try(Label) *WAM*

This instruction creates a choicepoint with the next try set to the instruction after this one. Then the execution proceeds at Label.

retry(Label) *WAM*

The topmost choicepoint is updated with the next try set to the next instruction. Then the execution proceeds at Label.

trust(Label) *WAM*

The topmost choicepoint is popped of and the execution proceeds at Label.

try-me-else(Label) *WAM*

This instruction creates a new choicepoint on top of the backtrack area. The next try in this choicepoint is Label, and the execution proceeds to the following instruction.

retry-me-else(Label) *WAM*

The topmost choicepoint is updated with the next try set to Label, and the execution proceeds to the next instruction.

trust-me-else-fail *WAM*

The topmost choicepoint is popped off the backtrack area, and the execution proceeds to the next instruction.

10.5 The Guard Instructions

These instructions are such instructions that shall put restrictions on the variables in the head of a clause, and which cannot be expressed in GCLA itself. Var, nonvar, atom, atomic, number and ground belong to this group of instructions. These instructions should always occur before the end-of-head instruction.

var(X) *new*

This instruction dereferences X and examines the value. If it is currently unbound the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

nonvar(X) *new*

This instruction dereferences X and examines the value. If it is currently bound the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

number(X) *new*

This instruction dereferences X and examines the value. If it is currently bound to a number the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

atom(X) *new*

This instruction dereferences X and examines the value. If it is currently bound to a constant the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

atomic(X) *new*

This instruction dereferences X and examines the value. If it is currently bound to a number or a constant the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

ground(X) *new*

This instruction dereferences X and examines the value. If it is currently bound to an object not containing an unbound variable the execution proceeds, otherwise backtracking occurs. If mode is `left`, X refers to one of the L-registers, otherwise X refers to one of the R-registers.

10.6 The Fail Instruction

This instruction restores a former state which was a choicepoint in the execution. Depending on the topmost choicepoint different areas and registers are updated, and the execution proceeds from the point stored in the choicepoint. The choicepoints can be of 5 different types:

- rule, where it represents the next choice of a mode (`axiom`, `right` or `left`)
- axiom, where it represents the next choice of an assumption
- right, where it represents the next choice of a clause in the program
- left1, where it represents the next choice of an order to try among the clauses in the program
- left2, where it represents the next choice of an assumption (actually the same kind of choice as axiom, but we must perform different tasks upon backtracking).

The procedure first restores all the different registers stored in the choicepoint, and the trail is unwound to the address stored in the choicepoint. Now the state is restored to what it was when the choicepoint was created. Then, depending on the type of the choicepoint, different actions are performed in order to create the next choice:

- If the type is axiom, a new assumption is searched for with the same functor as the current consequent, which is held by the register RC. If one is found, this is tried, otherwise failure to the next choicepoint occurs.
- If the type is rule, the execution simply proceeds to the instruction pointed to by the choicepoint (somewhere in the rule code).
- If the type equals right, the execution proceeds to the instruction pointed to by the choicepoint.
- If the type of the choicepoint is left1, the next possible order of the program´s clauses is tried, which is stored in this choicepoint. If there is no more trials in this plane to do (see section 8), the register TA is examined. If it is false, there was at least one clause order that has succeeded in this plane, and the next choicepoint is examined. If TA is true, the next plane is tried, updating the choicepoint with the next plane number and the next order to try, which is the first one on this new plane. If there are no more planes to search, the current goal sequent holds and the next continuation is examined.
- If the type is left2, there is much to do; another assumption should be tried. The next assumption is searched for and examined. If the next assumption is the last one in the list (indicated by an empty pointer in the linking field of the assumption), this choicepoint of type left2 is popped off the stack, otherwise this choicepoint is updated with the next assumption in the list (the value of the linking field in the chosen

assumption). The annotation value is then examined, and if it is equal to zero (meaning that this assumption actually does not exist), backtracking occurs again, otherwise it is decreased by one and a pointer to this assumption is pushed onto the trail. A "continuation" is stored on the stack of type "restore-prem", saying that the annotation value of this assumption should be added by one.

The register TA is set to false for every choicepoint except those of type left1 (where it is not affected at all) and left2 where it is set to true except when the assumption´s name is the primitive ->, in which case it is set to false.

11. The Unification Instructions

These instructions have almost the same semantics as in the WAM. One of the major differences is that under the mode `left`, the registers B_1 to B_m are used instead of A_1 to A_n.

There is one unification instruction per symbol in the original program. The put-instructions and get-instructions handle (mostly) objects in the stack, while the unify-instructions always handle objects in the heap.

Every time a binding of a variable is done, the binding is trailed if necessary. This means that if a cell in the heap is bound, and it resides below the HB-pointer, this binding is trailed. A permanent variable is trailed if it resides below the corresponding stackpointer in the most recent choicepoint. In both the cases this suffices because when backtracking occurs all things made since the most recent choicepoint was created will be undefined by reseting the stackpointers.

An unsafe variable is a variable that can have references to it when the environment where it resides is popped off the stack. These variables have to get "globalized" if necessary, and this is accomplished by moving the value from the cell in the stack to a cell in the heap.

In the following X_i stands for a temporary variable, Y_i stands for a permanent variable and V_n stands for either a temporary variable or a permanent variable. If the mode is `left`, temporary variables refer to the L-registers (numbered B_1 ... B_n), otherwise the temporary registers refer to the R-registers (numbered A_1 ... A_n) .

11.1 The Get Instructions

These instructions occur in the head of a clause. They can fail, that is, if the argument that they are representing is not unifiable with the corresponding argument register, the procedure fail is called and backtracking occurs. These instructions can be called in mode `left` and in mode `right`.

get-variable(V_n, X_i) *WAM*

The variable Vn is bound to the dereferenced value of X_i. This instruction corresponds to a variable occurring for the first time in a clause.

get-value(V_n, X_i) *WAM*

The instruction gets the value of X_i and tries to unify it with the contents of V_n. If it succeeds the dereferenced value of X_i is left in V_n if V_n is a temporary variable. If it fails to unify V_n and X_i, backtracking occurs. This instruction should occur when V_n is initialized by another instruction before.

get-constant(C,X_i) *WAM*

This instruction dereferences X_i and the value is compared with C. If they are not unifiable (that is, if the value and C are not identical and the value is not an unbound variable) backtracking occurs, otherwise X_i is bound to C.

get-nil(X_i) *WAM*

This instruction is the same as the instruction get-constant, but the constant is the constant nil.

get-structure(F/A, X_i) *WAM*

This instruction marks the beginning of a structure occurring in the head of a clause. The register X_i is dereferenced and the value examined. If it is an unbound variable, that variable is bound to a structure pointer pointing to the top of the heap, F/A is pushed onto the heap and the unification mode is set to write. Otherwise if the dereferenced value is a structure pointer pointing to a cell identical to F/A, the unification mode is set to read and the S-register is set to point to the beginning of the arguments of the structure, otherwise backtracking occurs.

get-list(X_i) *WAM*

This instruction is analogous to the instruction get-structure, except that the structure is replaced by a list and the structure pointer is replaced by a list pointer.

11.2 The Put Instructions

These instructions cannot fail in mode `left` or `right`, they simply store data/references in the argument registers. They can be called in mode `left`, mode `right` and mode `axiom`. In mode `axiom` they are treated as their corresponding get-instructions in mode `right` (for example, in mode `axiom` the instruction put-value will be executed as the instruction get-value in mode `right`), except for the instruction put-variable, which is executed as get-value. The exception is due to the fact that the

assumptions are not guaranteed to be executed in the order the body goals are ordered. Therefore we always have to perform the more expensive unification of two variables which the instruction get-value does, instead of just setting one variable to the content of the other. This is also the reason why all the permanent variables in an environment have to be initialized to unbound variables when they are created by the instruction allocate. The description below is just for `right` mode and `left` mode.

put-variable(Y_n, X_i) *WAM*

This instruction corresponds to a variable occurring for the first time in the (main) body of a clause. If the mode equals `right`, the variable Y_n is initialized to an unbound variable, and X_i is bound to Y_n. If the mode equals `left` this instruction performs the same actions as put-value. (This instruction is actually not necessary, because the instruction allocate initializes all its variables to unbound. It could be replaced by put-value(Y_n,X_i).)

put-variable(X_n, X_i) *WAM*

This instruction corresponds to a variable occurring in just one atomic goal (not a goal containing ->) of a body. If the mode equals `right`, both the variable X_n and X_i are bound to a new unbound variable on the top of the heap. If the mode equals `left` this instruction performs the same actions as put-value.

put-value(V_n, X_i) *WAM*

The instruction puts the value of V_n into X_i. It occurs when V_n has been initialized by another instruction.

put-unsafe-value(Y_n, X_i, N) *WAM*

In mode `right` this instruction dereferences Y_n and examines the result. If the result is a variable in the current environment, a new global variable is created on top of the heap, the variable in the environment is set to the new variable and X_i is set to the new variable. Otherwise the instruction performs the same actions as put-value.
In mode `left` this instruction does the same thing as put-value, and in mode `axiom` this instruction does the same thing as get-value.

put-constant(C, X_i) *WAM*

This instruction puts the constant C into X_i.

put-nil(X_i) *WAM*

This instruction is the same as put-constant, where the constant is the constant nil.

put-structure(F/A, X_i) *WAM*

This instruction marks the beginning of a structure occurring in a body goal. F/A is pushed onto the heap and the corresponding structure pointer is put into the register X_i. Then the unification mode is set to write.

put-list(X_i) *WAM*

This instruction is analogous to put-structure, except that the structure is replaced by a list and the structure pointer is replaced by a list pointer.

11.3 The Unify Instructions

These instructions occur in a complex argument to a head in a clause (i.e. a structure or a list). They can be executed in unification mode read or write, depending on what the instructions get-list, put-list, get-structure and put-structure have set the unification mode to.

unify-variable(V_n) *WAM*

This instruction corresponds to a variable that is not previously initialized by another instruction. If it is executed in `right` and, furthermore, if it is executed in write unification mode, it pushes a new unbound variable onto the heap and sets V_n to point to it. In read mode it stores the value of what the Sp-register points to in V_n, and increments Sp.
Otherwise if it is executed in `axiom` mode or `left` mode, it shall perform the same actions as unify-local-value.

unify-value(V_n) *WAM*

This instruction corresponds to a variable V_n which has been initialized by another instruction. If the mode equals `right` and if it is executed in write mode, it pushes the value of V_n onto the heap. If it is executed in read mode, it unifies V_n and what Sp points to, leaving the dereferenced value in V_n if V_n is a temporary variable.
Otherwise, if it is executed in mode `axiom` or mode `left` it shall perform the same actions as unify-local-value to prevent references from the heap into the stack.

unify-local-value(V_n) *WAM*

This instruction does almost the same thing as unify-value. It is used where V_n is not guaranteed to be bound to a global value (i.e. not an unbound variable on the stack). It performs the same things as unify-value, except in write unification mode where the dereferenced value of V_n is a reference to a variable on the stack. Then a new variable is created on top of the heap and the variable on the stack is bound to the new variable and V_n is set to point to the new variable if V_n is a temporary variable.

unify-constant(C) *WAM*

If this instruction is executed in write unification mode the constant is simply pushed onto the heap. If the unification mode is read, this instruction dereferences the cell pointed to by the register Sp, and if that is unifiable with C (i.e. it is identical to C or is

an unbound variable, which is bound to C) the register Sp will be incremented and the execution proceeds, otherwise backtracking occurs.

unify-nil *WAM*

This instruction is the same instruction as unify-constant, except that the constant is nil.

unify-void(N) *WAM*

This instruction is used for variables whose values are not needed. In read unification mode this instruction simply adds N to the register Sp. In write unification mode N unbound variables are created on top of the heap.

