Experiments with enhancing a Tabled Logic Programming System with Constraint Search Trees

Daniel Aldman

October 22, 2003
Abstract

The XXX system is a tabled logic programming system which currently uses tries for storing and accessing terms in tables. This master thesis investigates the implementation of a more efficient method of storing constraints in the XXX system, using constraint search trees. The thesis discusses how the trie data structures were altered to cater for constraint search trees and reports on the system’s performance after the introduction of the constraint search trees.
Contents

1 Introduction
   1.1 The aim and the goals of this thesis ........................................... 1
   1.2 Implementation framework ......................................................... 2

2 Tabling
   2.1 The description ........................................................................... 3
   2.2 An example .................................................................................. 4

3 Basics of the emulator
   3.1 Cells ......................................................................................... 6
   3.2 The heap and the stack ............................................................... 7
   3.3 The choicepoint stack .............................................................. 8

4 Storing constraints in attributed variables
   4.1 The storage used ....................................................................... 9
   4.2 Example attributes ..................................................................... 10

5 Constraints and lists thereof
   5.1 Constraint data structure ........................................................... 11
   5.2 Constraint lists ......................................................................... 11
   5.3 Algorithm for finding all constraints in a call or answer ............. 12

6 Constraint search trees
   6.1 The logical data structure .......................................................... 13
   6.2 The specific data structure ....................................................... 14
   6.3 Algorithm for insertion ............................................................. 14
   6.3.1 Example insertion .................................................................. 15
   6.4 Algorithm for extraction .......................................................... 16
   6.5 True nodes and empty sets of constraints ................................. 16

7 Tabling Tries
   7.1 Structure of a trie node .............................................................. 17
   7.2 The old structure of tries ........................................................... 18
   7.2.1 Attributed variables in the old structure ............................... 19
   7.3 The new structure of tries ........................................................ 19
   7.3.1 Attributed variables in the new structure ............................. 19
   7.4 Examples ................................................................................. 19
8 Connecting the constraint search tree to the tries 22
  8.1 Call and answer generation ........................................ 22
  8.2 Returning answers .................................................... 22
  8.3 Trie deletion ............................................................ 22

9 Performance evaluation 23
  9.1 The results .............................................................. 23

10 Conclusions 28
  10.1 Redundancy in CSTs .................................................. 28
  10.2 Limitations on attributes .......................................... 28
  10.3 Construction time .................................................... 28
Chapter 1

Introduction

Top-down evaluators of logic programming, like those typically used in Prolog systems have disadvantages. (For more information about Prolog, see [5].) One of the main disadvantages is that a specific class of queries does not terminate. This class of queries is distinguished by the fact that the query ends up asking a query. Which is the same up to variable renaming. For example, given the following predicates:

Program 1

\ [% parent(X,Y) means: Y is the parent of X. parent(daniel, claes). parent(claes, bertil). parent(bertil, joel).\]

\ [% ancestor(X,Y) means: Y is the ancestor of X. ancestor(X,Y) :- parent(X,Y). ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).\]

the query ?- ancestor(daniel, X). does not terminate for this very reason. Although in this case there exists a very simple solution to the problem, namely to rewrite the ancestor predicate like so:

Program 2

\ [ancestor(X,Y) :- parent(X,Y). ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).\]

Such a solution may not be possible to find, due to the nature of the program.

One way of removing some nontermination is to use a technique called tabling; see e.g. [4].

Tabling not only ensures termination for a larger class of programs than Prolog-style execution does, but it can also improve the execution time of a program. The naive Fibonacci function is an example of this.

Program 3

\ [fib(0, 1). fib(1, 1).\]
fib(N, R) :-
    N > 1,
    N1 is N-1,
    N2 is N-2,
    fib(N1, R1),
    fib(N2, R2),
    R is R1 + R2.

Normally, the naïve implementation of the Fibonacci function has an $O(2^n)$ time complexity on
the input number, as in most programming languages, but since tabling stores answers to queries,
executing the recursive Fibonacci function above using tabling has an $O(n)$ time complexity.

A separate issue from that of tabling, that is also of interest, is that when a program like this
one:

**Program 4**

p(X) :- X > 30.

is given the query ?- p(X) , the query would only return an error, telling the user that you can’t
directly compare an unbound variable with a value. This is where Constraint Logic Programming
[2] comes in. Using constraints, program 4 could return an answer in the form ‘yes, but only if
X > 30’.

