LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

Implementation of a PROLOG-like Logic Programming Language Based on B&D Search

Matthias Benkard

Projektarbeit

Beginn der Arbeit:	01.03.2010
Abgabe der Arbeit:	7. Februar 2013
Betreuer:	Dr. Norbert Eisinger
	Simon Brodt

Erklärung

Hiermit versichere ich, dass ich diese Projektarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 7. Februar 2013 .....

Matthias Benkard

Zusammenfassung

Wir stellen eine Umsetzung der von Simon Brodt et al.[3] vorgeschlagenen Techniken zur informationsfreien Suche in Suchräumen vor. Es werden sowohl allgemeine Implementierungstechniken für backtrackinglose PROLOG-Systeme als auch für B&D-Suche spezifische Optimierungen besprochen.

Abstract

We present an implementation of the ideas set forth by Simon Brodt et al. in [3]. Both general techniques for implementing PROLOG systems not based on backtracking and optimization approaches specific to B&D search are discussed.

Inhaltsverzeichnis

Teil I.
Introduction

1. Motivation

Traditionally, most PROLOG-like logic programming systems have been based on a depth-first, backtracking search (DFS) of tree-shaped search spaces. While this makes the behaviour of programs in such systems algorithmically predictable and efficient, it also exposes the programmer to non-declarative behaviour even in the absence of extralogical operations such as cuts. For example, a PROLOG program enumerating the set of ordered pairs of natural numbers might naively be written as follows:

```
nat(0).
nat(X) ← nat(Y), X is Y + 1.

nats(X, Y) ← nat(X), nat(Y).
```

Unfortunately, in contrast to declarative expectations, this program will fail to enumerate any pairs with the first component greater than 0, since backtracking in `nats` will only ever backtrack over `Y`. In other words, backtracking is *incomplete* with regard to the above programme.

A number of alternative approaches have been suggested in the past, most notably breadth-first search and iterative deepening. In breadth-first search (BFS), the decision tree is traversed horizontally, one level at a time. In this way, we get a complete enumeration of all nodes even if one or more branches are infinite. In iterative deepening, every iteration I_ℓ traverses the decision tree up to a level of ℓ while discarding any intermediate results produced by the last iteration.

Breadth-first search has traditionally been avoided for a number of reasons, one of which is the $O(2^d)$ memory complexity, where d is the depth of the part of the search space traversed so far. Iterative deepening, on the other hand, needs quadratic time to execute linear programs such as *while*-loops. Therefore, neither technique is suitable for most classes of large, real-world applications.

B&D Search[3] combines breadth-first search with depth-first search, achieving $O(n)$ time complexity with regard to the number n of nodes as well as polynomial space complexity with regard to search-space depth d while still preserving completeness, which makes it more suitable for a general-purpose, real-world programming language. It has so far lacked an implementation proving its feasibility in the face of the constraints imposed by logic programming (such as the need for variable propagation). The implementation presented in this work is designed to fill that hole.

This report assumes readers to be familiar with the essentials in [3].

2. B&D Search

As noted, general tree traversal is very efficiently done by a depth-first search in the case of a finite tree. On infinite trees, however, a depth-first search fails to be complete. In this case, breadth-first search, while suboptimal in terms of memory usage[3], is a more adequate choice[1]. The idea of B&D search[3] is to do both traversals simultaneously while restricting their execution so as to preserve their complexity guarantees. This is done by using a sophisticated scheduling scheme that awards execution credits to breadth-first search whenever depth-first search advances further down the tree. In this scheme, breadth-first search may advance only by using up credits, while depth-first search must wait for breadth-first search to complete its current tree level whenever it is allowed to.

A flexible crediting function, adjustable through a constant which may be any non-negative numeric value, allows the user to decide what weight to give breadth-first or depth-first search, respectively, depending on the problem domain. In essence, the value determines how much effort is spent on completeness and fairness. Thus, the algorithm can be used even in known finite settings by setting the credit constant to 0, effectively disabling breadth-first search and maximizing efficiency; likewise, where a large number of non-terminating subgoals is expected, the user can switch to pure or nearly pure breadth-first search by setting the credit constant to a high enough value, trading memory efficiency for completeness.

3. SLD Resolution

In our B&D-based PROLOG, as in regular PROLOG, a program consists of a set of rules of the form $P \leftarrow A_1, ..., A_n$. A query is of the form $B_1, ..., B_m$ and represents a goal to be proven. The evaluator uses SLD resolution[7] to try to prove a query using the database of rules in the program. Assuming that $C_1, ..., C_\ell$ is the current goal, it proceeds as follows:

1. Select one of the C_i for expansion.
2. Unless C_i is trivial, look for a matching rule $P \leftarrow A_1, ..., A_n$ in the database whose head P unifies with C_i, and perform the unification. This yields a unifier ϕ.
3. Set $(C_1, ..., C_{i-1}, A_1, ..., A_n, C_{i+1}..., C_\ell)\,\phi$ as the new goal.
4. Repeat.

The intricacies of the process, in particular the unification mechanism, will not be discussed in this work. Instead, see [7] for details.

[1] Certainly, it is a choice with room for improvement. See [3] for alternative approaches.

4. SLD Search Spaces

4.1. Search Space Representation

Whenever multiple rules match in an SLD resolution step, the computation forks, yielding a tree-shaped search space. There are various ways of representing the search space of a logic program (i.e., the tree that the interpreter needs to traverse). The simplest representation is the so-called *or-tree* (see figure 4.2). In this representation, each node represents all of the pending subgoals yet to be proven, while the parent-child relationship represents possible proof paths. In this scheme, disjunctions manifest themselves in the sibling relationship. During execution, the tree grows downward, with each processed node spawning a (possibly empty) set of successors depending on the number of possible choices at the node's branching point.

Or-trees are useful because they mirror the basic process of running down the tree until a solution to a query is found, as well as making the processing of single nodes easy to illustrate. However, since they lack a representation of the relationship between a goal and its subgoals (and, in consequence, a goal and its sibling goals), each node must contain a list of all goals that remain to be proven.

