Jekejeke Runtime Benchmark

Version 1.0.6, May 4th, 2015

XLOG Technologies GmbH

Jekejeke Prolog
Runtime Library 1.0.6
Benchmark Core Results

Author: XLOG Technologies GmbH
Jan Burse
Freischützgasse 14
8004 Zürich
Switzerland

Date: May 4th, 2015
Version: 0.21

Participants: None

Warranty & Liability

To the extent permitted by applicable law and unless explicitly otherwise agreed upon, XLOG Technologies GmbH makes no warranties regarding the provided information. XLOG Technologies GmbH assumes no liability that any problems might be solved with the information provided by XLOG Technologies GmbH.

Rights & License

All industrial property rights regarding the information - copyright and patent rights in particular - are the sole property of XLOG Technologies GmbH. If the company was not the originator of some excerpts, XLOG Technologies GmbH has at least obtained the right to reproduce, change and translate the information.

Reproduction is restricted to the whole unaltered document. Reproduction of the information is only allowed for non-commercial uses. Small excerpts can be used if properly cited. Citations must at least include the document title, the product family, the product version, the company, the date and the page. Example:

> … Defined predicates with arity>0, both static and dynamic, are indexed on the functor of their first argument [1, p.17] ...
>
> [1] Language Reference, Jekejeke Prolog 0.8.1, XLOG Technologies GmbH, Switzerland, February 22nd, 2010

Trademarks

Jekejeke is a registered trademark of XLOG Technologies GmbH.

Table of Contents

Change History

Jan Burse, July 2nd, 2010, 0.1:

- Initial version with benchmarks, harness and raw data.

Jan Burse, July 20th, 2010, 0.2:

- Clause indexing section removed and discussions completed.

Jan Burse, January 2nd, 2011, 0.3:

- Updated to results of release 0.8.7.

Jan Burse, Mai 7th, 2011, 0.4:

- Updated to results of release 0.8.9.

Jan Burse, August 8th, 2011, 0.5:

- Updated to results of release 0.9.0.

Jan Burse, August 10th, 2011, 0.6:

- Added new benchmark cases perfect number and DCG calculator.

Jan Burse, August 13th, 2011, 0.7:

- Removed benchmark case primes and paths explained.

Jan Burse, August 18th, 2011, 0.8:

- Stack frame elimination and head variable elimination updated.

Jan Burse, August 22th, 2011, 0.9:

- Clause indexing comparison and Ciao Prolog system comparison introduced.

Jan Burse, September 20th, 2011, 0.10:

- New benchmark Tic-tac-toe and intermediate code changes.

Jan Burse, September 27th, 2011, 0.11:

- Updated to JDK 1.7 measurements and recent improvements.

Jan Burse, November 21th, 2011, 0.12:

- Updated to release 0.9.2 improvements.

Jan Burse, April 9th, 2012, 0.13:

- Updated to release 0.9.3 improvements.

Jan Burse, July 18th, 2012, 0.14:

- Updated to release 0.9.4 improvements.

Jan Burse, October 26th, 2012, 0.15:

- Updated to release 0.9.6 improvements.

Jan Burse, March 1st, 2013, 0.16:

- Updated to release 0.9.8 changes.

Jan Burse, August 22th, 2013, 0.17:

- ECLiPSe Prolog introduced and SICStus Prolog removed.

Jan Burse, December 12th, 2013, 0.18:

- Updated to SWI-Prolog 7 non-traditional.

Jan Burse, April 6th, 2014, 0.19:

- Updated to release 1.0.1 regression and use of ensure_loaded/1.

Jan Burse, June 12th, 2014, 0.20:

- New uses based benchmark file organization.

Jan Burse, Mai 4th, 2015, 0.21:

- Updated to release 1.0.6 changes and Prolog sources moved out.

1 Introduction

This document describes some benchmarking results for the Jekejeke Prolog system.

- **Study Object:** In this section we give a brief introduction into the architecture of the Jekejeke Runtime interpreter.

- **Available Optimizations:** We will also highlight some optimizations that were implemented for the interpreter.

- **Strategies Comparison:** We conducted a couple of performance tests. The main indicator that was measured was the elapsed time for various test cases.

- **Interpreter Comparison:** We conducted a couple of performance tests. The main indicator that was measured was the elapsed time for various test cases.

- **Appendix Harness Listings:** The full source code of the Java classes and the Prolog texts for the test harness is given.

- **Appendix Test Program Listings:** The full source code of the Prolog texts for the test programs is given.

2 Study Object

In this section we give a brief introduction into the architecture of the Jekejeke Runtime interpreter. We will highlight the following points:

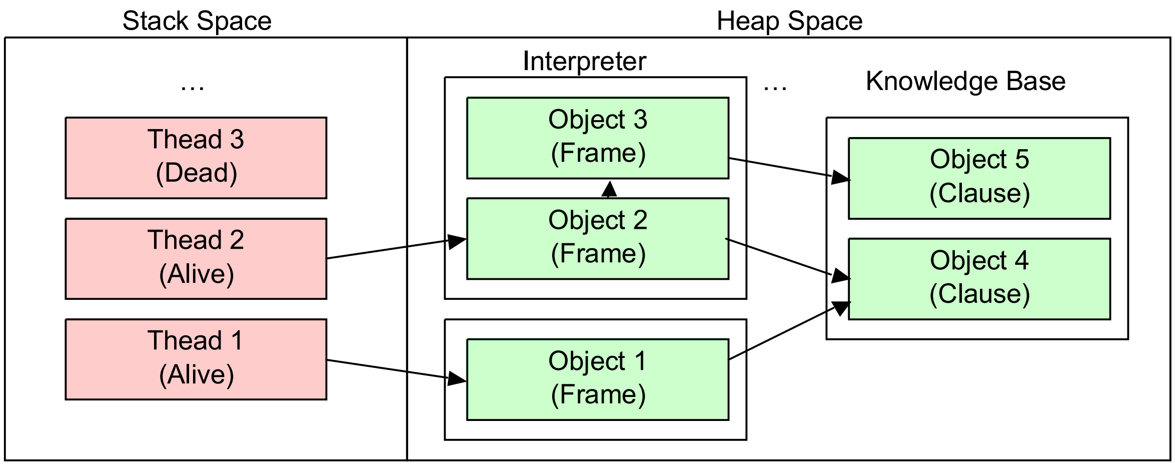
- **Memory Organization:** The Jekejeke Prolog system is based on the notion of interpreter and knowledge base Java objects. We will explain how these objects are laid out in memory and how they basically interact.

- **Resolution Step:** To later understand the available optimizations, we must first understand how an interpreter frame is further structured into sub objects.

- **Test Scope:** In this section we will describe our testing approach in more detail. In particular we will give details how we measured the test cases and under what circumstances we run the test cases.

2.1 Memory Organization

The Jekejeke Prolog system is based on the notion of interpreter and knowledge base Java objects. We will explain how these objects are laid out in memory and how they basically interact. The Java virtual machine allocates to each thread its own stack. The stack size is a parameter during the creation of the thread. During their execution, the threads then share the heap where the Java objects reside.



Picture 1: Memory Organization

The Java virtual machine automatically reclaims used heap and stack space by means of a garbage collection. When a thread dies the stack space is returned. When a Java object is not anymore referenced by the stack of a living thread either directly or via other Java objects than its heap space is returned. The reclamation of Java objects is done periodically or when certain conditions meet. There is no means to explicitly return heap space.

The knowledge base consists basically of a set of clauses, which are in turn Java objects. These clauses can be referenced and manipulated during the execution of a Prolog program. We did not need to implement any special mechanism for the reclaim of clauses, since we delegated the process fully to the Java virtual machine. When a clause is removed it will not be any more referenced by the knowledge base. If an active thread does not anymore use the clause, it will then automatically be removed by the garbage collection sooner or later.

Whether an interpreter organization benefits from automatic garbage collection is less obvious. The interpreter will need a new frame whenever he performs a resolution step with another clause. The basic backtracking algorithm will create and release frames in high frequency and it could benefit from explicitly returned heap space. Also there are hardly any references between frames that could be used by an automatic garbage collector since it will not understand the indexing structure of Prolog variables into frames.

The Java virtual machine offers help here in the form of generation scavenging. This means that the Java virtual machine can distinguishes between young and old objects, and young objects are faster and more often reclaimed than old objects. For the tip of the backtracking algorithm the interpreter fames will qualify for young objects. As a result these frames will be automatically reclaimed by the garbage collection very quickly. The performance is even better compared to explicit returned space, since the garbage collector can work in bulk.

Nevertheless having frames as allocated entities is costly. This holds for any Prolog system implementation. Therefore various optimizations have been developed over the time. The Jekejeke Prolog system implements a couple of these optimizations. We will explain them in the following sections. In the context of garbage collection, these optimizations have all in common that they try to archive either of the following:

- **Circumvent Allocation:** Avoid the creation of a Java object early on. Java objects that have never been created don't need to be reclaimed.

- **Decrease Connectedness:** Lower the reachability of a Java object early on. Java objects that are not reachable anymore can be reclaimed.

2.2 Resolution Step

To later understand the available optimizations, we must first understand how an interpreter frame is further structured into sub objects. Interpreter frames are required to represent a successful resolution step and they should also accommodate for the undoing and the continuation of a resolution step. Both capabilities are needed in the backtracking algorithm.

The resolution step is based on the following input:

```
    A :- B₁, .., Bₙ    The old answer term and old list

    C :- D₁, .., Dₘ    The picked clause
```

The answer term can be viewed as a list of the variables from the initial query. It is carried around so that we can display the answer substitution when the initial query has been successfully solved.

Now suppose that ρ is a substitution that will rename the variables of the picked clause by new fresh variables. Further suppose that θ is the most general unifier of B_1 and ρC. Then the resolution step will yield the following new current answer term and current goal list:

```
    θA :- θρD₁, .., θρDₘ, θB₂, .., θBₙ    The new answer term and goal list
```

The above can be efficiently implemented by means of skeleton and display pairs. Displays serve the purpose of providing fresh un-instantiated variables for a clause. They are thus capable of representing the substitution ρ. The variable place holders inside a display are then modified during unification. They are thus also capable of representing the most general unifier θ. Therefore the dominant structure of a frame will be the display.

But with the display alone we could not implement backtracking, since it will not allow us the undoing and the continuation of a resolution step. For undoing the resolution step it is not enough to simply forget the current display, since unification might also have modified previously allocated displays. Therefore the unification will produce a trail which records the modified variable place holders and which can later be used to undo the unification.

Besides the trail we will also need a pointer into the input list of the clauses. The pointer will refer to the currently picked clause. This will allow us to continue our search with the next clause upon backtracking. Besides that a further continuation structure will be needed. Namely instead of building the new goal list by appending the old goal list and the picked clause body, we simply store the old goal list as a later continuation in the display. And the search continues immediately with the picked clause body.

In summary an interpreter frame is structured as follows. The fields stemming from the resolution step are marked by an asterisk (*). All other fields have been introduced to provide further functionality needed by the Jekejeke Prolog implementation:

Display

- ***Place Holders:** Might refer to a skeleton and display pair.
- ***Continuation:** Points to the old goal list.
- **Prune:** Points to the true cut interpreter frame.
- **Number:** The number of choice points when this display was created.
- **Flags:** Flags of this display.

- **Clause:** The originating clause for which this display was created.

Choice Point

- ***At:** The currently picked clause.
- ***Trail:** Lists the instantiated place holders.

It should be noted, that we divert from the architecture of the Warren Abstract Machine (WAM) in two respects. The first difference can be seen in the way goals of a clause are invoked. In the WAM machine the skeleton display pointer pair approach is broken for goals. It is explicitly noted in [4] that goal invocations are dynamically created by assembling the arguments and then invoking the predicate. In our architecture we stick to the skeleton display pointer pair approach. Each goal in a clause is simply a different skeleton pointer which is completed by the display pointer of the clause instantiation.

Sticking to the skeleton display pointer pair approach for goals can be very efficient when there are efficient means to move from one skeleton pointer to the other. This is archived by turning each body of a clause into a goal list representation. But Prolog would not be complete if it would only be based on an efficient resolution step. What is typically also needed is a construct that allows the pruning of the search tree. The usual predicate for this is the cut and which will destroy any choice points up to its own frame.

The second difference is now found in the way how we deal with the cut. Jekejeke Prolog differs from other Prolog as far as we allow naked variables and cut transparent predicates. For this purpose the frame also contains a prune field, which points to the true frame that will belong to a cut. As a result we can invoke cuts inside meta-predicates with the desired result. Eventually this could also be realized inside a WAM, but the default mode of analysis and execution is that a cut is statically recognized inside a clause and that the cut will only prune its own frame and not an eventually inherited frame.

We have already explained most of the fields of the interpreter frame. The remaining “number”, “perms” and “clause” fields have various purposes. The “number” field is used by the body variable elimination optimization to notice whether the intermediate goals were deterministic, i.e. whether they have not created any new choice points. The “perms” field is used to determine whether the current execution belongs to the user or the system, and also to avoid double work in the stack frame elimination optimization. The “clause” field is simply used to provide source file and line number information in a stack trace.