When using Constraint Logic Programming, each variable that has some form of constraint on
its possible values is given an attribute. The variable is then regarded as an attributed variable,
which is in some cases handled differently than normal variables. These cases are discussed in
chapter 3. As an example, when inserting an attributed variable into a tabling trie\(^1\), the attribute
must be inserted in such a way that whenever the variable is asked for again, the attributes are also
returned. The standard way to implement this is to insert the actual structure of the attribute into
the tabling trie, which is not a memory efficient way of storing constraints. This is where constraint
search trees, which will be refered to in this document as CSTs for brevity, are able to help by
storing all variables’ constraints in one data structure, to reduce redundancy and memory usage.

### 1.1 The aim and the goals of this thesis

Attributed variables are used to express constraints on variables. Before the implementation of
constraint search trees, the attributes of attributed variables were stored in the tabling tries. Using
constraint search trees to store the constraints allows for more efficient storage. Finding a working
implementation of constraint search trees and their connection to the tabling tries is the goal of this
master thesis.

### 1.2 Implementation framework

The work was limited to modifying the existing system, by adding constraint search trees. While the
choice of data structure and algorithms connected with the constraint search trees were completely
free, the semantics of constraint search trees were set from the start. The same applies to the way
the tabling system worked.

\(^{1}\) Tries will be discussed in chapter 7

2
Chapter 2

Tabling

The following description of tabling is the result of applying a different perspective on it than the ones that were studied as part of the preparation for this thesis. It assumes that the reader has a basic knowledge of queries, predicates, variables and unification in Prolog.

The purpose of this description of tabling is not to explain any particular variant of tabling, and especially not the tabling used by the XSB/XXX Prolog system. It is meant to give readers who are not familiar with the idea of tabling an idea of what tabling is. The description here is as if the tabling system acts somewhat like an operating system, which is in theory a good implementation and would work just fine. It is not a preferable implementation in practice, however, mainly because the time taken to generate copies of processes outweighs the actual execution time of the processes themselves. A more comprehensive explanation of tabling systems in general, and the XSB/XXX tabling system in particular can be found in [4] or [1].

2.1 The description

When using tabling to handle evaluation of a logic program, the system can be viewed as a triple $T = (P, Q, L)$, composed of three components: processes, queries and listboxes, and a set of rules that describe the actions taken to compute the results of queries.

- A process $p \in P$ is defined as a tuple $(\psi, l, V)$ consisting of a series of atoms $\psi$, a reference to a return value listbox $l$ and a mapping $V: \text{Variable} \to \text{Value}$ of variable instantiations.

- A query $q \in Q$ is defined as a tuple $(P, V, l)$ consisting of a predicate $P$, a mapping $V: \text{Variable} \to \text{Value}$ of variable instantiations and a listbox.

  Two atomic queries are considered equal if they are identical up to variable renaming. Two mappings are considered equal if, for each variable instantiation of a mapping, the other mapping contains an equal variable instantiation. Finally, two variable mappings are considered equal if they both refer to the same variable and value. The listbox reference is not considered when comparing two queries.

- A listbox $l \in L$ functions a lot like a standard mailbox; a queue of messages which either returns a message or suspends the execution of the calling process pending a new message being put in the queue. The differences from most implementations of mailboxes are that a listbox is unlimited in length and that all inserted messages are retained after delivery. Each process which requests messages from the listbox will eventually receive any message put in
the mailbox in the order they were inserted. A further difference from mailboxes is that a message is not inserted into a listbox if it is not unique.

When the system is first initiated, it creates the tuple $T = (\emptyset, \emptyset, \emptyset)$. When a tabled predicate is queried by the query $q$, the system attempts to insert $q$ into $Q_T$ (the set $Q$ in the tuple $T$) using the $Q$-insert rules:

- If $q \notin Q_T$ then the system creates a listbox $l$ and a set of processes $P'$, each of which corresponds to a program clause matching the query $q$. These processes’ listbox reference is set to $l$, and the mapping is set to the same as in $q$. Then $T \leftarrow (P_T \cup P', \{q\} \cup Q_T, \{l\} \cup L_T)$ and $l$ is returned to the calling process, which will use $l$ to retrieve results.

- Otherwise, since $q \in Q_T$, the listbox which was originally created when $q$ was inserted into $Q_T$ using the rule above is returned to the calling process.

A scheduler is used to create new processes and, through calls to listboxes, suspend a process’ execution. It is also used to select which process gets to execute next. For the latter, no specific algorithm is assumed to be used, however a round-robin approach is used in the example below.

Each process follows the $P$-execution rules for evaluation:

- If the first predicate in $\psi_p$, of the process $p \in P_T$, is a tabled predicate, then create a query from the first conjunction and use the $Q$-insert rules to get a listbox $l$, from which to retrieve the results of the predicate. For each result retrieved, create a process $p'$, which has the same logical expression as $p$ except for the first conjunct, the same return value listbox reference as $p$ and which has the same variable-to-value mapping as $p$, except that the mapping is altered in respect to the variable mapping gotten from $l$.

