A Kernel of Data Structures and Algorithms for Automated Deduction with Equality Clauses

Robert Nieuwenhuis, José Miguel Rivero and Miguel Angel Vallejo*

Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain
E-mail: {roberto,rivero,vallejo}@lsi.upc.es.
Phone/Fax: +34-3-4017330/7014

keywords: Automated deduction, first-order clauses with equality, implementation techniques, rewriting.

Abstract. We present *Dedam* (Deduction Abstact Machine), a standard architecture around which high-performance theorem provers for clausal logic with equality can be built. It includes a WAM-like heap structure for storing terms as DAG's, several indexing data structures, and mechanisms for efficient memory management and demodulation. These data structures turn out to be surprisingly well combinable due to conceptual similarities.
We provide statistics showing how we reduce two well-known bottlenecks in equational deduction. The first one, matching for demodulation is done by a new indexing method, substitution trees with equality constraints. Regarding the other bottle neck, memory management, WAM-based techniques allow us to completely avoid it during unification, matching, and demodulation. Static clause (sub)sets can be compiled in this framework into efficient abstract machine code for inference computation and redundancy proving.

1 Introduction

During the last decade, research on automated deduction in our group has mainly focussed on theoretical results for first-order logic with equality. New techniques for e.g., clausal rewriting and deduction with constrained clauses have been developed and completeness results established. Many necessary underlying results on term orderings, constraint solving and answer computation have been given, with their decidability and complexity characteristics. (cf. `http://www-lsi.upc.es/dept/sectp.html`). In order to better understand these new techniques and learn more about their practical behaviour, during their development we have always worked with prototype implementations written in Prolog. These experimental systems converged in 1992 into our laboratory

* This work has been partially supported by the European Esprit Working Group CCL-II, ref. 22457 and the European HCM network ConSolE.

implementation *Saturate* [NN93, GNN95]. Although it has been applied successfully in many contexts and has led to interesting new insight and theoretical results (see also [BG96, Nie96]), the *Saturate* system is not a high-performance theorem prover.

Therefore, in order to focus more on practice, some years ago we decided to work on efficiency-oriented systems as well. When studying the existing provers, we found a wide range of different *ad-hoc* data structures for implementing very similar calculi and control mechanisms. E.g., for term representation there are the linear flatterms [Chr93] and diverse other types of nonlinear terms, and for indexing, discrimination trees [Chr93, McC92], path indexing [Sti89, McC92] and substitution trees [Gra95b], among others.

In spite of the evident qualities of all these data structures, we somewhat missed a more uniform framework, like the Warren Abstract Machine (WAM) [War83] in logic programming. In such a framework structure sharing, indexing for all the main operations and perhaps even compilation of static parts of the clause sets should be possible in a standard, elegant, well-structured and reusable way. After a number of experiments with many techniques (most of them with rather negative results), we finally came to a kernel of data structures satisfying these requirements.

We chose a WAM-like heap structure for storing terms as DAG's, where structure sharing is possible but not forced. Regarding indexing, we developed several techniques based on substitution trees, which turn out to be surprisingly well combinable with WAM terms due to conceptual similarities. Many known techniques from the WAM, like variable binding, backtracking and memory management, which are the result of many years of work in logic programming implementation, can be smoothly incorporated here. Another requirement is satisfied as well: static clause (sub)sets can be compiled in this framework into efficient abstract machine code for inference computation and redundancy proving. All this led to a toolkit called *Dedam* (Deduction Abstract Machine) in which all basic operations (indexing, variable management, I/O, etc.) are provided, and on top of which one can build high-performance theorem provers in a simple way.

This paper is structured as follows. In section 2 we describe some basic ideas about our WAM-like term heap and about indexing data structures. Section 3 shows how unification is done and how sharing in our data structures leads to efficiency in time and space. In section 4 we describe our new indexing data structure for matching. Our method for compiling static clause (sub)sets into efficient code for inference computation and redundancy proving is shortly outlined in section 5. Finally, we show our techniques for efficient demodulation, and, in particular, how to avoid memory management by means of WAM-based techniques in section 6.