12. Realization of the Procedural Semantics

To get a combination of efficiency and flexibility a new code area is introduced, called the *rule code*. The idea is to specify a number of primitives, and in these primitives should the operational and procedural semantics be expressed. With this rule code there is the ability to remove one rule by simply removing some of the primitives, or add a new rule, or reorder the trials in an efficient way. If this is a good idea or not is not clear at the moment, as it has not been implemented yet. As it is now there are just the default order and the default rules described earlier in the sections about operational semantics and procedural semantics. We will here describe how choicepoints for the choice of an assumption and how choicepoints for the clause order are treated and how the machine acts before trying a rule.

The default order is to try the different modes `axiom`, `right` and last `left`. The first thing that happens when a call-instruction or an execute instruction is executed in mode `right` is that the mode is set to `rule` and the machine starts a "mode-cycle", consisting of the mode-choices. First it creates a choicepoint for the next mode to try, which is the mode `right`. Then it tries to find an assumption for the axiom to try. If there is no assumption of the same functor and arity as the consequent, or if there is no assumption at all, backtracking occurs and the most recent choicepoint is examined (which was the one for trying the mode `right`). If there is at least one assumption with the same functor and arity as the current consequent, the PC is set to the code for that assumption, and a choicepoint for the next assumption (if there is one) is made. The mode is then set to `axiom`.

Now, suppose that the axiom fails. The rule-choicepoint is then popped off from the B-area (backtrack), and a new choicepoint is created for the mode `left`. Then the code for the procedure with the name held by the register RC is searched for . If the code exists for the current consequent, the PC is set to that code and execution proceeds in mode `right`. If there is no code backtracking occurs.

Now there is no mode to try after `left`, which is to be tried now, so the rule-choicepoint is popped off the stack, and an assumption is again searched for. If there are no assumptions, backtracking occurs. If there is one assumption, this assumption is

examined by setting the mode to `left` and loading the PC with the address hold by the assumption. This address points towards the main code of a body of some clause in the program. A continuation of type rule is stored and the environment pointer is set to the value hold by the assumption, and the annotation value is decreased by one before the execution proceeds in mode `left`.
If there are more than one assumption, a choicepoint for trying the next assumption is created before the chosen assumption is examined.
Now, if the execution succeeds in mode `left`, the execution returns to mode `rule` when the saved continuation is restored and then another mode-cycle takes place (now with the new assumptions, and perhaps with some new goals on the stack to solve, which were created by P⊢).

13. Compiling Schema

There are a lot of things that could be done, and should be done, by the compiler. A compiler for a somewhat simplified version of this machine has been implemented. But there is certainly room for a lot of ideas.
To fulfill the operational semantics a variable check on the variables in the body of a clause should be done. When the program-left rule is chosen, a clause which introduces new variables in the body is not allowed and that try of clause order should fail. This check should be done by the compiler, which should generate code where this has been taken care of in the clause order. However, there are a lot of places where this check is not necessary and, therefore, we have sofar not implemented it.
A difference with WAM is that an environment has to be generated more often. In WAM the variables in the first body goal can be temporary, and the other ones in the rest of the body goals must be permanent. In GAM an environment has to be generated for every clause that is not an atomic clause, and all variables that occur in a body goal have to be made permanent. This is done because the program-left rule saves the environment pointer E in the assumptions it creates, and when these assumptions are examined the environment pointer is restored again. The axiom rule (I) has the same behaviour.
There is no instruction for handling nested structures. These have to be lifted out and unified separately. For example, the code for the head argument s(f(X)) occurring inside a head is separated to s(Y), Y = f(X), which is then compiled into

```
...get-structure(s/1, Ai),
   unify-variable(Aj),
   get-structure(f/1, Aj),
   unify-variable(Ak), ...
```

One of the main tasks the compiler has to perform is to generate the clause orders, so they is built as specified in section 8.

In order to be sure that there will be no pointers from the heap to the stack, the instructions unify-variable and unify-value have the same definition as unify-local-value when the mode equals `left` or `axiom`. This solution is expensive and a thorough examination should be made in order to find the occasions where there is a risk for pointers in the wrong direction.

14. Some Examples

The first example of code generation is the example of negation introduced in section 3. Recall that it should decide whether an object (an elephant) was grey or not. The GCLA code for this program is:

```
elephant(clyde).
elephant(fido).
elephant(P) :-
  albino_elephant(P).

albino_elephant(karo).

grey(P) :-
  elephant(P),
  (albino_elephant(P) -> false).
```

And below is the corresponding GAM-code.

```
   elephant / 1,                                          elephant(
       [[[L19, L9], [L19, L14]], [[L19]]],
       switch-on-term(L8, L3, L19, L19),
L3     switch-on-constant([[clyde,L4], [fido, L6]], L19),
L4     try(L9),
       trust(L19),
L6     try(L14),
       trust(L19),
L8     try-me-else(L13),
L9     get-constant(clyde, X0),                                  clyde
       end-of-head(L12),                                            )
       take-next-clause,                                    .
L12    end-of-clause,                                       .
L13    retry-me-else(L18),
       get-constant(fido, X0),                                   fido
       end-of-head(L17),                                            )
       take-next-clause,                                    .
L17    end-of-clause,                                       .
L18    trust-me-else-fail,
L19    allocate(1),
       get-variable(Y0, X0),                                     X
       end-of-head(L24),                                            ):-
       create-premise(albino-elephant / 1, 1, L24),       albino-elephant(X)
```

```
        take-next-clause,
L24     put-value(Y0, X0),
        execute(albino-elephant / 1),

    albino-elephant / 1,
        [[[L4]]],
        switch-on-term(L4, L3, fail, fail),
L3      switch-on-constant([[karo, L4]], fail),
L4      get-constant(karo, X0),
        end-of-head(L7),
        take-next-clause,
L7      end-of-clause.

    grey / 1,
        [[[L3]]],
        switch-on-term(L3, L3, L3, L3),
L3      allocate(1),
        get-variable(Y0, X0),
        end-of-head(L9),
        create-premise(elephant / 1, 1, L9),
        create-premise(-> / 2), 1, L11),
        take-next-clause
L9      put-value(Y0, X0),
        call(elephant / 1),
L11     push-premise(albino-elephant / 1, 1, L13),
        goto(L16, false / 0),
L13     put-value(Y0, X0),
        call(albino-elephant / 1),
        end-of-premise,
L16     call(false / 0),
        restore-premises,
        end-of-clause,
```

```
.
albino-elephant(X)
.

albino-elephant(
                karo
                )
.
.

grey(
   X
   ):-
elephant(X),
(albino-elephant(X) -> false)
.
elephant(X
         ),
(albino-elephant(
            X
            )
            ->
               false
                )
.
```

The next program is a program for intuitionistic predicate logic. This is an interesting example because it shows how most of the different possibilities and primitives in GCLA are compiled into GAM-code.
The program looks like:

```
t(not X) :- (t(X) -> false).
t(X => Y) :- (t(X) ->  t(Y)).
t(X & y) :- t(X),t(Y).
t(X $ Y) :- t(X).
t(X $ Y) :- t(Y).
t(contr(Num,Goal)) :- con(Num,t(Goal)).
```

and the corresponding GAM-code looks like:
(This is perhaps the most "expanding" GCLA-program that could be written...)

```
    t / 1,                                                    t(
          [[[L7], [L22], [L39], [L52 ,L62], [L72], otherwise]],
          switch-on-term(L6, fail, fail, L3),
L3        switch-on-structure([[not/1, L7], [=>/2, L22]
                               [&/2, L39], [$/2, L4]
                               [contr/2, L72], fail),
L4        try(L52),
          trust(L62),

L6        try-me-else(L21),
L7        allocate(1),
          get-structure(not/1, X0),                             (not(
          unify-variable(Y0),                                       X))
          end-of-head(L13),                                             :-
          create-premise(-> /2, 1, L13),                        (t(X)  ->  false)
          take-next-clause,                                                 .
L13       push-premise(t/1, 1, L15),                            t(X)
          goto(L18, false/0),
L15       put-value(Y0, X0),                                    t(X)
          call(t/1),
          end-of-premise,                                           ->
L18       call(false/0),                                                false
          restore-premises,
          end-of-clause,                                                    .

L21       retry-me-else(L38),
L22       allocate(2),
          get-structure(=>/2, X0),                              => (
          unify-variable(Y0),                                       X,
          unify-variable(Y1),                                           Y))
          end-of-head(L29),                                             :-
          create-premise(->/2, 1,L 29),                         t(X) -> t(Y)
          take-next-clause,                                                 .
L29       push-premise(t/1, 1, L31),                            t(X)
          goto(L34, t/1),
L31       put-value(Y0, X0),                                    t(X)
          call(t/1)),
          end-of-premise(t/1),                                      ->
L34       put-value(Y1, X0),                                            t(Y)
          call(t/1),
          restore-premises,
          end-of-clause,                                                    .

L38       retry-me-else(L51),
L39       allocate(2),
          get-structure(&/2, X0),                               &(
          unify-variable(Y0),                                     X,
          unify-variable(Y1),                                           Y))
```

```
        end-of-head(L47),                          :-
        create-premise(t/1, 1, L47),               t(X),
        create-premise(t/1, 1, L49),               t(Y)
        take-next-clause,                            .
L47     put-value(Y0, X0),                         t(Y),
        call(t/1),
L49     put-value(Y1, X0),                         t(Y)
        execute(t/1),                                .

L51     retry-me-else(L61),
L52     allocate(1),
        get-structure($/2, X0),                    $(
        unify-variable(Y0),                          X,
        unify-void(1),                                 Y))
        end-of-head(L59),                                :-
        create-premise(t/1, 1, L59),               t(X)
        take-next-clause,                            .
L59     put-value(Y0, X0),                         t(X)
        execute(t/1),                                .

L61     retry-me-else(L71),
L62     allocate(1),
        get-structure($/2, X0),                    $(
        unify-void(1),                               X,
        unify-variable(Y0),                            Y))
        end-of-head(L69),                                :-
        create-premise(t/1, 1, L69),               t(X)
        take-next-clause,                            .
L69     put-value(Y0, X0),                         t(X)
        execute(t/1),                                .

L71     trust-me-else-fail,
        allocate(2),
        get-structure(contr/2, X0),                contr(
        unify-variable(Y1),                              Num,
        unify-variable(Y0),                                      Goal))
        end-of-head(L79),                                                :-
        create-premise(t/1, Y1, L79),              con(Num,t(Goal))
        take-next-clause,                                              .
L79     put-value(Y0, X0),                         t(Goal))
        execute(t/1).                                                  .
```

This code needs some explanation. The symbol "otherwise" is the declaration of an open predicate. When the end of the unification order is reached, the symbol "otherwise" prevents the execution from succeeding with an assumption that is not unifiable with any clause in the program.

In the last clause the treatment of contraction is shown. Y_1 corresponds to the variable "Num" in the original GCLA. This variable must be checked so that it is currently

instantiated to a number. What happens (semantically) when it is not a number is not clear. With this treatment the current goals succeed because the guard-instruction number will fail. Perhaps the best solution is that the variable Y_1 should make use of something called *freeze*, a primitive for "freezing" a variable and corresponding goals until the variable has been bound [4]. However, this has not been implemented yet.