- If the first predicate in $\psi_p$, of the process $p \in P_T$, is not a tabled predicate, then execute it as in Prolog-style execution.

- If there are no further conjuncts, the process inserts its variable-to-value mapping $V$ into the its return value listbox. Then the process backtracks to find new answers. If the process has to backtrack to before its first conjunct to find a choicepoint, it terminates.

When all processes in $P$ are simultaneously waiting for messages to arrive in a listbox, i.e. a global deadlock has occurred, the scheduler knows that there are no more answers. Therefore, $T \leftarrow (\emptyset, Q_T, L_T)$.

### 2.2 An example

As an example, if the ancestor predicate in program 1 on page 1 was tabled, the same query would use $Q$-insert rule 1 to insert the query into the set of queries, create one receiving listbox for the answers and start two processes, one for each program clause. The first of these two processes queries for the parent of daniel, to which there is only one matching program clause, which is a fact. The answer to the query is then inserted into the return value listbox.

The first action that the second process takes is to issue a query for ancestors of daniel, which has already been queried for. Therefore, no new process is started, but the process is suspended on the return value listbox of that query, which happens to be its own. For each answer it retrieves from the listbox, the process creates a new sub-process, which handles the rest of the program clause, with the variables unified properly.
So, after the first process has entered its answer into its return value listbox, the answer is returned to the second process, which therefore creates another process, process 3. Process 3, in turn, continues by querying for the parent of class. The answer returned will be bertil. Process 3 would now enter the answer that bertil is an ancestor to daniel into its return value listbox, which allows the second process to start another sub-process with this new answer.

This procedure is repeated until all processes have terminated or been halted, i.e. a global deadlock has occurred. The system then knows that no further messages will be placed into the listboxes, and can therefore terminate all processes.
Chapter 3

Basics of the emulator

This chapter explains some parts of the Prolog emulator XXX, and how these parts work in relation to the constraint search trees.

3.1 Cells

The most basic data structure used in the emulator is the cell. The cell is a piece of data the size of the word length on the computer for which the system has been compiled. The lowest three bits are used to signify what kind of data the cell represents. A cell can represent an integer, a floating point number, a list, a structure, an atom, a variable or an attributed variable. Each of these has its own 3-bit identifier, or tag, to separate it from the others.

![Diagram of cell representation]

Figure 3.1: Representation of integers, floats and atoms

As indicated by figure 3.1, integers, floating point numbers and atoms all fit within one cell, so their value is tagged and placed in the cell. The value of an atom is a pointer to an atom table entry, where the strings containing the names of the atoms are stored.

As figure 3.2 shows, a list cons cell is stored as two consecutive cells on the heap. The first is a reference to the list element’s data. The second is a tagged pointer, referencing the next element in the list, or NULL if the element is the last in the list.

Figure 3.3 shows how a structure is represented. First comes a tagged pointer, referencing a section of memory indicating both the name and number of arguments of the structure, also known as the arity. The tagged pointer is immediately followed by the structure’s arguments.
Variables, as shown in Figure 3.4, are represented as pointers to the cells where what they are bound to lies, even if this is a variable. Free, or unbound, variables point to themselves. To make it easier to deal with variables, and because all variables point to a 4-byte aligned address, variables have two tags: 0 and 4. This means that variables don’t have to be encoded or decoded before their value is used.

Attributed variables, as shown in Figure 3.5, consist of three elements: two consecutive cells, the first of which is a free variable, the second the actual attributes. The attributes are stored as a structure v/N. The third element is that the reference to the first cell is tagged as an attributed variable, not as a variable.

### 3.2 The heap and the stack

The heap and the stack are storage areas for data created when the Prolog program executes. These areas are both in the same chunk of memory, growing towards each other from opposite sides. When there is no longer any room for the two to grow, the emulator takes a series of steps to make space. First of all, the heap is subjected to garbage collection, and if that fails to free up enough memory, the emulator allocates a new heap and stack area that is sufficiently big to accommodate the need.
3.3 The choicepoint stack

The way Prolog works, there is need for a stack of choicepoints. These choicepoints contain the data necessary to go back and try a different execution path, also known as backtracking. Among the things stored in a choicepoint is a pointer to the instruction to be executed next and a pair of pointers to what was the top of the stack and the heap, at the time when the choicepoint was created.
Chapter 4

Storing constraints in attributed variables

From within a Prolog program, storing constraints using attributed variables can be done by using the predicates \texttt{get_atts/2} and \texttt{put_atts/2}.

The \texttt{get_atts/2} predicate takes an attributed variable as its first argument, and in its second argument returns whatever previously has been associated with that variable. The \texttt{put_atts/2} predicate does the opposite: given a variable in its first argument, and a list in its second argument containing atoms and/or structures that have previously been declared to be attributes, it associates whatever is in the list with the variable in the first argument. If the first argument isn’t already an attributed variable, it also makes sure that it is.