2 Heaps, Indexing and other Basic Notions

Efficient ATP systems typically require very fast operations, when given a large set of terms $\{s_1, \ldots, s_n\}$ (the *index*) and another *query* term t, for unification (finding the mgus of t and s_i for all i), matching for demodulation of the query (finding some s_i and substitution σ s.t. $s_i\sigma = t$), and for matching for demodulation of the set (finding all subterms $s_i|p$ and σ s.t. $t\sigma = s_i|p$). Substitution trees [Gra95b] and discrimination trees [Chr93, McC92] are two well-known *indexing* data structures for doing this. For example, the set $\{\ f(a, g(x), h(y)),\ \ f(h(b), g(y), h(y))\ \}$ can be stored:

substitution tree:

$$\begin{tikzcd}[column sep=tiny] & x_0 \mapsto f(x_1, g(x_2), h(y)) \arrow[dl] \arrow[dr] & \\ x_1 \mapsto a, x_2 \mapsto x & & x_1 \mapsto h(b), x_2 \mapsto y \end{tikzcd}$$

discrimination tree:

$$\begin{tikzcd}[column sep=tiny] & f \arrow[dl] \arrow[dr] & \\ a \arrow[d] & & h \arrow[d] \\ g \arrow[d] & & b \arrow[d] \\ \vdots & & \vdots \end{tikzcd}$$

A substitution tree has in each node a substitution, let's say a list of pairs $x_i \mapsto t$ where x_i is an *internal* variable and t is a term. The remaining (*external*) variables are the ones of the terms stored. Roughly, in such a tree all terms $x_0\sigma$ are stored s.t. σ is the composition of all substitutions on some path from the root to a leaf of the tree.

The main advantage of substitution trees is obvious: contrarily to trie-like tree structures as discrimination trees, not only common *prefixes* can be shared (only the head symbol f in the example), but also other common parts. The amount of sharing not only reduces the space needed, but is also essential for reducing the time complexity of the main retrieval operations. Furthermore, in substitution trees the terms are not stored in a linear way, but they rather preserve their tree (or DAG) structure, which has some other advantages as we will see. The main advantage of discrimination trees is that they are easy to implement efficiently due to their simplicity.

Insertion in substitution trees can sometimes be done in more than one way. For instance, inserting $f(a, g(y), h(y))$ in the previous tree can give:

$$\begin{tikzcd}[column sep=tiny] & & x_0 \mapsto f(x_1, g(x_2), h(y)) \arrow[dl] \arrow[dr] & \\ & x_1 \mapsto a \arrow[dl] \arrow[dr] & & x_1 \mapsto h(b), x_2 \mapsto y \\ x_2 \mapsto x & & x_2 \mapsto y & \end{tikzcd}$$

but it could clearly also be inserted using the right branch. It is clear that when storing two terms $f(x, a)$ and $f(y, b)$ there is more sharing if x and y are considered as the same variable and hence terms are normalized before inserting them. Insertion is a relatively expensive operation, since it requires the computation of a most specific common generalization (mscg) of two substitutions. However, for highly dynamic sets, this is compensated by a very fast deletion procedure [Gra95c].

2.1 The Heap

Like the WAM heap, it is an array of pairs ⟨ *tag, address* ⟩, where *tag* can be **`ref`** or a function symbol `f`. The field *address* contains a heap address when *tag* is **`ref`** (when *tag* is a function symbol it can be used for other purposes). Terms are stored on the heap as in the WAM: each function symbol of arity n is followed by n contiguous **`ref`** positions pointing to its arguments. The whole term needs not to be contiguous; common subterms can be shared, and each variable corresponds to a **`ref`** position pointing to itself. The first part of the heap is reserved for several *variable banks*. The remaining part is administrated by a memory manager for allocating and freeing contiguous blocks of all sizes. As an example, the heap below at the left contains $f(x, g(x), g(x), y)$ at the address 20:

1	`ref`	1
2	`ref`	2
	⋮	⋮
20	`f`	
21	`ref`	1
22	`ref`	30
23	`ref`	30
24	`ref`	2
	⋮	⋮
30	`g`	
31	`ref`	1
	⋮	⋮

A substitution tree, its internal representation and its pre-construction on the heap:

$$x_0 \mapsto f(x', g(x_1), g(y'), y')$$
$$\swarrow \qquad \searrow$$
$$x_1 \mapsto x' \qquad x_1 \mapsto y'$$

Internal tree:

$$x_0 = 40$$
$$\swarrow \qquad \searrow$$
$$56 = 11 \qquad 56 = 12$$

11	`ref`	11
12	`ref`	12
	⋮	⋮
40	`f`	
41	`ref`	11
42	`ref`	55
43	`ref`	70
44	`ref`	12
	⋮	⋮
55	`g`	
56	`ref`	56
	⋮	⋮
70	`g`	
71	`ref`	12
	⋮	⋮

3 Many-to-one Unification and Sharing

Each indexing tree has a corresponding variable bank in the heap for its external variables[2]. In the right hand part of the previous figure, the variable bank starts at position 11. The terms of the substitutions of the tree are *pre-built* on the heap (in the example, the term of the root is at position 40), which allows us to internally represent nodes of substitution trees simply as lists of pairs of heap addresses (see the internal tree of the example). Note that the internal variables are self-referencing positions (position 56 in the example).