2.3 Test Scope

In this section we will describe our testing approach in more detail. In particular we will give details how we measured the test cases and under what circumstances we run the test cases. The test harness was used to perform internal and external tests. In the internal tests we compared Jekejeke Prolog in different configurations. In the external tests we compared the optimal Jekejeke Prolog configuration with different existing products.

The test scope included the following features of a Prolog interpreter:

- The elapsed runtime time.
- Built-in predicates for arithmetic (is/2, </2, etc..).
- Defined predicates with argument unification.
- Basic control constructs (!/0, fail/0, etc..).
- List comprehension predicates (findall/3, etc..).
- Definite clause grammar rules (-->/2, phrase/3, etc..).
- Advanced control constructs (;/2, ->/2, \+/1, etc..).

The test scope did not include the following features of a Prolog interpreter:

- The memory usage at runtime time.
- Built-in predicates for structures (=../2, =/2, functor/3, @</2, etc..).
- Built-in predicates for atoms (atom_concat/3, atom_length/2, etc..).
- Dynamic database access (assertz/1, retract/1, clause/1, etc..).
- Text and byte stream input/output (put_char/1, get_byte/2, etc..).
- List sorting predicates (sort/2, keysort/2, setoff/3, etc..).
- Perpetual processes via infinite recursive loops.
- The elapsed consult time.
- The memory footprint of consulted files.

To perform the internal and external tests we used the same set of test programs and the same kind of harness. Each test program consists of an entry point that will compute the solution to a problem. This entry point was then iterated via the test harness. The number of iterations was chosen in such a way that for each test program a time measure of the same order can be obtained. The following test programs and number of iterations were used:

Table 1: Iterations of the Test Programs

Iterations	Name	Description
6’001	nrev	Naïve reverse
301	crypt	Crypt arithmetic.
30’001	deriv	Symbolic derivation.
61	poly	Polynomial reduction.
6’001	qsort	Quick sort of a list.
11	tictac	Tic-Tac-Toe game.
16	queens	9-queens problem.
3’001	query	Database query.
31	mtak	McCarthy Tak function.
11	perfect	Perfect numbers.
20’001	calc	DCG calculator.

When doing the internal tests we varied some Prolog flags. We used a Java class and our Prolog API as the main entry point so that the runtime library is tested. For the external tests we used the Java class again. For each compared Prolog system we had to adapt the code that retrieves the elapsed runtime. Details on the test harness and the test programs can be found in the appendix.

3 Test Programs

In the following we give a short description of each test program. The following test programs are provided:

- nrev Test Program
- crypt Test Program
- deriv Test Program
- poly Test Program
- qsort Test Program
- tictac Test Program
- queens Test Program
- query Test Program
- mtak Test Program
- perfect Test Program
- calc Test Program

This document does not contain the source code of the test programs. A browsable version of the source code of the test programs can be found on the following web site:

www.jekejeke.ch/idatab/doclet/blog/en/docs/05_run/07_benchmark/package.jsp

Further the source code is also bundled in the suprun.zip when downloading the Jekejeke Prolog runtime library from the web site.

3.1 nrev Test Program

The naïve reverse computes the reverse of a list by invoking a concatenate function. The order of the algorithm thus becomes $O(n^2)$. An algorithm with an accumulator parameter would be more efficient. Nevertheless the predicate is a popular benchmark and it was already found in D.H.D. Warren’s thesis [2, page 219].

One test iteration will reverse a list with 30 numbers.

3.2 crypt Test Program

The crypt riddle is a complication of the well-known SEND + MORE = MONEY riddle. Our riddle is found in W.C. Trigg’s collection [1, problem 223] and it will not only involve addition but also multiplication. We solve for a result of the following riddle. The goal is to find digits that can replace the letters:

```
  O E E
    E E
------- *
E O E E
E O E
-------
O O E E
```

A letter E stands for an even digit. The letter O stands for an odd digit. The initial E letters are not allowed to receive the zero value. Otherwise all digits of the corresponding class are allowed. The riddle has exactly one solution as follows:

```
  3 4 8
    2 8
------- *
2 7 8 4
6 9 6
-------
9 7 4 4
```

One test iteration will solve the riddle.

3.3 deriv Test Program

Symbolic derivation of formulas is mainly based on the chain rule. This rule reduces the derivation of the composition of two functions to the derivation of each separate function. The chain rule reads as follows:

```
      dy(u)    dy(u) du
      ----- =  ----- --
      dx       du    dx
```

Early implementations of symbolic derivation were reported with LISP. The Prolog implementation can take advantage of pattern matching. Our test predicate will not perform a simplification of the resulting expressions. The benchmark is already found in D.H.D. Warren’s thesis [2, page 222].

One test iteration will derive the ops8, the divide10, the log10 and the times10 examples.

3.4 poly Test Program

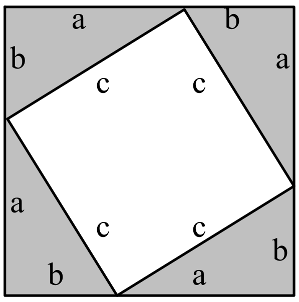
Simple polynomials are known to most of us from basic school algebra. They can be used to express geometrical identities such as the Pythagorean Theorem $a^2 + b^2 = c^2$. Algebraic manipulations can even mimic geometrical reasoning. Here is a proof of the Pythagorean Theorem based on some areal equations:

square area = grey area + white area

$$\therefore (a+b)^2 = 4ab/2 + c^2$$

$$\therefore a^2 + 2ab + b^2 = 2ab + c^2$$

$$\therefore a^2 + b^2 = c^2$$



Picture 2: Pythagorean Theorem

In higher mathematics polynomials can be viewed as algebraic extensions of rings. An interesting operation is the evaluation of an algebraic expression in a polynomial ring. This is exactly what the following Prolog text does. The original Prolog text is an adaption of a Lisp program and can be found in the Aquarius test suite [3].

One test iteration computes $(1\text{-}x+y\text{-}z)^{10}$.

3.5 qsort Test Program

This predicate sorts a list by the quick sort algorithm developed by C.A.R. Hoare. The algorithm recursively proceeds by splitting the given list into two halves and then sorting each half. In the given benchmark the splitting is simply based on the head of the list. The algorithm has a good average order of O(n log n), but it can have an order of O(n^2) in the worst case. The benchmark is already found in D.H.D. Warren's thesis [2, page 220].

One test iteration will sort a list with 50 numbers.

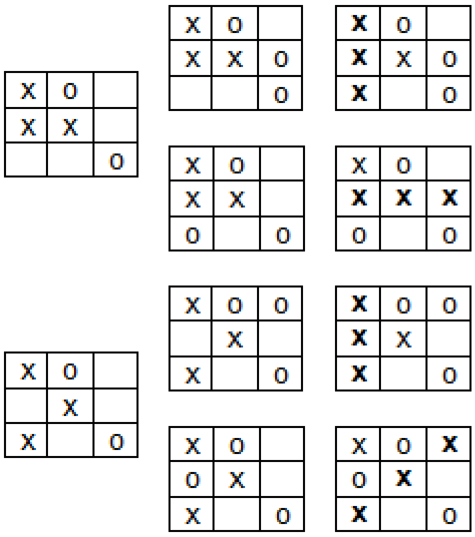
3.6 tictac Test Program

Tic-tac-toe is the classic toy example for min-max search [13]. The search space of the game is small enough so that a full depth search can be deployed. End positions in the game can be valued -1, 0 and +1 depending on whether the opponent wins, a tie has been reached or the current player wins.

Since the valuation is tri-state we have implemented the min-max search via negation. The predicate best/3 should indicate that from the given position there is a move that puts the given player in a position with a winning strategy. This can be used to play the game. We can for example pose the following query and will get the following answers:

```
?- best([[x,o,-],[-,x,-],[-,-,o]], x, X).
X = [[x, o, -], [x, x, -], [-, -, o]] ;
X = [[x, o, -], [-, x, -], [x, -, o]] ;
No
```

We can easily verify that the computed positions are indeed winning positions:



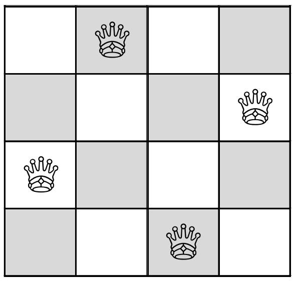
Picture 3: Search Tree Excerpt

In one test iteration we will verify that from start the first player has no winning strategy. This can be verified in that the following query fails:

```
?- init(X), best(X,x,_).
No
```

3.7 queens Test Program

In the modern chess play the queen piece has the most power, as it can move along the diagonals, the verticals and the horizontals. It thus combines the power of the bishop piece and the rook piece. Instead of playing chess the chess board can also be used to pose riddles. One such riddle asks for the placement of nine queens so that they don't attack each other. Here is a solution on the 4x4 checker board:



Picture 4: 4 Queens

The solution algorithm works similar to the solution algorithm for the Sieve of Eratosthenes in that occupied places are crossed out. The difference is that backtracking is involved in the placement of the queens and that we do not represent the whole board. Only the distinct positions of the placed queens are remembered in a list, which makes the check for horizontal and vertical obsolete. The diagonal check is then implemented via arithmetic.

One test iteration will compute all 352 solutions for the 9x9 checker board.

3.8 query Test Program

The query benchmark tests the suitability of Prolog for database problems. Prolog facts are used to represent information about countries such as area and population. The query then computes countries of similar population density. The benchmark is already found in D.H.D. Warren's thesis [2, page 220].

One test iteration will find all similar countries.

3.9 mtak Test Program

Highly recursive functions that majored primitive recursive functions have already been invented by Ackerman in the late 1920s. In 1970s Takeuchi came up with a highly recursive function that has astonishing simple closed form solution. The recursive function is:

```
                    | Y                  for X <= Y
    tarai(X,Y,Z) := |
                    | tarai(tarai(x-1,y,z),tarai(y-1,z,x),tarai(z-1,x,y))
```

And its close form:

```
                    | Y                  for X <= Y
                    |
    tarai(X,Y,Z) := | Z                  for Y <= Z
                    |
                    | X
```

But McCarthy spoiled it all by a slight modification. Instead of returning Y in the first case of the recursive definition, in his version the value of Z is returned. This function has to become known as the tak function.

One test iteration will compute the tak function for the arguments (18, 12, 6).

3.10 perfect Test Program

A perfect number is defined to be one which is equal to the sum of its aliquot parts. The four perfect numbers 6, 28, 496 and 8128 seem to have been known from ancient times and there is no record of these discoveries:

$6 = 1 + 2 + 3$,
$28 = 1 + 2 + 4 + 7 + 14$,
$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$
$8128 = 1 + 2 + 4 + 8 + 16 + 32 + 64 + 127 + 254 + 508 + 1016 + 2032 + 4064$

The first recorded mathematical result concerning perfect numbers which is known occurs in Euclid’s Elements written around 300BC. He gave a condition on perfect numbers based on what later became known as Mersenne prime numbers.

One test iteration will find the perfect numbers below 500.

3.11 calc Test Program

Definite clause grammars (DCG) are an extension of context free grammars. DCGs allow arbitrary tree structures to be built in the course of parsing and they allow extra conditions dependent on auxiliary computations. DCGs can also be used for text generation.

In our test program we want to use a DCG to parse an arithmetic expression and to return the evaluation of that expression. A first attempt would lead to the following grammar rule to handle for example subtraction:

```
    expr(Z) --> expr(X), "-", term(Y), {Z is X-Y}.
```

Unfortunately the above rule is left-recursive and top down parsing via DCG would run into an infinite loop. Therefore we use a modified set of grammar rules that implements the syntax via tail recursion and the evaluation via additional accumulator variables. We have also put cuts as to make the parser greedy and deterministic.

One test iteration will parse and evaluate the following expressions:

```
    -12+34*56+78            /*  Evaluates to 1970 */
    (-12+34)*(56+78)        /*  Evaluates to 2948 */
```

4 Available Optimizations

We will also highlight some optimizations that were implemented for the interpreter. In particular we will further explain the following points:

- **Choice Point Elimination:** The Jekejeke Prolog system is capable of omitting choice points for deterministic predicates. We will describe in more detail the used technique.
- **Clause Indexing:** The Jekejeke Prolog system is capable of dynamically creating indexes for multiple argument positions and that might span multiple arguments.
- **Body Variable Elimination:** The Jekejeke Prolog system is capable of reclaiming variables during its execution. We will describe in more detail the used technique.
- **Stack Frame Elimination:** The Jekejeke Prolog system is capable of dropping stack frames for last call goals. We will describe in more detail the used technique.
- **Head Variable Elimination:** The Jekejeke Prolog system is capable of eliminating the need for some head variables. We will describe in more detail the used technique.

4.1 Choice Point Elimination

The Jekejeke Prolog system is capable of omitting choice points for deterministic predicates. We will describe in more detail the used technique. The point of departure for choice point elimination is a naïve implementation of Prolog with cut. In this implementation defined predicates always create choice points after success and the cut only removes choice points upon redo. Choice point elimination now implements the following optimizations to reduce the consumption of choice points:

1) **Exit from Call last Clause:** When a defined predicate was invoked for the first time and the last clause was found then don’t create a choice point.