4.1 The storage used

The extensions made in this master thesis require that there is only one declared attribute, \texttt{constraints/1}. This attribute has to contain a list of structures describing the constraints related to the variable. The table below lists all the different types of constraints that the extensions made in this master thesis recognize and how to construct them from Prolog.
<table>
<thead>
<tr>
<th>Prolog expression</th>
<th>Type of constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>X &lt; Y + c</td>
<td>x &lt; y + c</td>
</tr>
<tr>
<td>X &lt; Y</td>
<td>x &lt; y</td>
</tr>
<tr>
<td>X &lt; c</td>
<td>x &lt; c</td>
</tr>
<tr>
<td>X &lt;= Y + c</td>
<td>x &lt;= y + c</td>
</tr>
<tr>
<td>X =&lt; Y</td>
<td>x =&lt; y</td>
</tr>
<tr>
<td>X =&lt; c</td>
<td>x =&lt; c</td>
</tr>
<tr>
<td>X = Y + c</td>
<td>x = y + c</td>
</tr>
<tr>
<td>X = Y</td>
<td>x = y</td>
</tr>
<tr>
<td>X = c</td>
<td>x = c</td>
</tr>
<tr>
<td>X != Y + c</td>
<td>x != y + c</td>
</tr>
<tr>
<td>X != Y</td>
<td>x != y</td>
</tr>
<tr>
<td>X != c</td>
<td>x != c</td>
</tr>
<tr>
<td>X &gt;= Y + c</td>
<td>x &gt;= y + c</td>
</tr>
<tr>
<td>X =&lt; Y</td>
<td>x &gt;= y</td>
</tr>
<tr>
<td>X =&lt; c</td>
<td>x &gt;= c</td>
</tr>
<tr>
<td>X &gt; Y + c</td>
<td>x &gt; y + c</td>
</tr>
<tr>
<td>X &gt; Y</td>
<td>x &gt; y</td>
</tr>
<tr>
<td>X &gt; c</td>
<td>x &gt; c</td>
</tr>
</tbody>
</table>

4.2 Example attributes

As an example, the predicate p/2 in program 5 sets the constraints between the two variables in the argument to be \( X \leq Y + 4 \)

Program 5

```prolog
:- import set_atts/2 from atts.

% This has to be specified.
:- attribute constraints/1.

p(X, Y) :- set_atts(X, [constraints([X =< Y + 4])]),
           set_atts(Y, [constraints([X =< Y + 4])]).
```
Chapter 5

Constraints and lists thereof

This chapter will explain the data structure used to store a single constraint, as well as the data structure used to store multiple constraints. It will also explain how these constraint lists are gathered.

5.1 Constraint data structure

To store a constraint, the system uses a structure containing four elements: three cells of data and an enumerated value. The enumerated value denotes what kind of constraint the structure is representing. These values are named similarly to the atoms used in the Prolog-level to represent the same thing, see chapter 4.

The three cells of data represent the two variables and the constant in the constraint.

Program 6

typedef struct {
    enum {
        CST_less_than,
        CST_less_than_or_equal,
        CST_equal,
        CST_more_than_or_equal,
        CST_more_than
    } type;
    Cell variable_1, variable_2, constant;
} CST_constraint;

Two constraints are considered to be equal if all elements of their constraint structures are equal to one another.

5.2 Constraint lists

When storing more or less than one constraint, the constraint data structure above isn’t enough. For that purpose, constraint lists are used.

Constraint lists are ordinary singly linked lists with a CST_constraint structure as it’s data element.
5.3 Algorithm for finding all constraints in a call or answer

When a call or answer is to be inserted into the appropriate table, all the constraints referenced by that call have to be collected into a constraint list, which is then inserted into the constraint search tree. The collection is done using the following steps:

- First, the constraint list is set to the empty list.
- When going through the call or answer, if a new attributed variable is encountered, a function, `CL_insert_constraints`, is called to parse the attributed variable’s attributes.
- The function first checks that the attribute is a particular structure, v/2.
- The second argument of this structure should be a list. If it is, then the list is traversed to locate all the constraint structures\(^1\). When one is found, it is placed into the list of constraints.
- The constraint structure may contain a reference to another attributed variable, which is also checked to see if it is new. If it is, then the function is called recursively to handle that variable.
- When all attributes have been processed, the list of constraints is checked for duplicate elements, any that are found are removed.

\(^1\)those listed in chapter 4
Chapter 6

Constraint search trees

Note that the following discussion on CSTs is not a complete explanation of the data structure, but a discussion regarding the implementation chosen for this master thesis. For a more extensive explanation of constraint search trees, see [6].