Now suppose we want to compute all unifiers with the query $f(x, g(x), g(x), y)$ drawn at the left at address 20. We simply traverse the internal representation of the substitution tree with backtracking, and call the standard WAM unification procedure for heap terms: first we unify, according to the root, positions 20

[2] Several trees may share the same variable bank. This is useful in some cases.

and 40, and after that we unify 56 either with 11 or with 12. The backtracking mechanism is very simple. The *backtracking stack* (a stack of heap addresses) contains `ref` positions in the heap to be restored to self-references upon backtracking; for each node of the substitution tree we keep the height of the stack when arriving to that node. Therefore, when backtracking to a node, we only need to pop and restore all `ref` positions above this height. Note that no bookkeeping (e.g. by additional stacks, like in other implementations) is needed for keeping track of the current working position of the query term or for reminding where to continue unification in the query term, etc.

Standard WAM unification between two heap terms does not write any new terms on the heap and hence does not require any further memory allocation. It only changes `ref` positions. This is therefore also the case for our many-to-one unification procedure: no memory management is needed during unification. Due to the fact that we only use standard WAM term representations, it will be relatively easy to integrate our data structures in ATP systems[3].

If heap unification is performed using the occurs check, this technique provides a so-called perfect filter, i.e. we succeed iff a unifier exists. But this check can also easily be left for the end if no other (more frequent and cheaper to check) type of clash occurs before. Some other additional optimizations are possible on these basic algorithms as well.

Both types of matching by means of a substitution tree can be implemented very similarly to unification. For example, in matching for demodulation of the query (e.g. when the terms of the tree are left hand sides of rewrite rules), the main difference is that only variables of the substitution tree can get instantiated. However, matching can be done far more efficiently with specialized data structures, as we will see in section 4.

The statistics below give some intuition about the efficiency of our data structures for unification. Let us remark that we have often seen in the litarature interpretations about the efficiency of different *techniques* based on a comparison between different concrete *implementations* of them; many times these implementations are rather uncomparable due to different degrees of optimization and refinement (sometimes they are even run on different machines). In [Gra95c] comparisons are given between similarly refined and optimized implementations of discrimination trees, abstraction trees and substitution trees (path indexing was omitted because it was much slower than substitution trees). Substitution trees were clearly fastest for both matching operations and unification.

Since we did not want to compare with the ACID toolbox [Gra95a] implementation of substitution trees, which is not the latest optimized version and because in toolbox implementations some time may be lost in interfaces, last summer we asked Christoph Weidenbach to provide us with the best state-of-the-art implementation of substitution trees available at the Max-Planck-Institut (as used in the Spass theorem prover [Wei97]). Below we compare our Dedam im-

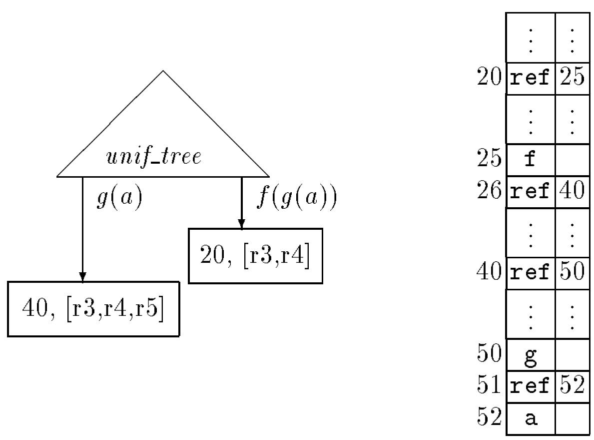
[3] If one does not care about the efficiency gained from a full integration of indexing data structures, one can of course use a package like ACID [Gra95a] whose internal term representation is hidden, but which provides interfaces for Prolog and C.

plementation and the MPI-substitution trees run on the same SPARC-10 (with 64 Mbytes of memory) on a number of different term sets obtained by typical Otter applications (see [Gra95c]). The times indicate the cost in seconds of finding all unifiers between the terms in the first *index* set and the second *query* set[4]. Numbers in names indicate the number of terms in the sets. For example, in the third row the index set is the set of 500 terms EC500+ and the query terms are from EC500-.