Indeed, the goal-subgoal relationship is yet another child-parent relationship; this means we can illustrate it using a tree as well. In such a tree, however, the thread of execution is not as clear as in an or-tree, which makes it impractical to depict all of the processing of a query in a single tree.

Because of this, an alternative "boxy tree" representation is chosen for the explanations in this paper. In a boxy tree, nodes are represented as nested labeled boxes with arrows in between, where the nesting is interpreted as the goal-subgoal relationship, whereas the arrows denote the successor relation as in an or-tree. Our rendition of boxy trees subdivides a conjunctive goal into vertical areas which represent the subgoals of the goal, aligned vertically over all branches. This makes it easier to follow the depicted program both conceptually and execution-wise. Variable bindings in a node are represented as a bracketed set of equations (e.g. $[X = carol]$) at the end of the node's label. In addition, we omit arrows between a box and its immediate nesting children, if present, since they are redundant, which yields a "simplified boxy tree."

4.2. Example

Consider the set of facts and rules written down in figure 4.1.

The code models a typical example of a logic programming problem. The female/1 predicate ranges over people, asserting who is female. The parent/2 predicate associates a parent (first argument) with a child (second argument). As we can see, Carol's mother is Mary, whose father, in turn, is Dan. Finally, the ancestor/2 predicate generalizes the parent/2 relationship by allowing multiple generations in between a person and their ancestor.

ancestor/2, as defined, is a transitive relation, and thus is a classical example of the care that must be taken regarding the order of rule definitions when doing traditional depth-first

```
female(mary).
female(carol).

parent(mary, carol).
parent(dan, mary).

ancestor(X, Y) ← parent(Z, Y), ancestor(X, Z).
ancestor(X, Y) ← parent(X, Y).
```

Abbildung 4.1: Example code.

search.[2]

Figure 4.5 shows the search space induced by the query female(X), not(ancestor(X, carol)). For compactness, predicate names and constants have been abbreviated to a single character each.

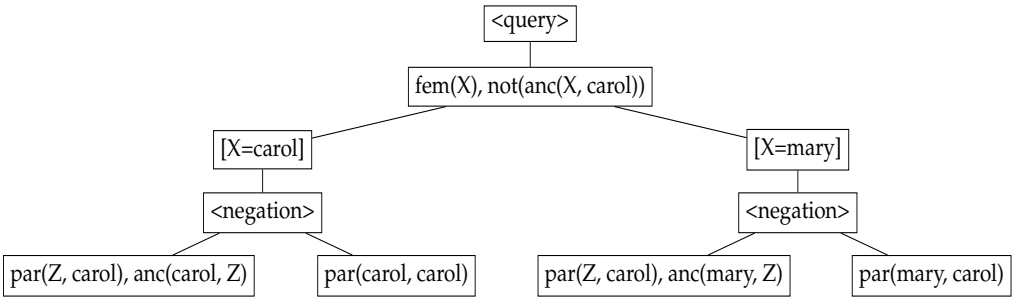

Abbildung 4.2: An or-tree.

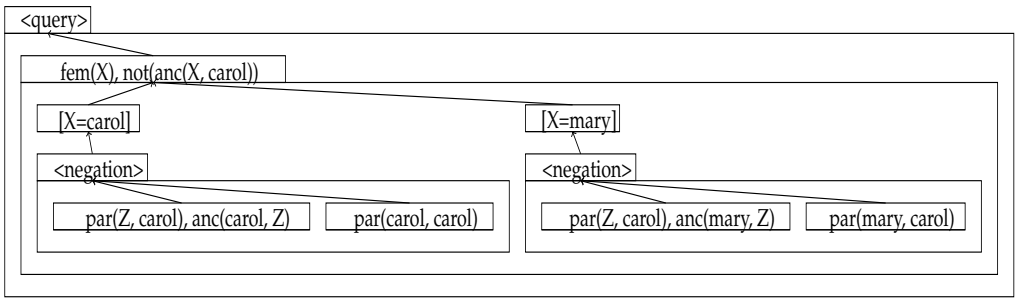

Abbildung 4.3: Boxy tree.

[2] You may have noticed that, in order to illustrate the advantages of B&D search, the ancestor/2 relation is deliberately defined in such a way that traditional depth-first search will never find an answer to any given query that asks for someone's ancestors.

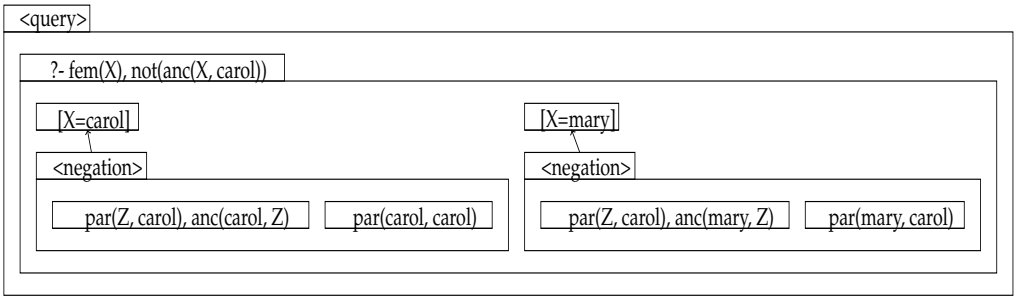

Abbildung 4.4: Simplified boxy tree.