This chapter will first explain the semantics of CST nodes and the trees themselves, then it will describe the actual data structure. Then the algorithms for inserting and extracting data will be described. Lastly, the way empty constraint sets are handled will be explained.

6.1 The logical data structure

A CST is a data structure for holding sets of constraints\(^1\). It is a binary tree with three types of nodes: slim, fat and true.

![Figure 6.1: The logical structure of a CST node](image)

The difference between the different node types is the number of constraints in a node. Slim nodes hold only one constraint, fat nodes hold more than one constraint and true nodes hold no constraints. In the implementation, each internal node is a slim node and each leaf node is either a slim node or a fat node. There is one exception to this rule and that is when handling empty sets of constraints, as explained in section 6.5. It should be noted that true nodes can only appear as the rightmost leaf of a CST.

An example CST is shown in figure 6.2.

\(^1\)The constraint sets are stored as lists, as discussed in chapter 5

13
6.2 The specific data structure

As is evident from listing 1, the CST nodes contain seven fields: an opcode used for backtracking purposes, the constraint list which holds the constraints, pointers to the parent node and the left and right children of the node, an integer that is used to check if a CST node is in a call or answer CST, and a pointer to the auxiliary data field. The auxiliary data field is NULL at all times except when the node represents a set of constraints. It then contains a pointer to a data structure holding more information about the call or answer that the node represents.

Listing 1

typedef struct CST_node *CST;

struct CST_node {
    #ifdef THREADED
        Cell instr_addr;
    #endif
        byte instr;
    #ifdef BITS64
        byte padding[7];
    #else
        byte padding[3];
    #endif
    CL constraints;
    void *auxdata;
    int is_in_answer_trie;
    CST parent, left, right;
};

6.3 Algorithm for insertion

When inserting a set of constraints into a CST, the following algorithm is applied, beginning at the root node:
• If the end of the CST has been reached, insert a new node containing all constraints.

• Otherwise, if the current node is a slim node, check whether or not the constraint in the current node is in the set of constraints that are to be inserted. If it is, then remove the constraint from the set of constraints and go down into the right subtree, otherwise go down into the left subtree.

• If the current node is a fat node, and the set of constraints that are to be inserted and the set of constraints in this node are not equal, the node is split. Splitting is accomplished by finding one constraint C in either the set of constraints in the current node or the set of constraints that are to be inserted, in that order, that is unique to its own set. C is then removed from its set and used to create a slim node N. The left subtree of N is made up of a fat node, containing the set of constraints that C was deleted from. The right subtree of N is made up of a fat node, containing the other set.

### 6.3.1 Example insertion

![Example CST before insertion](image1)

Figure 6.3: Example CST before insertion

To further explain the insertion algorithm, this example shows how two constraint sets are inserted into an example CST. The first constraint set consists of the constraints $X < 5$, $X > -7$ and $Y < X$. This set is inserted into the CST in figure 6.3. The marked nodes are nodes which represent a complete constraint set.

![Example CST after inserting the first constraint set](image2)

Figure 6.4: Example CST after inserting the first constraint set

Since the root node is a slim node, the new constraint list is tested to see if the constraint in the node is in the list. This holds, so the algorithm removes the constraint from the list of constraints that are to be inserted, and proceeds down into the left child. The left child is also a slim node, and this time the constraint is not in the list, so the list of new constraints is not modified. The algorithm moves into the right subtree, which is empty. Therefore the algorithm inserts a new node here, which is a fat node. The CST then looks as in figure 6.4.

The second constraint set consists of the constraints $X < 5$, $X > -7$, $Y < X$ and $Y < 3$. This insertion happens the same way as the last insertion, up until it reaches the new, fat, node. At this
point the algorithm splits the node, since the remaining constraint set is not identical to the set in the node. Splitting gives the discriminating element as $Y < 3$, which is in the new list. Had there been a discriminating element in the node, that element would have been used instead.

The discriminating element is at this point inserted into its own node and the old node is put as the right child, and the algorithm then creates another new node which it places as the left child of the other new node. The tree will then look as shown in figure 6.5.

### 6.4 Algorithm for extraction

When extracting a set of constraints out of a CST, the following algorithm is applied, starting at the node in the CST that represents that set of constraints.

- If the last node visited was the root node, stop.
- Otherwise, if the last node visited was the left child of the current node, or if the current node is the first node visited, add the constraint(s) in the current node to the collected set of constraints, then visit the current node’s parent.
- If the last node visited was the right child of the current node, then ignore the current node and move on to the current node’s parent.

### 6.5 True nodes and empty sets of constraints

The insertion algorithm outlined above works for non-empty sets of constraints only, but sometimes there will be no constraints to insert, since the set to insert is the empty set. In that case, the insertion algorithm used is the following, starting at the root node:

- If the end of the tree has been reached, then insert a true node as the right child.
- Otherwise, recursively insert into the right subtree.