Example	**Dedam**	**MPI − ST**
$EC500++$	2.35	3.65
$EC500+-$	4.7	6.02
$EC500-+$	4.73	1.83
$EC500--$	28.92	16.07
$CL1000++$	2.02	4.35
$CL1000+-$	0.65	1.30
$CL1000-+$	5.67	−
$CL1000--$	1.48	−
$BO6000++$	2.17	6.3
$BO6000+-$	1.42	2.33
$BO6000-+$	0.55	0.97
$BO6000--$	0.82	2.28

3.1 Sharing

As a simple example, in the figure below we show how sharing in our data structures leads to efficiency for inference computation (for operations like backward demodulation there are very similar mechanisms).



[4] The "–" indicates information missing due to unability in building the indexing tree of CL1000-.

The figure at the left represents an indexing tree for unification, called *unif_tree* and two of its leaves, corresponding to the terms $g(a)$ and $f(g(a))$ respectively. At each leaf a heap address and a list of rule numbers is stored.

Let us analyze an example of ordered equational paramodulation, i.e. the computation of critial pairs. Suppose we are looking for inferences with the rewrite rule $g(x) \to h(x)$ (not shown here on the heap) using *unif_tree*. When we arrive at the leaf corresponding to the term $g(a)$, the variable x will have been instantiated accordingly, and we find the heap address (40) of a `ref` that points to $g(a)$ at position 50. By temporarily changing this `ref` at position 40, making it point instead to the term $h(x)$, we can simply read off all the critical pairs from the given list of rules [r3,r4,r5] which share the subterm $g(a)$ through position 40. Note that $g(a)$ is also shared by other terms in the tree like $f(g(a))$, and hence the rule set for $g(a)$ contains the one at $f(g(a))$ as a subset.

4 Efficient Matching by Substitution Trees with Equality Constraints

This section is on efficient matching for demodulation of a query term t, that is, finding some term s_i of the index and a substitution σ such that $s_i\sigma = t$. This operation is a well-known main bottleneck in equational theorem proving.

For this operation, one can apply standard substitution trees like it was done for unification in section 3, with the additional restriction that the variables of the query are considered constants. However, we found that this operation can be implemented far more efficiently. In particular, the stack used for saving and restoring both internal and external variables can be complety avoided. Furthermore, in the standard approach when there are non-linear index terms (that is, index terms where some variable occurs more than once), sometimes two query subterms are matched against each other. With an adequate control mechanism, this matching operation can be replaced by a cheaper syntactic equality test (which is what is actually needed since the query term variables are considered constants).

10	`f`	
11	`ref`	30
12	`ref`	40
	⋮	⋮
30	`s`	
	⋮	⋮
40	`t`	
	⋮	⋮

$$\begin{array}{ccc} & x_0 \mapsto f(x_1, x) & \\ \swarrow & & \searrow \\ x_1 \mapsto x' \textbf{ if } x' = x & & x_1 \mapsto a \end{array}$$

Internal tree:

$$\begin{array}{ccc} & x_0 = 120 & \\ \swarrow & & \searrow \\ 121 \mapsto 101 \textbf{ if } 100 = 101 & & 121 \mapsto 150 \end{array}$$

	⋮	⋮
100	`ref`	100
101	`ref`	101
	⋮	⋮
120	`f`	
121	`ref`	121
122	`ref`	100
	⋮	⋮
150	`a`	
	⋮	⋮

Let us explain the method by a very simple example. Suppose the query term is of the form $f(s,t)$, where s and t are arbitrary subterms, and let the index contain the two terms $f(x,x)$ and $f(a,x)$. Then we have the representation in our data structures as in the previous figure.

Before storing the terms in the indexing data structure they are linearized (repeated occurrences of variables are renemed). The nodes in the substitution tree now have substitutions and *equality constraints*: expressions that are conjunctions of equality classes between variables. For example, $f(x,x)$ is inserted as $f(x,x')$ with the equality constraint $x = x'$ (internally $100 = 101$). During the matching operation with such trees, all variables (both internal and external) in the substitution part of the tree are always simply overwritten, and hence no stack is needed for restoring their values upon backtracking. The constraints deal with the non-linearities in the indexed terms. The equality constraints are checked with a syntactic equality test operation.