2) **Exit from Redo last Clause:** When a defined predicate was invoked an additional time and the last clause was found then release the current choice point.

3) **Cut on Call:** The cut will remove all choice points up to its own true frame before succeeding. In doing so it will not destroy the bindings from the intermediate goals.

To explain the optimizations 1) and 2) we can use the following example Prolog text:

```
p(a).
p(b).
```

We have two facts. When the first fact matches, we need a choice point so that we can go to the second fact. In the naïve implementation we keep the choice point even when the second fact matches. We can test this as follows:

```
?- set_prolog_flag(sys_choice_point, off).
Yes
?- p(b).
Yes ;
No
?- p(X).
```

```
X = a ;
X = b ;
No
```

The set_prolog_flag/2 system predicate allows us to switch off choice point elimination. When we invoke p(b), we see that after the success the system will prompt. This is an indicative that choice points are still around. Further when we invoke p(X), we see that after the first and the second solution the system will prompt. The prompt is again an indicative that choice points are still around. In both cases we quit the prompt with a redo request.

Now let’s run the example with choice point elimination enabled. We can use the directive set_prolog_flag/2 to switch on the choice point elimination optimization:

```
?- set_prolog_flag(sys_choice_point, on).
Yes
?- p(b).
Yes
?- p(X).
X = a ;
X = b
```

As could be seen, after the success of p(b) we will did not get a prompt anymore. This is now an indicative that the creation of choice point in situation 1) has been omitted. Further we also don’t get a prompt anymore after the last solution of p(X). Therefore the creation of choice point in situation 2) has been omitted.

To explain the optimizations 3) we can use the following example Prolog text:

```
p(X) :- q(X), !.
p(b).

q(a).
```

The predicate q has just one fact and is invoked by the predicate p. The predicate p has one rule and one fact. The rule that precedes the fact has a cut, so that when the first rule has passed its cut, the fact will not anymore be considered. In the naive implementation of the cut, the cut will only remove the choice point of the first rule in a redo. It will therefore automatically keep the bindings of the intermediate goals. We can test this as follows:

```
?- set_prolog_flag(sys_choice_point, off).
Yes
?- p(b).
Yes ;
No
?- p(X).
X = a ;
No
```

The cut works as expected. Procedurally we can derive p(b) since the cut was not passed as the q invocation failed. Procedurally we can derive for p(X) only one solution X=a, since this time the cut was passed. Both queries again result in a prompt which we quit with a redo request again. For the second query the prompt is an indicative that cut left the choice point untouched upon exit.

Now let’s run the example with choice point elimination enabled:

```
?- set_prolog_flag(sys_choice_point, on).
Yes
?- p(b).
Yes
?- p(X).
X = a
```

For both queries we do not get a prompt anymore. For the second query this means that in situation 3) the choice points where already removed when the cut was passed. This completes our exemplar exposition of the choice point elimination optimization for defined predicates and the cut.

The clean implementation of the choice point elimination optimization took us quite some time. Especially the cut was quite challenging. Because we want to keep the instantiations of the intermediate goals the trail of the eliminated choice points cannot be undone. On the other hand the trail of an eliminated choice point has to be merged with the other trails, so that upon an outer backtracking the instantiated variables are still correctly reset. The solution to this problem is a global trail with pointers into it.

The optimization is quite fundamental, since without the ability to eliminate choice points the head variable elimination and the stack frame elimination would not be possible. For more details on these two optimizations see some later sections. It should also be noted that choice point elimination and clause indexing have a synergetic effect, since optimization 1) and 2) might happen more often to smaller groups of clauses. For more details on the clause indexing optimization see the next sections.

Choice point elimination is also available for system and custom built-ins. System built-ins are free to create and destroy arbitrary choice points. System built-ins can thus follow choice point elimination strategies that even reach into optimization 3). For custom built-ins defined via Java methods there is a special protocol for the elimination of choice points. The Java foreign predicate can indicate via the try flag in the call out object whether they want to create respectively keep a choice point. This caters for the optimizations 1) and 2).

4.2 Clause Indexing

The Jekejeke Prolog system is capable of dynamically creating indexes for multiple argument positions and that might span multiple arguments. We will describe in more detail the used technique. The point of departure for clause indexing is a naïve implementation of Prolog without any indexing. In this implementation defined predicates are always scanned in full to find a matching clause. Clause indexing now implements the following indexes on the clause set of a predicate:

1) **First Argument Indexing:** When the defined predicate is called with a non-variable first argument then the functor of this argument is used in a hash table lookup to find a smaller set of clauses for scanning.

2) **Non-First Argument Indexing:** If the first argument is a variable then the calling goal is searched for another argument which is non-variable and the functor of this argument is then picked for a hash table lookup.

3) **Multi-Argument Indexing:** The hash table does not point to clause sets but to further hash tables so that recursively after a found non-variable argument further non-variable arguments can be searched in the calling goal.

Older Prolog systems did first argument indexing. This indexing was done independent of the call pattern so that the index was already created during the consult of the clauses. Jekejeke Prolog follows a different strategy. When clauses are consulted no indexes are necessarily maintained. Only after a defined predicate has been first invoked and the arguments have been analysed indexes are created. Additional indexes might later be dynamically created when the predicate is called with new non-variable argument patterns.

To explain the optimization 1), 2) and 3) we can use the following example Prolog text. We will use a binary predicate for illustration but the clause indexing method has also been implemented for defined predicates for less or more arguments:

```
r(a, b).
r(a, c).
r(d, c).
r(d, e).
```

We have four facts. We will be able to see what clause sets are picked in that we can see whether a choice point is left for the last clause in the set or not. In the naïve implementation always the full clause set is picked. We can test this as follows:

```
?- set_prolog_flag(sys_clause_index, off).
Yes
?- r(a, X).
X = b ;
X = c ;
No
```

The set_prolog_flag/2 system predicate allows us to switch off clause indexing. When we invoke r(a, X) then we see that after the first and second solution the system will prompt. This indicates that choice points are still around since the full clause set is scanned. Now let's run the example with clause indexing enabled. We can use the directive set_prolog_flag/2 to switch on clause indexing:

```
?- set_prolog_flag(sys_clause_index, on).
Yes
?- r(a, X).
X = b ;
X = c
```

As could be seen we don’t get a prompt anymore after the last solution of r(a, X). Therefore the clause indexing provided us with a smaller clause. This verifies the working of clause indexing in situation 1).

To explain situation 2) we can use the following query:

```
?- set_prolog_flag(sys_clause_index, off).
Yes
?- r(X, c).
X = a ;
X = d ;
No
```

When we invoke r(X, c) then we see that after the first and second solution the system will prompt. This indicates that choice points are still around since the full clause set is scanned. Now let’s run the example with clause indexing enabled:

```
?- set_prolog_flag(sys_clause_index, on).
Yes
?- r(X, c).
X = a ;
X = d
```

As could be seen we don’t get a prompt anymore after the last solution of r(a, X). Therefore the clause indexing provided us with a smaller clause.

We can use the following query to finally explain situation 3):

```
?- set_prolog_flag(sys_clause_index, off).
Yes
?- r(a, c).
Yes ;
No
```

When we invoke r(a, c) then we see that after the success the system will prompt. This is an indicative that choice points are still around since the full clause set is scanned. Now let’s run the example with clause indexing enabled:

```
?- set_prolog_flag(sys_clause_index, on).
Yes
?- r(a, c).
Yes
```

As could be seen, after the success of r(a, c) we will did not get a prompt anymore.

4.3 Body Variable Elimination

The Jekejeke Prolog system is capable of reclaiming variables during its execution. We will describe in more detail the used technique. This optimization technique introduces reference counting for variable place holders. During unification when a variable is bound to a term, the reference count of all the variables in the term is incremented by one. When the variable instantiation is undone the reference count of the variables in the unbound term is decremented by one. Since variable place holders start with a reference count of one, the reference count of a variable place holder should never reach zero this way.

Now a new instruction comes into play. The new instruction will be “dispose_bind” and it allows the explicit decrementing of the reference count of a variable place holder in the body. This instruction will be idempotent and it will only be applied when no choice points are left for the given clause. This allows constantly reclaiming variable place holders and the term structures that are bound by the variable place holders.

The decrementing of the reference count of a variable place holder and its delay of the corresponding “dispose_bind” instruction pose some overhead not necessary for non-deterministic predicates. The overhead can be minimized to simple counter checks and some speed up structure in the terms, leading to a surprisingly low additional cost. The implementation itself is based on a liveliness analysis of the clause variables. For each variable we determine the following two attributes:

- **Max Goal:** The value -1 if the variable only occurs in the head, otherwise the index of the last goal where the variable occurs.

The new instruction is then placed according to the above value. The “dispose_bind” instruction will be found in front of a goal call or a body end for those variables with max goal + 1 = goal index respectively clause size. Thus the “dispose_bind” instructions are placed at the earliest moment possible moment.

The effect of this optimization can be observed by listing the intermediate form. We will start with the head variable, stack frame and body variable optimization switched off. These optimizations are explained in a later sections and its presence would only complicate the current explanations. The switching off is done as follows:

```
?- set_prolog_flag(sys_head_variable, off).
?- set_prolog_flag(sys_stack_frame, off).
?- set_prolog_flag(sys_body_variable, off).
```

We will explain the effect by means of the following Prolog text. It is the Prolog code for the naïve reverse implementation:

```
rev([],[]).
rev([X|Rest],Ans) :- rev(Rest,L), concatenate(L,[X],Ans).

concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

We can now list the intermediate form of the rev/2 predicate via the built-in friendly/1. As can be seen the instruction “dispose_bind” is not yet used:

```
?- friendly(rev/2).
```

```
[...]

rev([X | Rest], Ans) :-
      rev(Rest, L),
      concatenate(L, [X], Ans).
   0 new_bind X, Rest
   1 unify_compound _0, [X | Rest]
   2 new_bind Ans
   3 unify_term _1, Ans
   4 init_display
   5 new_bind L
   6 call_goal rev(Rest, L)
   7 call_goal concatenate(L, [X], Ans)
   8 call_cont
```

The rest of the intermediate form is practically a literal translation of the head and the body of the clause. The head results in unify term instructions for each argument. The body results in goal call instructions “call_goal” for each goal. The intermediate form ends with a continuation call instruction “call_cont”.

The predicate rev/2 works as expected:

```
?- rev([1,2,3],X).
X = [3, 2, 1]
```

We will now switch on the body variable elimination optimization. This is done as follows:

```
?- set_prolog_flag(sys_body_variable, on).
```

We can now re-consult the Prolog text for the naïve reverse implementation. When generating the intermediate form during consult, the Prolog interpreter will now apply the body variable elimination optimization. We can again list the intermediate form of the rev/2 predicate via the built-in friendly/1:

```
?- friendly(rev/2).
[...]

rev([X | Rest], Ans) :-
      rev(Rest, L),
      concatenate(L, [X], Ans).
   0 new_bind X, Rest
   1 unify_compound _0, [X | Rest]
   2 new_bind Ans
   3 unify_term _1, Ans
   4 init_display
   5 new_bind L
   6 call_goal rev(Rest, L)
   7 dispose_bind Rest
   8 call_goal concatenate(L, [X], Ans)
   9 dispose_bind X, Ans, L
  10 call_cont
```

In the second rule of the predicate rev/2 we find that the “dispose_bind” instructions have been placed. Not all “dispose_bind” instructions have been placed before the continuation call. The “dispose_bind” instruction for the variable “Rest” has been placed before the last goal call, since this is the only variable that is not anymore used in the clause after this point.

The predicate rev/2 still works as expected:

```
?- rev([1,2,3],X).
X = [3, 2, 1]
```

The current implementation of body variable elimination is not based on some mode analysis such as free or ground [6]. The decrementing instructions are placed independent of the mode of the arguments. Nevertheless the mode has an effect on the decrementing. For example if a head argument is ground then the term unification will not increase the reference count of a variable place holder since the clause variable will be instantiated with a part of the ground term and not vice versa. As a result decrementing will always immediately remove the variable from the display and the trail.

On the other if a head argument is non-ground then there are two possible situations. The first situation is similar to the ground case. It happens when a clause variable will be instantiated with a part of the non-ground term. Again as a result the decrementing will always remove the variable. It can even be arranged that this situation also applies in a variable to variable instantiation. The second situation is when a variable of the non-ground term unifies with a part of a clause term. The reference count of the clause variables in the clause term will then be increased and the clause variables survive as long as the non-ground term survives.

The delay of the decrementing of a variable place holder has been implemented in the spirit of [7, section 4.4.1]. Namely we also reorder the variables of a clause with respect to their max goal value and can then simply move an index. Further we also pose a choice point condition. But it seems that our solution is a little bit more flexible, since we can still move the index when the choice point is later removed. But our system might forget about the decrementing when an outer choice point removal happens. So there is room for implementation improvement and better coverage in the benchmarks.

4.4 Stack Frame Elimination

The Jekejeke Prolog system is capable of dropping stack frames for last call goals. We will describe in more detail the used technique. We have already seen the ingredients of a stack frame in the previous chapter. Among the fields of the stack frame we find the continuation. This field points to the old goal list skeleton and display. There are various options to implement this field. For the requirement of the resolution step it would be enough to point to the old goal list minus the old goal. For the purpose of obtaining a call chain it is more convenient to also include the old goal in the pointer.

When building the stack frame the Jekejeke Prolog interpreter includes the old goal skeleton and display in the continuation frame. There is a slight speed up by this method, since the access to the goal list succeeding the old goal is thus deferred to the new end body. Following this field from display to display the Jekejeke Prolog interpreter can generate a call chain at any time. This is for example used in the exception handling to generate a meaningful exception context. The call chain is available independent of some debugging mode. We use the following Prolog text as our first example to explain the stack frame:

```
?- set_prolog_flag(sys_stack_frame, on).
?- [user].
p :- q.
p.

q :- abc.
q.
^D
```

When the predicate p is invoked the Prolog interpreter will in turn invoke the predicate q, and subsequently the predicate r. There it will run into an undefined predicate exception. The exception will display the full call chain:

```
?- p.
Error: Undefined predicate abc / 0.
      abc / 0
      q / 0
      p / 0
```

The old goal skeleton and display is needed in a choice point to try further unifications. This gives us the idea that when there is no choice point that the old goal skeleton and display is not needed anymore. Unfortunately we cannot drop the old goal skeleton and display in all cases, since it serves us as the continuation. But there is one case where a drop of the old goal skeleton and display is feasible. We can drop it in case that the goal list succeeding the old goal is in itself an end body. We can then overwrite the continuation by the parent of the continuation, an early execution of the old end body.

The overwriting of the continuation by the parent continuation will shorten the call chain. That the goal list succeeding the old goal is an end body indicates that the old goal was a last call. Thus the call chain will be shortened when two conditions meet. An old last call and a new omitted choice point. We can make the two conditions visible by the following example.