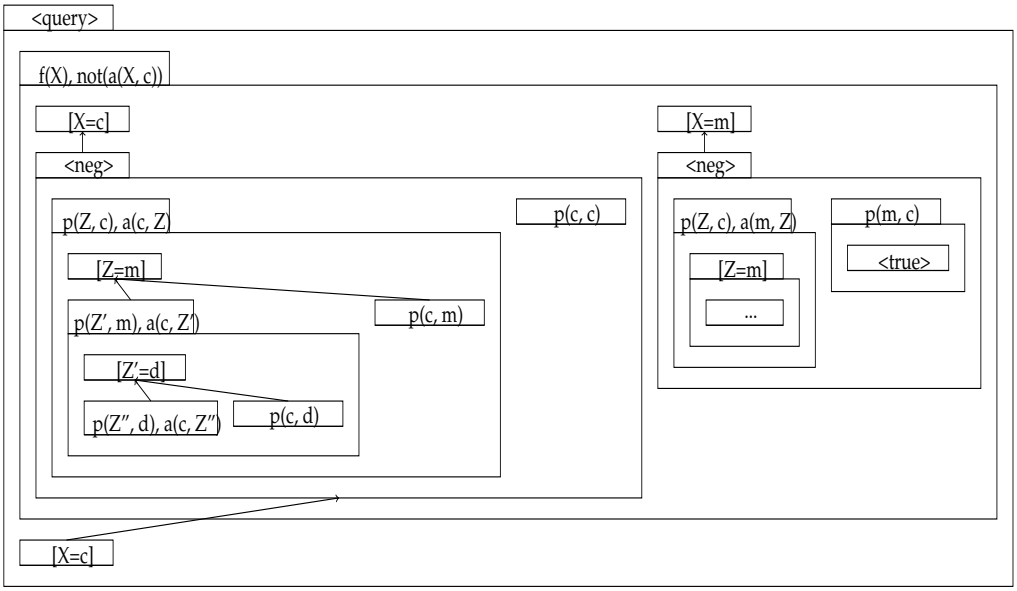

Abbildung 4.5: Search space of the example query.

Teil II.
Implementation Strategy for a Dialect of Pure Prolog using B&D Search

5. Object Language (Pure Prolog)

In the following, we will only consider a "pure" subset of PROLOG, that is, a PROLOG-like language stripped of most non-declarative operations. Included in this subset are:

- Equality
- Conjunctions
- Disjunctions
- Arithmetic operators

The following non-declarative features are notably absent from our PROLOG dialect:

- The cut operator (though a substitute is available; see 10.2)
- Side-effecting operators like write/1

6. Execution Cycle

The execution cycle of the evaluator follows a simple trampoline[3]-like structure:

Algorithm 1 Execution Cycle

while nodes remaining **do**
  $node \leftarrow$ choose node
  $expand(node)$
end while

In this representation, the `choose node` pseudo-expression is the manifestation of the tree traversal algorithm, while the `expand` procedure is responsible for all node processing including variable propagation and the handling of every kind of query.

[3]The word "trampoline" is used here in the sense of a top-level loop that is used to call successive continuations in a procedural or functional program. Use of a trampoline makes it possible to implement first-class continuations and tail-call optimization on top of a language that lacks these features.[12] Also see [1] for an interesting approach to the same problem that manages to avoid trampolines in a specific setting.

7. Subgoal Trees and Execution Trees

As noted in section 4.1, several tree structures are relevant to representing the relationships within the search space of a logic program that are needed by an implementation:

1. the tree of subgoals, which can change during execution
2. the tree of successors (representing the relevant part of the search space), which can only change by either growing downward or unlinking nodes

The tree of subgoals is implemented simply by storing a link from every child node to its parent node. For reasons explained in 10, we have the parent node refer to its children as well, albeit in a more limited way.

The tree of successors, on the other hand, needs to be more intricately tuned for efficiency, since it is the primary data structure the algorithm needs to operate on. Since the processing of nodes proceeds left-to-right as well as top-to-bottom, a list-like structure is appropriate for modelling the search space. In particular, in order to also enable efficient removal of nodes from each tree level, we implement the search space as a vector of doubly-linked lists, where each list represents one level of the tree. We call this structure the *execution tree*, as its purpose is to direct the behavior of the execution cycle. Graphically, execution trees are represented by boxy trees (see section 4.1).

Note that the boxy-tree representation obscures the identity of the execution tree levels by decoupling the search space depth of a given node from its height within the drawing. While this is mostly a minor point, it has to be kept in mind, since D&B search operates directly on execution tree levels. Therefore, the behavior of the tree traversal algorithm is hard to follow based on a boxy tree diagram.

8. Propagation of Variable Bindings

Traditional backtracking implementations of PROLOG such as the Warren Abstract Machine[13], keep variable bindings on a single execution stack, undoing them upon backtracking. When processing multiple search-tree branches in parallel, this approach is inappropriate, since multiple execution states with (at least partially) independent variable bindings must be maintained simultaneously. Instead, the set of active variable bindings must be a property of each individual branch, and thus, of each individual tree node.

This restriction necessitates a way of propagating variable bindings up the subgoal tree, since each branch, just as in a backtracking search, will both grow downwards at times and shrink upwards at others.

Note that when a branch shrinks, this is represented in a boxy tree diagram by the respective path leaving its enclosing box and thereby moving upward with regard to the nesting relationship (see figure 8.1). That is, variable propagation points can be recognized by looking for intersections between arrows and box boundaries, and final query results are represented by leaves on the top level with regard to box inclusion, whereas all other leaves represent either unprocessed or failed nodes.

In the following, we outline a couple of approaches to variable propagation, along with their respective flaws and virtues.

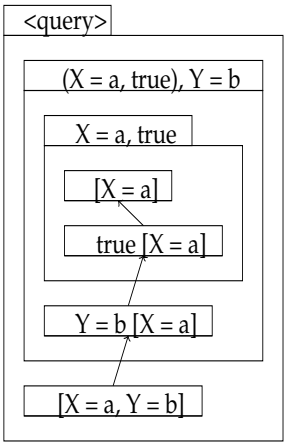

Abbildung 8.1: Variable propagation. Note: Execution simply proceeds from top to bottom by successively adding new nodes.

8.1. Naïve Approach

A naïve implementation of variable propagation might closely follow the SLD resolution algorithm outlined in section 3. In this scheme, all free variables contained in the current goal are renamed prior to the unification step, ensuring that unification will not affect the variables in other branches of the search space. A node representing a successful subgoal is turned into ("spawns") a new node by applying step 3 of the SLD resolution algorithm.

Note that the naïve approach does not require a node n to maintain a link to its parent $p(n)$ or vice-versa. However, it entails copying large sets of goal terms including possibly unchanged variable bindings at each propagation step, which is unacceptably inefficient in practice.