The next example is the program for the functional definition of the arithmetic function add. Remember the GCLA program

```
add(0,N) :- N.
add(s(M),N) :- s(add(M,N)).

s(Y) :- ((Y -> X) -> &axiom(s(X))).
```

The corresponding GAM code is

```
    add/2,                                  add(
        [[[L4], [L13]]],
        switch-on-term(L3, L4, fail, L13),
L3      try-me-else(L12),
L4      allocate(1),
        get-constant(0, X0),                    0,
        get-variable(Y0, X1),                     N)
        end-of-head(L10),                           :-
        create-premise(call/0, 1, L10),       N
        take-next-clause,                       .
L10     put-value(Y0, X0),                    N
        execute(call/1),                        .
L12     trust-me-else-fail,
L13     allocate(2),
        get-structure(s/1, X0),                 s(
        unify-variable(Y0),                       M),
        get-variable(Y1, X1),                       N)
        end-of-head(L20),                             :-
        create-premise(s/1, 1, L20),          s(add(M,N))
        take-next-clause,                                .
L20     put-structure(add/2, X0),             s(add(
        unify-local-value(Y0),                      M,
        unify-local-value(Y1),                        N))
        execute(s/1)),                                  .

    s/1,                                    s(
        [[[L2]]],
L2      allocate(2),
        get-variable(Y0, X0),                 Y)
        end-of-head(L7),                        :-
        create-premise(->/2, 1, L7),          (Y -> X) -> &axiom(s(X))
```

```
        take-next-clause,
L7      push-premise(->/2, 1, L9),
        goto(L18, s/1, 0),
L9      push-premise(call/1, 1, L11),
        goto(L14, call/1),
L11     put-value(Y0, X0),
        call(call/1),
        end-of-premise,
L14     put-value(Y1, X0),
        call(call/1),
        restore-premises,
        end-of-premises,
L18     put-value(Y1, X0),
        call(s/1),
        restore-premises,
        end-of-clause.
```

```
(Y ->X)

Y

(Y

   ->

      X)

         ->

            s(X)
```

This also needs some explanation. The "predicate" call is a primitive which takes a term as argument. If it is currently instantiated to a structure or constant (not a number) it performs the same actions as if that structure were the goal. If it is not instantiated the only rule that currently could be used is the axiom. It should be pointed out that this is a restriction in the current definition of GAM, and that the uninstantiated goal in the theory could, of course, be matched against the program clauses as well.
The third argument in the goto-instruction in the code for s is the annotation value for the guiding primitive &axiom. The axiom rule can always use an assumption whatever the annotation value is, and therefore if the value is set to 0 initially, the only rule that could use the assumption is the axiom rule.

15. Conclusions and Future Work

To be able to do some performance tests we have implemented both the GAM and WAM in Common lisp, and made the two implementations as similar as possible. Then the same Prolog program was executed in the two implementations, and they are indicating that pure Prolog programs are executed in two-third´s of the time that it takes an "original" WAM machine executes it. If there are assumptions in the goal GAM runs the program up to two-third´s of the time an "original" WAM, depending on the assumptions.
In this implementation we have not made an effort to be as efficient as possible in all the steps the machine takes. For example, if there is no assumption, the axiom will still be tried and an assumption for the rules P⊢ and →⊢ will still be searched for before backtracking occurs (i.e. a choicepoint for the modes `axiom` and `left` is created although there is no assumption to examine). Here is certainly room for making the machine more efficient. Some of the registers could perhaps also be removed if another test conditions were used.

The code size for GCLA program compiled into GAM code is something like 10 - 20% larger as for a Prolog program compiled into WAM code. The increase is mostly due to the new code for P⊢.

Also in mode `left` and mode `axiom` the more expensive instruction unify-local-value is used even though it is not needed in all places (or actually the instructions unify-variable and unify-value have the same definition as unify-local-value in mode `axiom` and mode `left`). It is used to be sure that there will not be pointers from the heap to the stack. Where unify-local-value is needed and where it is not should be clarified, perhaps new instruction(s) should be introduced.

Another idea is to introduce three arguments in the put and get instructions. The first two are the same as in the WAM, while the third one should be used instead of the first one when the mode equals `right` or `axiom`. This should increase the possibility to use temporary registers instead of permanent registers.

Something that is also wanted is a delaying mechanism for variables, called delay or freeze [4]. With an ability to delay a goal until its variables are grounded, there is (perhaps) a possibility to implement the so-called anti-unification, or disunification, of the rule P⊢, and also to put constraints on the variables.

Yet another idea to get more efficient code for the rule P⊢ is that the unifiers, which is generated by the compiler of all the clauses in a predicate, could be stored together with the predicate in one way or another (i.e. the unifier as it is or by some, perhaps new, instructions). This should increase the efficiency of P⊢ when there are several clauses that are unifiable with each other in a predicate. This solution should then partially replace some of the existing instructions (for example, take-next-clause becomes obsolete).

The next step is to systematize the code for the rules. The idea is that the user should define the inference rules himself, and that a file containing this code, the so-called operators, should be loaded together with a program into the rule code area. The user should also have the possibility to change the inference mechanism under the execution from inside the program. How this will influence the code described herein is not clear, but the idea has some very nice properties, for example, it is clear that these operators could also be treated as definitions, and thus the semantics for them is clear.

However, this experimental work with GAM has shown that GCLA could be implemented efficiently, and that there are a lot of promising ways to examine towards a more efficient and flexible implementation.

Acknowledgement

I am indebted to a lot of people at SICS for many valuable and helpful discussions. Special thanks goes to Kent Boortz and Peter Olin for many valuable and fruitful discussions, to Lars-Henrik Eriksson and Mats Carlsson for several suggested improvements to this paper, to Vicki Carleson for language support (all language ererors are by me made after her careful work), to Torbjörn Åhs at UPMAIL for comments on both an earlier version of this paper and this paper and to the project leader, Lars Hallnäs, for his never ceasing flow of new ideas and faith in the project.

References

[1] K. Appleby, M. Carlsson, S. Haridi, D. Sahlin, *Garbage Collection for Prolog Based on WAM*, SICS Research Report R86009B

[2] M. Aronsson, L. Hallnäs, *GCLA, Generalized Horn Clauses as a Programming Language*, SICS Research Report R88014

[3] J. Barklund, *A Garbage Collection Algorithm for Tricia*, UPMAIL Technical report No. 37B

[4] M. Carlsson, *An Implementation of dif and freeze in the WAM*, SICS Research Report R86012

[5] L-H. Eriksson, L. Hallnäs, *A programming Calculus Based on Partial Inductive Definitions*, SICS Research Report R88013

[6] L. Hallnäs, *Partial Inductive Definitions*, SICS Research Report R86005C

[7]. L. Hallnäs, P.Schröder-Heister, *A Proof-Theoretic Approach to Logic Programming, I. Generalized Horn Clauses*, SICS Research Report R 88005

[8] P. Henderson, *Functional Programming*, Prentice/Hall 1980

[9]. D.H.D. Warren, *An Abstract Prolog Instruction Set*, SRI Technical Note 309, Menlo Park 1983

APPENDICES

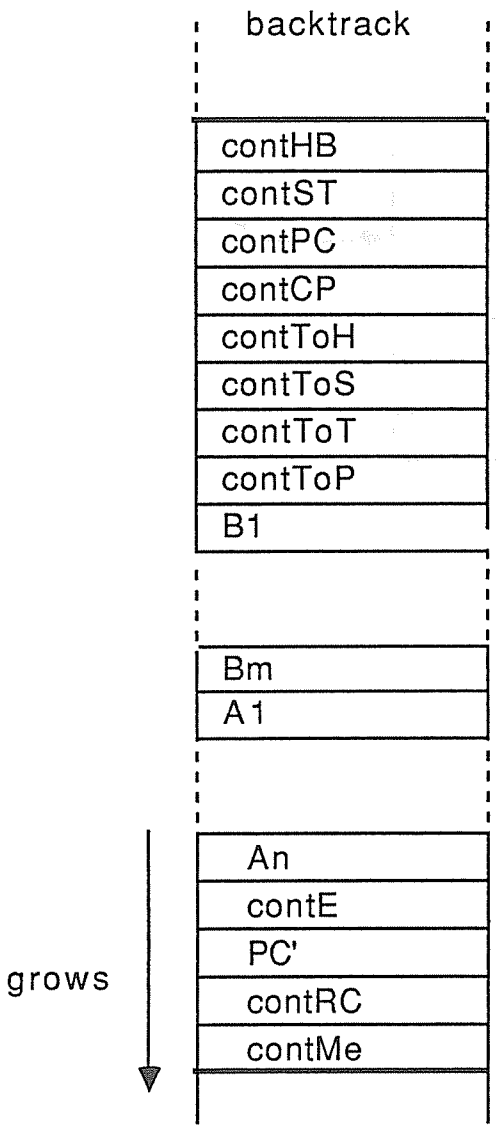
A Formats of Different Frames:

A.1 Choicepoints
A.2 Environments
A.3 Continuations
A.4 Premises
A.5 Different trail-formats

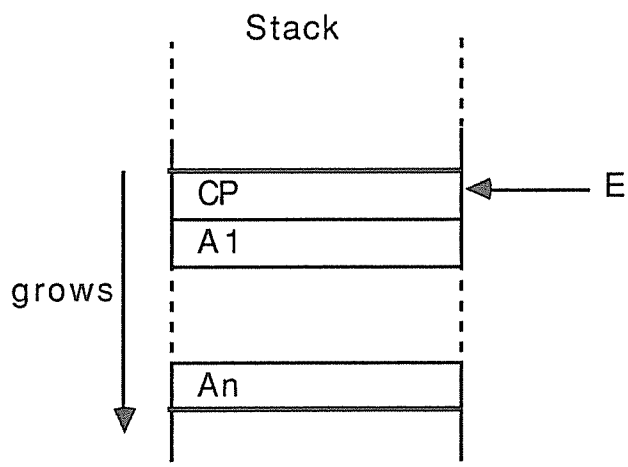
B Transitions Describing GAM

B.1 Calling Instructions
B.2 Premise Instructions
B.3 Indexing Instructions
B.4 Choicepoint Instructions
B.5 Guard Instructions
B.6 Fail Instruction
B.7 Get Instructions
B.8 Put Instructions
B.9 Unify Instructions
B.10 Rule Transitions

