Łukasiewicz Problems: Experiments Using Program Analysis and Transformation

André de Waal

Forschungsbericht AIDA–95–05

Darmstadt,

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstraße 10
D–64283 Darmstadt
Germany

Łukasiewicz Problems: Experiments Using Program Analysis and Transformation

D.A. de Waal*
FG Intellektik, FB Informatik
Technische Hochschule Darmstadt
Darmstadt, Germany
andre@intellektik.informatik.th-darmstadt.de

Abstract

In this report we describe experiments performed on the Łukasiewicz problems with the aim of reducing the explosion of the search space experienced with even the simplest of these problems. In the experiments we apply various logic program analysis and transformation techniques to selected problems. The results of the experiments are reported and preliminary conclusions drawn about the relevance of the tried approaches.

1 Introduction

In this report we informally describe experiments performed on the Łukasiewicz problems with the aim of reducing the explosion of the search space experienced with even the simplest of these problems. Some of the ideas noted here are "speculative", to say the least. However, as most of the "standard" techniques tried do not deliver results, solving this problem clearly calls for an "alternative" approach.

The two Łukasiewicz axioms are given below (in the TPTP-Library [8] format):

$$
\begin{array}{l}
input\_clause(condensed\_detachment, axiom, \\
\quad [++th(Y), \\
\quad --th(imp(X,Y)), \\
\quad --th(X)]). \\
input\_clause(ic\_JLukasiewicz, axiom, \\
\quad [++th(imp(imp(imp(X,Y),Z),imp(imp(Z,X),imp(U,X))))]).
\end{array}
$$

with two examples of theorems:

$$
\begin{array}{l}
input\_clause(prove\_simple, theorem, \\
\quad [--th(imp(a,a))]). \\
input\_clause(prove\_ic\_5, theorem, \\
\quad [--th(imp(a,imp(imp(a,b),b)))]).
\end{array}
$$

*This author was supported by HCM Project: Compulog Group—Cooperation Group in Computational Logic under contract no. ERBCHBGCT930365.

Our aim is to do a **syntactic** analysis of the above problem. We therefore do not go into a detailed description or explanation of the problem. A short description can be found in the TPTP-Library and in the Proceedings of CADE-9 [6]. As this is a Horn-problem, we may therefore rewrite the problem as a logic program. Furthermore, to remove any meaning we may associate with the constants and function symbols appearing in the problem, we replace the binary functor $implies(X, Y)$ by a list and reverse the order of the arguments, e.g. $imp(X, Y)$ is replaced by $[Y, X]$. The transformed two axioms written as a logic program are given below:

$$
\begin{array}{l}
theorem([[[X, \_], [X, Z]], [Z, [\neg\, X]]]) \leftarrow \\
theorem(Y) \leftarrow \\
\qquad theorem([Y, X]), \\
\qquad theorem(X)
\end{array}
$$

with example theorem (goal):

$$\leftarrow theorem([a, a])$$

This transformed program is used in some of the experiments. It is well known and easily provable (using the techniques described in [1]) that reduction steps (ancestor resolution) is not needed in a model elimination derived proof procedure to prove any theorem of the form given previously. It should be noted that although the given program is in a form suitable for analysis purposes, it should not be run using a standard Prolog compiler as the lack of an occur check may create infinite terms and a search strategy other than depth-first search is needed to find a successful proof.

Various experiments were tried to try and infer some "pattern" needed to find proof or to identify redundant parts of the search space. The experiments are listed below in the order in which they were tried. The order does not indicate any preference by the author with respect to the tried experiments or any degree of "success" or "failure" of the experiments.

1. Compute a safe approximation using logic program analysis and transformation.
2. Compute a safe approximation using logic program analysis and transformation, but do a finer grained analysis as in 1.
3. Compute an under-approximation by restricting arguments to ground terms.
4. Use partial evaluation with a specialized unfolding rule to "crystallize" out some patterns in the proof.
5. Rewrite the problem in a constraint logic programming language to take advantage of the power of constraint solvers.
6. Solve a more general problem and collect constraints that have to be satisfied to solve instances of this problem.

Each experiment is explained (some in more detail than others) in the following sections. Some conclusions are drawn from these experiments in the final section.

We assume the reader is familiar with the basic concepts of logic programming, partial evaluation, abstract interpretation, constraint logic programming and automated theorem proving.

2 Compute a Safe Approximation Using Logic Program Analysis and Transformation

The idea with this experiment is to approximate the theorems provable given the two Łukasiewicz axioms. This is done as follows. First, we specialize a specification of some first-order theorem prover (we use model elimination), written as a logic program, with respect to the two Łukasiewicz axioms. This is done using partial evaluation. Second, we compute an approximation of the resulting program with respect to some formula (possibly a theorem) using some approximation tool. In this case a regular safe approximation is computed. The result of this process is analysis information about the behavior of the model elimination proof procedure given the two axioms and the formula ([1] contains detailed algorithms). Inspection of the results may indicate parts of the search space that need not be explored as it can never contribute to a successful proof or it may give some structure necessary for a proof.

First, we give a specification of model elimination written as a logic meta-program (see [1] for further explanations).

$\%$ *Let* T *be a first−order theory and* F *a literal. If* $T \models \exists F$ *then*
$\%$ *there exists a proof for* F *from the augmented theory* $T \wedge \neg F$,
$\%$ *that is* $\leftarrow$ *solve*$(F, [\,])$ *will succeed.*
$solve(G, A) \leftarrow$
    $depth\_bound(D)$,
    $prove(G, A, D)$

$prove(G, A, D) \leftarrow member(G, A)$
$prove(G, A, D) \leftarrow D1 > 1,\ D1\ is\ D - 1,$
    $neg(G, GN)$,
    $clause(G, B)$,
    $proveall(B, [GN|A], D1)$

$proveall([\,], A, D) \leftarrow$
$proveall([G|R], A, D) \leftarrow$
    $prove(G, A, D)$,
    $proveall(R, A, D)$

Note that for analysis purposes the depth bound may be ignored.