As is evident from this algorithm, a true node’s parent is not always a slim node.
Chapter 7

Tabling Tries

The data structure that stores tabled calls and their answers is called a “tabling trie”, or simply a “trie”. For more information on tries, see [3].

This chapter will discuss tries from the perspective of the implementation of CSTs. It will not address all aspects of the implementation of tries in the system, nor will it address the exact nature of the data structure, since the actual data structure has not been changed by the implementation of the CSTs. The discussion will first address the parts of a trie node that are of interest, and then discuss the way trie nodes and other data structures were organized before and after the introduction of CSTs. The chapter finishes with an example of how a tabled call would be stored before and after the implementation of CSTs.

7.1 Structure of a trie node

Each trie node holds 5 pieces of information that are of interest in the following discussion: the trie node type, a data cell and pointers to the trie’s parent, child and sibling nodes. The actual data structure is defined as shown in listing 2.

Listing 2

typedef struct InstructionPlusTypeFrame {
    #ifdef THREADED
        Cell instr_addr;
    #endif
    byte instr;       /* contains compiled trie code */
    byte status;     /* whether the node has been deleted */
    byte trie_type;  /* global info: what type of trie is this? */
    byte node_type;  /* local info: what is the role of this struct? */
    #ifdef BITS64
        byte padding[4];
    #endif
} InstrPlusType;

typedef struct Basic_Trie_Node *BTNptr;
typedef struct Basic_Trie_Node {
    InstrPlusType info;

    17
BTNptr sibling;
BTNptr child;
BTNptr parent;
Cell symbol;
} BasicTrieNode;

There are, for the purpose of this discussion, only two kinds of trie node types: internal nodes and leaf nodes.

The data held in a trie node is of the same basic type as the heap - a cell. Since data in WAM stack space disappears during backtracking, there can be no references from a trie to this space. For this reason, the entire calls and answers are copied to the trie.

To handle non-linearity, each variable is stored in the trie as a tagged index rather than as a self-referencing pointer. Attributed variables are stored the same way, only with a different tag.

A trie is an N-ary tree. To accomplish the N-ary structure with a node of fixed size, each node has pointers to its right sibling and leftmost child.

### 7.2 The old structure of tries

When tabling a call and its answers, the emulator sets up a structure that is composed of four parts: A call trie and a subgoal frame (SGFrame), optionally followed by an answer trie and an answer substitution information structure (AS_info) per answer. The call trie stores the parameters of each call, the subgoal frame points out the associated answer trie, and the answer trie stores the new values of the variables in the call trie.

![Diagram of trie structure](image)

**Figure 7.1: The original macrostructure of the tabling tries**

In order for the emulator to be able to retrieve the answers of a tabled call, the answer trie has to be pointed out by the call trie. This is done by making the child pointer of the leaf node of the call trie point to the subgoal frame, which in turn points out the associated answer trie. Because the node is a leaf node, the child pointer is not assumed to contain a pointer to a valid trie node.
7.2.1 Attributed variables in the old structure

To accurately record information about attributed variables that are stored in a trie, the attribute of each variable must also be stored. Before using CSTs to represent the constraints, the attribute structure \( v/H \) was stored in the trie as it appeared on the heap, directly after the attributed variable.

7.3 The new structure of tries

The incorporation of CSTs have changed the structure of the tries.

![Diagram](image)

Figure 7.2: The new macrostructure of tabling tries and constraint search trees

Since the attributes of attributed variables have been removed and placed in a CST, this CST can represent several different calls. Therefore the CST has been made to point out the subgoal frames and answer substitution factors, while the call and answer tries point out the CSTs the same way they pointed out subgoal frames and answer substitution factors.

7.3.1 Attributed variables in the new structure

In the new structure, the attribute structures of attributed variables are no longer stored in the trie at all. An attributed variable still appears as it did before, but it is no longer followed by a constraint structure.

7.4 Examples

When looking at the logical structure of the new and the old structure, it might appear as if the new structure takes up more space than the old. This is not the case.
Program 7

:- attribute constraints/1.

p(X) :- put_attributes(X, [constraints([X < 5])]),
q(X).

table q/1.

q(X) :- put_attributes(X,
    [constraints([X < 3,
    X > -4])]).

Program 7 above, when queried p(X), would create the structure displayed in figure 7.3 using the old trie structure, or the structure displayed in figure 7.4 using the new trie structure.