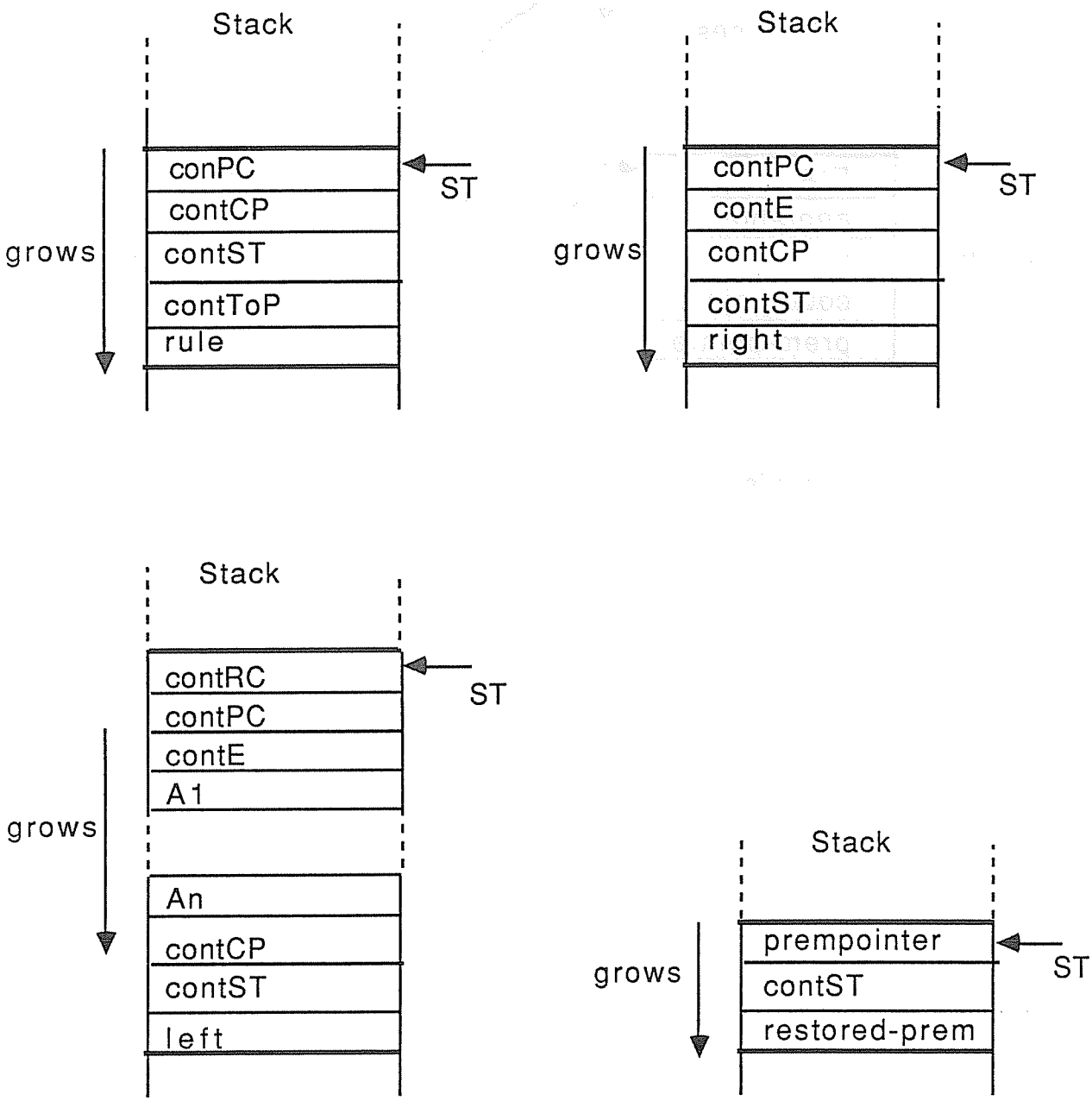
A.1 Choicepoints



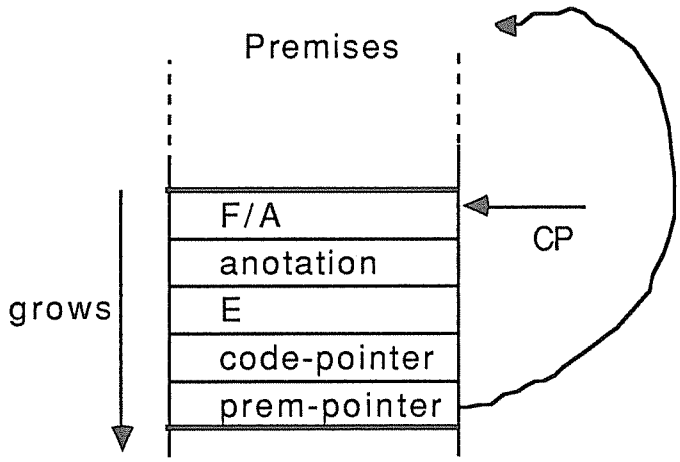
A.2 Environments



A.3 Continuations

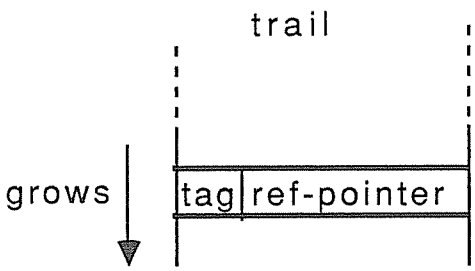


A.4 Premises

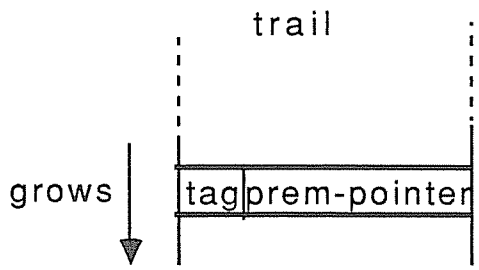


A premise

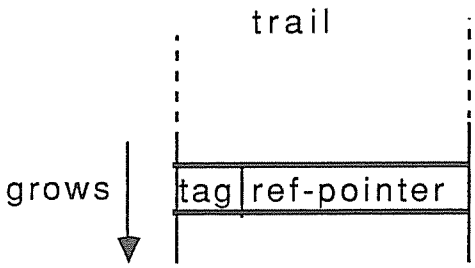
A.5 Different Trail Formats



A variable which should be unbound



A premise whose anotation-value should be increased by one



A premise whose anotation-value should be decreased by one

Appendix B

Transitions Describing GAM

This appendix does not pretend to be a logical description of the machine. We have tried to give a understandable and exact description without going into details that are obvious. Therefore we for example are talking about an element being in the heap without exactly pointing it out, or the "predicate" unify which updates the stack and/or the heap if there is a variable that should be bound. We think that this is sufficient, without loosing exactness. It should also be mentioned that this is a *description* of the machine. An implementation can be made in a more efficient way, for example in these transitions we have made use of the predicate unify at some places where it could be replaced by a more efficient one-way matching (e.g. the instruction get-constant).

We begin with a summary of the data areas and the registers:

Me	Mode, register holding the mode, one of rule, right, axiom, left. For the sake of clearness two additional modes are also defined: finish, which represents a successful computation, and fail which represents an unsuccessful computation.
CA	Current Assumptions, points to the leftmost premise (the last created).
CAP	Current Assumption Pointer, temporary pointer used to point at a specific chosen assumption.
TA	Treat As Absurd
CL	Clauses Left, a register holding the clauses left to unify with under the mode left
ST	Succeed To, points to the next continuation on the stack
PC	Program Counter
Sp	Structure pointer, help-register for unifying structures, points always into the heap. (The same register as in the WAM)
Um	Unification mode, read or write. (The same as in the WAM)
E	Environment pointer, points to the current environment on the stack
HB	Heap Backtrack pointer, points into the heap where the top of the heap was when the last continuation was created
RC	Right Caller, holds the functor/arity of the current consequence
S	Stack, contains environments and continuations
H	Heap, holds global values such as structures, lists etc (The same as in the WAM)
P	Premises, holds the defined premises, which forms a tree
B	Backtrack, a stack containing the choicepoints
T	Trail, a stack containing things to be undone upon backtracking
R	The argument registers for the consequence, numbered A1..An

L The argument registers for the premises, numbered B1..Bm

The registers below are not mentioned together with the other registers in the transitions, although they of course are part of the state. They are implicitly updated when a push or a pop are done on the stack, premises, heap, backtrack and trail respectively, or if something is said explicitly in the comments below the transition.

ToS Top of stack
ToH Top of heap
ToP Top of premises
ToB Top of backtrack
ToT Top of trail

There is a number of functions used in the transitions below. The functions are:

unify(X,Y) Tries to unify X and Y. If X and Y is unifiable, this relation is true and as a side effect variables on the stack and/or the heap can get bound. These bindings in the heap are trailed if necessary, that is, if the variable is in the heap, it is trailed if it occurs below HB, and if the variable occurs in the stack it is trailed if it occurs before the most recent choicepoint´s environment pointer.

deref(X) X is dereferenced to its value (an integer, constant, structure pointer, list pointer or reference pointer indicating an unbound variable)

push(X,Y) pushes X on the top of Y (i.e. in the cell pointed to by the corresponding top-of-area-pointer, described below. It also adds one to this pointer).

pop(Y) pops the topmost object from Y (i.e. returns the object in the cell just below the cell pointed to by top-of-area pointer described below, and subtracts this pointer with one).

bind(X,Y) X gets bound to Y

address_of(X) gives the address of X

value(X) gets the second half of X´s value (i.e. the next clause order)

next_plane(X) gets the first half of X´s value (i.e. the next list of possible clause orders)

constr_struct_pointer(X) construct a structure pointer

constr_list_pointer(X) construct a list pointer

struct_pointer(X) true if X is a structure pointer, false otherwise

list_pointer(X) true if X is a list pointer, false otherwise

bound(X) true if X is currently not unbound, false otherwise

unbound(X) true if X is currently unbound, false otherwise

These symbols should also be treated as functions:

$\in$ meaning that an element resides somewhere in the area, stack or list

'.' which is a concatenation
{...} means that the elements inside the brackets are sequentially pushed onto an area
$\leq$ and $\geq$ checking whether an address is greater that or equal to another
'_' denotes that we do not care about that cell´s value

The transitions should be read as: if the state in the upper row is fulfilled, and the if-statement (if there is one) also is fulfilled, then the state in the bottom row is the result.

B.1 The Calling Instructions

call

call(F/A)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
rule	CA	CAP	TA	CL	ToS-1	*strc*	Sp	Um	E	HB	F/A	*co*.S	H	P	B	T	R	L

---

where *strc* is the address of the first instruction of the rule code,

co is the continuation {right.ToP.ST.CA.E.PC} pushed sequentially on the stack

call(F/A)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	*locl*	ST	*fcou*	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L

if F/A is defined in the program

---

where *fcou* is the first label in the first list of the unifying order for the clause F/A ,

locl is the the rest of the labels in the first list of the unifying order for the clause F/A,

chpt is the choicepoint {left1.RC.PC.E.A1...An.B1...Bm.ToP.ToT.ToS.ToH.CA.(1.1).ST.HB} pushed sequentially on backtrack, the (1.1) in the third cell from the end in *chpt* represents that it is the first unifying order

in the first plane that is under consideration.

call(F/A)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for end-of-clause.

if F/A is <u>not</u> defined in the program

---

call(F/A)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for end-of-clause.

---

execute

execute(F/A)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
rule	CA	CAP	TA	CL	ST	*strc*	Sp	Um	E	HB	F/A	S	H	P	B	T	R	L

---

where *strc* is the address of the first instruction of the rule code,

execute(F/A)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for call(F/A).

---

execute(F/A)

axiom CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for end-of-clause.

---

end-of-premise

end-of-premise

axiom CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for end-of-clause.

---

end-of-premise

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for end-of-clause.

---

end-of-head

end-of-head(label)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

right CA CAP TA CL ST label Sp Um E HB RC S H P B T R L

---

end-of-head(label)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

left CA CA false CL ST PC+1 Sp Um E HB RC S H P B T R L

---

end-of-head(label)

axiom CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for end-of-clause.

---

take-next-clause

take-next-clause

left	CA	CAP	TA	null	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
rule	CA	CAP	TA	null	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

take-next-clause

left	CA	CAP	TA	f.r	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CAP	CAP	TA	r	ToS-1	f	Sp	Um	E	HB	RC	*cont*.S	H	P	B	T	R	L

where *cont* is the continuation {rule,ToP,ST,CA,PC}

end-of-clause

end-of-clause

Me	CA	CAP	TA	CL	null	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
finish	CA	CAP	TA	CL	null	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

end-of-clause

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$c \in$ S	H	P	B	T	R	L
right	cca	CAP	TA	CL	cst	cpc	Sp	Um	ce	HB	RC	S	H	P	B	T	R	L

where *c* is pointed to by ST, *c* is {right,cst,cca,ce,cpc} (*c* is popped if the most recent choicepoint's ST-value points below *c*).

end-of-clause

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$c \in$ S	H	P	B	T	R	L
rule	cca	CAP	TA	CL	cst	cpc	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

where *c* is pointed to by ST, *c* is {rule,cst,cca,cpc} (*c* is popped if the most recent choicepoint's ST-value points below *c*).

end-of-clause

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$c \in$ S	H	P	B	T	R	L
left	cca	CAP	TA	CL	cst	cpc	Sp	Um	ce	HB	crc	S	H	P	B	rp.T	A1..An	L

where *c* is pointed to by ST, *c* is {left,cst,cca,A1...An,ce,cpc,crc} (*c* is popped if the most recent choicepoint's ST-value points below *c*).

end-of-clause

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$c \in S$	H	$p \in P$	B	T	R	L
Me	CA	CAP	TA	CL	cst	*fieoc*	Sp	Um	E	HB	RC	S	H	$p' \in P$	B	T	*rp*.R	L

where *c* is pointed to by ST, *c* is {restored-prem,cst,ca} (*c* is popped if the most recent choicepoint's ST-value points below *c*).