Partial evaluation of this meta-program with respect to the Łukasiewicz axioms (assuming we are going to restrict our formula F to the formula $imp(a, a)$) gives:

```
solve(++th(_6870),A) :- prove_1(++th(_6870),A).
solve(--th(_6870),A) :- prove_2(--th(_6870),A).

prove_1(++th(_6870),A) :- member_1(++th(_6870),A).
prove_2(--th(_6870),A) :- member_2(--th(_6870),A).

member_1(++th(_6870),[++th(_6870)|_]).
```

```
member_1(++th(_6870),[_|A]) :- member_1(++th(_6870),A).
member_2(--th(_6870),[--th(_6870)|_]).
member_2(--th(_6870),[_|A]) :- member_2(--th(_6870),A).

prove_1(++th(_6877),A) :-
     prove_1(++th(imp(_6885,_6877)),[--th(_6877)|A]),
     prove_1(++th(_6885),[--th(_6877)|A]).
prove_1(++th(imp(imp(imp(_6885,_6886),_6883),imp(imp(_6883,_6885),
                                                 imp(_6894,_6885)))),A) :-
     true.
prove_2(--th(imp(a,a)),A) :-
     true.
prove_2(--th(imp(_6887,_6879)),A) :-
     prove_2(--th(_6879),[++th(imp(_6887,_6879))|A]),
     prove_1(++th(_6887),[++th(imp(_6887,_6879))|A]).
prove_2(--th(_6887),A) :-
     prove_2(--th(_6879),[++th(_6887)|A]),
     prove_1(++th(imp(_6887,_6879)),[++th(_6887)|A]).
```

As this program was computed using a Prolog implementation, we give the program as generated and do not convert back to the more elegant logic programming notation. The following regular approximation is computed (in 1.73 seconds):

```
solve_ans(X1,X2) :-t307(X1),t16(X2).
t307(++X1) :-t308(X1).
t16([]) :-true.
t308(th(X1)) :-t13(X1).
t13(imp(X1,X2)) :-t14(X1),t15(X2).
t14(a) :-true.
t15(a) :-true.
prove_1_ans(X1,X2) :-t294(X1),t248(X2).
t294(++X1) :-t295(X1).
t248([X1|X2]) :-t211(X1),t248(X2).
t248([]) :-true.
t295(th(X1)) :-any(X1).
t211(--X1) :-t212(X1).
t212(th(X1)) :-any(X1).
```

Note that most of the partially evaluated program may be deleted as empty approximations for the *member* predicates and for *prove_2* indicate that all reduction steps and extension steps with the negative literal $--th(X)$ can never contribute to proving this theorem. This is the expected result.

Unfortunately the approximation for *prove_1*, corresponding to the positive literal $++th(X)$ is very imprecise (the $any(X)$'s occurring in the approximation indicating any term in the Herbrand universe of the program). We can therefore not infer any additional information not already known to us that may be used to prune the search space of this problem. What is the reason for the imprecision of our analysis method? There are several reasons that we discuss briefly:

1. The Y and Z variables occurring only once in the *ic_JŁukasiewicz* axiom. We have to assume the worst-case analysis results for these arguments.

2. There is only one predicate symbol th (*theorem*), one function symbol imp (*implies*) and one constant a. No differentiation between different predicates, functors and constants are possible.

3. Variable sharing information (variables occurring more than once in a literal) in the *ic_JŁukasiewicz* axiom is a crucial part of the axiom. Our regular approximation is very imprecise regarding keeping track of sharing information and therefore "throws away" important bits of information that may be needed to prune the search space effectively.

In the next section we show how we can do a finer grained analysis that may assist in deriving more precise information.

3 Compute a Safe Approximation Using Logic Program Analysis and Transformation - A Finer Grained Analysis

In the previous section we did an analysis on predicate symbol level. This can obviously be improved by also taking functors and arguments to functors into account during the specialization. Instead of only creating renamed procedures for

$$++th(X)$$
$$--th(X)$$

we may create renamed specialized procedures for

$$++th(a)$$
$$++th(imp(X,Y))$$
$$--th(a)$$
$$--th(imp(X,Y))$$

or

$$++th(a)$$
$$++th(imp(a,a))$$
$$++th(imp(a,imp(X,Y)))$$
$$++th(imp(imp(X,Y),a))$$
$$++th(imp(imp(X,Y),imp(Z,U)))$$
$$--th(a)$$
$$--th(imp(a,a))$$
$$--th(imp(a,imp(X,Y)))$$
$$--th(imp(imp(X,Y),a))$$
$$--th(imp(imp(X,Y),imp(Z,U)))$$

and so on ...

This gives a much larger partially evaluated program with, for the last given set of predicates, ten renamed specialized *prove* procedures and ten renamed *member* procedures. Analysis of this program which has a much finer grained differentiation than our previous program, may now allow us to infer more precise information as we can now differentiate between different instances of $++th(X)$ and $--th(X)$.

Unfortunately this extra precision up to a term depth of two or three is not enough to infer anything better than what we inferred before. Furthermore, our partially evaluated program becomes much bigger with every increase in term depth we specify and the analysis therefore also takes much longer. However, as was shown in [1], for some problems this extra differentiation is exactly what is needed to obtain precise analysis results. The question here is: Up to what term depth do we need to go before we infer any useful results? Also, is it practical as our analysis get slower and more complicated with each increase in precision we desire?

4 Compute an Under-Approximation by Restricting Arguments to Ground Terms

The idea here is to try to open a small window in the search space just big enough to allow one proof. If this can be done in such a way that the whole search space up to some depth do not need to be explored, we may find a proof faster than before. Our restriction of the search space is based on restricting arguments to ground terms of a specified depth.

