Ulrich John, Petra Hofstedt, Armin Wolf

Deklarative Ansätze zur Künstlichen Intelligenz – punktuelle Beiträge

- Post-Proceedings des 32. Workshops für (Constraint) Logische Programmierung -

W(C)LP 2018

Herausgeber/ Organisation W(C)LP 2018

Prof. Dr. Ulrich John (Chair)
Hochschule für Wirtschaft, Technik und Kultur (HWTK)
ulrich.john@hwtk.de

Prof. Dr. Petra Hofstedt
Brandenburgische Technische Universität Cottbus-Senftenberg
hofstedt@b-tu.de

Dr. Armin Wolf
Fraunhofer FOKUS
armin.wolf@fokus.fraunhofer.de

GLP - Gesellschaft für Logische Programmierung e.V.

Programmkomitee

Erscheinungsdatum: September 2019

© Hochschule für Wirtschaft, Technik und Kultur | hwtk – University of Applied Sciences
Bernburger Straße 24/25, 10963 Berlin
www.hwtk.de

ISBN 978-3-00-065784-9

Vorwort

Der Begriff „*Künstliche Intelligenz*“ ist seit einigen Monaten (wieder) in aller Munde, KI ist das Hauptthema des aktuellen Wissenschaftsjahres 2019, Hauptthema diverser Konferenzen, Debatten und Initiativen etc. in Politik, Gesellschaft und Wirtschaft. Gereifte und weiterzuentwickelnde KI-Technologien werden dabei als hauptsächliche, technologische Treiber einer zweiten Digitalisierungswelle gesehen und bieten erhebliche, revolutionierende Potenziale. Damit verbunden sind natürlich nicht zu unterschätzende Risiken und enorme gesellschaftliche Herausforderungen. Relevante Stichworte/ Themen der nächsten Jahre sind diesbezüglich zum Beispiel „*Intelligente Digitalisierung*“, „*Intelligente Prozesse*“, „*Intelligentes Unternehmen*“, „*Intelligente Hochschule*“. „*Intelligente Behörde*“, „*Intelligenter Staat*“ und „*Intelligentes Europa*“.

Im derzeitigen Hauptfokus der KI-Publikationen und KI–Anwendungen scheinen insbesondere Data Mining, Neuronale Netze und Big-Data-Technologien zu stehen, wobei dies unter anderem durch die erzielte Reife der Technologien und durch die Verfügbarkeit von Cloud Computing ermöglicht wird. Stellt man sich den großen Themen/ Herausforderungen (siehe oben), wird man in zunehmendem Maße erkennen, dass zusätzlich zu den „rein datenbasierten Ansätzen“, *deklarative KI-Komponenten* eine entscheidende Rolle spielen werden. Seit diversen Jahren werden diesbezügliche Arbeiten und Überlegungen in den jährlich stattfindenden W(C)LP[1]-Workshops der *Gesellschaft für Logische Programmierung (GLP e.V.)* und den MOC[2]-Workshops thematisiert, diskutiert und weiterentwickelt.

In diesem online-Buch *„Deklarative Ansätze zur Künstlichen Intelligenz – punktuelle Beiträge“* werden ausgewählte, teilweise überarbeitete und erweiterte, Beiträge des 32. Workshops für (Constraint) Logische Programmierung (W(C)LP 2018) präsentiert, der im September 2018 an der Hochschule für Wirtschaft, Technik und Kultur (HWTK) in Berlin stattfand. Die ausgewählten Beiträge sind in einem relativ breiten Spektrum angesiedelt: im Kontext relationaler Datenbanken, theoretische Erweiterungen von Prolog, Ansätze zum beschleunigten Lösen von CSP-Problemen, Integration mit Nicht-KI-Programmiersprachen und Aspekte der Lehre von Deklarativen-KI-Sprachen in Studiengängen der Betriebswirtschaftslehre. Zusätzlich enthalten ist ein Initialbeitrag für die auf dem Workshop durchgeführte Podiumsdiskussion *„Künstliche Intelligenz und Digitalisierung – sind wir auf dem richtigen Weg?“*.