p is a premise pointed to by ca in the continuation *c*. *p'* is the same premise as *p* except that the anotation value is increased by one.

fieoc is the first instruction of end-of-clause

rp is a tagged pointer to the premise *p'*.

allocate

allocate(N)

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC	Sp	Um	ToS	HB	RC	*e*.S	H	P	B	T	R	L

where *e* is a new environment with N permanent variables, all initiated to unbound

ToS is set to ToS + 1 + N

(i.e. $e = \{Y_0, Y_1, \ldots, Y_{N-1}, CA\}$, push(*e*, S) => *e*.S)

B.2 The Premise Instructions

push-premise

push-premise(F/A,anot,address)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	ToP+5	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	*np*.P	B	T	R	L

if deref(anot) is currently instantiated to a positive integer, otherwise an error is siganled and the execution stops

---

where *np* is the new premise {F/A,deref(anot),E,address,CA}, which is pushed on top of P

push-premise(F/A,anot,address)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ToS+7+n	address	Sp	Um	E	HB	RC	*sc*.S	H	P	B	T	A1..An	L

---

where *sc* is the saved continuation {left,ToP,ST,CA,A1..An,E,PC,RC} and n is the arity hold by RC

restore-premise

restore-premise

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
Me	*rca*	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L

---

where *rca* is the old CA value which is to be restored,

e is the current environment pointed to by E, *e* = {*rca*,Y1..Yn} where Y1..Yn are permanent variables

goto

goto(address,F/A,anot)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	address	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

---

goto(address,F/A,anot)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
rule	ToP+5	CAP	TA	CL	ST	*strc*	Sp	Um	E	HB	RC	S	H	*p*.P	B	T	R	L

if deref(anot) is currently instantiated to a positive integer, otherwise an error is siganled and the execution stops

---

where *p* is a new premise {F/A,deref(anot),E,address,CA} pushed on top of P

strc is the address of the first instruction of the rule code

create-premise

create-premise(address,anot,F/A)

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
Me	ToP+5	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	*p*.P	B	T	R	L

if deref(anot) is currently instantiated to a positive integer, otherwise an error is siganled and the execution stops

---

where *p* is a new premise {F/A,deref(anot),E,address,CA} pushed on top of P

B.3 The Indexing Instructions

These instructins are executed only in right mode

switch-on-term

switch-on-term(var,const,list,struct)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ST	var	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L

if the dereferenced value of A1 is unbound

---

switch-on-term(var,const,list,struct)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ST	const	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L

if the dereferenced value of A1 is a constant (i.e. a functor of arity 0)

---

switch-on-term(var,const,list,struct)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ST	list	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L

if the dereferenced value of A1 is a list

---

switch-on-term(var,const,list,struct)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ST	struct	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L

if the dereferenced value of A1 is a structure

---

switch-on-term(var,const,list,struct)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	L

if the dereferenced value of A1 is a unbound and var = fail or
the dereferenced value of A1 is a constant and const = fail or
the dereferenced value of A1 is a list and list = fail or
the dereferenced value of A1 is a structure and struct = fail

---

switch-on-structure

switch-on-structure([[F1/A1,label1] ... [Fn/An,labeln]],default)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST $label_i$ Sp Um E HB RC S H P B T A1..Am L

if Fi/Ai is equal to the dereferenced value of register A1

---

switch-on-structure([[F1/A1,label1] ... [Fn/An,labeln]],default)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST default Sp Um E HB RC S H P B T A1..Am L

if for all i, $1 \leq i \leq n$, Fi/Ai is not equal to the dereferenced value of register A1

---

where default is an address into the current clause

switch-on-structure([[F1/A1,label1] ... [Fn/An,labeln]],fail)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T A1..Am L

if for all i, $1 \leq i \leq n$, Fi/Ai is not equal to the dereferenced value of register A1

---

where *fiof* is the address of the first instruction of the procedure fail

switch-on-constant

switch-on-constant([[F1,label1] ... [Fn,labeln]],default)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST $label_i$ Sp Um E HB RC S H P B T A1..Am L

if Fi is equal to the dereferenced value of register A1

---

switch-on-constant([[F1,label1] ... [Fn,labeln]],default)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST default Sp Um E HB RC S H P B T A1..Am L

if for all i, $1 \leq i \leq n$, Fi is not equal to the dereferenced value of register A1

---

where default is an address into the current clause

switch-on-constant([[F1,label1] ... [Fn,labeln]],fail)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T A1..Am L

right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T A1..Am L

if for all i, $1 \leq i \leq n$, Fi is not equal to the dereferenced value of register A1

---

where *fiof* is the address of the first instruction of the procedure fail

B.4 The Choicepoint´s instructions

try

try(label)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	label	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm

where *chpt* is the choicepoint {right.RC.PC.E.A1...An.B1...Bm.ToP.ToT.ToS.ToH.CA.PC+1.ST.HB} pushed sequentially on backtrack

retry

retry(label)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	label	Sp	Um	E	HB	RC	S	H	P	*chpt'*.B	T	A1..An	B1..Bm

where *chpt* is the choicepoint {right.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.PC".ST.HB} popped from backtrack

chpt' is the choicepoint {right.RC.PC.E.A1...An.B1...Bm.ToP.ToT.ToS.ToH.CA.PC+1.ST.HB} pushed sequentially on backtrack

trust

trust(label)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	label	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	B1..Bm

where *chpt* is the choicepoint {right.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.PC".ST.HB} popped from backtrack

try-me-else

try-me-else(label)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm

where *chpt* is the choicepoint {right.RC.PC.E.A1...An.B1...Bm.ToP.ToT.ToS.ToH.CA.label.ST.HB} pushed sequentially on backtrack

retry-me-else

retry-me-else(label)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	*chpt'*.B	T	A1..An	B1..Bm

---

where *chpt* is the choicepoint {right.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.PC".ST.HB} popped from backtrack

chpt' is the choicepoint {right.RC.PC.E.A1...An.B1...Bm.ToP.ToT.ToS.ToH.CA.label.ST.HB} pushed sequentially on backtrack

trust-me-else-fail

trust-me-else-fail

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	A1..An	B1..Bm
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	A1..An	B1..Bm

---

where *chpt* is the choicepoint {right.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.PC".ST.HB} popped from backtrack

B.5 The Guard Instructions

These instructions should always occur before the instruction end-of-head, and is terefore never executed in axiom mode.

var

var(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is unbound

---

where Xi ∈ L

var(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is unbound

---

where Xi ∈ R

var(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not unbound

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ L

var(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not unbound

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

nonvar

nonvar(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

left CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is not unbound

---

where Xi ∈ L

nonvar(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is not unbound

---

where Xi ∈ R

nonvar(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is unbound

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ L

nonvar(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is unbound

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

number

number(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is a number

---

where Xi ∈ L

number(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is a number

---

where Xi ∈ R

number(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not a number

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ L

number(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not a number

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

ground

ground(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not bound to anything containing an unbound variable

---

where Xi ∈ L

ground(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is not bound to anything containing an unbound variable

---

where Xi ∈ R

ground(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if Xi is bound to something containing an unbound variable

---

where *fiof* is the first instruction of the procedure fail

Xi ∈ L

ground(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is bound to something containing an unbound variable

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

atom

atom(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is bound to a constant (i.e. an functor of arity 0)

---

where Xi ∈ L

atom(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is bound to a constant (i.e. an functor of arity 0)

---

where Xi ∈ R

atom(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is not bound to a constant (i.e. an functor of arity 0)

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ L

atom(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is not bound to a constant (i.e. an functor of arity 0)

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

atomic

atomic(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is bound to an constant (i.e. an functor of arity 0) **or** an integer

---

where Xi ∈ L

atomic(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST PC+1 Sp Um E HB RC S H P B T R L
if Xi is bound to an constant (i.e. an functor of arity 0) **or** an integer

---

where Xi ∈ R

atomic(Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is not bound to an constant (i.e. an functor of arity 0) **and** not bound to an integer

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ L

atomic(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if Xi is not bound to an constant (i.e. an functor of arity 0) **and** not bound to an integer

---

where *fiof* is the first instruction of the procedure fail
Xi ∈ R

B.6 The Fail Instruction

fail

fail

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	null	T	R	L
fail	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	null	T	R	L

---

fail

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L
right	cCA	CAP	false	CL	cST	cPC	Sp	Um	cE	cHB	F/A	S'	H'	P'	*chpt*.B	T'	A1..An	B1..Bm

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {right, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cPC, cST, cHB}

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L
rule	cCA	CAP	false	CL	cST	*absp*	Sp	Um	cE	cHB	F/A	S'	H'	P'	*chpt*.B	T'	A1..An	B1..Bm

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {axiom, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cPC, cST, cHB}

absp is the instruction to jump to when backtracking should occur in axiom mode (somewhere in the rule code, not specified in this report). The search for a premise should continue from cPC.

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L
rule	cCA	CAP	false	CL	cST	cPC	Sp	Um	cE	cHB	F/A	S'	H'	P'	B	T'	A1..An	B1..Bm

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {rule, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cPC, cST, cHB}

nrt is the next try of a rule, if there is any left. cPC points into the rule code, which is not specified in this report.

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me	CA	CAP	TA	nil	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L
left	cCA	CAP	TA	*rcl*	cST	*ntfuo*	Sp	Um	cE	cHB	F/A	S'	H'	P'	*chpt'*.B	T'	A1..An	B1..Bm

if not empty(value(cUO)) and not value(cOU) = *otherwise*

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cUO, cST, cHB}

chpt' is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, nextUO, cST, cHB}

nextUO is the next unifying order

rcl is the rest of the clauses to try, i.e. rest of the unifying order pointed to by cUO

ntfuo is the first label in the unifying order pointed to by cUO

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me	CA	CAP	true	nil	ST	PC	Sp	Um	E	HB	RC	S	H	P	*chpt*.B	T	R	L
left	cCA	CAP	true	nil	cST	*fioeoc*	Sp	Um	cE	cHB	F/A	S'	H'	P'	B	T'	A1..An	B1..Bm

if empty(value(cUO)) **and** empty(next-plane(cUO))

After the new state is initialized, call the instruction end-of-clause

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cUO, cST, cHB}

fioeoc is the first instruction of end-of-clause

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or

[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me CA CAP true nil ST PC Sp Um E HB RC S H P *chpt*.B T R L

left cCA CAP true *rcl* cST *ntfuo* Sp Um cE cHB F/A S' H' P' *chpt'*.B T' A1..An B1..Bm

if empty(value(cUO)) **and** not empty(next-plane(cUO))

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cUO, cST, cHB}

chpt' is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, (*np*. 1), cST, cHB}

np is the next plane

rcl is the rest of the clauses to try, i.e. rest of the unifying order in the first order of the next plane

ntfuo is the first label in the unifying order in the first order of the next plane

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or

[if restore_contraction_value(c) then add_one(c)] or

[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me CA CAP false nil ST PC Sp Um E HB RC S H P *chpt*.B T R L

left cCA CAP false nil cST *fiof* Sp Um cE cHB F/A S' H' P' B T' A1..An B1..Bm

if empty(value(cUO))