Consider again the finer grained analysis we did in Section 3. If we restrict predicates to the form:

$$++th(a)$$
$$++th(imp(a,a))$$
$$--th(a)$$
$$--th(imp(a,a))$$

we can only prove a subset of the theorems we could previously. This corresponds to opening up a small window in the search space. We now approximate the resulting specialized program as before (as in Section 2 and Section 3). For the goal $\leftarrow solve(++th(imp(a,a)),[])$ it is trivially clear that we can not prove this theorem with the given restrictions (the window is too small). An empty approximation is computed for $solve(++th(imp(a,a)),[])$ confirming this.

We now gradually increase the complexity of ground terms (depth) until an empty approximation is not any more computed. We then have an indication that the first proof may have been found (however, we can not be sure as we have now computed an unsafe under-approximation) and use this restriction of terms to prune the search space of our real theorem prover. If our analysis results are precise enough, a proof will be found with the given restrictions. Hopefully, our analysis results will also allow us to prune the search space even further.

As in Section 2 and Section 3 we quickly run into problems with this approach as we get an explosion in the number of ground terms we have to specify and then also a very large partially evaluated program and slow approximation times as before. However, in [6] it is stated that to solve one of the basic Łukasiewicz theorems, the proof has length 29 and the deepest nesting of implications is 5. This may indicate that this approach may turn to be useful if similar

properties about depth of nesting of implications hold for other Łukasiewicz problems as well. Further experimentation is needed to resolve these questions.

5 Use Partial Evaluation with a Specialized Unfolding Rule to "Crystallize" out some Patterns in the Proof

Partial evaluation is an optimization technique whose use may result in gains in efficiency. Partial evaluation takes a program P and part of its input In and constructs a new program P_{In} that given the rest of the input *Rest*, yields the same result that P would have produced given both inputs. The origins of partial evaluation can be traced back to Kleene's s-m-n theorem [4] and others (see [5, 1, 3] for more details).

One of the basic problems encountered in Partial Evaluation is to create the right number of specialized versions of definitions (procedures). If we create too few specialized procedures, we loose precision and may miss some specialization opportunities. Creating too many specialized versions of definitions obviously wastes time and memory as our resulting program becomes very large with many redundant pieces of code.

The notion of a characteristic path was introduced in [2] to try and resolve this issue and it was successfully used in the SP-system to control partial evaluation. Informally, a characteristic path gives the structure of an SLDNF derivation and can be used to control the granularity of the specialized code. The intuitive idea behind this abstraction is that if two atoms B and C have the same characteristic path, they are regarded as equivalent from the point of view of partial evaluation and are presented by a single atom (which will lead to shared specialized code for B and C as they are indistinguishable to the partial evaluator). If [1] it was shown that this notion may need refinement to create the right number of specialized procedures for analysis purposes. Argument information was included in the notion to obtain the required specialization.

The aim here is to use partial evaluation to create "the right number" of specialized versions of the Łukasiewicz axioms so that from the structure of the specialized program we may infer the structure of a proof.

For the simplest of Łukasiewicz theorems, we decided as a first experiment on the following abstraction of terms:

$$[a, a] \rightarrow [a, a]$$
$$[a, X] \rightarrow [a, var]$$
$$[X, a] \rightarrow [var, a]$$
$$[X, X] \rightarrow [var, var]$$

a and var are constants and X any variable. All other structures are basically ignored, and the improved characteristic path abstraction therefore consists of the usual characteristic path abstraction information plus additional information abstracted in the way just described. This allows for a finer grained analysis and therefore "better" specialisation.

The aim with the given abstraction is as follows. We want specialized versions of the axioms for all the different "base cases" of terms that may occur. As we have a binary functor and one constant there are four possibilities if we consider that an argument may only be a variable or a constant. All variables are therefore "the same". Newly generated definitions will therefore

only be regarded as "equivalent" (and therefore reduced to one definition) when they stem from the same characteristic paths (similar SLDNF derivations) and they agree on all arguments as specified by the given abstraction operation. Partial evaluation of the logic program given in the introduction with respect to the goal $\leftarrow theorem([a,a])$ gives:

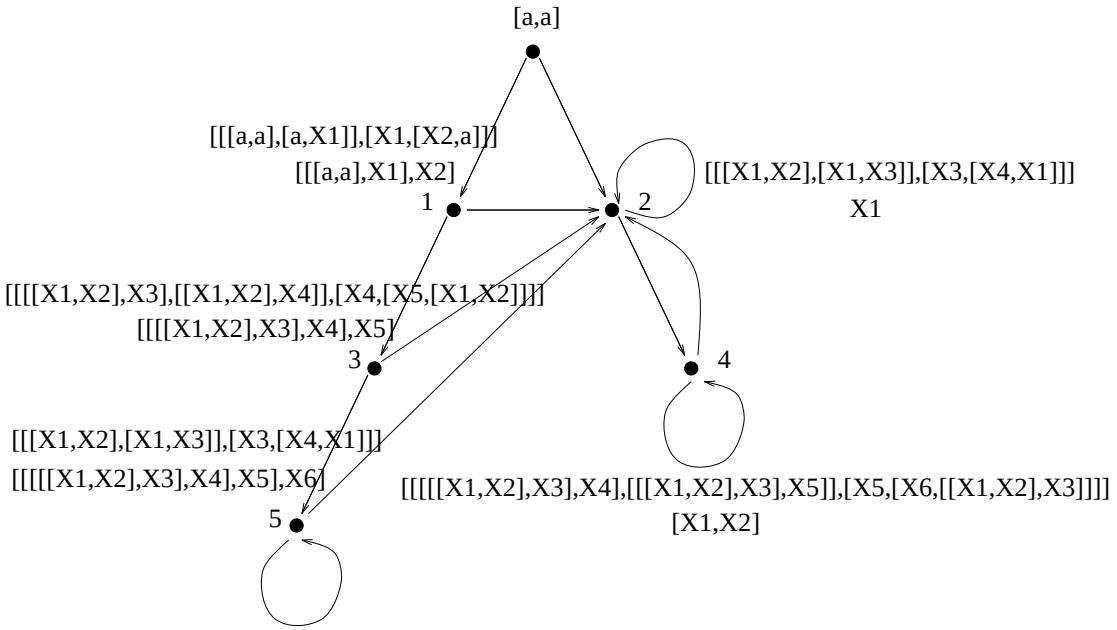
```
theorem([a,a]) :-
      theorem_1(X4,X5),
      theorem_2(X5),
      theorem_2(X4).
theorem_1([a,X1],[X1,[X2,a]]) :-
      true.
theorem_1(X1,X2) :-
      theorem_3(a,a,X1,X2,X5),
      theorem_2(X5).
theorem_2([[[X1,X2],[X1,X3]],[X3,[X4,X1]]]) :-
      true.
theorem_2(X1) :-
      theorem_4(X1,X4),
      theorem_2(X4).
theorem_3(X1,X2,X3,[[X1,X2],X4],[X4,[X5,[X1,X2]]]) :-
      true.
theorem_3(X1,X2,X3,X4,X5) :-
      theorem_5(X1,X2,X3,X4,X5,X8),
      theorem_2(X8).
theorem_4([[X1,X2],[X1,X3]],[X3,[X4,X1]]) :-
      true.
theorem_4(X1,X2) :-
      theorem_4([X1,X2],X5),
      theorem_2(X5).
theorem_5(X1,X2,X3,X4,[[[X1,X2],X3],X5],[X5,[X6,[[X1,X2],X3]]]) :-
      true.
theorem_5(X1,X2,X3,X4,X5,X6) :-
      theorem_5([X1,X2],X3,X4,X5,X6,X9),
      theorem_2(X9).
```