The statistics below give some intuition about the efficiency of our data structures for matching. Here we compare with the currently best state-of-the-art implementation of substitution trees available at the Max-Planck-Institut, again provided by Christoph Weidenbach. The first two columns indicate for each implementation the time (in seconds) spent in finding all generalizations of each query term in the index. The last two columns indicate the times in finding 100 times the first generalization (this is in fact the operation used for demodulation). All the comments about statistics and the term sets made at the end of section 3 again apply here.

	all gener.		100 times first gener.	
Example	**Dedam**	**MPI − ST**	**Dedam**	**MPI − ST**
$EC500++$	0.05	0.12	1.87	5.90
$EC500+-$	0.07	0.10	4.88	11.02
$EC500-+$	0.00	0.02	0.25	1.03
$EC500--$	0.13	0.25	8.57	17.67
$CL1000++$	0.12	0.33	4.98	23.60
$CL1000+-$	0.02	0.13	0.87	11.57
$CL1000-+$	0.00	0.02	0.48	0.62
$CL1000--$	0.30	0.70	17.42	46.53
$BO6000++$	0.55	1.38	21.35	69.28
$BO6000+-$	0.68	1.23	47.28	113.40
$BO6000-+$	0.08	0.30	7.52	26.20
$BO6000--$	0.27	1.00	13.48	50.68

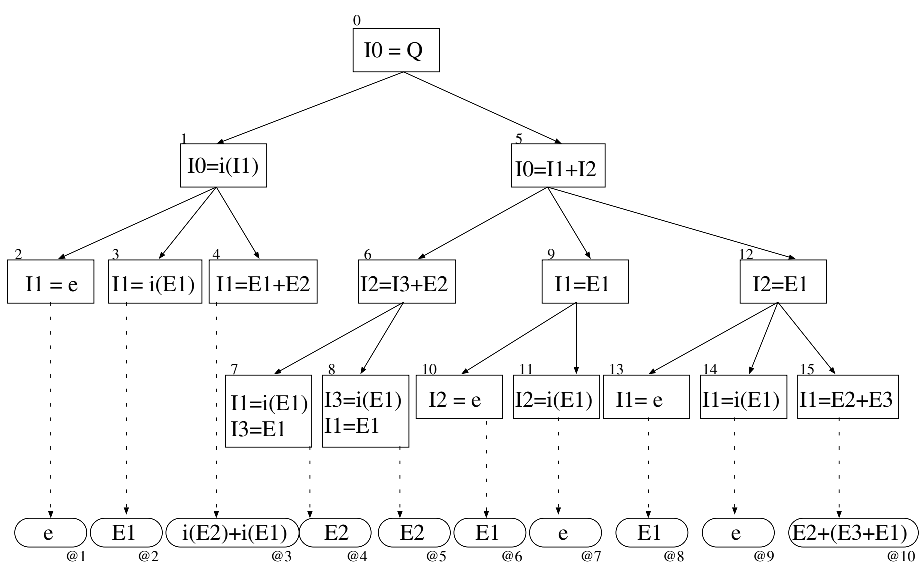
5 Compiling Static Clause (Sub)Sets

In this section we describe how static subsets of the clause data base can be compiled for increasing the performance of the basic operations. For example, in an interactive saturation-based ATP system, one can compile saturated parts of the theory obtained form partial runs of the system. This is done here by compiling in a rather straightforward manner the substitution trees in which their terms are stored. No real compiler has been implemented yet and the code shown here has been obtained “by hand”. The compiled code will not only behave exactly as the dynamic tree would do, but it also proceeds operationally in a very similar way. Efficiency is gained due to the fact that the control does not need to be looked up and because the compiler includes some optimizations which are possible only by a non-local analysis (including look-aheads) of the substitution tree.

5.1 Compilation for Matching

The input to the compilation process is a substitution tree pre-built on the heap as described in section 3. There are two sets of registers: `I0,I1,...` is used for the internal variables of the tree, and `E0,E1,...` for the external ones. The basic abstract machine operations for matching are simple:

- `deref` (dereferentiation) is like in the WAM: given a heap address, it follows the chain of `ref`'s until a self referentiating `ref` or a function symbol is reached and returns the last heap address of the chain.
- `jump_when_different_struct( f, heapaddr, #label )` compares the function symbol `f` with the *tag* field of the given heapaddress, and jumps to `#label` if different.
- `jump_when_different_terms( heapaddr, heapaddr, #label )` compares the terms starting at both given heapaddresses, and jumps to `#label` if they are different.
- `success(leaf info)` success rutine returning the information at this leaf of the tree (cf. the previous section).