8.2. Restriction of Variable Binding Storage and Copying

To mitigate this, instead of storing all variable bindings in each and every node that potentially contains references to them, we store only two sets of variables in each node n: the set $V(n)$ of variables directly referenced by the currently processed goal (which does not contain goals that are waiting to be processed further up the tree) and the set $V_{sync}(n)$ of synchronization variables, which is originally an independent copy of the goal variables $V(p(n))$ of the parent $p(n)$ but *synchronized* with the current node's goal variables $V(n)$ such that whenever there is a common reference to some variable v in both $V(n)$ and $V_{sync}(n)$, an update to v with respect to one set will be reflected in the other.

Variable propagation, then, proceeds by shallow-copying the parent node, replacing the copy's list V of goal variables with the child's list V_{sync} of synchronization variables. This

allows for a far greater degree of structure sharing while remaining reasonably simple. See figure 8.2 for an illustration of the technique.

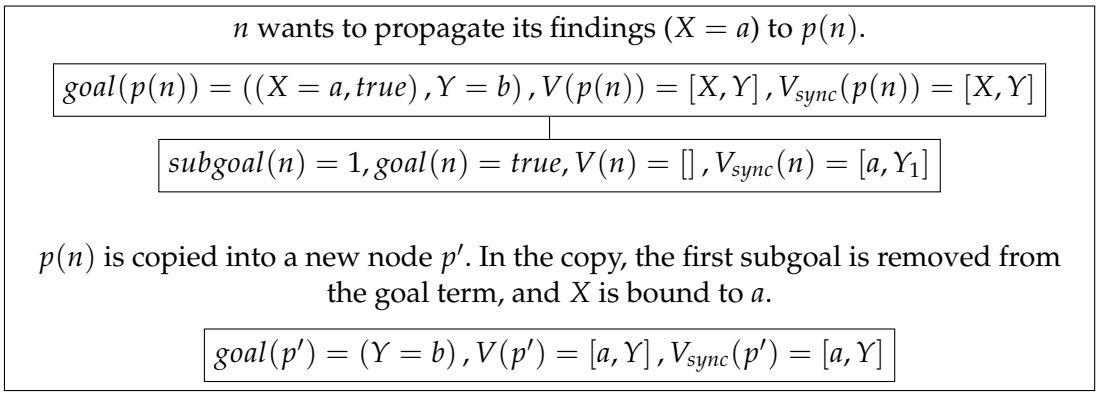

Abbildung 8.2: Variable propagation.

8.3. Parent Sharing Between Siblings

Structure sharing can be even further improved. Consider the case where a node n propagates a set of variables to its parent $p(n)$. Assume that n does not represent its parent's last goal. In this case, we would normally make a shallow copy p' of the parent $p(n)$ with node n's synchronization variables $V_{sync}(n)$ in place of the list $V(p(n))$ of the parent's variables and one goal removed (see figure 8.2). When the copied parent p' is processed, it will spawn a number of children c_i responsible for processing the next subgoal, where each child represents an alternative branch. This is illustrated in figure 8.3.

Now, observe that for every newly spawned node c_i, $V_{sync}(c_i)$ will be a copy of the original node n's synchronization variables $V_{sync}(n)$. Furthermore, while the list of synchronization variables $V_{sync}(c_i)$ clearly depends on the results returned by n, no other part of it does (see figure 8.4). Since the bindings done with regard to the new list $V_{sync}(c_i)$ of synchronization variables contain all bindings done in $V_{sync}(n)$, these will be propagated to the parent through c_i eventually. This means that variable propagation from the original node n to its parent node $p(n)$ is actually redundant; but since variable propagation was the only reason for copying the parent in the first place, we could have saved us the trouble of doing so.

Indeed, we can share the same parent across all of its subgoals. To do this, instead of having a copy of the parent node spawn new children, we directly create *siblings* by copying the list $V_{sync}(n)$ of the original node's synchronization variables and expanding the next subgoal. Figure 8.5 illustrates this approach.

Since we are now using the same parent for all of its subgoals, we need a way of remembering which subgoals have already been processed and need to be skipped the next time a child returns. Since this is branch-specific information, we need to store it in the child. In our implementation, this is a simple integer value that represents an index into the parent's list of subgoals, but for conceptual clarity, we will express the relationship as a node *pointing to* a specific subgoal in the parent.

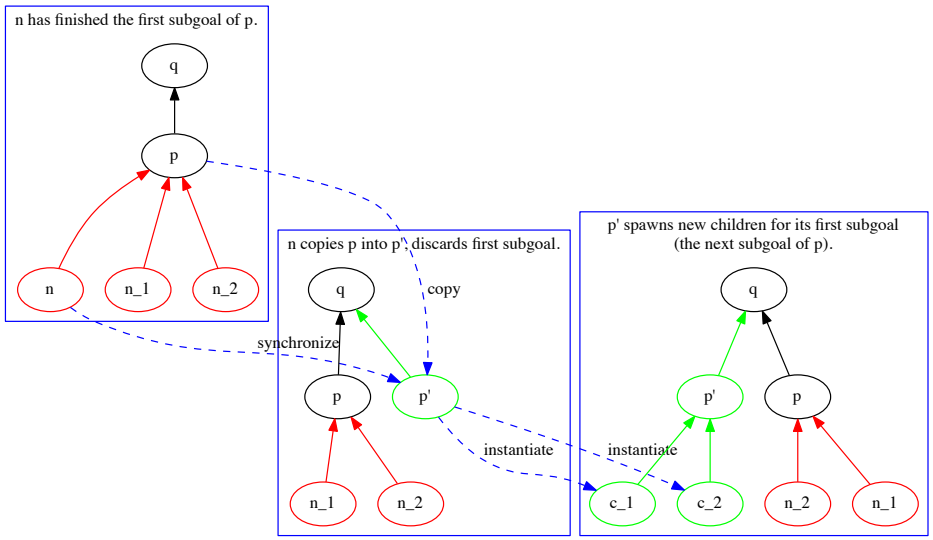

Abbildung 8.3: Propagation as per section 8.2 needlessly copies the parent node.

n wants to propagate its findings $(X = a)$ to $p(n)$.

$$goal(p(n)) = ((X = a, true), Y = b), V(p(n)) = [X, Y], V_{sync}(p(n)) = [X, Y]$$

$$subgoal(n) = 1, goal(n) = true, V(n) = [], V_{sync}(n) = [a, Y_1]$$

$p(n)$ spawns c_1.

$$goal(p(n)) = ((X = a, true), Y = b), V(p(n)) = [X, Y], V_{sync}(p(n)) = [X, Y]$$

$$subgoal(c_1) = 2, goal(c_1) = (Y = b), V(c_1) = [Y], V_{sync}(c_1) = [a, Y_1]$$

Abbildung 8.4: Consecutive stages of variable propagation. Note that nothing in c_1 depends on the results found by n except for $V_{sync}(c_1)$.