```
p :- q.
p.

q :- abc.
```

The invocation of the predicate q is a last call. The predicate q itself will omit the choice point since there is no additional rule. Therefore when the optimization is in place, we should observe that the predicate q is not anymore part of the call chain:

```
?- p.
Error: Undefined predicate abc / 0.
      abc / 0
      p / 0
```

In the above example both conditions have met. In the first example the condition of an omitted choice point was not met. We will now produce an example where the condition of a last call is not met. The example is as follows:

```
p :- q, h.
p.

q :- abc.
```

The invocation of the predicate q is not a last call anymore since it is followed by call of the predicate h. But the predicate q itself will still omit the choice point. Nevertheless the optimization cannot take effect anymore and we will thus see the full call chain:

```
?- p.
Error: Undefined predicate abc / 0.
      abc / 0
      q / 0
      p / 0
```

The last call condition will be detected at runtime from the intermediate form. There is no special compile time instruction like “execute” in the intermediate form for the last call [4] of the parent frame. Instead we will simply detect that the goal call is followed by a continuation call. The goal call needs not to be a callable term at compile time it can also be a naked variable so that we will only know about the called predicate at runtime. Therefore the optimization applies also from within meta-predicates that have goal arguments. Among the meta-predicates that work with this optimization are call/1, (,)/2, (;)/2 and (->)/2.

The last call condition on the parent frame does not change at runtime since it depends on the compile time intermediate form. On the other hand the condition that a choice point has been omitted is not stable. When a clause matches we might detected that we cannot omit the choice point. But still at runtime the choice point can be removed later on by a cut. We have implemented the stack frame elimination in such a way that the omission of the choice point is constantly monitored. This is expressed by the new instructions “last_goal” and “last_cont”. This is seen in the intermediate form as follows:

```
?- friendly(p/0).
p :-
      q,
      h.
   0 init_display
   1 last_goal q
   2 last_goal h
   3 last_cont
```

```
[...]
```

As can be seen the monitoring is done on all goals and on the continuation. We will see in the next section about head variable elimination that under this optimization fewer goals will be monitored. The monitoring of all goals is necessary since any of the goals could be cut transparent and issue a cut. So particularly checking whether a goal has the form of the cut (!) would be of no avail. Since other meta-predicates such as (,)/2, (;)/2 and (->)/2 can issue a cut depending on the followed solution path during their execution.

That the cut can also provoke the shortening of the call as chain can be seen by the following example. The example is very similar to our first example where the optimization did not apply. The optimization did not apply since q had a second rule, and thus the choice point was not omitted. In the new example the cut will remove the choice point later on:

```
p :- q.
p.

q :- catch(abc, X, sys_print_exception(X)), !, abc.
q.
```

We use a catch/3 and sys_print_exception/1 to check the call chain of the first call of abc/0 before continuing with the cut and the second abc/0. The first call of abc/0 will show the full call chain. Whereas the second call of abc/0 will only show a shortened call chain:

```
?- p.
Error: Undefined predicate abc / 0.
      abc / 0
      catch / 3
      q / 0
      p / 0
Error: Undefined predicate abc / 0.
      abc / 0
      p / 0
```

It should be noted that the optimization has also a positive effect on body variable elimination. Since we will simulate the end body of the last call, we will also invoke the "dispose_bind" instructions preceding the end body. As a result we will eliminate more body variables early on.

4.5 Head Variable Elimination

The Jekejeke Prolog system is capable of eliminating the need for some head variables. We will describe in more detail the used technique. Compared to the other optimizations, this optimization does not have the greatest gain in average. Nevertheless we have implemented it. Since it is a compile time optimization no bad cases are to be expected. The optimization technique introduces an improved unification instruction. Normally during head unification each goal argument is unified with the corresponding clause skeleton. The new unification instruction allows unification between goal arguments.

To detect the situation when the improved unification instruction is applicable we had to broaden our variable analysis. Based on the variable analysis for body variable elimination we added additional attributes to a variable:

- **Min Goal:** The value -1 if the variable occurs in the head, otherwise the index of the first goal where the variable occurs.

- **Structure Flag:** True if the variable occurs in some head position and this head position does not equal the variable, otherwise false.

- **Min Argument:** The index of the first head position which equals the variable.

- **Min Body:** The index of the first goal where the variable occurs. Available for temporary variables but not for extra variables.

The min goal field can be used whether a variable occurs in the head and is thus a head variable. If the value of the min goal field is -1 it is a head variable. The structure flag can be used to determine whether a head variable needs special treatment. If the structure flag is false then we do not perform the ordinary unification with the variable during head matching. We call such a variable a temporary variable.

The min argument field is used when a temporary variable occurs multiple times in a head in argument position. We then place a combined unification instruction to directly perform unification between the corresponding argument positions in the calling goal. This saves ordinary unification since the combined unification covers two ordinary unifications:

Ordinary Unifications	Combined Unification
unify_term _1, X unify_term _2, X	unify_term _1, _2

The min body field is then used to determine when the place holder needs to be unified. We place corresponding specialized unification instruction “unify_var” before the goal indicated by the min body position. This instruction is aware that the second argument is a variable and takes a slightly faster path for unification:

Ordinary Unification	Specialized Unification
unify_term _1, X	unify_var _1, X

Temporary variables that don’t have an occurrence in the body are called extra variables. These variables will never see a corresponding specialized unification instruction “unify_var” instruction in the body. Therefore we arrange that these variables are allocated at the end of the display. The clause is then invoked with a shortened display without these variables.

The maximum of the min body field has a further purpose. It indicates when stack frame elimination monitoring can be started. Before the maximum of this field there will be still specialized unification “unify_var” instructions for temporary variables. An early eliminating would prevent us from unifying the temporary variable with the corresponding argument in the calling goal since the calling goal is stored in the parent stack frame.

The effect of this optimization can be observed by listing the intermediate form. We will start with the head variable optimization switched off:

```
?- set_prolog_flag(sys_head_variable, off).
```

We will explain the effect by means of the following Prolog text. It is the Prolog code for the naïve reverse implementation:

```
rev([],[]).
rev([X|Rest],Ans) :- rev(Rest,L), concatenate(L,[X],Ans).

concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

We can now list the intermediate form of the concatenate/3 and the rev/2 predicate via the built-in friendly/1. As can be seen the enhanced unification instruction “unify_term” with two goal arguments is not yet used. Only the ordinary unification instruction “unify_term” with one parameter argument and one skeleton argument is used. Also no specialized unification “unify_var” for non-extra temporary variables is not yet seen:

```
?- friendly(concatenate/3).
concatenate([], L, L).
   0 unify_atomic _0, []
   1 new_bind L
   2 unify_term _1, L
   3 unify_term _2, L
   4 init_display
   5 dispose_bind L
   6 last_cont

[...]
?- friendly(rev/2).
[...]

rev([X | Rest], Ans) :-
      rev(Rest, L),
      concatenate(L, [X], Ans).
   0 new_bind X, Rest
   1 unify_compound _0, [X | Rest]
   2 new_bind Ans
   3 unify_term _1, Ans
   4 init_display
   5 new_bind L
   6 last_goal rev(Rest, L)
   7 dispose_bind Rest
   8 last_goal concatenate(L, [X], Ans)
   9 dispose_bind X, Ans, L
  10 last_cont
```

The intermediate form also contains the results of the body variable elimination and the stack frame elimination from the previous sections. For the body variable elimination this means the intermediate code is scattered with “dispose_bind” instructions. For the stack frame elimination this means that “last_goal” and “last_cont” instructions are seen.

The predicate concatenate/3 works as expected:

```
?- concatenate(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3]
?- concatenate([1],[2,3],X).
X = [1, 2, 3]
```

We will now switch on the head variable elimination optimization. This is done as follows:

```
?- set_prolog_flag(sys_head_variable, on).
```

We can now re-consult the Prolog text for the naïve reverse implementation. When generating the intermediate form during consult, the Prolog interpreter will now apply the head variable elimination optimization. We can again list the intermediate form of the concatenate/3 predicate and the predicate rev/2 via the built-in friendly/1:

```
?- friendly(concatenate/3).
concatenate([], L, L).
   0 unify_atomic _0, []
   1 unify_term _1, _2
   2 init_display
   3 last_cont

[...]
?- friendly(rev/2).
[...]

rev([X | Rest], Ans) :-
      rev(Rest, L),
      concatenate(L, [X], Ans).
   0 new_bind X, Rest
   1 unify_compound _0, [X | Rest]
   2 init_display
   3 new_bind L
   4 call_goal rev(Rest, L)
   5 dispose_bind Rest
   6 new_bind Ans
   7 unify_var _1, Ans
   8 last_goal concatenate(L, [X], Ans)
   9 dispose_bind X, L, Ans
  10 last_cont
```

In the first rule of the concatenate/3 predicate we see that the variable “L” has been judged as an extra variable. Therefore the two ordinary unification instructions in line 3 and 4 were replaced by one enhanced unification instruction in line 2. Subsequently the “new_bind” instruction for the variable “L” was not anymore needed.

In the second rule we see that the variable “Ans” has been judged as a non-extra temporary variable. Therefore the ordinary unification in line 5 was replaced by a specialized unification in line 8. The “new_bind” instruction for the variable “Ans” was moved as well.

Further the call of the rev/2 goal does not do stack frame monitoring anymore.

The predicate concatenate/3 still works as expected:

```
?- concatenate(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3]
?- concatenate([1],[2,3],X).
X = [1, 2, 3]
```

Prior to release 0.9.0 there was only the concept of an extra variable. The concept of a temporary variable has then been introduced. The movement of the “unify_term” instruction still put penalty for non-deterministic predicates since the instruction has to be repeated during backtracking. The reason is that the variable binding caused by this instruction is logged in the trail and thus automatically undone during backtracking.

On the other hand the optimization seems to be beneficial to deterministic predicates. Especially to programming patterns of the following form:

```
     P(X,Y) :- G(X), B(X,Y).
```

If G is a guard that might or might not succeed execution without temporary variables turns out costly. The variable Y would always be unified, even when the guard fails. With temporary variables the variable is only unified when we pass the guard.

5 Strategies Comparison

We conducted a couple of internal performance tests. The main indicator that was measured was the elapsed time for various test cases. The test is documented along the following lines:

- **Test Results:** In this section we will present the figures that were obtained from our test runs. Figures for various versions of the Jekejeke Prolog system will be given.
- **Discussion Choice Point Elimination:** In this section we will provide a critical discussion of the impact of the choice point elimination optimization.
- **Discussion Clause Indexing:** In this section we will provide a critical discussion of the impact of the clause indexing optimization.
- **Discussion Body Variable Elimination:** In this section we will provide a critical discussion of the impact of the body variable elimination optimization.
- **Discussion Stack Frame Elimination:** In this section we will provide a critical discussion of the impact of the stack frame elimination optimization.
- **Discussion Head Variable Elimination:** In this section we will provide a critical discussion of the impact of the head variable elimination optimization.