As more arguments have been generated, it is difficult to interpret the results. We therefore give the same result, but with specialization restricted to the same number of argument positions as was the case in the original program must be used. This gives the revised program below.

```
theorem([a,a]) :-
      theorem([[[a,a],X4],X5]),
      theorem(X5),
      theorem(X4).
theorem([[[a,a],[a,X1]],[X1,[X2,a]]]) :-
      true.
theorem([[[a,a],X1],X2]) :-
      theorem([[[[a,a],X1],X2],X5]),
      theorem(X5).
```

```
theorem([[[X1,X2],[X1,X3]],[X3,[X4,X1]]]) :-
      true.
theorem(X1) :-
      theorem([X1,X4]),
      theorem(X4).
theorem([[[[X1,X2],X3],[[X1,X2],X4]],[X4,[X5,[X1,X2]]]]) :-
      true.
theorem([[[[X1,X2],X3],X4],X5]) :-
      theorem([[[[[X1,X2],X3],X4],X5],X8]),
      theorem(X8).
theorem([[[X1,X2],[X1,X3]],[X3,[X4,X1]]]) :-
      true.
theorem([X1,X2]) :-
      theorem([[X1,X2],X5]),
      theorem(X5).
theorem([[[[[X1,X2],X3],X4],[[[X1,X2],X3],X5]],[X5,[X6,[[X1,X2],X3]]]]) :-
      true.
theorem([[[[[X1,X2],X3],X4],X5],X6]) :-
      theorem([[[[[[X1,X2],X3],X4],X5],X6],X9]),
      theorem(X9).
```

This however, is still difficult to interpret and we give a graph showing the calls that may occur and the pattern that crystallized with corresponding term structure. The numbers in the graph correspond to the numbers n of the specialized procedures *theorem_n*. All nodes are *and*-nodes and each path indicates one or more calls.



Each of the nodes in the graph may also be used as a termination node. The top-term at each node gives the term structure in the head of the non-recursive clause and the second term the term structure in the head of the recursive clause at each node. If a node is not a termination

node, both paths leading from the node has to succeed. The structure of the bodies of each recursive clause can be seen in the program just given. Although we have some structure in the graph, it is still difficult (impossible?) to decide which path may lead to a successful proof as at least one new variable is created at each node (see the previous program).

The first specialized Prolog program may be regarded as a new set of Łukasiewicz axioms derived from the two original axioms, but specialized (restricted) with respect to the theorem $theorem([a, a])$. We get eleven axioms, which we hope will assist the theorem prover in getting a faster proof. The only way to check this is to compare the times needed to find a proof using the original set of axioms and the transformed sets of axioms. The result of the experiment using KoMeT on a Sun SPARCstation 10 is as follows (using the list of KoMeT options given in Appendix A):

Axioms	**Runtime**
Original set of axioms	154.5 sec
Derived set of axioms	3.3 sec

However, the times do not tell the full story. The two following tables give the statistics for preprocessing operations.

```
 Statistic of simple reductions: Original set of axioms
 ------------------------------
Number of PURE -Reductions: 0
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 0
Number of UNIT -Reductions: 0
Number of ISOL -Reductions: 0
Number of ISOL*-Reductions: 2
Number of SUBS -Reductions: 0
```

```
 Statistic of simple reductions: Transformed set of axioms
 ------------------------------
Number of PURE -Reductions: 3
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 1
Number of UNIT -Reductions: 1
Number of ISOL -Reductions: 0
Number of ISOL*-Reductions: 9
Number of SUBS -Reductions: 0
```

This shows that the specialization process made some "hidden" information explicit that allowed the theorem prover to make better use of the axioms which in turn led to better preprocessing and therefore a faster proof. The argument differentiation (splitting of one complex argument into several less complex arguments) may be a possible cause for the obtained speedup. The number of extension steps has been reduced as follows:

Axioms	Number of extension steps
Original set of axioms	29
Derived set of axioms	17

An immediate question is if the speedup is the result of the switching of arguments of the binary function. The order of arguments are important from a logic programming specialization point of view, but is not should not be important from a theorem proving point of view. If we reverse the arguments (change it back to the original order), we get the following results.