After the new state is initialized, backtracking occurs again

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cUO, cST, cHB}

fiof is the first instruction of the procedure fail

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or

[if restore_contraction_value(c) then add_one(c)] or

[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me CA CAP TA nil ST PC Sp Um E HB RC S H P *chpt*.B T R L

left cCA CAP TA nil cST *fiof* Sp Um cE cHB F/A S' H' P' B T' A1..An B1..Bm

if value(cUO) = otherwise

After the new state is initialized, backtracking occurs again

---

where *chpt* is the most recent choicepoint (the topmost),
chpt is the choicepoint {left1, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCA, cUO, cST, cHB}
fiof is the first instruction of the procedure fail
T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'
(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))
ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

fail

Me	CA	CAP	TA	nil	ST	PC	Sp	Um	E	HB	RC	S	H	*p*∈P	*chpt*.B	T	R	L
left	cCA	cCAP	TA'	nil	cST+3	*ppc*	Sp	Um	*pE*	cHB	F/A	*c*.S'	H'	*p*'∈P'	B	*up*.T'	A1..An	B1..Bm

if empty(next-premise(cCAP)) **and** anot-value(cCAP) > 0

---

where *chpt* is the most recent choicepoint (the topmost),
chpt is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCAP, cUO, cST, cHB}
c is the "continuation" {restored-prem,ST,CAP}
p is the premise pointed to by cCAP
p' is the premise *p* after its anotation value has been reduced by one
ppc is the address stored in the next premise in whose anotation value is greater than 0, pointed to by cCAP
pE is the environment pointer stored in the premise *p*.
T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'
up is a tagged pointer to the premise *p'* whose anotation value should be restored upon backtracking
(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))
ToS, ToH, ToP are reset to cToS, cToH, cToP respectively
TA' is set to false except when the functor of the premise *p* is the arrow "->" when it is set to true.

fail

Me	CA	CAP	TA	nil	ST	PC	Sp	Um	E	HB	RC	S	H	*p*∈P	*chpt*.B	T	R	L
left	cCA	cCAP	false	nil	cST+3	*fiof*	Sp	Um	*pE*	cHB	F/A	*c*.S'	H'	*p*'∈P'	B	*up*.T'	A1..An	B1..Bm

if empty(next-premise(cCAP)) **and** anot-value(cCAP) = 0

---

where *chpt* is the most recent choicepoint (the topmost),
chpt is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCAP, cUO, cST, cHB}
c is the "continuation" {restored-prem,ST,CAP}
p is the premise pointed to by cCAP
p' is the premise *p* after its anotation value has been reduced by one

fiof is the first instruction of the procedure fail

pE is the environment pointer stored in the premise *p*.

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

up is a tagged pointer to the premise *p'* whose anotation value should be restored upon backtracking

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

TA' is set to false except when the functor of the premise *p* is the arrow "->" when it is set to true.

fail

Me CA CAP TA nil ST PC Sp Um E HB RC S H *p*∈P *chpt*.B T R L

left cCA cCAP TA' nil cST+3 *ppc* Sp Um *pE* cHB F/A *c*.S' H' *p'*∈P' *chpt'*.B *up*.T' A1..An B1..Bm

if not empty(next-premise(cCAP)) **and** anot-value(cCAP) > 0

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCAP, cUO, cST, cHB}

chpt' is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, next-prem, cUO, cST, cHB}

and next-prem is the pointer returned by nedxt-premise(cCAP)

c is the "continuation" {restored-prem,ST,CAP}

p is the premise pointed to by cCAP

p' is the premise *p* after its anotation value has been reduced by one

ppc is the address stored in the next premise in whose anotation value is greater than 0, pointed to by cCAP

pE is the environment pointer stored in the premise *p*.

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

up is a tagged pointer to the premise *p'* whose anotation value should be restored upon backtracking

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

TA' is set to false except when the functor of the premise *p* is the arrow "->" when it is set to true.

fail

Me CA CAP TA nil ST PC Sp Um E HB RC S H *p*∈P *chpt*.B T R L

left cCA cCAP false nil cST+3 *fiof* Sp Um *pE* cHB F/A *c*.S' H' *p'*∈P' *chpt'*.B *up*.T' A1..An B1..Bm

if empty(next-premise(cCAP)) **and** anot-value(cCAP) = 0

---

where *chpt* is the most recent choicepoint (the topmost),

chpt is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, cCAP, cUO, cST, cHB}

chpt' is the choicepoint {left2, F/A, _, cE, A1..An, B1..Bm, cToP, cToT, cToS, cToH, next-prem, cUO, cST, cHB}

and next-prem is the pointer returned by nedxt-premise(cCAP)

c is the "continuation" {restored-prem,ST,CAP}

p is the premise pointed to by cCAP

p' is the premise *p* after its anotation value has been reduced by one

fiof is the first instruction of the procedure fail

pE is the environment pointer stored in the premise *p*.

T' is the trail after it has been unwound to cToT from ToT, and as this proceeds, things pointed to by the elements in T are reset to their previous value, creating S', H' and P'

up is a tagged pointer to the premise *p'* whose anotation value should be restored upon backtracking

(i.e. for all c, ToT < c < cToT ([if variable_pointer(c) then reset(c)] or
[if restore_contraction_value(c) then add_one(c)] or
[if decrease_contraction_value(c) then sub_one(c)]))

ToS, ToH, ToP are reset to cToS, cToH, cToP respectively

TA' is set to false except when the functor of the premise *p* is the arrow "->" when it is set to true.

B.7 The Get Instructions

get-variable

get-variable(Xj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R'	L

---

where the temporary variable Xj in R is bound to the dereferenced value of Xi in R creating R'
(i.e. Xi ∈ R, Xj ∈ R, Xi ∈ R' and bind(Xj,deref(Xi)) => R')

get-variable(Yj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	*e'*∈S	H	P	B	T	R	L

---

where *e* is the environment pointed to by E
e' is the environment where the permanent variable Yj in *e* is bound to the dereferenced value of Xi in R
(i.e. Xi ∈ R, Yj ∈ *e* and bind(Yj, deref(Xi)) => *e'*)

get-variable(Xj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L'

---

where the temporary variable Xj in L is bound to the dereferenced value of Xi in L creating L'
(i.e. Xi ∈ L, Xj ∈ L, Xi ∈ L' and bind(Xj, deref(Xi)) => L')

get-variable(Yj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	*e'*∈S	H	P	B	T	R	L

---

where *e* is the environment pointed to by E
e' is the environment where the permanent variable Yj in *e* is bound to the dereferenced value of Xi in L
(i.e. Xi ∈ L, Yj ∈ *e*and bind(Yj, deref(Xi)) => *e'*)

get-value

get-value(Xj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L
if	unify(deref(Xi),deref(Xj))																

---

where the dereferenced value of Xj in R is unifyed with the dereferenced value of Xi in R creating S' and H'
(i.e. Xi ∈ R, Xj ∈ R and unify(deref(Xi),deref(Xj) => S', H')

get-value(Yj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	*e'*∈S'	H'	P	B	T	R	L
if						unify(deref(Xi),deref(Yj))												

---

where *e* is the environment pointed to by E
the dereferenced value of Yj is unifyed with the dereferenced value of Xi, creating S' and H'
(i.e. Xi ∈ R, Yj ∈ *e*, and unify(deref(Xi),deref(Yj) => S', H')

get-value(Xj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
if						not unify(deref(Xi),deref(Xj))												

---

where the dereferenced value of Xj in R is not unifyable with the dereferenced value of Xi in R
fiof is the address to the first instruction of the procedure fail
Xi ∈ R, Xj ∈ R

get-value(Yj,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
if						not unify(deref(Xi),deref(Yj))												

---

where *e* is the environment pointed to by E
the dereferenced value of Yj in *e* is not unifyable with the dereferenced value of Xi
fiof is the address to the first instruction of the procedure fail
Xi ∈ R, Yj ∈ *e*

get-value(Xj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L
if						unify(deref(Xi),deref(Xj))												

---

where the dereferenced value of Xj in L is unifyed with the dereferenced value of Xi in L creating S' and H'
(i.e. Xi ∈ L, Xj ∈ L, and unify(deref(Xi),deref(Xj) => S', H')

get-value(Yj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	*e'*∈S'	H'	P	B	T	R	L
if						unify(deref(Xi),deref(Yj))												

---

where *e* is the environment pointed to by E

the dereferenced value of Yj is unifyed with the dereferenced value of Xi, creating S' and H'
(i.e. Xi ∈ L,Yj ∈ *e*, and unify(deref(Xi),deref(Yj) => S', H')

get-value(Xj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if not unify(deref(Xi),deref(Xj))

---

where the dereferenced value of Xj in L is not unifyable with the dereferenced value of Xi in L
fiof is the first instructon of the procedure fail
Xi ∈ L, Xj ∈ L

get-value(Yj,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	*e*∈S	H	P	B	T	R	L

if not unify(deref(Xi),deref(Yj))

---

where *e* is the environment pointed to by E
the dereferenced value of Yj in *e* is not unifyable with the dereferenced value of Xi in L
fiof is the address of the first instruction of the procedure fail.
Xi ∈ L, Yj ∈ *e*

get-constant

get-constant(C,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L

if bound(deref(Xi)) and C = deref(Xi)

---

where the constant C is bound with the dereferenced value of Xi in A creating S' or H', depending on where the dereferenced value is located
(i.e. Xi ∈ R, and (deref(Xi) ∈ S, bind(deref(Xi),C) => S', H = H' or
deref(Xi) ∈ H, bind(deref(Xi),C) => H', S = S')

get-constant(C,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if not unify(C,deref(Xi))

---

where *fiof* is the address to the first instruction of the procedure fail.
Xi ∈ R

get- constant (C,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L

if unify(C,deref(Xi))

---

where the dereferenced value of Xi in L is unified with the constant C creating S' or H', depending on where the dereferenced value is located
(i.e. Xi ∈ L, and (deref(Xi) ∈ S, bind(deref(Xi),C) => S', H = H' or
deref(Xi) ∈ H, bind(deref(Xi),C) => H', S = S')

get-constant(C,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if not unify(C,deref(Xi))

---

where *fiof* is the address to the first instruction of the procedure fail.
Xi ∈ L

get-structure

get-structure(F/A,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S'	F/A.H'	P	B	T	R	L

if unbound(deref(Xi))

---

where the dereferenced value of Xi in R is bound to a structure pointer pointing at the top of the heap, and F/A is pushed onto the heap
(i.e. Xi ∈ R, (deref(Xi) ∈ S, bind(deref(Xi),constr_struct_pointer(ToH)) => S', H = H', push(F/A,H') => F/A.H' or
deref(Xi) ∈ H, bind(deref(Xi),constr_struct_pointer(ToH)) => H', S = S', push(F/A,H') => F/A.H')

get-structure(F/A,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	F/A∈H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	deref(Xi)	read	E	HB	RC	S	F/A∈H	P	B	T	R	L

if bound(deref(Xi)), F/A = deref(Xi)

---

where the dereferenced value of Xi in R is equal to a structure pointer pointing at the functor F/A
Xi ∈ A

get-structure(F/A,Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if bound(deref(Xi)) **and** F/A ≠ deref(Xi)

---

where *fiof* is the address of the first instruction of the procedure fail.
Xi ∈ R

get-structure(F/A,Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST PC+1 Sp write E HB RC S' F/A.H' P B T R L
if unbound(deref(Xi))

---

where the dereferenced value of Xi in L is bound to a structure pointer pointing at the top of the heap, and F/A is pushed onto the heap
(i.e. Xi ∈ L, (deref(Xi) ∈ S, bind(deref(Xi),constr_struct_pointer(ToH)) => S', H = H', push(F/A,H') => F/A.H' or
deref(Xi) ∈ H, bind(deref(Xi),constr_struct_pointer(ToH)) => H', S = S', push(F/A,H') => F/A.H')

get-structure(F/A,Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S F/A∈H P B T R L
left CA CAP TA CL ST PC+1 deref(Xi) read E HB RC S F/A∈H P B T R L
if bound(deref(Xi)) **and** F/A = deref(Xi)

---

where the dereferenced value of Xi in L is equal to a structure pointer pointing at the functor F/A
Xi ∈ L

get-structure(F/A,Xi)

left CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
left CA CAP TA CL ST *fiof* Sp Um E HB RC S H P B T R L
if bound(deref(Xi)) **and** F/A ≠ deref(Xi)

---

where *fiof* is the address of the first instruction of the procedure fail.
Xi ∈ C

get-list

get-list(Xi)

right CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L
right CA CAP TA CL ST PC+1 Sp write E HB RC S' H' P B T R L
if unbound(deref(Xi))