5.1 Test Results

In this section we will present the figures that were obtained from our test runs. Figures for various versions of the Jekejeke Prolog system will be given. We were running our test harness on the following machine:

Operating System: Windows 7 Professional
Processor: Intel Core i7-2620M @ 2.70GHz
Memory: 4.00 GB
Energy Settings: HP Optimized

We used the following Jekejeke Prolog system version. The test harness was started from the runtime library via the Java Class Harness. We did not use the development environment to run the test harness:

Java Version: JDK 1.8.0_25-b18 (64-bit)
VM Parameters: -mx512m -Duser.language=en
Library Version: Jekejeke Prolog Runtime Library 1.0.4
Concurrently Idle Applications: Word, Excel, Tomcat, SQL Server

We were then comparing the Jekejeke Prolog system under various optimization settings. In particular we were running our test harness for the following different settings, gradually increasing the number of optimizations:

Table 2: Compared Optimization Settings

Setting	Choice Point	Clause Indexing	Body Variable	Stack Frame	Head Variable
1) None	off	off	off	off	off
2) Discussion Choice Point	on	off	off	off	off
3) Discussion Clause Indexing	on	on	on	off	off
4) Discussion Body Variable	on	on	on	off	off
5) Discussion Stack Frame	on	on	on	on	off
6) Discussion Head Variable	on	on	on	on	on

We were running the test suite two times in a row, and were only measuring the second run. The first run would be a cold start with incomplete caching of the predicate references. The second run is a warm start which we wanted to measure.

Measuring warm start means also that we will not see some costs incurred by class loading or by un-optimized Java code. The Java runtime had enough time to load classes and to apply just in time compilation and as well adaptive optimizations.

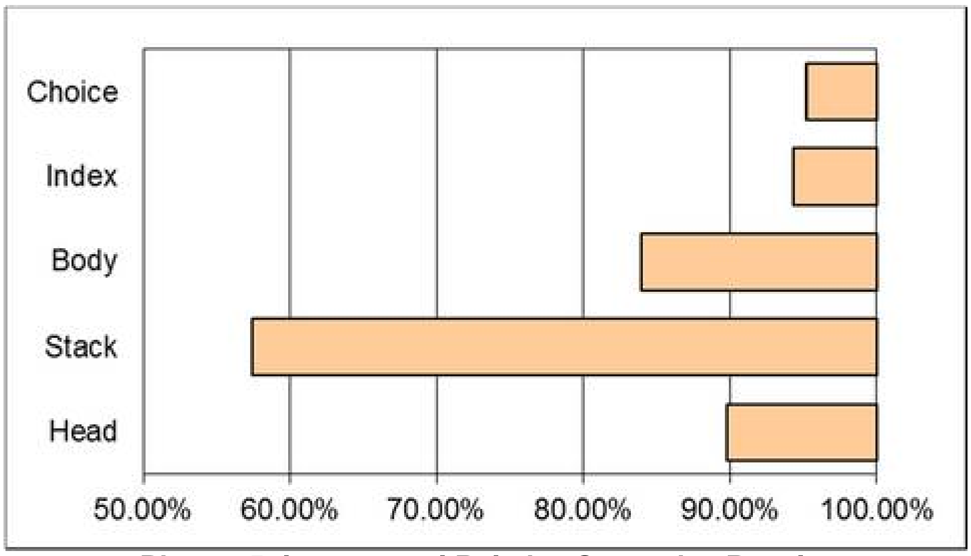
On the other hand the memory might already be fragmented. To avoid seeing a severe degradation because of constant garbage collection we use a memory allocation that is larger than the default of the Java virtual machine.

The absolute raw results measured in milliseconds are displayed in the table below:

Table 3: Absolute Detailed Strategies Results (ms)

Test	1)	2)	3)	4)	5)	6)
nrev	995	989	942	911	748	720
crypt	667	667	621	643	620	624
deriv	6'634	5'506	5'924	4'107	509	391
poly	662	671	594	556	554	502
qsort	1'092	1'095	1'065	1'046	870	779
tictac	951	930	968	985	994	969
queens	823	812	806	796	784	739
query	2'276	2'339	1'039	1'001	968	985
mtak	1'006	1'012	1'035	836	815	676
perfect	1'271	1'159	1'205	1'238	740	527
calc	2'659	2'948	2'904	2'237	641	492
Total	19'036	18'128	17'103	14'356	8'243	7'404

The picture below shows the total results relative to their previous discussion:

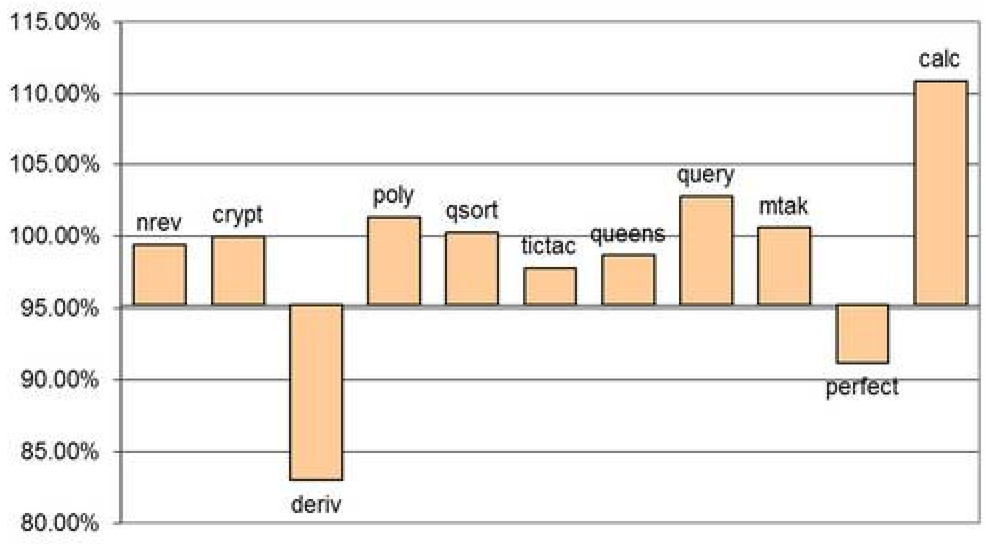


Picture 5: Incremental Relative Strategies Results

We did not subtract the execution time for a dummy test from the results. Our measuring of a dummy test revealed that its execution time is in the range of 2 milliseconds. Therefore all our test results also measure the slight overhead of the test harness itself.

5.2 Discussion Choice Point Elimination

In this section we will provide a critical discussion of the impact of the choice point elimination optimization. The choice point elimination already causes a speed-up in the average. We observe an average speed-up factor of 95.2%. The choice point elimination is the worst optimization. There are other optimizations down the pipeline which show a better speed-up. The best speed up factor of around 83.0% was observed for the test program deriv. The worst slow-down factor of around 110.9% was observed for the test program calc.



Picture 6: Choice Point Elimination Impact

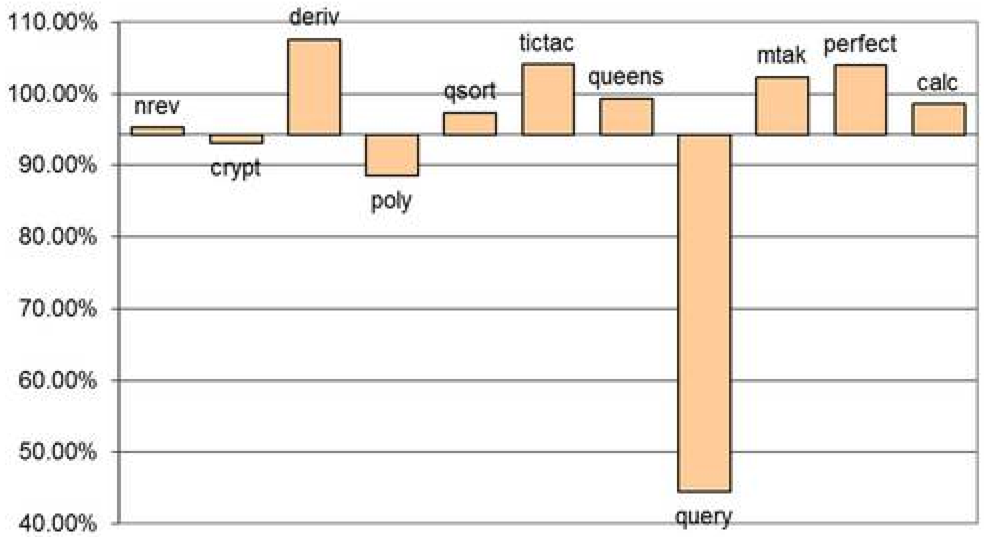
Bare bone choice point elimination has two advantages. First a list of clauses will not need a choice point for the last clause so that the corresponding space and time can be saved. Second a cut itself does not create a choice point anymore so that again space and time can be saved. If we sum up these two advantages then all programs should see some speed up, but this cannot be verified in the present case. We speculate that there are a couple of reasons for this phenomenon.

One reason is that non-deterministic programs might not enough profit from choice point elimination. Backtracking has the same effect since it will also force a predicate to remove choice points. Therefore the early elimination of choice points does not give any advantage since the choice points are anyway not long living and quickly returned to the Java memory management by the garbage collection. The choice point elimination only incurs an overhead here by the additional checks needed.

It should be noted that choice point elimination is an enabler for a couple of subsequent optimizations. When we switch off these optimizations, some code branches will be nevertheless executed when choice point elimination is on. There is a problem of ordering additional checks. What we currently do in the current implementation we always first check for determinism and then go through all the enabled subsequent optimizations. Choice point elimination causes the determinism check to succeed more often. If no subsequent optimization is enabled, then only overhead is incurred.

5.3 Discussion Clause Indexing

In this section we will provide a critical discussion of the impact of the clause indexing optimization. The average speed up factor is around 94.4%. The best speed up factor is seen for the test program query with 44.4%. We also find test programs with a slowdown. The test program deriv has the worst slowdown factor of 107.6%. The average speed up factor is not overwhelming. Nevertheless it should be noted that this optimization is also important, since it improves the choice point elimination and therefore also the body variable elimination and the stack frame elimination.



Picture 7: Clause Indexing Impact

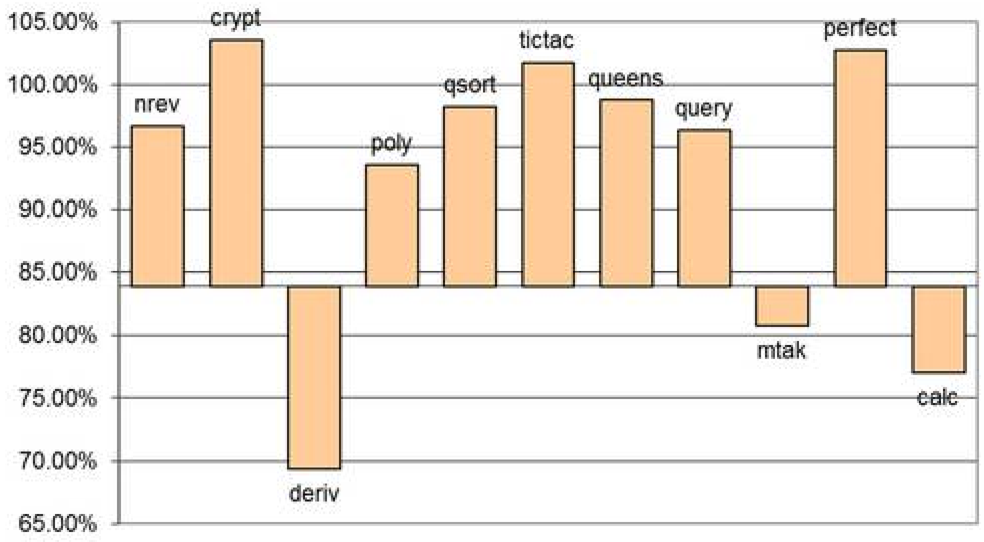
When the clause indexing picks a non-variable argument it performs a lookup in a hash table. The effort for this lookup can be assumed as a constant effort. The hash table will then return a smaller clause set compared to the full clause set. The smaller this clause set the less effort is needed by the Prolog system finding a matching clause. The clause sets are usually smaller when the hash table has more different keys. As such we can understand our benchmarking results and the good result for query. The predicate area/2 of the query test program does show a great number of different keys.

The other test programs then have hash table keys to varying degree. List processing predicates typically profit from hash table keys for '[]' and '.'. These keys help making the necessary case decisions faster since lengthy multiple failed unifications are bypassed. We find here the test programs nrev. The test programs poly and crypt will even make use of clause indexes that span multiple arguments. In the latter case it is not that list processing keys are important, rather the hash table key 'poly' is of importance.

But we also find the situation of clauses where an argument position consists of a variable. These clauses are included in all the clause sets of all the hash table keys since a calling goal might match such a clause. This enlarges the size of the clause set and lowers the impact of clause indexing. If the argument position leads to a trivial hash table the argument position is skipped and a better argument position is searched. If no better argument position is found then only overhead is generated.

5.4 Discussion Body Variable Elimination

In this section we will provide a critical discussion of the impact of the body variable elimination optimization. The average speed up factor is around 83.9%. The worst slow-down factor is around 103.5% for the test program crypt. The result is closely followed by the test program perfect. The best speed up factor is around 69.3% for the test program deriv. The result is closely followed by the test program calc.