Axioms	Runtime
Original set of axioms	166.8 sec
Derived set of axioms	1.2 sec

For this version of the problems fifteen axioms were generated which produced better results. Again, the preprocessing statistics provide useful information.

```
 Statistic of simple reductions: Original set of axioms
 ------------------------------
Number of PURE -Reductions: 0
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 0
Number of UNIT -Reductions: 0
Number of ISOL -Reductions: 0
Number of ISOL*-Reductions: 2
Number of SUBS -Reductions: 0
```

```
 Statistic of simple reductions: Transformed set of axioms
 ------------------------------
Number of PURE -Reductions: 4
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 2
Number of UNIT -Reductions: 1
Number of ISOL -Reductions: 1
Number of ISOL*-Reductions: 10
Number of SUBS -Reductions: 0
```

The number of extension steps has been reduced as follows:

Axioms	Number of extension steps
Original set of axioms	20
Derived set of axioms	16

Furthermore, the maximum depth of search needed to find a proof has also been reduced.

Axioms	Maximum depth of search
Original set of axioms	6
Derived set of axioms	4

The speedup obtained for these experiments may have been partly due to the "non-optimal" options selected for KoMeT given in Appendix A. A well tested set of options in given in Appendix B (this was developed by Thomas Rath). We then repeated the last experiment with the new options and the results are as follows:

Axioms	Runtime
Original set of axioms	4.42 sec
Derived set of axioms	3.5 sec

The number of extension steps has been reduced as follows:

Axioms	Number of extension steps
Original set of axioms	49
Derived set of axioms	40

Furthermore, the maximum depth of searh needed to find a proof has also been reduced.

Axioms	Maximum depth of search
Original set of axioms	5
Derived set of axioms	4

The reduction in depth of search needed to find a proof as well as the reduction in the number of extension steps seems to show a reduction in the complexity of the transformed problem.

A slightly more difficult theorem to prove is $[[[a, b], a], a]$. The specialisatiom method generated nineteen specialised axioms. The abstraction function used was a combination of that described earlier and that used for the theorem $[a, [b, a]]$ (see below for more details). The experimental results using the KoMeT options in Appendix C are given below.

```
 Statistic of simple reductions: Original set of axioms
 ------------------------------
Number of PURE -Reductions: 0
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 0
Number of UNIT -Reductions: 0
Number of ISOL -Reductions: 0
Number of ISOL*-Reductions: 2
Number of SUBS -Reductions: 0
```

```
 Statistic of simple reductions: Transformed set of axioms
 ------------------------------
Number of PURE -Reductions: 4
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 2
Number of UNIT -Reductions: 2
Number of ISOL -Reductions: 3
Number of ISOL*-Reductions: 13
Number of SUBS -Reductions: 0
```

Axioms	**Runtime**
Original set of axioms	651.5 sec
Derived set of axioms	8.0 sec

The number of extension steps has been reduced as follows:

Axioms	**Number of extension steps**
Original set of axioms	29
Derived set of axioms	12

and the maximum depth of searh needed to find a proof has also been reduced:

Axioms	**Maximum depth of search**
Original set of axioms	10
Derived set of axioms	5

A very useful reduction in the number of extension steps, depth of search and time it took to prove the theorem were obtained.

Some of the more difficult theorems such as $[[a, b], [[b, c], [a, c]]]$ have also been tried, but yet without the success achieved for some of the simpler theorems. This may indicate that the generalization of the abstraction function used previously, namely

$$[constant, constant] \rightarrow [constant, constant]$$
$$[constant, X] \rightarrow [constant, var]$$
$$[X, constant] \rightarrow [var, constant]$$
$$[X, X] \rightarrow [var, var]$$

where *constant* indicates any constant and is not abstracted is too imprecise and do not produce enough new procedures (axioms). The structure of the theorem may be a better base for developing an abstraction function from (e.g. for the theorem $[a, [b, a]]$ we should have base cases of the form $[a, [b, a]]$ abstracted in a similar way as described earlier). For theorems with a more complicated structure more complex abstractions preserving more structure may be needed. The theorem $[a, [b, a]]$ is relatively easy to prove and the described techniques also apply to this theorem. The time reduction is much less as there is much less room for optimization.

As a final experiment, we used the improved abstraction operation described in the previous paragraphs for the theorem $[[a,b],[[b,c],[a,c]]]$. The following tables of preprocessing operations give some indication that we may be moving in the right direction.

```
 Statistic of simple reductions: Original set of axioms
 ------------------------------
Number of PURE -Reductions: 0
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 0
Number of UNIT -Reductions: 0
Number of ISOL -Reductions: 0
Number of ISOL*-Reductions: 2
Number of SUBS -Reductions: 0
```

```
Statistic of simple reductions: Transformed set of axioms
 ------------------------------
Number of PURE -Reductions: 4
Number of TAUT -Reductions: 0
Number of MULT -Reductions: 5
Number of UNIT -Reductions: 2
Number of ISOL -Reductions: 6
Number of ISOL*-Reductions: 19
Number of SUBS -Reductions: 0
```

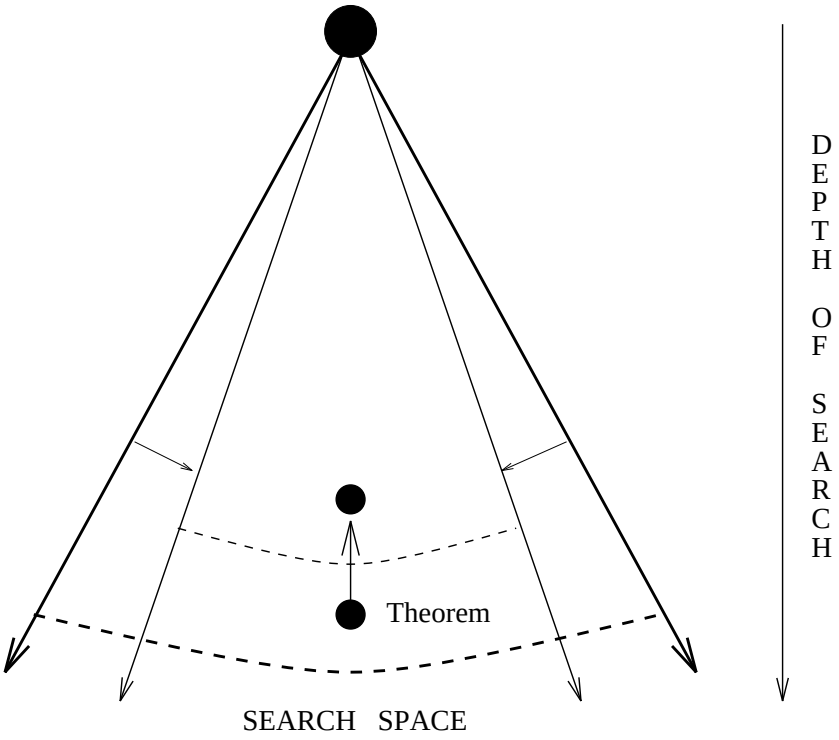
Twenty three new axioms were generated with a very complex structure in some of the argument positions. Some axioms now also have more than ten arguments.