Berlin, September 2019

Ulrich John

[1] Workshop on (Constraint) Logic Programming
[2] Workshop zur Deklarativen Modellierung und effizienten Optimierung komplexer Probleme

Inhaltsverzeichnis

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

A New Benchmark Database and An Analysis of Transitive Closure Runtimes

Stefan Brass and Mario Wenzel

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
`brass@informatik.uni-halle.de`, `mario.wenzel@informatik.uni-halle.de`

Abstract. In this paper, we compare the performance of logic programming systems such as XSB with the performance of relational database systems such as SQLite and RDF stores such as Apache Jena.
There are three main contributions compared to the OpenRuleBench [6]: (1) We created a database for storing and evaluating the runtimes of the benchmarks. In particular, this manages several executions of the same benchmark configuration, which is missing in the OpenRuleBench scripts. We also measure more parameters. (2) We identified and solved a problem with the OpenRuleBench test graphs for the transitive closure benchmark. We also added a relatively large number of additional test graphs of different structure. (3) We propose a simple cost measure for the transitive closure benchmark based on parameters of the graph, and compare the measured run times with this cost measure.
Currently, we did measurements and analysis only for the transitive closure benchmark. The OpenRuleBench collection is much larger. Our database is general enough to store all benchmark results, but the other work is just the beginning of a larger project.

1 Introduction

Benchmarks can be a useful tool to compare performance between systems and evaluate how well different kinds of systems and evaluation schemes perform against each other. Especially comparing different models of computation and evaluation schemes can be helpful for system developers in identifying shortcomings and furthering cross-pollination between disciplines.

In the area of deductive databases/rule-based systems, OpenRuleBench [6] is a well-known benchmark suite. It is a quite large collection of problems, basically 12 logic programs, but some with different queries and different data files. The original paper [6] contains 18 tables with benchmark results.

However, the OpenRuleBench scripts run each benchmark only once, and do not support comparing benchmarks run with different option settings or program versions, or even on different machines. We have developed a database and new benchmark execution scripts in order to overcome these restrictions. Furthermore, the test graphs used in OpenRuleBench for the transitive closure have a very specific (dense) structure, and even graphs declared as non-cyclic contain loops. We solved this problem and used a much larger collection of different test graphs.

In this paper, we report results for only one of the OpenRuleBench benchmark problems, namely the transitive closure, but we look at it in much more detail. In particular, we define a cost measure, namely the number of applicable rule instances, and compare the actual runtimes with this cost measure. We believe that for declarative programming, it is important that one can get a rough estimate of the runtime on an abstract level (without need to understand implementation details). At least, one wants to be sure that for all inputs that might occur (even larger ones), the runtime will be acceptable. If the runtime would suddenly "explode" for specific inputs, the system would be unreliable.

We checked not only logic programming systems such as XSB [9], YAP [5], and SWI-Prolog, but also relational databases (currently only SQLite3, but PostgreSQL and HSQLDB are coming soon), and RDF triple stores/graph databases (Apache Jena) as well as DataScript (a Datalog query engine for JavaScript). Of course, we also included our own abstract machine for Push bottom-up evaluation of Datalog [4]. Our own system development project also made it necessary to develop a database for our performance measurements that permits to evaluate the effects of updates and different implementation alternatives.

The runtimes of transitive closure in relational databases have already been investigated in [7], but there has been progress in the implementations of relational database systems. For instance, the number of iterations, which used to be important eight years ago, has become much less important now.

In addition, we did some measurements on main memory consumption, also giving a cost measure for that resource.

2 A Benchmark Database

The OpenRuleBench scripts [6] run each benchmark only once. However, it is common practice to execute benchmark programs several times. There is some variation in the runtimes of the same program on the same data on the same machine. For instance, the total runtime of XSB for the `tcff` benchmark with the largest tested random graph (`f_2k_1m`) in 10 runs varied between 64.1 s and 75.9 s (with average 69.7 s and standard deviation 5.1 s). Since we are developing our own system, we want a reliable indication of the effects of changes in the implementation, even if they are not dramatic.

When measuring other systems, we feel obliged to do some tests with different option settings (or different indexes) and choose the best settings we could find (with our limited expertise). Thus, the situation gets more complicated than in the OpenRuleBench, since we have different implementation variants of the same benchmark for the same system.