Substitution Tree containing the completed Group Axioms

Below we give (part of) the compilation for rewriting of the substitution tree given in the previous figure, which stores the well-known ten-rule rewrite system for groups. Each node of the tree has a number `i`, and its corresponding piece of code is labeled with `#i`. Regarding the code, only note that absolutely no information for backtracking is stored, because at compile time we know the addresses of the points to which we backtrack, and for the variable registers at any point it is known whether their contents can be overwritten or not. We only call `jump_when_different_terms` when it is absolutely necessary, i.e. when an external variable appears for the second time in the tree, which is when no information about the structure of the terms is available at compile time because both terms come from the query.

```
Procedure match(QueryAddress):
#0:  I0 = deref(QueryAddress)
#1:  jump_when_different_struct(i,I0,#5)
     I1 = deref(I0+1)
#2:  jump_when_different_struct(e,I1,#3)
     success(@Rule_1)
#3:  jump_when_different_struct(i,I1,#4)
     E1 = deref(I1+1)
     success(@Rule_2)
#4:  jump_when_different_struct(+,I1,#5)
     E1 = deref(I1+1)
     E2 = deref(I1+2)
     success(@Rule_3)
#5:  jump_when_different_struct(+,I0,#END)
     I1 = deref(I0+1)
     I2 = deref(I0+2)
#6:  jump_when_different_struct(+,I2,#9)
     I3 = deref(I2+1)
     E2 = deref(I2+2)
#7:  jump_when_different_struct(i,I1,#8)
     E1 = deref(I1+1)
     jump_when_different_terms(I3,E1,#8)
     success(@Rule_4)
      .
      .
#15: jump_when_different_struct(+,I1,#END)
     E2 = deref(I1+1)
     E3 = deref(I1+2)
     success(@Rule_10)
#END:exit
```

5.2 Compilation for Unification

For finding all unifiers between the terms in the tree and the query term, the code is again simple. Essentially, the only additional data structure involved is the *Backtracking Stack* BTStack, which contains the ref positions of the query term that have to be restored to self-referentiations upon backtracking. No more information for backtracking needs to be stored, because again at compile time we know the addresses to which we backtrack, and whether the variable registers can be overwritten or not. Apart form the registers I0,I1,... and E0,E1,... for the internal and external variables of the tree, we use registers H0,H1,.... There will be a register Hi for each node i of the substitution tree which has more than one son, and it will contain the height of BTStack when leaving node i. The code to which the abstract machine instructions are expanded is as follows:

– assign_and_jump_when_clash(f, heapaddr, #label):

```
If Heap[heapaddr].tag == REF Then
   push(BTStack,heapaddr)
   Heap[heapaddr].addr = <prebuilt_address>
ElseIf Heap[heapaddr].tag != f Then
   backtrack(#label)
EndIf
```

  where <prebuilt_address> denotes the address where the corresponding term in this node is pre-built on the heap (cf. Section 3).

– unify_and_jump_when_fail(heapaddr1, heapaddr2, #label):

```
unify(heapaddr1,heapaddr2)
If unify_failure Then backtrack(#label) EndIf
```

  Here the procedure unify is the standard WAM unification procedure for heap terms. It pushes ref positions on BTStack whenever they get instantiated. Note that this procedure is called only when necessary, i.e. when an external variable appears for the second time in the tree, which is when no information about the structure of the terms being unified is available at compile time because both terms come from the query.

– backtrack(#label):

```
While height(BTStack) > H Do
   A = pop(BTStack)
   Heap[A].addr = A
EndWhile
GoTo #label
```

  Here H is the register Hi where i is the parent node (known at compile time) of #label.