This approach seems to produce some structure needed for a proof, better than just applying self-resolution an arbitrary number of times. There are several "problems" that need to be addressed before we can draw any real conclusions namely:

1. do we need a different abstraction function for each theorem we want to prove?
2. if the answer to 1. is affirmative, how do we automate this?
3. is the given abstraction function "perfect" or still too general or specific?

We can therefore summarize the proposed procedure as follows:

1. Write the Łukasiewicz axioms as a logic program.
2 Specialize the logic program with respect to a given theorem (using partial evaluation with a special unfolding rule) and create specialized procedures (axioms) distinguishing different cases that may occur based on the structure of the theorem.
3. Treat the specialized program as a new set of axioms and use these axioms to find a proof.



The transformation of the search space we are trying to achieve is illustrated in the given diagram.

The aim is to reduce the number of extension steps needed to find a proof as well to limit the explosion in the size of the search space.

6 Rewrite the Problem in a Constraint Logic Programming Language to Take Advantage of the Power of Constraint Solvers

We illustrate the idea with the well-known logic programming naive reverse example. The program is given below:

$$
\begin{aligned}
&reverse([\,],[\,]) \leftarrow \\
&reverse([X|Xs],Z) \leftarrow \\
&\qquad reverse(Xs,Y), \\
&\qquad append(Y,[X],Z)
\end{aligned}
$$

This program can be rewritten in PrologIII (a constraint logic programming language [7]) as follows (keeping in mind that angled brackets now replace the Prolog-style list notation, $< X >$ indicates the head or tail of a list, '.' indicates concatenation and $\leftarrow$ is replaced by $\rightarrow$):

$$
\begin{aligned}
&reverse(<>,<>) \rightarrow; \\
&reverse(< X > .Y, Z. < X >) \rightarrow
\end{aligned}
$$

$reverse(Y, Z)$;

Note that in PrologIII we can also access the element at the tail of a list. This program reverses a list in a time linear to the length of the list. However, we can still improve as we show in the following program:

$$
\begin{array}{l}
reverse(<>, <>) \rightarrow; \\
reverse(< X >, < X >) \rightarrow; \\
reverse(< X > .W. < Y >, < Y > .Z. < X >) \rightarrow \\
\quad reverse(W, Z);
\end{array}
$$

We can now reverse a list in a time approximately linear to one half the length of a list. The resulting PrologIII list operations are more complex as we have a much more powerful inference engine running in the background. We may also add constraints where needed to further exploit the power of other constraint solvers. The idea is to do a similar transformation of the Łukasiewicz axioms as illustrated above.

Our Łukasiewicz problem looks as follows rewritten in PrologIII:

$$
\begin{array}{l}
theorem(<<< X, \_ >, < X, Z >>, < Z, < \_, X >>>) \rightarrow; \\
theorem(Y) \rightarrow \\
\quad theorem(< Y, X >), \\
\quad theorem(X);
\end{array}
$$

Unfortunately, because we have a binary functor and therefore the length of each list is known, we could not find a way to rewrite the problem to successfully exploit the extra power of the PrologIII symmetrical list mechanism (or any one of the other powerful domain solvers such as interval arithmetic or boolean algebra). The question is: Is this possible to transform or rewrite the Łukasiewicz axioms and if so, how? Also, is it possible to automate this transformation?

7 Solve a More General Problem and Collect Constraints that have to be Satisfied to Solve Instances of this Problem

Consider the following PrologIII variation (given below) of the Łukasiewicz axioms (this time with the same argument order as in the original axioms). The aim is to solve a more general problem, infer some constraints and then inspect the constraints to obtain information about the proofs. We first give the PrologIII session and then explain what happened.

```
PROLOG III v1.4, Avril 93 (C)PrologIA 1989-1993

> input("test.p3");
{}
> > > Err 17: Fin de fichier
> list;
theorem(<<<X,Y>,Z>,<<Z,X>,<U,X>>>) ->  ;
theorem(Y) ->
```

```
    theorem(<X,Y>)
    theorem(X)  ;

{}
> theorem(<a,a>);
{a = <<<U_1,X_1>,<U_1,<U_1,X_1>>>,<U_1,<U_1,X_1>>>, X_1 = <U_1,X_1>}
{a = <<<Y_2,X_3>,<U_3,X_3>>,<X_3,Y_3>>}
{a = <<U_4,<X_5,Y_5>>,<<Y_4,X_5>,<U_5,X_5>>>}
{a = <<U_4,<<Y_6,X_7>,<U_7,X_7>>>,<U_6,<X_7,Y_7>>>}
{a = <<U_4,<U_8,<X_9,Y_9>>>,<U_6,<<Y_8,X_9>,<U_9,X_9>>>>}
{a = <<U_4,<U_8,<<Y_10,X_11>,<U_11,X_11>>>>,<U_6,<U_10,<X_11,Y_11>>>>}
{a = <<U_4,<U_8,<U_12,<X_13,Y_13>>>>,<U_6,<U_10,<<Y_12,X_13>,<U_13,X_13>>>>>}
.
.
.
```

Note that in $theorem(< a, a >)$, a is a PrologIII variable and NOT a constant. Our query asks PrologIII to solve all problems where the first and second arguments to our binary functor is equivalent. When the variable is instantiated to the constant a, we get our *simple* theorem given in the introduction which is then an instance of this more general theorem.