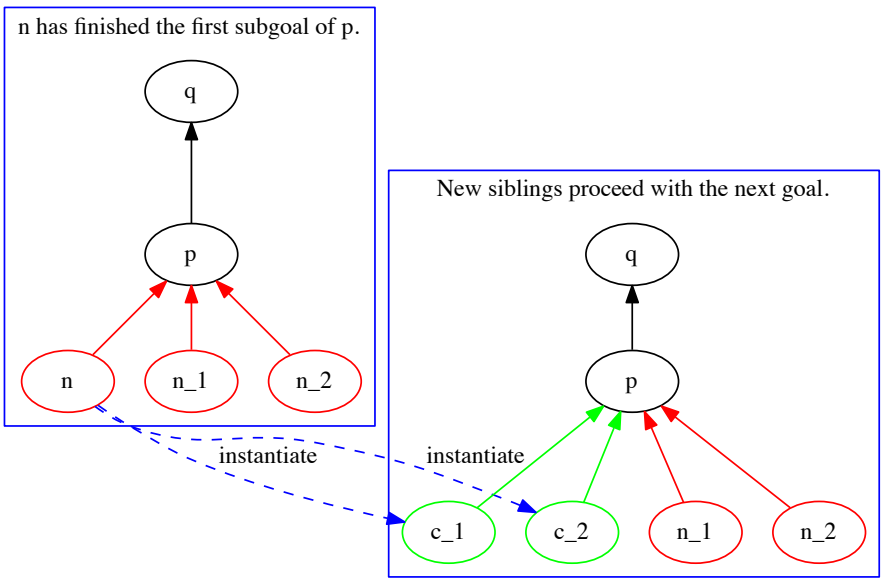

Abbildung 8.5: By copying $V_{sync}(n_3)$ only and remembering the subgoal number, we can avoid duplicating the parent node on each subgoal propagation.

9. Branches Without Solutions

Given a node with the current subgoal $p(t_1, ..., t_n)$, unification may fail for all possible rule heads in the definition of p/n. In this case, the currently processed node spawns neither children nor siblings and in consequence is immediately removed from the tree. Since there are no siblings, this may cause the parent node's children list to become empty, which causes the parent to be removed from the tree in turn. This node removal process proceeds up the tree until a node is encountered whose children list remains nonempty after removal of the respective child node.

10. Negation and Cut

10.1. Negation

Negation in PROLOG is a special case to consider when implementing a non-backtracking implementation because PROLOG's negation, which is known as *negation by failure*, is a control construct as much as it is a logical operator. In negation by failure, a negation is true if and only if the negatee fails (i.e. does not find a solution). This is a useful definition for a logic programming language because it permits (and, in the case of backtracking, more or less enforces) *short-circuiting* in the sense that as soon as one branch of the negatee finds a solution, the negation may return failure, discarding any unprocessed branches dynamically below the negation. On the other hand, if all children of a negation node fail, the negation succeeds but does not bind any variables (it can't, since by definition, no consistent set of variable bindings could be found).

We handle negations by introducing a special node type, below which the negatee is processed as if it were a goal to be proven. If some branch of the goal representing the negatee succeeds, it will eventually try to propagate up through the negation node. When this happens, all children of the negation node fail (this can be done by recursively traversing each subtree and processing every leaf node in the way it would be processed if it failed; a more efficient approach is outlined further below), as does the negation node itself. This concludes the processing of the negation node.

If a branch fails, on the other hand, it is processed as usual by being removed from the tree. Remember that this is done by removing the leaf node and recursively moving upward along the ancestor chain until we reach an ancestor that retains some children after the removal. On the way, we may visit a negation node. If so, and if the negation node is losing its last child through this process, we stop, and a new node is created from the negation node's parent node as if the negation node returned successfully with an empty set of variable bindings.

Note that the above implementation of negation modifies the procedural semantics of the not/1 operator only in that it manages to find a solution in some cases that PROLOG does not[4] (since our tree traversal algorithm is complete). In all other cases, the behavior does

[4] If PROLOG does not find a solution, this means that PROLOG does not terminate when evaluating the negation. In some of these cases, our algorithm is able to falsify the negation and thus terminate.

not differ between our implementation and traditional PROLOG implementations.

10.2. Cut and once/1

10.2.1. once/1 as an Alternative to Declarative Use of Cut

In PROLOG, there is another short-circuiting operator known as *cut*. The cut operator is an extralogical operator with very little inherent declarative meaning. In fact, *cut* can be understood as a pure control flow construct whose declarative meaning depends on the precise execution order. In our system, since the execution order is not usefully predictable, *cut*, in a sense, has no meaning whatever. However, we can replace the *cut* operator with the slightly more declarative once/1 operator.[4] once/1 evaluates its argument as if invoked directly, but restricts its output to the first solution found. In PROLOG, once/1 can be trivially defined by using *cut*.

```
once(X) ← X, !.
```

cut cannot itself be expressed using once/1. However, many declarative uses of *cut* are representable using once/1.[9, pp. 25-27][9, pp. 85-111][10, pp. 103-120][6]

10.2.2. Implementation of once/1

once/1 is implemented just like not/1 except that the success and failure cases are reversed and, in the case of success, the set of variable bindings yielded by the succeeding branch is propagated upwards.