![Figure 7.3: The structure of the first example trie in the old system](image)

Program 8

:- attribute constraints/1.

p(X) :- q(X, _Y).
table q/2.

\[
q(X, Y) :- \text{put\_attributes}(X, \\
\text{[constraints}([X < 3 \\
\quad X > -4]))]).
\]

Program 8 above, when queried \( p(X) \), would create the structure displayed in figure 7.5 using the new macrostructure.

Figure 7.4: The structure of the first example trie in the new system

Figure 7.5: The structure of the second example trie in the new system
Chapter 8

Connecting the constraint search tree to the tries

There were a number of changes that had to be made to the system to accommodate the CSTs.

8.1 Call and answer generation

When a new call or answer is ready to be processed, it is placed on the heap, and each cell of it is placed into its own trie node, which is inserted into the call or answer trie, respectively. When doing this, it is essential that every cell referenced is put into the trie. Before the implementation of CSTs it was therefore necessary to put an attributed variable’s attribute into the trie. Now that the attributes are stored in the CST, that part of the code was removed.

8.2 Returning answers

When retrieving an answer from a table, the system first creates a structure called the return skeleton in WAM stack space, whose arguments are references to the variables in the call which should be bound, or, in the case of attributed variables, changed, by the answer. Then the content of the answer trie is placed on the heap, and the dereferenced values of the return skeleton’s references are bound to it.

Each trie node contains an opcode; see listing 2 in chapter 7. This opcode is used to place the content of the answer trie onto the heap. The reason for using opcodes to return answers from completed tables is because it is faster than parsing the data structure. It also simplifies backtracking when there are more than one answer in the trie. Constraint search tree nodes contain no opcode, but when the end of the answer trie has been reached, a structure is allocated to hold data relevant to backtracking, including an opcode, the location of the return skeleton, and the next CST node to return an answer from.

8.3 Trie deletion

Because of the addition of constraint search trees, the code used to delete entries in the table had to be altered to handle the CSTs.
Chapter 9

Performance evaluation

This report has so far covered how the main tabling data structures were changed to accommodate for the CSTs, how the CSTs and constraint lists were implemented and how they relate to each other, and finally how the CSTs interact with the rest of the emulator.

The aim of this thesis was to implement CSTs, in order to reduce the amount of memory used when tabling terms with constraints. CSTs have been implemented and work as intended. The question is: how much memory, if any, does this implementation save?

For the performance evaluation, the following benchmark program was used.

Program 9

test(_):- abolish_all_tables, fail.
test(N):- X lt 3, Y gt 5, p(N,X,Y), fail.
test(_).

:- table p/3.
p(0,_,_).
p(N,X,Y):- N > 0, N1 is N-1, p(N1,X,Y), X lt Y + N.
p(N,X,Y):- N > 0, N1 is N-1, p(N1,X,Y).

Given a number N, test/1 creates N + 1 generators, N consumers, and inserts a total of $2^{N+1} - 1$ different constraint answers in the tables for p/3.

The results were gathered by calling the test/1 predicate, measuring how much time the call took to complete\(^1\) and how much space was used by the CST and Trie systems\(^2\). To get measurable times, the test was run 10 times. The results were then summed up and divided by 10.

9.1 The results

The reason the tests results for the CST system go further than the test results for the old system is because as the system stores more and more answers, eventually the memory for the application

\(^1\)Using the built-in predicate `cputime/1`.
\(^2\)Using the built-in predicate `statistics/0`.
<table>
<thead>
<tr>
<th>Size</th>
<th>with CSTs</th>
<th>without CSTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>2</td>
<td>59</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>62</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>63</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>78</td>
<td>83</td>
</tr>
<tr>
<td>7</td>
<td>116</td>
<td>97</td>
</tr>
<tr>
<td>8</td>
<td>187</td>
<td>154</td>
</tr>
<tr>
<td>9</td>
<td>314</td>
<td>252</td>
</tr>
<tr>
<td>10</td>
<td>594</td>
<td>474</td>
</tr>
<tr>
<td>11</td>
<td>1124</td>
<td>943</td>
</tr>
<tr>
<td>12</td>
<td>2288</td>
<td>1875</td>
</tr>
<tr>
<td>13</td>
<td>4455</td>
<td>3804</td>
</tr>
<tr>
<td>14</td>
<td>8276</td>
<td>7906</td>
</tr>
<tr>
<td>15</td>
<td>15215</td>
<td>15609</td>
</tr>
<tr>
<td>16</td>
<td>28715</td>
<td>32029</td>
</tr>
<tr>
<td>17</td>
<td>50878</td>
<td>64975</td>
</tr>
<tr>
<td>18</td>
<td>92817</td>
<td>134816</td>
</tr>
<tr>
<td>19</td>
<td>154206</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>274866</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>469717</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>765936</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1239674</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>2016477</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>3121408</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.1: Execution times (ms) with and without CSTs.

runs out\(^3\). This happens earlier for the non-CST system because it uses more memory, as is evident in the size tables and graphs.