An infinite number of solutions expressed as constraints are produced. Unfortunately, by inspecting the constraints we do not learn that much about the structure of a proof. Is the query we tried to general? How can we make the query more precise without solving the original problem? Are the inferred constraints useful?

8 Summary

One conclusion we can draw from the experiments is that there are not a lot of syntactic information available in the Łukasiewicz problems that may be used to prune the search space effectively. This may indicate that the axioms in their given form are just too compact for a precise syntactic analysis.

Of the six experiments we tried the partial evaluation experiment seems to hold the most promise. Some structure was inferred based on the "base-cases" that may occur. However, there are still many unanswered questions as indicated in Section 5 that need to be resolved before we can draw any final conclusions about the suitability of this approach.

These experiments confirm that the Łukasiewicz problems are extremely challenging problems that remain a challenge not only to state of the art theorem provers, but also to state of the art analysis and transformation techniques.

As a final comment, we hope that one of the analysis and transformation methods proposed in this report will turn out to provide THE solution everybody is searching for.

Acknowledgements

I would like to thank Uwe Egly and Michael Thielscher for interesting discussions about how (and how not) to approach these problems and Daniel Korn for introducing me to KoMeT.

Lastly, I would like to thank prof. Wolfgang Bibel for suggesting these problems as a subject for analysis and transformation. They turned out to be very challenging indeed.

References

[1] D.A. de Waal. *Analysis and Transformation of Proof Procedures.* PhD thesis, University of Bristol, October 1994.

[2] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of ICLP'90*, pages 732–746, 1990.

[3] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, University of Bristol, 1993.

[4] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

[5] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[6] F. Pfenning. Single axioms in the implicational propositional calculus. In E. Lusk and R. Overbeek, editors, *Automated Deduction—CADE-9*, pages 710–713. Springer-Verlag, 1988.

[7] PROLOGIA. *PrologIII Reference Manual*, version 1.2 edition, 1991.

[8] C. Sutter, Sutcliffe G, and T. Yemenis. The TPTP problem library. Technical Report TPTP v1.0.0 - TR 12.11.93, James Cook University, Australia, November 1993.

A List of KoMeT Options Selected for Experiments

```
simple_reductions = on.
pure_reduction = on.
unit_reduction = on.
mult_reduction = on.
taut_reduction = on.
isol_reduction = on.
isol_star_reduction = on.
subs_reduction = on.
munch_reduction = no_equality.
munch_star_reduction = off.
equality_reductions = on.
parser_delete_equality_axioms = on.
equality_handling = add_equality_axioms.
% e_modification.
optimize_connections_in_equality_clauses = on.
select_as_goals = neg_clauses.
check_preprocessing_constraints = on.
build_tautology_constraints = on.
build_subsumtion_constraints = on.
reorder_literals = connection_graph.
% connection_graph.
% instantiation_value.
show_xptree = on.
proof_for_original_matrix = on.
proof_presentation = xptree.
prover = compiler.
compile_dynamic_literal_selection = off.
compile_check_proof_depth = off.
compile_check_number_of_inferences = off.
compile_search = iterative_deepening(1,1,*).
% iterative_deepening(1,1,*).
% restrict_number_of_inferences(1,*,*).
compile_eclipse_unification = on.
compile_restrict_positive_transitivity = on.
compile_restrict_negative_transitivity = on.
compile_restrict_complex_negative_transitivity = on.
compile_create_statistic = off.
compile_create_detailed_statistic = off.
compile_write_extension_steps = off.
compile_create_proof = on.
compile_reduction_steps = off.
compiler_literal_insertation = breath_first.
compiler_literal_selection = equality.
compile_identical_ancestor_pruning = on.
compile_create_prooftree = on.
compile_check_for_subsuming_unit_clauses = off.
```

```
compile_check_for_subsuming_unit_lemma = off.
compile_check_for_identical_compl_path_literals = off.
compile_regularity_pruning = off.
compile_local_lemmata = off.
compile_unit_lemmata = off.
compile_restrict_number_of_unit_lemmata = off.
compile_hard_restrict_number_of_unit_lemmata = off.
compile_minimal_proof_length_of_unit_lemmata = off.
compile_show_unit_lemmata = off.
compile_use_all_generated_unit_lemmata = off.
compile_use_c_reductions_for_identical_literals = off.
compile_use_c_reductions_for_unifiable_literals = off.
compile_minimal_proof_length_for_c_reduction = off.
compile_maximal_path_length_for_c_reduction = off.
compile_check_tautology_constraints = off.
compile_check_subsumtion_constraints = off.
compile_check_antilemmata_constraints = off.
compile_reset_antilemmata_constraints = clause.
compile_generalize_lemmata = off.
compile_delay_activation_of_unit_lemmata = off.
compiler_reorder_connections = shortest_clauses_first.
compile_unit_extension_steps_before_reductions = off.
```