Picture 8: Body Variable Elimination Impact

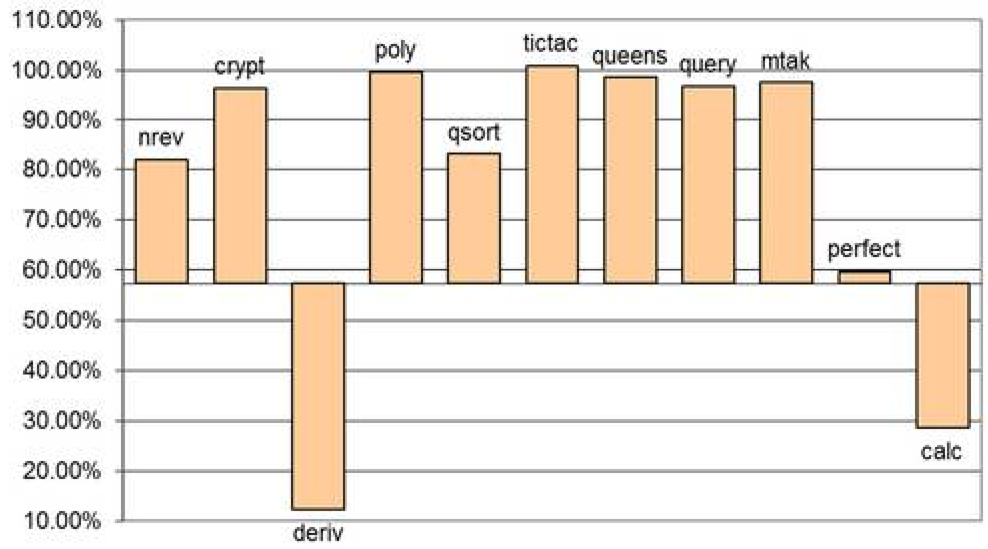
The body variable elimination optimization introduces a new instruction. The new instruction that is added to the intermediate form is “dispose_bind”. This instruction has only an optimizing effect for deterministic predicates. In case of non-deterministic predicates the instructions are invoked over and over again without any further effect. As consequence body variable elimination largely causes only overhead for non-deterministic predicates which can be seen in the lesser improvement for this group of test programs.

When a clause variable is part of a clause structure which is matched with a non-ground argument it might get referenced and thus cannot be disposed. Since we do not do display shrinking, this also prevents reclamation of displays. The last argument of the predicate concatenate/3 in the test program nrev is for example is structure resulting in this sense. We guess that these structures resulting arguments cause a considerable overhead which is seen in the timing diagram. It even affects non-deterministic test programs such as crypt when they make use of deterministic predicates with structure resulting arguments.

On the other hand the body variable elimination will see to it that non-structure resulting arguments are quickly released. Especially intermediate results that are atomic are detected by the body variable elimination. This means that for example intermediate number results from evaluations are quickly made available to the garbage collection of the Java memory management. Last but not least the destruction structure input arguments will need some place holders. These place holders will also be reclaimed as soon as the destructed components are passed to further predicates and the place holders are not anymore needed.

5.5 Discussion Stack Frame Elimination

In this section we will provide a critical discussion of the impact of the stack frame elimination optimization. The average speed up factor is around 57.4%. The best speed up factor is around 12.4% for deriv. deriv and to some extend calc are the only test programs that excel over the average. A couple of other test programs such as nrev, qsort and perfect saw also some significant speed up. The later test programs heavily making use of some typical tail recursive programming patterns. The worst slow-down was for tictac with 100.9%.



Picture 9: Stack Frame Elimination Impact

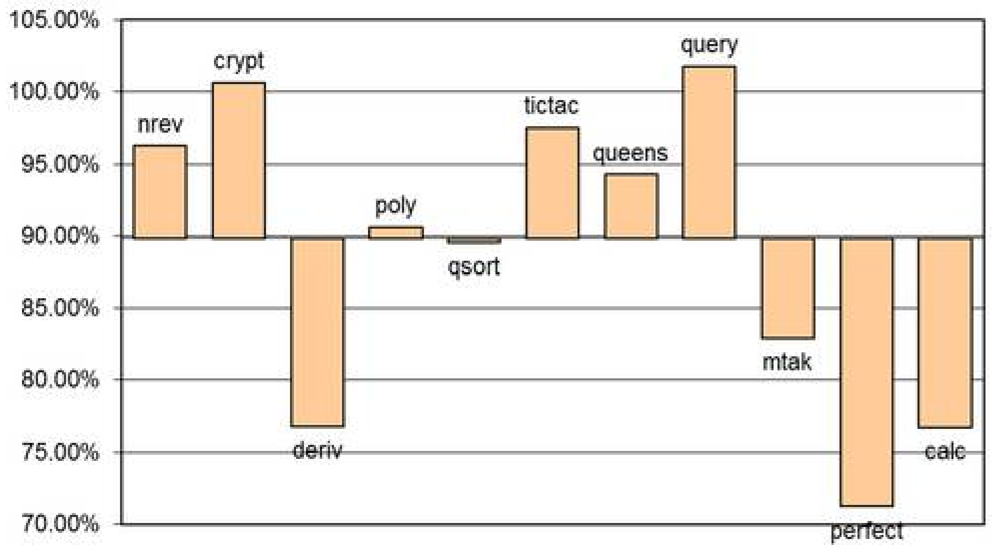
We can more or less identify a clear trend between the groups of deterministic and non-deterministic test programs. Test programs from the non-deterministic group show practically no impact from stack frame elimination. The reason is simple: Stack frame elimination cannot be applied to a non-deterministic clause since the parent stack frame with the call-site goal is needed during backtracking. A positive impact might be seen when the test program combines deterministic and non-deterministic predicates. This is for example the case for the benchmark program query, but the impact is not severe.

But it is not enough to detect the situation that the stack frame can be eliminated and then to shorten the call chain. We must also see to it that the stack frame is not anymore referenced by some variable place holders. For this purpose our stack frame elimination also invokes the disposal of the remaining variables of the calling clause. This disposal corresponds to the body variable elimination after the last goal of the calling clause, but it is done before the last call and not only when the last call succeeds. The stack frame elimination is therefore an additional enabler for body variable elimination.

Unfortunately the body variable elimination before the last call is also subject to the problems already identified in the previous section. Namely currently the display itself cannot be reclaimed as long as some variable is still in use. This is for example the case when there are structure result arguments. What has become known as tail recursion optimization can be viewed as a combination of our stack frame elimination and our body variable elimination. Tail recursion optimization is often found in Prolog systems since it is beneficial to perpetual processes [8]. But our test programs cannot be viewed as testing perpetual processes since we execute them repeatedly via backtracking.

5.6 Discussion Head Variable Elimination

In this section we will provide a critical discussion of the impact of the head variable elimination optimization. The head variable elimination might reduce the number of instructions in the case of temporary variables. Also the head variable elimination shortens the size of the display in the case of extra variables. The average speed up factor is around 89.8%. We observed the worst speed-up of 101.8% for the test program query. In the best case we observed a speed up of 71.2% for the test program perfect.



Picture 10: Head Variable Elimination Impact

The benefits of the head variable elimination are quickly eaten up by non-extra temporary variables. These variables are unified over and over again during backtracking which would explain the performance penalty for the test program query. On the other hand when such variables occur in the context of guards there is a clear advantage. This would explain the positive impact on the DCG calculator since the factor testing works in the form of a digit guard and eventually also the positive impact on perfect since the between1/3 predicate makes use of a >/2 guard.

Further there is a geometric argument which can explain some of the positive examples. The current head variable elimination most often only applies to facts. Facts form the leaves of a proof tree. The number of leaves depends on the shape of the proof tree. The lowest number of leaves is found in a linear proof such as found in the concatenate/3 predicate. The number of leaves of order O(1) relative to the total number of proof steps n. More leaves are found in a quadratic proof such as rev/2. It is of order O($n^{0.5}$). Finally a tree shaped proof such as sort/3 has the highest number of leaves. It is close to the order of O(n/2).

The head variable elimination is redundant to the body variable elimination. The effect of sparing a variable place holder allocation is already archived by the body variable elimination. The body variable elimination would first create and then later dispose the variables after the unification. Nevertheless the head variable elimination has the advantage of bypassing these runtime steps and reducing the code already at compile time.

6 Interpreter Comparison

We conducted a couple of external performance tests. The main indicator that was measured was the elapsed time for various test cases. The test is documented along the following lines:

- **Test Results:** In this section we will present the figures that were obtained from our test runs. Figures for various Prolog systems will be given.

- **Discussion ECLiPSe Prolog:** In this section we will provide a critical discussion of the comparison with the ECLiPSe Prolog system.

- **Discussion SWI Prolog:** In this section we will provide a critical discussion of the comparison with the SWI Prolog system.

- **Discussion GNU Prolog:** In this section we will provide a critical discussion of the comparison with the GNU Prolog system.

- **Discussion Ciao Prolog:** In this section we will provide a critical discussion of the comparison with the Ciao Prolog system.

- **Discussion B-Prolog:** In this section we will provide a critical discussion of the comparison with the B-Prolog system.

6.1 Test Results

In this section we will present the figures that were obtained from our test runs. Figures for various Prolog systems will be given. We were using the same machine as for our internal tests. And we were also using the same Jekejeke Prolog system as for our internal tests. The Jekejeke Prolog system was run in default mode, which means that all its optimizations are switched on.

We were comparing the Jekejeke Prolog system to other available Prolog systems. We compared it with 4 freely available Prolog systems and one commercially available Prolog system. In particular we compared the following Prolog systems:

- **ECLiPSe Prolog:** Version 6.1 #181 (x86_64_nt)
- **SWI Prolog:** SWI-Prolog (Multi-threaded, 64 bits, Version 7.1.11)
- **GNU Prolog:** GNU Prolog 1.4.4 (64 bits)
- **Ciao Prolog:** Ciao 1.15-1781 (interpreted mode)
- **B-Prolog:** B-Prolog Version 8.1 (interpreted mode)

We were running the test suite again two times in a row, and were only measuring the second run. The first run would be a cold start with incomplete caching of the predicate references. The second run is a warm start which we wanted to measure.

For the Jekejeke Prolog system we invoked the Harness Java Class. For the other Prolog systems we opened their console and manually started the test harness. B-Prolog and Ciao Prolog did provide a DOS console whereas the other Prolog systems had a custom console.

The manual start included first consulting the Prolog system specific test harness file and then invoking the test suite predicate. The test harness file for a Prolog system includes its predicate for measuring the elapsed time.

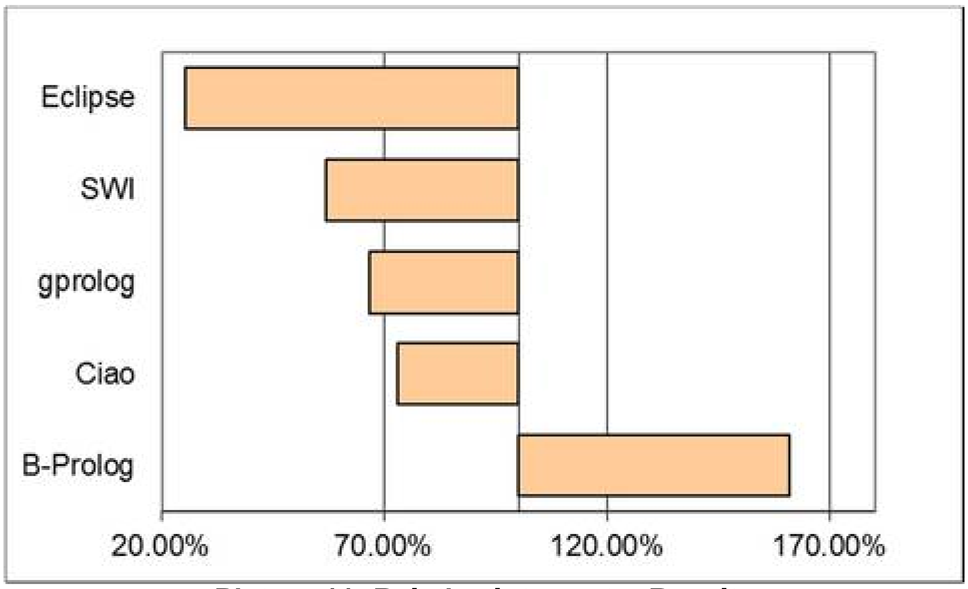
The test harness file for a Prolog system also includes the statements to read the common file and the test programs. Not in all Prolog systems this was possible via the consult/1 predicate. For the GNU Prolog system we had to refer to the include/1 predicate.