The test results for the first 5 sizes are very small, making them very susceptible to outside influences. This causes the test results to appear very random.

The graph in figure 9.1 shows the time taken using the CST system and the time taken using the non-CST system. The shorter data set is the non-CST system.

The graph in figure 9.2 shows the space used using the CST system and the time taken using the non-CST system. Again, the shorter data set is the non-CST system.

The graph in figure 9.3 shows the amount of memory used by the normal system divided by the amount of memory used by the CST system. As it clearly shows, the CST system uses increasingly less space than the normal system.

\(^3\)The test machine was a 32-bit machine, so each process can only access 4 GB of memory. This test eventually uses more than that.
<table>
<thead>
<tr>
<th>Size</th>
<th>with CSTs</th>
<th>without CSTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>380</td>
<td>560</td>
</tr>
<tr>
<td>1</td>
<td>884</td>
<td>1768</td>
</tr>
<tr>
<td>2</td>
<td>1684</td>
<td>3840</td>
</tr>
<tr>
<td>3</td>
<td>3076</td>
<td>7928</td>
</tr>
<tr>
<td>4</td>
<td>5652</td>
<td>16624</td>
</tr>
<tr>
<td>5</td>
<td>10596</td>
<td>35688</td>
</tr>
<tr>
<td>6</td>
<td>20276</td>
<td>77792</td>
</tr>
<tr>
<td>7</td>
<td>39428</td>
<td>170584</td>
</tr>
<tr>
<td>8</td>
<td>77808</td>
<td>374252</td>
</tr>
<tr>
<td>9</td>
<td>153768</td>
<td>817536</td>
</tr>
<tr>
<td>10</td>
<td>304312</td>
<td>1777200</td>
</tr>
<tr>
<td>11</td>
<td>600352</td>
<td>384352</td>
</tr>
<tr>
<td>12</td>
<td>1170916</td>
<td>8269936</td>
</tr>
<tr>
<td>13</td>
<td>2230704</td>
<td>17712296</td>
</tr>
<tr>
<td>14</td>
<td>4044624</td>
<td>37776032</td>
</tr>
<tr>
<td>15</td>
<td>7407388</td>
<td>80262168</td>
</tr>
<tr>
<td>16</td>
<td>13377672</td>
<td>169952400</td>
</tr>
<tr>
<td>17</td>
<td>23751392</td>
<td>358769416</td>
</tr>
<tr>
<td>18</td>
<td>41484176</td>
<td>755277184</td>
</tr>
<tr>
<td>19</td>
<td>70308636</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>120637144</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>202378556</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>327350888</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>526420200</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>843634068</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>1328406356</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.2: Table space (bytes) used with and without CSTs.
Figure 9.1: Comparing the time taken in the new and old systems.

Figure 9.2: Comparing the space taken up in the new and old systems.
Figure 9.3: Comparing the space taken up in the new and old systems.
Chapter 10

Conclusions

The performance evaluation in chapter 9 showed the CST system’s time and memory requirements as compared to the previous system. This chapter will discuss other aspects of the CST system which may not be immediately apparent.

10.1 Redundancy in CSTs

There is a situation in which CSTs will store more constraints than necessary. This is when two constraint sets that have a large number of constraints that are equal are stored into a CST. In this case, the first constraint set is stored as a fat node, and the second only takes out one of the discriminating elements.

CSTs could of course be modified to handle this event, but doing so is outside the scope of this thesis.

10.2 Limitations on attributes

The implementation of CSTs has meant that constraints other than the ones recognized by the CST system are removed when tabling a call or answer. This, in turn, means that attributed variables returned from a tabled call will not contain any other attributes.

The CST system considers the list of constraints given to a variable as a set. This means that it will delete any duplicate constraints it finds. It also assumes that both the involved variables in a constraint should have that constraint as an attribute after being returned from a tabled predicate.

10.3 Construction time

While inserting a set of constraints in a trie is a linear time operation, doing the same in a constraint search tree has a $O(n \times m)$ worst-case execution time, where $n$ is the number of constraints to insert and $m$ is the height of the tree they are to be inserted in. This is due to the fact that the constraint list must be traversed at each node it passes.

The constraint lists are constructed using a linear time algorithm, placing each encountered constraint first in the list. The list is then subjected to an algorithm that is guaranteed to run in $O(n \times \log(n))$ time, where $n$ is the number of constraints in the constraint list, to delete duplicate constraints. Because a constraint is stored in the attributes of all the variables that are affected by
the constraint\(^1\), the algorithm will always find that at least half of the constraints referring to two variables are duplicates.

\(^1\)Which is one or two, as seen in chapter 4.
Bibliography