Furthermore, as developers we use different machines with different operating systems. In order to keep the overview of the different runtime measurements, we decided to develop a benchmark database.

The database has the following tables:

- `MACHINE(`<u>`MACHINE`</u>`, SEQ_NO, STATUS, MODEL, YEAR, MONTH, PRICE, NUM_CPUS, CPU, CPU_CORES, CPU_THREADS, GHZ, ARCH, RAM, NUM_DISKS, DISK, SSD, OS, OS_CORE)`
  These are some data on the test machines. We underline primary key attributes, in this case the machine name. There is a single "standard machine" marked with `STATUS='S'`, and many views for evaluating the data are restricted to this machine. However, the database permits to store and evaluate benchmark measurements for different machines with different architectures and operating systems. The number in column `SEQ_NO` can be used for ordering the list of machines.

- `BENCHMARK(`<u>`BENCH`</u>`, SEQ_NO, HEADLINE)`
  This is the list of all benchmarks, e.g. `tcff` is a benchmark identifier in column `BENCH`. It corresponds to the computation of the transitive closure with query `tc(X,Y)`, i.e. with both arguments "free". Another benchmark, `tcbf`, is the computation of transitive closure with query `tc(1,X)`, i.e. the first argument "bound" (to a given value), and the second argument "free". The column `SEQ_NO` is for ordering benchmarks in the output, e.g. when a web page with all benchmark results should be generated. This would also need a `HEADLINE` for each benchmark.

- `PROGRAM(`<u>`PROG`</u>`, SEQ_NO, FULL_NAME, URL, REMARK)`
  This table lists the tested programs, such as `xsb`.

- `PROG_VER(`<u>`PROG`</u>`→PROGRAM, `<u>`VER`</u>`, `<u>`MACHINE`</u>`→`<u>`MACHINE`</u>`, VER_NAME, VER_ADDTEXT, VER_DATE, COMPILER, STATUS, REMARK)`
  This table lists program versions that are or were installed on a test machine. They are identified here by a number that is unique together with the program name and the machine. These three elements (Program, Version number, and machine) are also encoded in the file names with the test results. We use a simple sequential number `VER` for the version in the key, not the real version number (which can be an arbitrary string). The real version number is available in column `VER_NAME` and can be used for output.

  This table also defines a current version of a program on a machine (by `STATUS='C'`). Some views show only results for the current version. It is also possible to define a previous version (`STATUS='P'`) as reference, if the result of a program update should be evaluated.

- `BENCH_IMPL(`<u>`PROG`</u>`→PROGRAM, `<u>`BENCH`</u>`→BENCHMARK, `<u>`IMPL`</u>`, STATUS, DESCRIPTION)`
  In general, there are different option settings (e.g. index selections) that should be tested for the same program to execute a given benchmark as fast as possible. We call such a combination of benchmark, program and option settings a "benchmark implementation". `STATUS='B'` marks the best implementation that is used in some views to compare different programs. This is selected manually, but there are also views that help to find the best implementation.

– `DATA_FILE(`<u>`FILE_ID`</u>`, SEQ_NO, LINES, BYTES, FILENAME)`
  The filename is stored without extension, since we need the data files in different formats for different programs. The number of bytes is the size of the "Datalog Facts" version. There is one fact per line, i.e. `LINES` is the number of facts (tuples in the relation).
– `GRAPH_DATA(`<u>`FILE_ID`</u>`→DATA_FILE, NUM_NODES, NUM_EDGES, ..., TC_SIZE, COST)`
  For data files that are standard directed graphs, more information is available. This helps to evaluate the results for the transitive closure benchmark.

– `BENCH_INPUT(`<u>`BENCH`</u>`→BENCHMARKS, `<u>`FILE_ID`</u>`→DATA_FILE)`
  This is the many-to-many relation that states which input files should be used for which benchmark.
– `BENCH_RUN(`<u>`RUN_DATE`</u>`, `<u>`RUN_NO`</u>`, `<u>`PROG`</u>`, `<u>`VER`</u>`, `<u>`MACHINE`</u>`, BENCH, IMPL, FILE_ID, WITH_OUTPUT, LOAD_TIME, EXEC_TIME, TOTAL_TIME, USER_TIME, SYS_TIME, REAL_TIME, MEM, STATUS, REMARK)`
  This is the main table that contains the benchmark results. There are three foreign keys:
  - `PROG`, `BENCH`, `IMPL` → `PROG_IMPL`.
  - `BENCH`, `FILE_ID` → `BENCH_INPUT`.
  - `PROG`, `VER`, `MACHINE` → `PROG_VER`.