11. General Optimization Techniques

11.1. Tail-Call Optimization

As logic programming, and certainly PURE PROLOG, discourage imperative constructs, loops are generally not encoded directly. Instead, they are modelled using a special kind of recursion called *tail recursion*, where a rule contains a recursive call in *tail position*, i.e. as the expression that is evaluated last. Traditional PROLOG compilers take great care to convert tail calls into jumps, since overflowing the stack when doing tail recursion would make it effectively impossible to write tail-recursive loops, undermining competitiveness with imperative programming languages.[11]

At first glance, it may not be clear that we can compete with traditional PROLOG in terms of tail-call optimization, since our approach is not based on an execution stack, but rather an execution tree. However, we can indeed do effectively the same kind of optimization, albeit in a different way.

In the case of an evaluator operating on an execution tree like ours, when executing a largely linear piece of a program, it is quickly observed that the mechanisms outlined so far

tend to leave a long trail of *trivial* goal-subgoal relationships in the execution tree. These are those child-parent pairs where the child represents the last subgoal of the parent. Since apart from V_{sync}, there is no useful information left in the parent, such pairs can easily be collapsed into a single node by doing a sort of premature upward propagation, which is achieved by relinking the child so that it points directly to the grandparent while appropriately synchronizing the relevant variable lists between the grandparent, parent and child. This process is called *tail-call optimization*.[5] While it is most noticable in the presence of a tail-recursive loop, it actually applies to any kind of tail call.[5] For instance, after tail-call optimization, the following example definition of even/1 will only consume a constant amount of stack space.

```
even(0).
even(X) ← Y is X−1, odd(Y).

odd(1).
odd(X) ← Y is X−1, even(X).
```

Our implementation does tail-call optimization whenever possible in the way just described. Note that tail-call optimization *must not* be done when the parent is a negation or once/1 node. This is because in such a case, the link between the children of the negation or cut node would be severed, which would make it impossible to cut off the other branches in the case of success.

12. Specific Optimization Techniques

12.1. Tree Cutting

12.1.1. Idea

Whenever a negation node fails or a once/1 node succeeds, all of its children are immediately removed from the execution tree. Naively, this is done by recurring down the subtree and removing the leaves, which triggers removal of the parents in a loop ascending up to the negation or once/1 node. However, because of the way the execution scheme works, we can do better: Since although the exact order of node processing is complicated by the interaction between breadth-first search and depth-first search, the algorithm still processes nodes strictly left-to-right on each level, a node's children are bound to line up in a straight row. Moreover, this structural feature reproduces recursively when moving down the tree. That is, the descendants of a given node (such as a not/1 or once/1 node) on a given tree level are always adjacent to each other.[6] Moreover, leaves of a subtree will always be unprocessed nodes, and as such, located on the lowest possible level. This observation suggests an alternative approach to tree cutting in which whole subtrees are cut out of the

[5] This effectively enables the programmer to write any GOTO-based algorithm in PURE PROLOG without sacrificing memory efficiency. Of course, the desirability of exploiting this possibility to the fullest is debatable.

[6] We might call this property "horizontal continuity" of subtrees.

tree by traversing the subtree downward across its left and right boundaries, unlinking the traversed elements from the doubly linked level lists.

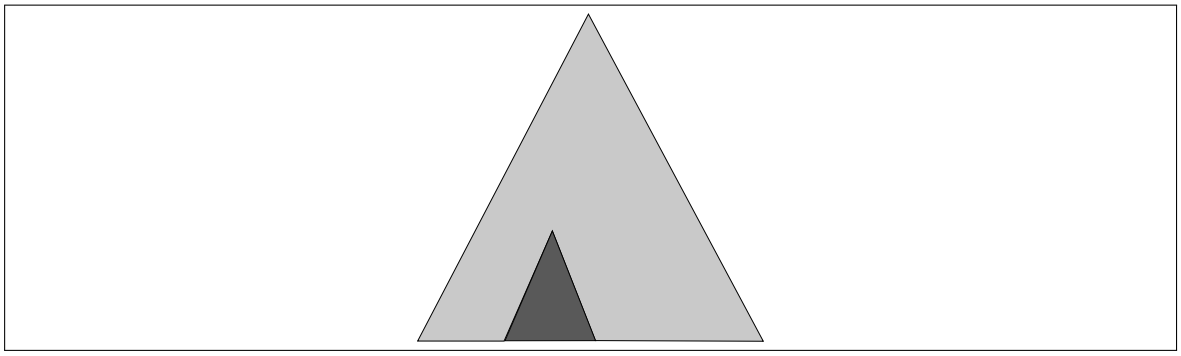

Abbildung 12.1: A tree (light-gray area) with a subtree (dark area) fit for cutting.

12.1.2. Implementation

Let n be the top node of the subtree to be cut. Moreover, let the *BFS cursor* be the node breadth-first search is waiting to process next. We need to distinguish three cases:

1. If n is on the execution tree's leftmost branch:
   a) If the subtree starts on a level below the current BFS cursor: Delete all levels below n.
   b) Otherwise, obliterate all levels more than one level below the current BFS cursor and proceed as in case 2.
2. Otherwise:
   a) If the subtree's rightmost descendant is on the same level as the leftmost one: Simultaneously traverse the left and right boundaries, unlinking them from the tree along with all intermediate nodes.
   b) Otherwise, the leftmost leaf must be one level below the BFS cursor while the rightmost leaf must be on the level of the BFS cursor. In this case, do the same thing as in case (2a) down to the deepest common level (which is the BFS cursor's level), and handle the deepest level by unlinking the subtree's leftmost descendant x, effectively removing everything on that level that is to the right of x.

Now recall the implementation of the execution tree as described in 7. With this, the above set of cases may be unified into a single algorithm:

1. Simultaneously traverse the left and right boundaries of the subtree. On each level, unlink the left and right boundary nodes from the level list and relink the level list so as to remove all the elements in between.
2. When the right boundary ends, proceed further down until the left boundary ends as well. This time, on each level, remove everything to the right of the left boundary.

12.1.3. Auxiliary data structures

For the above algorithm to work, the runtime environment must be able to determine the left and right boundaries of a subtree. This can be facilitated by maintaining yet another tree-shaped data structure which, for every node n, stores the leftmost and rightmost children of n. We call this data structure the *boundary tree*.