The absolute raw results measured in milliseconds are displayed in the table below:

Table 4: Absolute Detailed Interpreter Results (ms)

Test	ECLiPSe	SWI	gprolog	Ciao	B-Prolog	Jekejeke
nrev	59	156	269	405	655	720
crypt	94	515	483	141	905	624
deriv	105	249	312	1'841	842	391
poly	94	297	312	109	780	502
qsort	94	296	375	437	1'030	779
tictac	171	375	670	312	1'076	969
queens	94	405	530	124	1'186	739
query	265	967	797	468	1'919	985
mtak	109	281	405	94	1'435	676
perfect	658	328	359	234	967	527
calc	140	343	422	1'245	1'108	492
Total	1'883	4'212	4'934	5'410	11'903	7'404

The picture below shows the total results relative to the Jekejeke Prolog system:



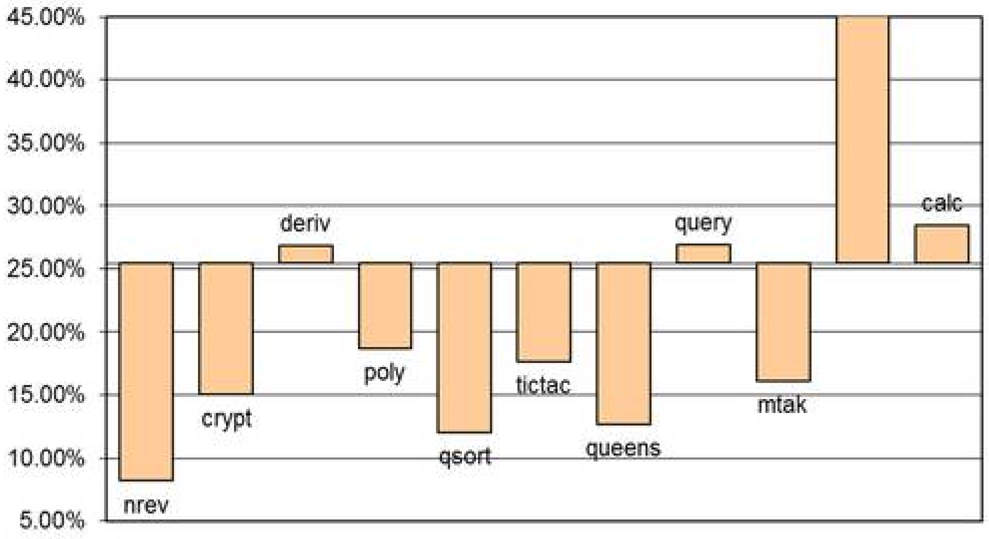
Picture 11: Relative Interpreter Results

Some of the tested Prolog systems do not distinguish between compiled and interpreter mode. We find SWI-Prolog and Jekejeke Prolog in this category. Some of the tested Prolog systems are declared as compilers. We find ECLiPSe Prolog and GNU Prolog in this category. For these systems there was not much choice in how to execute the benchmarks.

Some of the tested Prolog systems provide both compiled and interpreted mode. We find B-Prolog and Ciao Prolog in this category. For these systems we choose the interpreted mode to run our benchmarks. As the bar chart shows, there is no clear divide between compiled and interpreted. But the fastest Prolog system here is a compiled one.

6.2 Discussion ECLiPSe Prolog

In this section we will provide a critical discussion of the comparison with the ECLiPSe Prolog system. The ECLiPSe Prolog system is the only compiler that we have included in our external benchmark. The average speed up factor compared to Jekejeke Prolog is around 25.4%. The worst slow down factor was around 124.9% for perfect. The best speed up factor is around 8.2% for nrev.



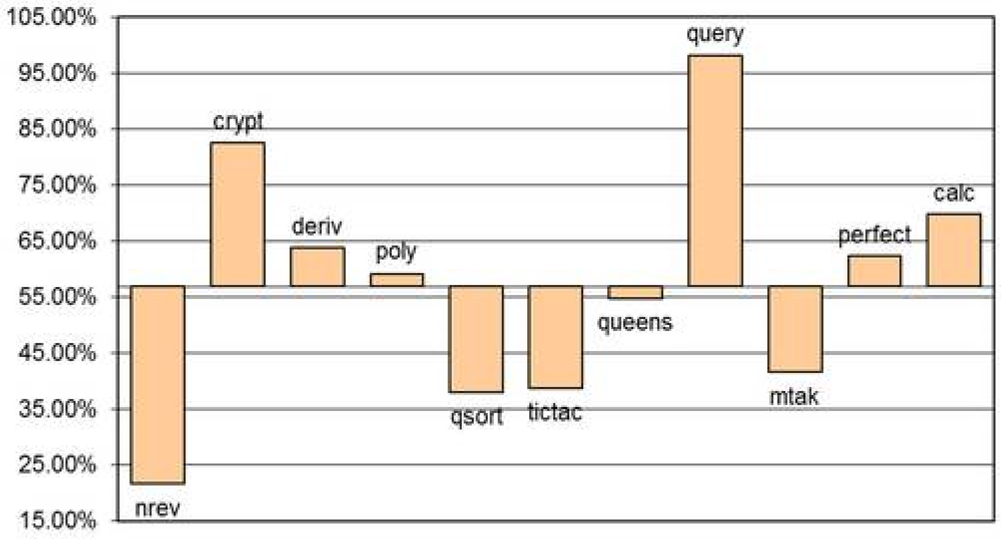
Picture 12: ECLiPSe Prolog Performance

History ECRC, Sepia, Cisco
WAM Code, ECLiPSe Object Code
C, Prolog

Multi Argument Indexing? Crypt, Query
Inline Choice Points? Tictac

6.3 Discussion SWI Prolog

In this section we will provide a critical discussion of the comparison with the SWI Prolog system. The SWI Prolog system is the fastest non-compiler that we have included in our external benchmark. The average speed up factor compared to Jekejeke Prolog is around 56.9%. The worst slow down factor was around 98.2% for query. The best speed up factor is around 21.7% for nrev.



Picture 13: SWI Prolog Performance

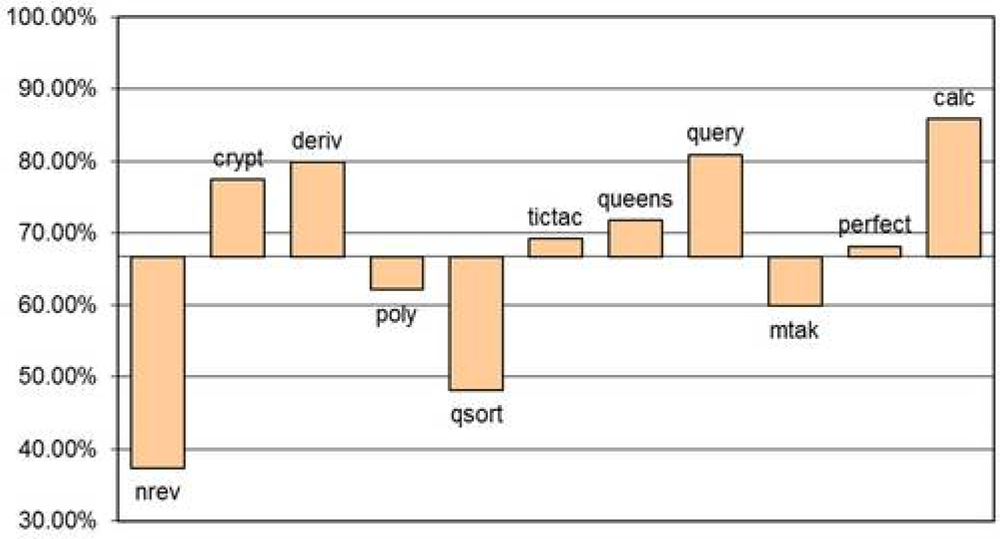
The SWI Prolog system is not the only system that cannot apply an optimization to query. All of the tested Prolog systems seem to have problems with this test program. This is very astonishing since it is based on a predicate lookup which is very well amenable to first argument indexing. A further test program that might profit from indexing is crypt. For this test program almost all Prolog systems show also a problem. Only the B-Prolog system and the Ciao Prolog do not suffer extremely in this case. The Jekejeke Prolog system on the other hand will use a multi-argument index for the predicate sum2/4 in crypt.

The separation of the speed ups into the two groups deterministic and non-deterministic is perfect for SWI Prolog. Unfortunately the calc example does not show some special speed up over the average. SWI Prolog does perform a DCG expansion similar to GNU Prolog. But SWI Prolog places additional unification steps subsequent to DCG actions into the body of a grammar rule to make it steadfast similar to the Jekejeke Prolog system. This could affect the performance of SWI Prolog for the calc test program.

The SWI Prolog system speed-up shows a modest span of 86.6%. A smaller span is only found for the GNU Prolog system with 58.7%. The biggest span is found for the SICStus Prolog system with 378.2%. The variance of the system speed-up shows a similar picture, means that we do have similar clustering on both sides of the speed spectrum for all Prolog systems. The SWI Prolog system speed-up shows again a modest variance of 22.9%. A smaller variance is only found for the GNU Prolog system with 16.1%. The biggest variance is found for the SICStus Prolog system with 138.5%.

6.4 Discussion GNU Prolog

In this section we will provide a critical discussion of the comparison with the GNU Prolog system. The GNU Prolog ranks second in our comparison of interpreters. The average speed up factor relative to the Jekejeke Prolog system is around 66.6%. This is a little bit worse than the average speed up factor of SWI Prolog. The worst speed up factor is around 85.8% for query. The best speed up factor is around 37.4% for nrev.



Picture 14: GNU Prolog Performance

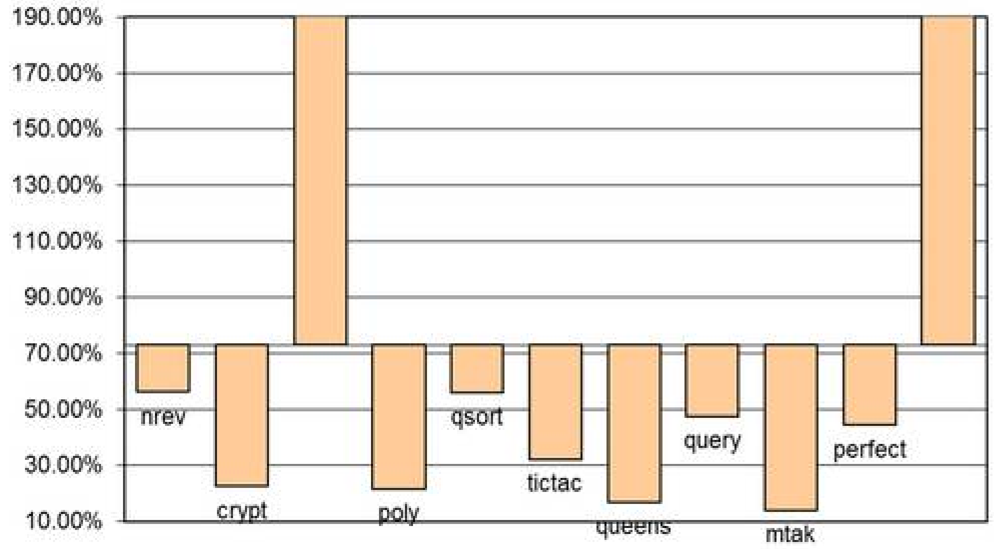
GNU Prolog and SWI Prolog are not only similar what concerns the average speed up factor. But the Prolog systems also share a similar pattern of speed ups among the test programs. We only see a considerable difference in the test programs tictac, queens and mtak. The SWI Prolog system shows a significantly better relative performance for these test programs. On the other hand the relative performance for the test program query is better.

Most of the test programs that are better than the average compared to Jekejeke Prolog are deterministic. Therefore we might conclude that GNU Prolog has a better handling of deterministic predicates than we do. In particular we find those test programs that have structure resulting arguments above the average. In our PLM based architecture the cost for structure resulting arguments is the variable instantiation and then a dispose instruction that executes to no avail. In a WAM based architecture the structure is simply newly allocated.

Concerning the DCG expansion the GNU Prolog system is able to merge terminals into the head like SWI-Prolog can do. But contrary to SWI Prolog it does not put additional unification steps inside the grammar rules for DCG actions. In theory this should give a relative speed advantage. But we do not find such a result, which is more astonishing since query performs better. We can only conclude that query performs better because of faster arithmetic and not because of faster indexing. Since the better indexing would be also seen in the merging of terminals into the head for the calc test program.

6.5 Discussion Ciao Prolog

In this section we will provide a critical discussion of the comparison with the Ciao Prolog system. The Prolog system shows an average slowdown very similar to the slowdown of SICStus Prolog. The average slowdown is around 73.1%. The best speed up is seen for the mtak test program with around 13.9%. The worst slow-down is seen for the deriv test program with around 470.8%.



Picture 15: Ciao Prolog Performance

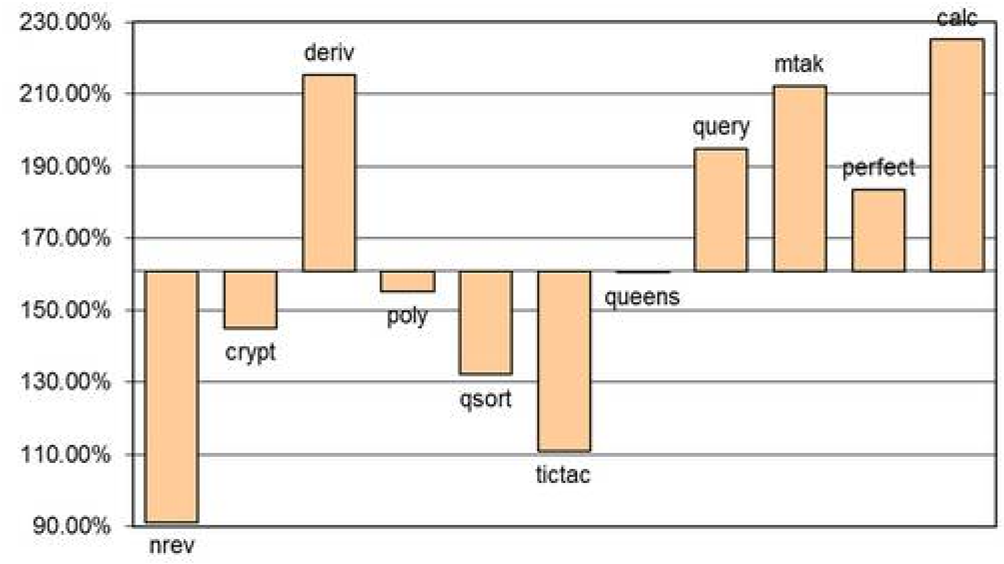
t.b.d.