Below we give the (first part of the) code obtained from (again) the tree of the group example. As before, labels correspond to node numbers in the tree; here the success instruction can be set to provoke backtracking to compute *all* unifiers.

```
Procedure Unify(QueryAddress):

#0:  I0 = deref(QueryAddress)
     H0 = height(BTStack)
#1:  assign_and_jump_when_clash(i,I0,#5)
     I1 = deref(I0+1)
     H1 = height(BTStack)
#2:  assign_and_jump_when_clash(e,I1,#3)
     success(@Rule_1)
#3:  assign_and_jump_when_clash(i,I1,#4)
     E1 = deref(I1+1)
     success(@Rule_2)
#4:  assign_and_jump_when_clash(+,I1,#5)
     E1 = deref(I1+1)
     E2 = deref(I1+2)
     success(@Rule_3)
#5:  assign_and_jump_when_clash(+,I0,#END)
     I1 = deref(I0+1)
     I2 = deref(I0+2)
     H5 = height(BTStack)
#6:  assign_and_jump_when_clash(+,I2,#9)
     I3 = deref(I2+1)
     E2 = deref(I2+2)
     H6 = height(BTStack)
#7:  assign_and_jump_when_clash(i,I1,#8)
     E1 = deref(I1+1)
     unify_and_jump_when_fail(I3,E1,#8)
     success(@Rule_4)
       .
       .
```

6 Efficient Data Structures for Demodulation

In this section we describe our techniques for efficient demodulation and, in particular, how we can completely avoid memory management for this purpose thanks to our WAM-like term representation[5].

Normally, in theorem provers demodulation is applied as a forward redundancy mechanism, that is, each conclusion of an inference is demodulated as soon as it is generated by (backtrackable) rewriting, and only if it is not rewritten into a tautology it is copied and stored as a new clause; then the demodulation steps are undone and the inference computation goes on. For simplicity, we do not consider here the ordering restrictions for assuring termination of the de-

[5] We are very grateful to Christoph Weidenbach who, during last Cade in New Brunswick, suggested the idea of exploiting our WAM heap like it is explained here.

modulation process, i.e., we assume here that all rewrite rules are oriented with respect to a reduction ordering.

A demodulation step on a clause C with a rewrite rule $l \rightarrow r$ amounts to replacing a subterm of the form $l\sigma$ for some substitution σ by $r\sigma$. A common rewriting strategy applied for complete normalization of clauses is the (leftmost) innermost strategy, because it permits *marking* irreducible subterms to avoid unnecessary reduction attempts. In our WAM-heap all non-`ref` positions can have such an irreducibility mark.

Apart from the search for a matching left-hand side (this aspect has been treated in section 4) the most expensive operation of applying a rewrite step is to obtain a fresh copy of the *skeleton* part r of the term $r\sigma$. The term $r\sigma$ has to be fresh: it cannot be shared with other instances $r\sigma'$ of other rewrite steps with the same rule, among other reasons because further steps inside $r\sigma$ may take place. The terms of the substitution part of $r\sigma$ are already present before the rewrite step and do not need to be created; this is the case even if some variable x occurs more times in r than in l, because the same copy of $x\sigma$ can be shared.

In a previous approach in the Dedam framework (similarly to what most provers end up doing) we used to create the skeleton of r by our general memory management module, which is able to allocate and free pieces of heap of all sizes. For example, when applying the rewrite rule of distributivity, $x*(y+z) \rightarrow x*y+x*z$, three blocks of size three have to be allocated. Furthermore, in order to be able to know which parts can be freed when undoing the rewrite step, special marks indicating "end_of_skeleton_part" are needed. Of course the memory management operations for allocating and freeing blocks of different sizes are relatively expensive, taking into account the number of demodulation steps performed in typical runs of a prover for clauses with equality. Here we propose the following much simpler approach.

10	$*$	
11	`ref`	39
12	`ref`	40
	$\vdots$	$\vdots$
39	`a`	
40	$*$	
41	`ref`	39
42	`ref`	50
	$\vdots$	$\vdots$
50	$+$	
51	`ref`	39
52	`ref`	39

Standard Term Heap (Left)

and

Rewrite Heap (Right)

1000	$*$	
1001	`ref`	41
1002	`ref`	51
1003	$*$	
1004	`ref`	41
1005	`ref`	52
1006	$+$	
1007	`ref`	1000
1008	`ref`	1003
1009	`ref`	1003
$H = 1010$	$\ldots$	$\ldots$
	$\vdots$	$\vdots$

In the left part of the previous figure the term $a*(a*(a+a))$ is represented, starting at position 10. After performing a rewrite step with the rule $x*(y+z) \rightarrow x*y+x*z$ on its subterm $a*(a+a)$ at the `ref` position 12, this `ref` position is changed from 40 to 1006, the address of a fresh copy of the right hand side

$x * y + x * z$. In a *rewrite stack* we keep the pair (12,40) in order to be able to undo the rewrite step.