12.2. Outlook: Stack-based Depth-First Search

While B&D Prolog makes use of a single execution tree for both BFS and DFS, and accordingly pays a price in efficiency relative to a stack-based DFS design as in the WAM[13], it may in fact be possible to implement the DFS part of the algorithm using a traditional execution stack.

In this scheme, DFS proceeds by a backtracking algorithm implemented in WAM style. Whenever DFS backtracks far enough to eliminate all of the stack, it converts the leftmost unprocessed node of the BFS tree into a new stack, which it then processes. Likewise, whenever BFS advances a level, it converts the topmost stack frame into a tree node, effectively unwinding the stack from below.

Presumably, there is considerable complication in the details of this approach. These lie outside the scope of this work and may be an interesting research option to consider in the future.

Teil III. Implementation of a Metaevaluator in Prolog

13. Implementation Language and Data Structures

In order to verify the feasibility of the implementation techniques outlined above, an implementation is provided as a PROLOG-based metaevaluator. PROLOG was chosen as the implementation language because it enables us to reuse its built-in facilities for variable binding, term representation and unification, and arithmetic. However, since efficiency was a concern, the metaevaluator is not written in a pure logic programming style, but rather an imperative style not typical of code written in PROLOG. Moreover, SWI-specific features are used to destructively modify terms. A more declarative style, while more desirable on aesthetic grounds, would have forced the implementation to copy the execution tree whenever it was modified. Since the execution tree can become very large during the execution of a program, this would likely have had less than desirable effects on run-time performance.

The evaluator's state is maintained as a number of global variables, such as the current maximum DFS and BFS cursors and the current credit value. These are managed through the use of SWI-PROLOG's fact database (figure 13.1).

```
global(Key, Value) ← nb_getval(Key, Value).
override(Key, Value) ← nb_setval(Key, Value).
override_eval(Key, Term) ← Value is Term, nb_setval(Key, Value).
```

Abbildung 13.1: Global variable management.

A mutable doubly-linked list datastructure is used as the basis for the individual search tree levels.

Nodes in the execution tree are represented as mutable data structures with the following fields:

1. goals
2. vars
3. Execution-tree-structural attributes:
   a) parent
   b) children
4. Success propagation information:
   a) sync_vars
   b) goal_index
5. queue_item
6. type
7. Boundary-tree-structural (or *boundary chain*) attributes:
   a) left_chain_sibling
   b) right_chain_sibling
   c) chain_parent
   d) leftmost_chain_child
   e) rightmost_chain_child

The data structure for a node consists of a Prolog term of the form node(1,2,3a,3b,4a,4b,5,6,7a,7b,7c,7d,7e) where the arguments are terms for the 13 fields listed above. Whenever a field is inapplicable or nonexistent, its value is the atom nil.

14. Evaluator Loop

The main entry point to the evaluator loop is provided by the auxiliary predicate depth_breadth/1, which takes an ordinary Prolog term as an argument and feeds it into the evaluator as a starting state by calling init/1 (figure 14.1), before finally handing control over to continue_depth_breadth_cycle/0, the main evaluator dispatch loop (figure 14.2), which conforms to the trampoline design described in 6. continue_depth_breadth_cycle/0 determines whether to do a DFS or BFS step according to the D&B algorithm and transfers control to depth_step/0 or breadth_step/0, respectively. All of these predicates are straightforward translations of the procedures given in [3] as pseudo-code.

```
init(Term) ←
    !,
    initialise_level_queues,
    override(number_of_unexpanded_nodes, 1),
    override(depth, 0),
    override(max_depth, 0),
    override(breadth, 0),
    override(credit, 0),
    term_vars(Term, Term_Vars),
    termify_conjunction(Term, Goals),
    level_queue_push_all(0,
      [node(Goals, Term_Vars, root(Term_Vars, []),
            Term_Vars, nil, [], nil, regular,
            nil, nil, nil, nil, nil)]).
```

Abbildung 14.1: init/1.

15. Node Expansion

Nodes are represented as terms and destructively modified as needed rather than copied. Because of the large number of fields, accessor predicates (figure 15.1) are used both to set and retrieve field values.

The entry point into the node expansion mechanism is the expand_first/2 predicate, which takes a node and the current cursor level as its arguments, modifies the tree by expanding the given node, optionally yields a result in the form of a set of variable bindings, and finally returns control to the main evaluator loop. Note that a result will only be returned in case the given node represents a query result. In any other case, expand_first/2 operates through the use of side effects.

There are a considerable number of cases that expand_first/2 needs to deal with. These are processed as follows:

1. Apply tail-call optimization as long as it is possible. This is done by lifting a node upward in the tree as described in section 11.1.

```
continue_depth_breadth_cycle ←
    global(breadth, Breadth),
    global(breadth, Depth),
    (   Depth < Breadth →
        !, fail
    ;   global(credit, Credit),
        level_count(Breadth, Level_Count),
        (   Credit < Level_Count, Credit > 0 →
            (   prefer(breadth) →
                breadth_step
            ;   depth_step
            )
        ;   (   Credit < Level_Count →
                depth_step
            ;   breadth_step
            )
        )
    ).
```

Abbildung 14.2: continue_depth_breadth_cycle/1.

2. If the currently processed node is non-trivial, expand the node into zero or more child nodes by applying the expand_goal_into_nodes/3 predicate to the node's first subgoal, and consider the expansion step finished.
3. If the currently processed node is a negated triviality (i.e., a node that corresponds to the term not(true)), simply fail. This concludes the expansion step.
4. Otherwise, the node is successful, that is, all of its subgoals have successfully been proven. To propagate the success upward, we must proceed according to the type of its parent node:

    a) If the currently processed node has no parent, we have found a solution: Propagate it to the calling program. If propagation fails, this means that the calling program is requesting another solution. (In case the calling program is the PROLOG toplevel, the user may have requested another solution interactively.) In this case, behave as if the node failed, and conclude the expansion step.

    b) If the node's parent is a regular node, there are still goals to prove there. (We know this is the case because otherwise, tail-call optimization would have occurred and eliminated the parent.) Therefore, proceed by expanding the current node as if in case 2, but instead of making the resulting nodes child nodes of the current node, make them its siblings by temporarily synchronizing with the parent node, and remove the current node from the tree. (This is a non-essential optimization, which circumvents the need for explicit tail-call optimization in this common case.) This concludes the expansion step.

    c) If the node's parent is a negation node, we have proven a negated proposition. In this case, make the negation node fail, and conclude the expansion step.

d) If the node's parent is a once/1 node, we have found a solution for it. In this case, make the current node replace the once/1 node by moving upward one level and then making the once/1 node (along with all of its children) fail. This concludes the expansion step.