The relatively good performance of tictac is a highlight. The test program demands efficient deep unification. This is an area that is not yet covered by any of our optimizations. On the downside we find the deriv test program. Here our own analysis shows that the deriv test program profits most from stack frame elimination. We might thus speculate that eventually the Ciao Prolog system might need some improvement in stack frame elimination.

The problem with calculate can be also due to issues with the DCG translation. The Ciao Prolog system uses the 'C'/3 construct to translate terminals. It is the only system besides our system that makes use of this construct. Our own analysis shows that this DCG translation can profit from head variable elimination, especially temporary variables that can speed up guards. This is consistent with the problem of the mtak test program which makes also use of a guard.

6.6 Discussion B-Prolog

In this section we will provide a critical discussion of the comparison with the B-Prolog system. The newest release of the B-Prolog system is still slower than the former release 7.5#8 of the B-Prolog system. The average slow-down factor is around 160.8%. The test program with the least slow down factor is nrev with around 91.0%. The test program with the worst slow down factor is calc with around 225.2%.



Picture 16: B-Prolog Performance

The pattern is much more balanced than the pattern for the other Prolog systems. Many of the test programs do not perform very differently from the average. What features could explain this behaviour? We cannot attribute this performance to jumbo instructions. Both B-Prolog [11] and SICStus Prolog [12] are known to do the merging and specialization of instructions. But we have the problem that this technology probably only applies to compiled code and not to interpreted code.

B-Prolog is based on TOAM which does eager environment allocation. The eager environment allocation leads to a degradation in the case of mtak [10]. Further the last call optimization in the TOAM is directly targeted towards reusing the eagerly created environments [11]. The optimization seems to work well when there are fewer goals between the head and the last call. Maybe this explains the better performance for crypt and qsort.

The performance of the calc test program should be much better. B-Prolog does not place additional unification steps subsequent to DCG actions into the body of a grammar rule to make it steadfast. Therefore it should not be subject to the corresponding performance penalty. But it does not integrate terminals into the head. Instead it handles these terminals via a list equation. This could give a small performance penalty again if list equation is not optimized away since it would prevent first argument indexing.

7 Appendix Harness Listings

The full source code of the Java classes and the Prolog texts for the test harness is given. The following source code has been included:

7.1 Common Files

The only common file consists of a Prolog text with the test suite and a Prolog text with utilities for the measurement of the elapsed time and of the garbage collection time.

For the common files there are the following sources:

- **suite.p:** The Prolog text that defines the test suite.
- **util.p:** The Prolog text for the testing utilities.

Prolog Text suite

```
/**
 * The Prolog text that defines the test suite.
 *
 * Copyright 2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0.3 (a fast and small prolog interpreter)
 */

:- ensure_loaded('util.p').

:- ensure_loaded('nrev.p').
:- ensure_loaded('crypt.p').
:- ensure_loaded('deriv.p').
:- ensure_loaded('poly.p').
:- ensure_loaded('qsort.p').
:- ensure_loaded('tictac.p').
:- ensure_loaded('queens.p').
:- ensure_loaded('query.p').
:- ensure_loaded('mtak.p').
:- ensure_loaded('perfect.p').
:- ensure_loaded('calc.p').

suite :-
   bench(3001, dummy, _, _),
   bench(6001, nrev, T1, G1),
   bench(301, crypt, T2, G2),
   bench(30001, deriv, T3, G3),
   bench(61, poly, T4, G4),
   bench(6001, qsort, T5, G5),
   bench(11, tictac, T6, G6),
   bench(16, queens, T7, G7),
   bench(3001, query, T8, G8),
   bench(31, mtak, T9, G9),
   bench(16, perfect, T10, G10),
   bench(20001, calc, T11, G11),
   T is T1+T2+T3+T4+T5+T6+T7+T8+T9+T10+T11,
   G is G1+G2+G3+G4+G5+G6+G7+G8+G9+G10+G11,
   write('Total'),
   show(T, G), nl.
```

Prolog Text util

```
/**
 * The Prolog text for the testing utilities.
 *
 * Copyright 2010-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

for(_).
for(N) :- N > 0, M is N - 1, for(M).

:- meta_predicate test(?,0).
test(N, X) :- for(N), call(X), fail.
test(_, _).

show(T, G) :-
   write('\tin '),
   write(T),
   write('\t('),
   write(G),
   write(' gc) ms'), nl.

:- meta_predicate bench(?,0,?,?).
bench(N, X, T, G) :-
   uptime(T1), gctime(G1),
   test(N,X),
   uptime(T2), gctime(G2),
   T is T2 - T1,
   G is G2 - G1,
   write(X),
   show(T,G).

dummy.
```

7.2 Jekejeke Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time.

For the Jekejeke Prolog harness there are the following sources:

- **jekejeke.p:** The Prolog text.

Prolog Text jekejeke

```
/**
 * Jekejeke Prolog code for the test harness.
 *
 * Copyright 2010-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

% ?- ensure_loaded('<base>/jekejeke.p').

uptime(X) :-
   statistics(uptime, X).

gctime(X) :-
   statistics(gctime, X).

:- ensure_loaded('suite.p').
```

7.3 ECLiPSe Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the ECLiPSe Prolog harness there are the following sources:

- **eclipse.p:** The Prolog text.

Prolog Text eclipse

```
/**
 * ECLiPSe Constraint Logic Programming System code for the test harness.
 *
 * Copyright 2011-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

% ?- ensure_loaded('<base>/eclipse.p').

uptime(X) :-
   statistics(times, [_,_,T]),
   X is round(T*1000).

gctime(X) :-
   statistics(gc_time, T),
   X is round(T*1000).

:- use_module(library(iso)).

:- ensure_loaded('suite.p').
```

7.4 SWI Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the SWI Prolog harness there are the following sources:

- **swi.p:** The Prolog text.

Prolog Text swi

```
/**
 * SWI Prolog code for the test harness.
 *
 * Copyright 2011-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

% ?- ensure_loaded('<base>\\swi.p').

uptime(X) :-
   statistics(cputime, T),
   X is round(T*1000).

gctime(T) :-
   statistics(garbage_collection, [_,_,T|_]).

:- set_prolog_flag(double_quotes, codes).

:- ensure_loaded('suite.p').
```

7.5 GNU Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime. Replace the <base> by the test program directory. Measurement of the garbage collection time is not possible.

For the GNU Prolog testing we need a special version of the suite.p Prolog text that uses include/1 instead of ensure_loaded/1 and we also need a special version of the util.p Prolog text that goes without meta_predcate/1 declarations.

For the GNU Prolog harness there are the following sources:

- **gprolog.p:** The Prolog text.
- **suiteinc.p:** The Prolog text of the modded suite.p.
- **utilinc.p:** The Prolog text of the modded util.p.

Prolog Text gprolog

```
/**
 * GNU Prolog code for the test harness.
 *
 * Copyright 2010-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.8.6 (a fast and small prolog interpreter)
 */

% ?- consult('<base>\\gprolog.p').

uptime(X) :-
   real_time(X).

gctime(0).

:- include('suiteinc.p').
```

Prolog Text suiteinc

```
/**
 * The Prolog text that defines the test suite.
 * Version that uses include instead of ensure_loaded.
 *
 * Copyright 2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0.3 (a fast and small prolog interpreter)
 */

:- include('utilinc.p').

:- include('nrev.p').
:- include('crypt.p').
:- include('deriv.p').
:- include('poly.p').
:- include('qsort.p').
:- include('tictac.p').
:- include('queens.p').
:- include('query.p').
:- include('mtak.p').
:- include('perfect.p').
:- include('calc.p').

suite :-
   bench(3001, dummy, _, _),
   bench(6001, nrev, T1, G1),
   bench(301, crypt, T2, G2),
   bench(30001, deriv, T3, G3),
   bench(61, poly, T4, G4),
   bench(6001, qsort, T5, G5),
   bench(11, tictac, T6, G6),
   bench(16, queens, T7, G7),
   bench(3001, query, T8, G8),
   bench(31, mtak, T9, G9),
   bench(16, perfect, T10, G10),
   bench(20001, calc, T11, G11),
   T is T1+T2+T3+T4+T5+T6+T7+T8+T9+T10+T11,
   G is G1+G2+G3+G4+G5+G6+G7+G8+G9+G10+G11,
   write('Total'),
   show(T, G), nl.
```

Prolog Text utilinc

```
/**
 * The Prolog text for the testing utilities.
 * Version that doesn't use meta_predicate declarations.
 *
 * Copyright 2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 1.0.3 (a fast and small prolog interpreter)
 */

for(_).
for(N) :- N > 1, M is N - 1, for(M).

test(N, X) :- for(N), call(X), fail.
test(_, _).

show(T, G) :-
   write('\tin '),
   write(T),
   write('\t('),
   write(G),
   write(' gc) ms'), nl.

bench(M, X, T, G) :-
   uptime(T1),
   gctime(G1),
   test(M, X),
   uptime(T2),
   gctime(G2),
   T is T2 - T1,
   G is G2 - G1,
   write(X),
   show(T, G).

dummy.
```

7.6 Ciao Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the Ciao Prolog harness there are the following sources:

- **ciao.p:** The Prolog text.

Prolog Text ciao

```
/**
 * Ciao code for the test harness.
 * Use ensure_loaded/1 on toplevel so that code will be interpreted.
 *
 * Copyright 2011-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

% :- ensure_loaded('<base>\\ciao.p').

uptime(Y) :-
   statistics(walltime, [X|_]),
   Y is round(X).

gctime(S) :-
   statistics(garbage_collection, [_,_,T]),
   S is round(T).

:- ensure_loaded('suite.p').
```

7.7 B-Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the B-Prolog harness there are the following sources:

- **bprolog.p:** The Prolog text.

Prolog Text bprolog

```
/**
 * B-Prolog code for the test harness.
 * Since consult/1 is used code will be interpreted.
 *
 * Copyright 2010-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.0 (a fast and small prolog interpreter)
 */

% ?- set_prolog_flag(redefine_builtin, on).
% ?- consult('<base>\\bprolog.p').

uptime(X) :-
   statistics(runtime, [X|_]).

gctime(X) :-
   statistics(gc_time, X).

ensure_loaded(X) :-
    atom_concat('<base>\\', X, Y),
    consult(Y).

:- ensure_loaded('suite.p').
```

Pictures

Tables

References

[1] Trigg, W. C. (1985): Mathematical Quickies, Dover Publications, Inc., New York, 1985

[2] Warren, D.H.D. (1983): Applied Logic – Its Use and Implementation as a Programming Tool, Technical Note 290, SRI International, 1983

[3] Haygood, R. (1989): A Prolog Benchmark Suite for Aquarius, Computer Science Division, University of California Berkley, April 30, 1989

[4] Warren, D.H.D. (1983): An Abstract Prolog Instruction Set, Technical Note 309, SRI International, October, 1983

[5] Aït-Kaci, H. (1991): Warren's Abstract Machine, A Tutorial Reconstruction, ICLP'91 Pre-Conference Tutorial, 1991

[6] Bruynooghe M., Janssens, G. and Kagedal, A. (1996): Live-structure Analysis for Logic Programming Languages with Declarations, Department of Computer Science, Leuven, May 6, 1996

[7] Hodas, J.S. (1990): From the PLM to the WAM and Beyond, University of Pennsylvania, Philadelphia, Fall 1990.

[8] Wielemaker, J. and Neumerkel, U. (2008): Precise Garbage Collection in Prolog, Proceedings of CICLOPS-08, Udine, Italy, 2008

[9] Zhou, N.-F. (2000): Garbage Collection in B-Prolog, In Proc. of the First Workshop on Memory Management in Logic Programming Implementations, 2000

[10] Demoen, B. and Nguyen, P.-L. (2008): Environment Reuse in the WAM, International Conference on Logic Programming, Udine, Italy, 2008

[11] Zhou, N.-F. (2007): A Register-free Abstract Prolog Machine with Jumbo Instructions, International Conference on Logic Programming, Porto, Portugal, 2007

[12] Nassen, H. (2000): Optimizing the SICStus Prolog Virtual Machine Instruction Set, Master's thesis, SICS Technical Report T2001-01

[13] Russell, S. and Norvig, P. (2010): Artificial Intelligence, A Modern Approach, Third Edition, Pearson Education, Inc.

[14] M. V. Hermenegildo et al. (2010): An Overview of Ciao and its Design Philosophy, University of Madrid, School of Computer Science