The address 1006 of the fresh copy belongs to a a general *rewrite heap* zone (shown at the right in the previous figure) that grows in one direction, with a register H (like in the WAM) pointing to the first free position. Allocating a a block of size n amounts to incrementing H with n units. Undoning the rewrite steps of the normalization of a clause simply amounts to restoring the `ref`'s of the rewrite stack and setting H to 1000. No "end_of_skeleton_part" marks are needed, since the part that has to be freed is, by construction, exactly what is built on the rewrite heap.

In the table below we compare the time spent in seconds for 3000 demodulation steps with rules of the form $s \rightarrow s$ without including the matching time. There are eight columns corresponding to different sizes of the term s. There are three rows corresponding to the "old" method with memory management, the "new" method without and the factor of speedup obtained, which grows from 2.7 to 6.4 as the term size grows from 2 to 75:

term size:	2	3	4	7	11	21	38	75
old:	0.08	0.12	0.17	0.30	0.45	0.90	1.62	3.20
new:	0.03	0.05	0.07	0.08	0.12	0.18	0.30	0.50
factor:	2.7	2.4	4.25	3.75	3.75	5.0	5.4	6.4

7 Conclusions and Further Work

We have presented Dedam, a basic kernel of data structures for high performance automated deduction with first-order equality clauses. Our ambition is that it will become a kind of standard WAM-like framework for this purpose, around which ATP systems can be built. This has been done already for the equational theorem prover Barcelona [NRV97], built on top of a first version of Dedam, and devised mainly to test the performance of this toolkit. During the CADE-13 theorem proving competition, the relatively high throughput of the data structures seems to have compensated for the fact that the prover itself is just an unfailing Knuth-Bendix completion procedure without many heuristics or tuning for the specific class of problems. In the near future we will also build a prover for full first-order clauses with equality on top of Dedam.

It is clear that much work remains to be done. On one hand, we have started to extend our basic architecture to include other standard operations. For example, if ordering constraints are represented as sets of edges between DAG's on the heap, constraint solving can be done more efficiently in space and time since the same term normally appears many times in a constraint. We are also writing a documentation of the data structures (with more details than an introductory paper like this can contain) in order to make it publicly available and useful to the ATP community.

References

[BG96] David Basin and Harald Ganzinger. Complexity Analysis Based on Ordered Resolution. In *11th IEEE LICS*, pages 456–465, New Brunswick, NJ, USA, July 27–30, 1996.

[Chr93] Jim Christian. Flatterms, Discrimination Nets, and Fast Term rewriting. *Journal of Automated Reasoning*, 10:95–113, 1993.

[GNN95] Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. The Saturate System, 1995. Software and documentation available: `http://www.mpi-sb.mpg.de/SATURATE/Saturate.html`.

[Gra95a] Peter Graf. ACID User Manual - Version 1.0. Technical Report MPI-I-94-DRAFT, Max-Planck-Institut für Informatik, Saarbrücken, November 1995.

[Gra95b] Peter Graf. Substitution Tree Indexing. In J. Hsiang, editor, *6th RTA*, LNCS 914, pages 117–131, Kaiserslautern, Germany, April 4–7, 1995. Springer-Verlag.

[Gra95c] Peter Graf. Substitution Tree Indexing. In J. Hsiang, editor, *6th International Conference on Rewriting Techniques and Applications*, LNCS 914, pages 117–131, Kaiserslautern, Germany, April 4–7, 1995. Springer-Verlag.

[McC92] William McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992.

[Nie96] Robert Nieuwenhuis. Basic paramodulation and decidable theories. In *11th IEEE LICS*, pages 473–482, New Brunswick, NJ, USA, July 27–30, 1996.

[NN93] Pilar Nivela and Robert Nieuwenhuis. Practical results on the saturation of full first-order clauses: Experiments with the saturate system. (system description). In C. Kirchner, editor, *5th RTA*, LNCS 690, Montreal, June 16–18, 1993. Springer-Verlag.

[NRV97] Robert Nieuwenhuis, José Miguel Rivero, and Miguel Ángel Vallejo. The Barcelona Prover. *Journal of Automated Reasoning*, 1997. Special issue CADE-13 ATP System Competition. To appear.

[Sti89] Mark Stickel. The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, October 1989.

[War83] David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[Wei97] Christoph Weidenbach. Spass. *Journal of Automated Reasoning*, 1997. Special issue CADE-13 ATP System Competition. To appear.