  There are two sources for timing information: Measurements done by the program itself, and external measurements done with `/usr/bin/time`. The following values are what the program that runs the benchmark reports (if available):
  - `LOAD_TIME`: This is the time for loading the input data. (All time values are stored in milliseconds.)
  - `EXEC_TIME`: This is the execution time for the benchmark query. If the column `WITH_OUTPUT` is `'Y'`, it contains the time for writing the query result. However, all normal measurements are done without writing the result. But we generated a query result file at least once for each system and benchmark in order to check whether the result is correct. Also, a very large time difference to the normal benchmark execution would indicate that the optimizer of the system is too clever and uses the fact that the result values are not really read.
  - `TOTAL_TIME`: This is the total runtime for executing the benchmark as reported by the system.

  If the system can report CPU time and real time ("wallclock time"), all three values are CPU time. If the two differ a lot, the `STATUS` column contains a warning. If only wallclock time is available, we take that. We use the external measurements for the main comparison on the benchmarks, so these three values are only interesting if one wants to see what fraction of the time is spent on loading the data. Furthermore, startup time and time for compiling the benchmark is not included in these data.

The following values are measured externally and should be comparable for all systems that run as a single process:

- `USER_TIME`: This is the CPU time the program spent in user mode.
- `SYS_TIME`: This is the CPU time the program spent in kernel mode, i.e. while executing operating system calls. Usually, this is small. The sum of `USER_TIME` and `SYS_TIME` is the total CPU time the program used for executing the benchmark.
- `REAL_TIME`: This is the total wallclock time that the execution of the benchmark took. For single-threaded programs, it is slightly longer than the total CPU time used (e.g. when waiting for I/O). For multi-threaded programs, it can be significantly less than the CPU time used (which is added up over all threads).
- `MEM`: This is the "maximum resident set size" (in kilobytes), i.e. the "high water mark" for the amount of memory a process uses and that is present in real RAM (e.g., not swapped out). It includes shared libraries (as far as actually used). In our previous paper [2] we checked that the amount of memory used for only starting and immediately stopping the tested systems is small compared to the memory used for actually executing the benchmarks.

We also implemented a number of views to analyze the data. The generation of tables in LaTeX and HTML format is done by means of SQL queries. Our views also help to check for outliers to get more confidence in the data. One can also query for the best implementation variant of a benchmark for a system.

3 Transitive Closure Problem on Different Systems

In this section we describe the systems that we used in this work.

For comparability we decided on the transitive closure as our first problem to benchmark the database systems on. The problem of finding the transitive closure of a graph has been extensively studied and easily formulated for the different database systems. The transitive closure is also one benchmark in the OpenRuleBench collection [6]. There, the standard tail-recursive formulation is chosen:

```
tc(X,Y) :- par(X,Y).
tc(X,Y) :- par(X,Z), tc(Z,Y).
```

"`par`" stands for "parent", but it is simply the edge relation of the graph. The nodes in the graph are integers, thus are large set of facts of the form `par(1,2)` are given. We will discuss the test graphs in Section 4.

3.1 Prolog-Systems with Tabling

Since many of the test graphs are cyclic, a standard Prolog system would not terminate with the above program. We need systems that support tabling to detect repeated calls to the `tc` predicate.

XSB Prolog version 3.8.0
XSB Prolog is a well-known deductive system [9] with a long experience in efficiently implementing Prolog evaluation with tabling. We ran our tests loading the data with `load_dync` and enabled subsumtive tabling, and a trie index on the `par` relation.

YAP Prolog version 6.2.2
YAP is a high performance Prolog compiler [5] that can be configured in a version with tabling.

SWI Prolog version 7.7.15
SWI Prolog is a robust and scalable implementation of the Prolog language. However, the support for tabling is very new, and declared as merely a first prototype. As mentioned on the SWI Prolog web page, it cannot be considered yet as a serious competitor to the above systems.