---

where the dereferenced value of Xi in R is bound to a list pointer pointing at the topmost cell in H, giving S' or H'

(i.e. Xi ∈ R, (deref(Xi) ∈ S,bind(deref(Xi),constr_list_pointer(ToH)) => S', H = H' or
deref(Xi) ∈ H,bind(deref(Xi),constr_list_pointer(ToH)) => H', S = S')

get-list(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	*car*∈H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	deref(Xi)	read	E	HB	RC	S	*car*∈H	P	B	T	R	L

if bound(deref(Xi)) **and** list_pointer = deref(Xi)

---

where the dereferenced value of Xi in R is equal to a list pointer pointing at *car*.
Xi ∈ R

get-list(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if bound(deref(Xi)) **and** not list_pointer = deref(Xi)

---

where *fiof* is the address of the first instruction of the procedure fail.
Xi ∈ R

get-list(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	H	P	B	T	R	L

if unbound(deref(Xi))

---

where the dereferenced value of Xi in L is bound to a list pointer pointing at the topmost cell in H, giving S' or H'
(i.e. Xi ∈ L, (deref(Xi) ∈ S,bind(deref(Xi),constr_list_pointer(ToH)) => S', H = H' or
deref(Xi) ∈ H,bind(deref(Xi),constr_list_pointer(ToH)) => H', S = S')

get-list(F/A,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	*car*∈H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	deref(Xi)	read	E	HB	RC	S	*car*∈H	P	B	T	R	L

if bound(deref(Xi)) **and** list_pointer = deref(Xi)

---

where the dereferenced value of Xi in L is equal to a list pointer pointing at *car*.
Xi ∈ L

get-list(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if bound(deref(Xi)) **and** not list_pointer(deref(Xi))

---

where *fiof* is the address of the first instruction of the procedure fail.
Xi ∈ R

get-nil

get-nil(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L

if unbound(deref(Xi))

---

where the dereferenced value of Xi in R is bound to the constant nil
(i.e. Xi ∈ A, and (deref(Xi) ∈ S, bind(deref(Xi), nil) => S', H' = H or
deref(Xi) ∈ H, bind(deref(Xi), nil) => H', S' = S)

get-nil(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if bound(deref(Xi)) **and** nil = deref(Xi)

---

where the dereferenced value of Xi in R is equal to the constant nil
Xi ∈ R

get-nil(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if bound(deref(Xi)) **and** not nil = deref(Xi)

---

where *fiof* is the address of the first instruction of the procedure fail.
Xi ∈ R

get-nil(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S'	H'	P	B	T	R	L

if unbound(deref(Xi))

---

where the dereferenced value of Xi in R is bound to the constant nil
(i.e. Xi ∈ L, and (deref(Xi) ∈ S, bind(deref(Xi), nil) => S', H' = H or
deref(Xi) ∈ H, bind(deref(Xi), nil) => H', S' = S)

get-nil(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if nil = deref(Xi)

---

where the dereferenced value of Xi in L is equal to the constant nil

Xi ∈ L

get-nil(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

if bound(deref(Xi)) **and** not nil = deref(Xi)

---

where *fiof* is the address of the first instruction of the procedure fail.

Xi ∈ L

B.8 The Put Instructions

put-variable

put-variable(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for get-value(Xj,Xi) with Me = right

---

put-variable(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in S$	H	P	B	T	R	L
axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in S$	H	P	B	T	R	L

This is the same transition as for get-value(Yj,Xi) with Me = right

---

put-variable(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	ToH.H	P	B	T	R'	L

---

where R' is the argument registers after Xi and Xj have been bound to the top of the heap.
(i.e. Xi ∈ R, Xj ∈ R, [bind(Xi, ToH), bind(Xj, ToH)] => R', push(ToH,H) => ToH.H)

put-variable(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in S$	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in S$	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	$e' \in S$	H	P	B	T	R'	L

---

where *e* is the environment pointed at by E, *e'* is the environment after Yj is bound to its own address
R' is the argument registers where Xi has been bound to Yj's address.
(i.e. Yj ∈ *e*, Xi ∈ R, bind(Yj, address_of(Yj)) => *e'*, bind(Xi, deref(Yj)) => R')

put-variable(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as put-value(Xj,Xi)

---

put-variable(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as put-value(Yj,Xi)

---

put-value

put-value(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
axiom																		

This is the same transition as for get-value(Xj,Xi) with Me = right

---

put-value(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in$ S	H	P	B	T	R	L
axiom																		

This is the same transition as for get-value(Yj,Xi) with Me = right

---

put-value(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right																		
right						PC+1											R'	

---

where R' is the argument registers after Xi has been bound to what Xj is bound to.
(i.e. Xi ∈ R, Xj ∈ R, bind(Xi, Xj) => R')

put-value(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in$ S	H	P	B	T	R	L
right																		
right						PC+1											R'	

---

where R' is the argument registers after Xi has been bound to what Yj is bound to.
(i.e. Xi ∈ R, Yj ∈ *e*, bind(Xi, Yj) => R')

put-value(Xj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left																		
left						PC+1												L'

---

where C' is the argument registers after Xi has been bound to what Xj is bound to.
(i.e. Xi ∈ L, Xj ∈ L, bind(Xi, Xj) => L')

put-value(Yj,Xi)

	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	$e \in$ S	H	P	B	T	R	L
left																		
left						PC+1												L'

---

where C' is the argument registers after Xi has been bound to what Yj is bound to.
(i.e. Xi ∈ L, Yj ∈ *e*, bind(Xi, Yj) => L')

put-constant

put-constant(C,Xi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for get-constant(C,Xi) with Me = right

---

put-constant(C,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R'	L

---

where A' is the argument registers after Xi has been bound to C.
(i.e. Xi ∈ R, bind(Xi, C) => R')

put-constant(C,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L'

---

where L' is the argument registers after Xi has been bound to C.
(i.e. Xi ∈ L, bind(Xi, C) => L')

put-structure

put-structure(F/A,Xi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for get-strucure(F/A,Xi) with Me = right

---

put-structure(F/A,Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	F/A.H	P	B	T	R'	L

---

where R' is the argument registers after Xi has been bound to a structure pointer pointing at the top of the heap.
(i.e. Xi ∈ R, bind(Xi, constr_struct_pointer(ToH)) => R', push(F/A, H) => F/A.H)

put-structure(F/A,Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	F/A.H	P	B	T	R	L'

---

where L' is the argument registers after Xi has been bound to a structure pointer pointing at the top of the heap.
(i.e. Xi ∈ L, bind(Xi, constr_struct_pointer(ToH)) => L', push(F/A, H) => F/A.H)

put-unsafe-value

put-unsafe-value(Yj,Xi,Esize)

axiom CA CAP TA CL ST PC Sp Um E HB RC S H P B T R L

This is the same transition as for get-value(Yj,Xi) with Me = right

---

put-unsafe-value(Yj,Xi,Esize)

right CA CAP TA CL ST PC Sp Um E HB RC *e*∈S H P B T R L

right CA CAP TA CL ST PC+1 Sp Um E HB RC *e'*∈S' ToH.H P B ToH.T R' L

if unbound(deref(Yj)), deref(Yj) ∈ *e*

---

where *e* is the environment pointed to by E,

e' is *e* after Yj has been bound to the top of the heap,

R' is the argument registers after Xi has been bound to top of the heap.

(i.e. Xi ∈ R, Yj ∈ *e*, bind(Xi,ToH) => R', bind(Yj, ToH) => *e'*, push(ToH,H) => ToH.H)

put-unsafe-value(Yj,Xi,Esize)

right CA CAP TA CL ST PC Sp Um E HB RC S H *e*∈P B T R L

if not deref(Yj) ∈ *e*

This is the same transition as for put-value(Yj,Xi).

---

where *e* is the environment pointed to by E

put-unsafe-value(Yj,Xi,Esize)

left CA CAP TA CL ST PC Sp Um E HB RC *e*∈S H P B T R L

left CA CAP TA CL ST PC+1 Sp Um E HB RC *e'*∈S' ToH.H P B ToH.T R L'

if unbound(deref(Yj)), deref(Yj) ∈ *e*

---

where *e* is the environment pointed to by E,

e' is *e* after Yj has been bound to the top of the heap,

L' is the argument registers after Xi has been bound to top of the heap.

(i.e. Xi ∈ L, Yj ∈ *e*, [bind(Xi,ToH), bind(Yj, ToH)] => L', push(ToH,H) => ToH.H)

put-unsafe-value(Yj,Xi,Esize)

left CA CAP TA CL ST PC Sp Um E HB RC *e*∈S H P B T R L

if not deref(Yj) ∈ *e*

This is the same transition as for put-value(Yj,Xi).

---

where *e* is the environment pointed to by E

put-list

put-list(Xi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for get-list(Xi) with Me = right

---

put-list(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	H	P	B	T	R'	L

---

where R' is the argument registers after Xi has been bound to a list pointer pointing at the top of the heap.
(i.e. Xi ∈ R, bind(Xi, constr_list_pointer(ToH)) => R')

put-list(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	H	P	B	T	R	L'

---

where L' is the argument registers after Xi has been bound to a list pointer pointing at the top of the heap.
(i.e. Xi ∈ L, bind(Xi, constr_list_pointer(ToH)) => L')

put-nil

put-nil(Xi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as for get-nil(Xi) with Me = right

---

put-nil(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R'	L

---

where R' is the argument registers after Xi has been bound to the constant nil
(i.e. Xi ∈ R, bind(Xi, nil) => R')

put-nil(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

left	CA	CAP	TA	CL	ST	PC+1	Sp	Um	E	HB	RC	S	H	P	B	T	R	L'

where L' is the argument registers after Xi has been bound to the constant nil
(i.e. Xi ∈ L, bind(Xi, nil) => L')

B.9 The Unify Instructions

unify-variable

unify-variable(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp+1	read	E	HB	RC	S	H	P	B	T	R'	L

where R' is the argument registers after Xi has been bound to what the structure pointer Sp points to (i.e. Xi ∈ R, bind(Xi, Sp) => R')

unify-variable(Yi)

right	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	$e \in$ S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp+1	read	E	HB	RC	$e' \in$ S	H	P	B	T	R	L

where *e* is the environment pointed to by E

e' is the environment after Yi is bound to what the structure pointer Sp points to (i.e. Yi ∈ *e*, bind(Yi, Sp) => *e'*)

unify-variable(Xi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Xi)

unify-variable(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Xi)

unify-variable(Yi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Yi)

unify-variable(Yi)

left	CA	CAP	TA	CL	ST	PC	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Yi)

unify-variable(Xi)

right	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	ToH.H	P	B	T	R'	L

---

where R' is the argument registers after Xi has been bound to the top of the heap, which is a new unbound cell
(i.e. Xi ∈ R, bind(Xi, ToH) => R', push(ToH,H) => ToH.H)

unify-variable(Yi)

right	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	*e*∈S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	*e'*∈S	ToH.H	P	B	T	R	L

---

where *e* is the environment pointed to by E
e' is the environment after Yi is bound to the top of the heap
(i.e. Yi ∈ *e*, bind(Yi, Sp) => *e'*)

unify-value

unify-value(Xi)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Xi)

---

unify-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	*e*∈S	H	P	B	T	R	L

This is the same transition as unify-local-value(Yi)

---

unify-value(Xi)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Xi)

---

unify-value(Yi)

right	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	*e*∈S	H	P	B	T	R	L
right	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	*e*∈S	*v*.H	P	B	T	R	L

---

where *e* is the environment pointed to by E
(i.e. Yi ∈ *e*, push(value(Yi), H) => *v*.H, *v* = value(Yi))

unify-value(Yi)

axiom	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L

This is the same transition as unify-local-value(Xi)

---

unify-value(Yi)

left CA CAP TA CL ST PC+1 Sp write E HB RC S H P B T R L

This is the same transition as unify-local-value(Xi)

---

unify-local-value

unify-local-value(Xi)

Me CA CAP TA CL ST PC Sp read E HB RC S H P B T R L

Me CA CAP TA CL ST PC+1 Sp+1 read E HB RC S H' P B T R L

if (Me = axiom or Me = right) **and** unify(deref(Xi), deref(Sp))

---

where H' is the heap after the dereferenced value of Xi has been unified with the dereferenced value of what the structure pointer Sp points to

(i.e. Xi ∈ R, unify(deref(Xi), deref(Sp)) => H')

unify-local-value(Xi)

Me CA CAP TA CL ST PC Sp read E HB RC S H P B T R L

Me CA CAP TA CL ST *fiof* Sp read E HB RC S H P B T R L

if (Me = axiom or Me = right) **and** not unify(deref(Xi), deref(Sp))

---

where The dereferenced value of Xi can not be unified with the dereferenced value of what the structure pointer Sp points to

fiof is the first instruction of the procedure fail.