Case 3 may seem gratuitous. Indeed, the case of having to process a negated triviality is a pathological one, since the expression not(true) is unlikely to be encountered often in practice. On the other hand, such code may well be generated by code generators.

However, case 3 is actually necessary in the case of negation, while a similar case for once/1 is not: Since once(true) is equivalent to true, the behavior of the node is the same as if it had not been a once/1 node in the first place. not(true), on the other hand, would *succeed* if not treated specially, since its single subgoal true is otherwise used as a marker for successful subgoals awaiting propagation.

Teil IV.
General Outlook

16. Native Compilation

Naturally, a PROLOG-based metaevaluator will always be less efficient than a native implementation, if only because the way variable bindings are managed is unlikely to be optimized for structure sharing.

A native implementation of a B&D-based PROLOG can be realized in a relatively straightforward manner by imitating the Warren Abstract Machine. Variables are managed on what looks like a stack from the point of view of each indiviual branch but is actually handled by a copy-on-write scheme. This can be optimized easily by the use of purely functional data structures that facilitate structure sharing. In particular, a bitmapped, little-endian Patricia Tree[8] is likely to be a very efficient way of representing the stack, with variables represented by sequential machine integers.

17. Alternative Tree Traversal Algorithms

As noted in the introduction, while B&D search is certainly an improvement over pure BFS or even DFS in the general case, it is not clear whether we cannot do better. In fact, the concept of B&D search itself tends to inspire one to consider similar approaches replacing the BFS part of B&D search with iterative deepening or other well-known traversal schemes.[2, 3] However, rigorous complexity analyses, let alone an implementation proving their feasibility, are yet to be done on such algorithms.

```
node_fieldname_index(goals, 1).
node_fieldname_index(vars, 2).
node_fieldname_index(parent, 3).
node_fieldname_index(sync_vars, 4).
node_fieldname_index(goal_index, 5).
node_fieldname_index(children, 6).
node_fieldname_index(queue_item, 7).
node_fieldname_index(type, 8).
node_fieldname_index(left_chain_sibling, 9).
node_fieldname_index(right_chain_sibling, 10).
node_fieldname_index(chain_parent, 11).
node_fieldname_index(leftmost_chain_child, 12).
node_fieldname_index(rightmost_chain_child, 13).

query_node(Node, Query) ←
    test_assertion((Node =.. [node|_])),
    Query =.. [Type|Forms],
    node_do_queries(Node, Type, Forms).

node_do_queries(Node, Type, [Form|Forms]) ←
    node_do_query(Node, Type, Form),
    node_do_queries(Node, Type, Forms).

node_do_queries(_, _, []).
node_do_query(Node, get, Form) ←
    Form =.. [Param, Value],
    node_fieldname_index(Param, Index),
    arg(Index, Node, Value).

node_do_query(Node, set, Form) ←
    Form =.. [Param, Value],
    node_fieldname_index(Param, Index),
    nb_setarg(Index, Node, Value).

node_do_query(Node, link, Form) ←
    Form =.. [Param, Value],
    node_fieldname_index(Param, Index),
    nb_linkarg(Index, Node, Value).
```

Abbildung 15.1: Accessor predicates.

Teil V.
Conclusion

B&D search, along with related techniques, enable a more declarative style of programming with less need to worry about unintended dependence on the order of rules in the program. As we have seen, various optimization techniques can be applied to an implementation of a B&D-based PROLOG variant.

While outside the scope of this project thesis, exactly what performance can be expected of an implementation using these techniques requires further study. In particular, no measurements have been done to empirically verify the memory complexity guarantees made by B&D search. Neither have we verified that our optimization techniques actually effect improvements in real-world situations.

Finally, many of the optimization techniques described in this work easily generalize to a larger class of tree traversal algorithms, suggesting that further research in the area may be worthwhile.

Literatur

[1] H. G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. comp.lang.scheme.c newsgroup, 1994.

[2] S. Brodt. Tree-search, Partial Orderings, and a New Family of Uninformed Algorithms. Forschungsbericht/research report PMS-FB-2009-7, Institute for Informatics, University of Munich, 2009.

[3] S. Brodt, F. Bry, and N. Eisinger. Search for More Declarativity — Backward Reasoning for Rule Languages Reconsidered. In *Web Reasoning and Rule Systems, Proceedings of Third International Conference on Web Reasoning and Rule Systems, Chantilly, Virginia, USA (25th–26th October 2009)*, volume 5837 of *LNCS*, pages 71–86, 2009.

[4] L. F. Castro and D. S. Warren. Approximate Pruning in Tabled Logic Programming. In P. Degano, editor, *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2003.

[5] W. D. Clinger. Proper tail recursion and space efficiency. pages 174–185. ACM Press, 1998.

[6] S. K. Debray and D. S. Warren. Towards banishing the cut from Prolog. *IEEE Trans. on Software Engineering*, 16:2–12, 1990.

[7] R. A. Kowalski. Predicate Logic as Programming Language. In *World Computer Congress*, pages 569–574, 1974.

[8] C. Okasaki and A. Gill. Fast Mergeable Integer Maps. In *In Workshop on ML*, pages 77–86, 1998.

[9] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

[10] L. Shapiro and E. Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, Apr. 1994.

[11] Y. Shoham. *Artificial Intelligence: Techniques in PROLOG*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[12] D. Tarditi, P. Lee, and A. Acharya. No Assembly Required: Compiling Standard ML to C. *LOPLAS*, 1(2):161–177, 1992.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.