B List of KoMeT Options Selected for Experiments

```
%use_specified_file_setting = on.
%option_file = '$KOMETHOME/Examples/Sonstige/luk3.opt'.
output_file = [].

normal_form_transformation = on.
%nft_input_file = [].
%nft_output_file = [].
definitional_nf_transformation = off.
input_file = '~andre/analysis/kometluk3.p'.
preprocessing = on.
%preprocessing_input_file = '~andre/analysis/kometluk3.p'.
preprocessing_output_file = [].
simple_reductions = off.
pure_pl_reduction = on.
unit_pl_reduction = on.
mult_pl_reduction = on.
taut_pl_reduction = on.
isol_pl_reduction = on.
isol_star_pl_reduction = on.
subs_pl_reduction = on.
munch_reduction = off.
bottom_up_evaluation = off.
indexing_method_for_subsumption_tests = abstraction_trees.
at_type = heuristic.
maximal_number_of_iterations = no_restriction.
type_of_generated_unit_clauses = facts.
maximal_number_of_generated_unit_clauses = no_restriction.
minimal_productivity = no_restriction.
predicate_names = no_restriction.
base_non_unit_clauses = no_restriction.
db_reductions = off.
equality_reductions = off.
equality_handling = off.
optimize_connections_in_equality_clauses = off.
use_completed_theory = off.
completed_theory_input_file = [].
build_reachability_graph = on.
select_as_goals = pos_clauses.
check_preprocessing_constraints = on.

show_xptree = off.
%xptree_input_file = '$KOMETHOME/Examples/Sonstige/luk3.xptree'.

prover = compiler.
compile_reachability_graph = on.
compile_dynamic_literal_selection = off.
```

```
%compiler_output_file = '$KOMETHOME/Examples/Sonstige/luk3.pl'.
compile_check_proof_depth = off.
compile_check_number_of_inferences = off.
compile_search = restrict_number_of_inferences(1,*,*).
% iterative_deepening(1,1,*).
% restrict_number_of_inferences(1,*,*).
compile_eclipse_unification = on.
compile_restrict_positive_transitivity = off.
compile_restrict_negative_transitivity = off.
compile_restrict_complex_negative_transitivity = off.
compile_create_statistic = on.
compile_create_detailed_statistic = off.
compile_write_extension_steps = on.
compile_create_proof = on.
compile_path = off.
compile_reduction_steps = off.
compiler_literal_insertation = breadth_first.
compiler_literal_selection = depth_first.
compile_identical_ancestor_pruning = on.
compile_create_prooftree = off.
compile_check_for_subsuming_unit_clauses = off.
compile_check_for_subsuming_unit_lemma = on.
compile_check_for_identical_compl_path_literals = off.
compile_regularity_pruning = off.
compile_local_lemmata = off.
compile_unit_lemmata = on.
compile_restrict_number_of_unit_lemmata = 12.
compile_hard_restrict_number_of_unit_lemmata = on.
compile_minimal_proof_length_of_unit_lemmata = 1.
compile_maximal_termdepth_of_unit_lemmata = 4.
compile_show_unit_lemmata = on.
compile_use_all_generated_unit_lemmata = off.
compile_check_tautology_constraints = off.
compile_check_subsumtion_constraints = off.
compile_check_antilemmata_constraints = off.
compile_generalize_lemmata = on.
compile_delay_activation_of_unit_lemmata = off.
compiler_reorder_connections = shortest_clauses_first.
```

C List of KoMeT Options Selected for Experiments

```
input_file = '~andre/analysis/abc.p'.
normal_form_transformation = off.

preprocessing = on.
simple_reductions = on.
pure_reduction = on.
unit_reduction = on.
mult_reduction = on.
taut_reduction = on.
isol_reduction = on.
isol_star_reduction = on.
subs_reduction = on.
munch_reduction = on.
db_reductions = off.
equality_reductions = off.
equality_handling = off.
optimize_connections_in_equality_clauses = off.
use_completed_theory = off.
build_reachability_graph = on.
select_as_goals = pos_clauses.
check_preprocessing_constraints = on.

prover = compiler.
literal_selection = breadth_first.
search = restrict_number_of_inferences(1,*,*).
proof_depth_restriction = off.
proof_inference_restriction = off.
reduction = on.
identical_ancestor_pruning = on.
test_for_identical_ancestor = act_clause_path_and_clauses.
check_tautologie_constraints = off.
check_anti_lemmata_constraints = off.
intelligent_backtracking = off.
restrict_dynamic_use_of_equality_clauses = off.
use_lemmata = off.
inference_restriction_for_lemma_storing = 1.
instance_restriction_for_lemma_storing = 1.
maximal_number_of_unit_lemmata = off.
show_created_lemmata = off.
use_eq_connection = off.
use_theory = off.
use_induction = off.

show_xptree = off.
use_program_for_prove_presentation = xptree.
%xptree_input_file = '$KOMETHOME/Examples/Sonstige/luk2.xptree'.
```

```
prover = compiler.
compile_reachability_graph = on.
compile_dynamic_literal_selection = on.
%compiler_output_file = '$KOMETHOME/Examples/Sonstige/luk2.pl'.
compile_check_proof_depth = off.
compile_check_number_of_inferences = off.
compile_search = restrict_number_of_inferences(1,*,*).
compile_eclipse_unification = on.
compile_restrict_positive_transitivity = off.
compile_restrict_negative_transitivity = off.
compile_restrict_complex_negative_transitivity = off.
compile_create_statistic = off.
compile_create_detailed_statistic = off.
compile_write_extension_steps = off.
compile_create_proof = on.
compile_path = off.
compile_reduction_steps = off.
compiler_literal_insertation = breadth_first.
compiler_literal_selection = equality.
compile_identical_ancestor_pruning = off.
compile_create_prooftree = on.
compile_check_for_subsuming_unit_clauses = off.
compile_check_for_subsuming_unit_lemma = off.
compile_check_for_identical_compl_path_literals = off.
compile_regularity_pruning = off.
compile_local_lemmata = off.
compile_unit_lemmata = off.
compile_restrict_number_of_unit_lemmata = 22.
compile_hard_restrict_number_of_unit_lemmata = on.
compile_minimal_proof_length_of_unit_lemmata = 1.
compile_maximal_termdepth_of_unit_lemmata = 5.
compile_show_unit_lemmata = on.
compile_use_all_generated_unit_lemmata = on.
compile_check_tautology_constraints = off.
compile_check_subsumption_constraints = off.
compile_check_antilemmata_constraints = off.
compile_generalize_lemmata = on.
compile_delay_activation_of_unit_lemmata = on.
compiler_reorder_connections = shortest_clauses_first.
```