(Xi ∈ R)

unify-local-value(Xi)

left CA CAP TA CL ST PC Sp read E HB RC S H P B T R L

left CA CAP TA CL ST PC+1 Sp+1 read E HB RC S H' P B T R L

if unify(deref(Xi), deref(Sp))

---

where H' is the heap after the dereferenced value of Xi has been unified with the dereference value of what the structure pointer Sp points to

(i.e. Xi ∈ L, unify(deref(Xi), deref(Sp)) => H')

unify-local-value(Xi)

left CA CAP TA CL ST PC Sp read E HB RC S H P B T R C

left CA CAP TA CL ST *fiof* Sp read E HB RC S H P B T R C

if not unify(deref(Xi), deref(Sp))

---

where The dereferenced value of Xi can not be unified with the dereference value of what the structure pointer Sp points to

fiof is the first indtruction of the procedure fail.
(Xi ∈ L)

unify-local-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	*e*∈S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp+1	read	E	HB	RC	*e*∈S	H'	P	B	T	R	L

if (Me = axiom or Me = right) **and** unify(deref(Yi), deref(Sp))

---

where *e* is the environment pointed to by E, where Yi should reside
H' is the heap after the value of Yi is unified with the dereferenced value of what Sp points to
(i.e. Yi ∈ *e*, unify(value(Yi), deref(Sp)) => H')

unify-local-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	*e*∈S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	*fiof*	Sp	read	E	HB	RC	*e*∈S	H'	P	B	T	R	L

if (Me = axiom or Me = right) **and** not unify(deref(Yi), deref(Sp))

---

where *e* is the environment pointed to by E, where Yi should reside
fiof is the first instruction of the procedure fail
(Yi ∈ *e*)

unify-local-value(Xi)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	*v*.H	P	B	T	R	L

if Me = axiom **or** Me = right

---

where the value of Xi is pushed on top of the heap
Xi ∈ R

unify-local-value(Xi)

left	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
left	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	*v*.H	P	B	T	R	L

if Me = axiom **or** Me = right

---

where the value of Xi is pushed on top of the heap
Xi ∈ L

unify-local-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	*e*∈S	H	P	*chpt*.B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	*e*∈S´	ToH.H	P	*chpt*.B	deref(Yi).T	R	L

if (Me = axiom **or** Me = right) **and** deref(Yi) ∈ S **and** deref(Yi) ≤ get_E(chpt)

---

where *e* is the environment pointed to by E where Yi should reside

S´ is the environment after deref(Yi) is bound to a new unbound variable on the top of the heap
get_E(*chpt*) gets the environment pointer stored in the choicepoint *chpt*
(i.e. Yi ∈ *e*, bind(deref(Yi), ToH) => S´, push(ToH, H) => ToH.H, push(deref(Yi),T) => deref(Yi).T)

unify-local-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	*e*∈S	H	P	*chpt*.B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	*e*∈S´	ToH.H	P	*chpt*.B	T	R	L

if (Me = axiom **or** Me = right) **and** deref(Yi) ∈ S **and** deref(Yi) > get_E(chpt)

---

where *e* is the environment pointed to by E where Yi should reside
S´ is the environment after deref(Yi) is bound to a new unbound variable on the top of the heap
get_E(*chpt*) gets the environment pointer stored in the choicepoint *chpt*
(i.e. Yi ∈ *e*, bind(deref(Yi), ToH) => S´, push(ToH, H) => ToH.H)

unify-local-value(Yi)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	*e*∈S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	*e*∈S	*v*.H	P	B	T	R	L

if (Me = axiom **or** Me = right) **and** deref(Yi) ∈ H

---

where *e* is the environment pointed to by E
(i.e. Yi ∈ *e*, push(value(Yi), H) => *v*.H, *v* = value(Yi))

unify-constant

unify-constant(C)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp+1	read	E	HB	RC	S	H'	P	B	T	R	L

if unify(C, deref(Sp))

---

where H' is the heap after the constant C has been unified with the dereference value of what
the structure pointer Sp points to
(i.e. unify(C, deref(Sp)) => H')

unify-constant(C)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	*fiof*	Sp	read	E	HB	RC	S	H	P	B	T	R	L

if not unify(C, deref(Sp))

---

where *fiof* is the first instruction of the procedure fail.

unify-constant(C)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	C.H	P	B	T	R	L

---

where push(C,H) => C.H

unify-nil

unify-nil

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp+1	read	E	HB	RC	S	H'	P	B	T	R	L

if unify(nil, deref(Sp))

---

where H' is the heap after the constant nil has been unified with the dereference value of what the structure pointer Sp points to
(i.e. unify(nil, deref(Sp)) => H')

unify-nil

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	*fiof*	Sp	read	E	HB	RC	S	H	P	B	T	R	L

if not unify(nil, deref(Sp))

---

where *fiof* is the first instruction of the procedure fail.

unify-nil

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	nil.H	P	B	T	R	L

---

where push(C,H) => C.H

unify-void

unify-void(N)

Me	CA	CAP	TA	CL	ST	PC	Sp	read	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp+N	read	E	HB	RC	S	H	P	B	T	R	L

---

unify-void(N)

Me	CA	CAP	TA	CL	ST	PC	Sp	write	E	HB	RC	S	H	P	B	T	R	L
Me	CA	CAP	TA	CL	ST	PC+1	Sp	write	E	HB	RC	S	H'	P	B	T	R	L

---

where H' is the heap where N unbound cells have been pushed onto the heap

B.10 The Rule Transitions

The transitions below describes the machine in mode `rule`. The rule code (guiding code) should perform the things described in here.

---

rule	CA	CAP	TA	CL	ST	*strc*	Sp	Um	E	HB	RC	S	H	*p*∈P		B	T	R	L
axiom	CA	CAP	TA	CL	ST	pPC	Sp	Um	pE	HB	RC	S	H	*p*∈P	*chpt2.chpt*.B		T	R	L

if there is a premise *p* s.t. its main functor is equal to the value of RC

---

where *strc* is the address of the first instruction of the rule code,

p is the first premise found in P with equal "name" as RC, start searching from the value of CA

chpt is the choicepoint {rule.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.*fiotpr*.ST.HB} where *fiotptr* in the third cell from the end represents that the next choice is to jump to the first instruction of trying the mode right.

chpt2 is the choicepoint {axiom.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.next-p.ST.HB} where next-p is the next premise to test if *p* is not unifyable with the current argument registers in R

pPC is the PC value stored in *p*

pE is the environment pointer stored in the premise *p*

---

rule	CA	CAP	TA	CL	ST	*strc*	Sp	Um	E	HB	RC	S	H	P		B	T	R	L
rule	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	*chpt*.B		T	R	L

if there is no premise *p* s.t. its main functor is equal to the value of RC

---

where *strc* is the address of the first instruction of the rule code,

fiof is the first instruction of the procedure fail

chpt is the choicepoint {rule.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.*fiotpr*.ST.HB} where *fiotptr* in the third cell from the end represents that the next choice is to jump to the first instruction of trying the mode right.

---

rule	CA	CAP	TA	CL	ST	*fioptr*	Sp	Um	E	HB	RC	S	H	P	B		T	R	L
right	CA	CAP	TA	CL	ST	fPC	Sp	Um	E	HB	RC	S	H	P	B		T	R	L

if RC is defined in the program

---

where *fiotptr* is the first instruction of trying with Me set to right

fPC is the first GAM instruction in the procedure with RC

---

rule	CA	CAP	TA	CL	ST	*fioptr*	Sp	Um	E	HB	RC	S	H	P	B		T	R	L
right	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B		T	R	L

if RC is not defined in the program

---

where *fiotptr* is the first instruction of trying with Me set to right
fiof is the first instruction of the procedure fail

---

rule	null	CAP	TA	CL	ST	*altpl*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L
left	null	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	P	B	T	R	L

---

where *altpl* is the first instruction of trying with Me set to left
fiof is the first instruction of the procedure fail

---

rule	CA	CAP	TA	CL	ST	*altpl*	Sp	Um	E	HB	RC	S	H	*p*∈P	B	T	R	L
left	CA	CAP	TA	CL	ST	*fiof*	Sp	Um	E	HB	RC	S	H	*p*∈P	B	T	R	L

if get_premise_for_left(CA) = null

---

where *altpl* is the first instruction of trying with Me set to left
fiof is the first instruction of the procedure fail
get_premise_for_left(CA) returns a premise *p* if there is a premise *p* whose anotation value is greater than zero, otherwise null

---

rule	CA	CAP	TA	CL	ST	*altpl*	Sp	Um	E	HB	RC	S	H	*p*∈P	B	T	R	L
left	CA	CA	TA'	CL	ST+3	pPC	Sp	Um	pE	HB	RC	*co*.S	H	*p'*∈P	B	*up*.T	R	L

if get_premise_for_left(CA) ≠ null **and** get_next_prem(*p*) = null

---

where *altpl* is the first instruction of trying with Me set to left
p' is the *p* when its anotation value has been reduced by one
up is a tagged pointer to the premise *p'* to increase its anotation value by one upon backtracking
co is {restored-prem,cst,ca} sequentially pushed onto the stack
get_premise_for_left(CA) returns a premise *p* if there is a premise *p* whose anotation value is greater than zero, otherwise null
get_next_prem(prem) returns the next premise in the list, which is stored as a pointer in *p* (i.e. the CA value when *p* was created).
pPC is the PC value stored in *p*
pE is the environment pointer stored in the premise *p*
TA' is equal to false if *p*'s mainfunctor is the arrow ->, otherwise it is set to true

---

rule	CA	CAP	TA	CL	ST	*altpl*	Sp	Um	E	HB	RC	S	H	*p*∈P	B	T	R	L
left	CA	CA	TA'	CL	ST+3	pPC	Sp	Um	pE	HB	RC	*co*.S	H	*p'*∈P	*chpt*.B	*up*.T	R	L

if get_premise_for_left(CA) **and** get_next_prem(*p*) ≠ null

---

where *altpl* is the first instruction of trying with Me set to left

chpt is the choicepoint {left2.RC.PC'.E.A1...An.B1'...Bm'.ToP.ToT.ToS.ToH.CA.next-p.ST.HB} where next-p

p' is the *p* when its anotation value has been reduced by one

up is a tagged pointer to the premise *p'* to increase its anotation value by one upon backtracking

co is {restored-prem,cst,ca} sequentially pushed onto the stack

is the next premise (i.e. what is returned by get_next_premise(*p*))

get_premise_for_left(CA) returns a premise *p* if there is a premise *p* whose anotation value is greater than zero, otherwise null

get_next_prem(prem) returns the next premise in the list, which is stored as a pointer in *p* (i.e. the CA value when *p* was created).

pPC is the PC value stored in *p*

pE is the environment pointer stored in the premise *p*

TA' is equal to false if *p's* mainfunctor is the arrow ->, otherwise it is set to true