For the three systems above, we used the query "`tc(_,_), fail.`". I.e. we backtrack over all solutions, but the (quite large) output is not actually written. Also in OpenRuleBench, it is done in this way. However, our scripts also support measuring the time with writing of the result in order to check that the correct relation is computed, and the systems are not doing too big optimizations in the main benchmark because the query result is not used.

3.2 Bottom-Up Evaluation, Datalog systems

Push-Method with Abstract Machine (BAM)
We developed the Push method for bottom-up evaluation of Datalog [1,3]. It uses each derived fact immediately and can be seen as an extreme form of seminaive evaluation that dates back to the PhD thesis of Heribert Schütz [10]. Whereas we defined the method originally as a translation from Datalog to C++, we recently developed an abstract machine "BAM" for bottom-up evaluation based on the push method [4]. For the performance comparison, this is better, because the other systems also interpret code of some abstract machine. The translation to C++ would result in native machine code.

DataScript 0.16.6
DataScript is a Datalog query engine for JavaScript. We ran it on node.js 10.5.0. Even though DataScript's performance is severely lacking, it's an example for a benchmark on newer Datalog query engines akin to Datomic or Mozilla Mentat (formerly Datomish).

DataScript accepts queries in extensible data notation (EDN). We had to define the `par` relation as a relation with the multiplicity "many" and allowed for an index on this relation. In EDN the query and the rules are defined in the following way (`$` is a forward-reference to the database and `%` is a forward-reference to the rules):

```
[:find ?e1 ?e2 :in $ % :where (tc ?e1 ?e2)]
[ [(tc ?e1 ?e2) [?e1 "par" ?e2] ]
  [(tc ?e1 ?e2) [?e1 "par" ?ex] (tc ?ex ?e2) ] ]
```

3.3 Relational Databases

Recursive view definitions were introduced in the SQL-99 standard. Most modern relational database systems support them, and therefore can compute the transitive closure.

SQLite3 version 3.24.0
SQLite is a self-contained, public-domain, SQL database engine.

To improve performance, we used a temporary in-memory table for the `par`-relation. We added an index on the second column.

```
PRAGMA temp_store = MEMORY;
CREATE TEMP TABLE par (
  a INT NOT NULL, b INT NOT NULL,
  CONSTRAINT par_pk PRIMARY KEY (a, b)
) WITHOUT ROWID;
CREATE INDEX par_fb ON par (b);
```

Without the `temp_store` pragma, SQLite would use temporary files to evaluate the recursive query, even though the input data is in a transient in-memory database.

The query is (using a recursive "common table subexpression", i.e. a local view definition):

```
WITH RECURSIVE tc(a,b) AS (
  SELECT par.a, par.b from par
  UNION
  SELECT par.a, tc.b from par JOIN tc ON par.b = tc.a
) SELECT Count(*) FROM tc;
```

3.4 Graph Databases / RDF Triple Stores

Apache Jena version 3.7.0
Apache Jena is a Java Framework for linked data that supports querying RDF data via SPARQL. We ran Jena on OpenJDK 1.8 (Java 8) using Property Paths for the queries. To allow for deeper recursions we increased the JVM's thread stack size to 16MB using the environment variable `JVM_ARGS=-Xss16m` .

Graph databases and and RDF Triple Stores accept queries in the SPARQL format. SPARQL does not allow for general recursion [8] but SPARQL 1.1 added Property Paths [11] where non-variable predicates could be combined in a similar fashion to regular expressions. This allows us to write this query in a concise way:

```
SELECT (count(*) as ?resultcount) WHERE {?a :par+ ?b}
```

3.5 Execution Methodology and Hardware

We executed the benchmarks on a HP Blade server with two Intel Xeon CPUs E5-2630 v4@2.20GHz with 10 cores and 20 threads each. However, the current version of our program "BAM" does not use multiple threads (we are currently working on this). A Java program (such as Jena) always uses multithreading at least for garbage collection. The machine has 64 GB of RAM. The operating system is Debian `x86_64` GNU/Linux 8.10 (3.16.0).

The overall execution time ("elapsed wall clock time" and CPU time) and the memory ("maximum resident set size") for each test was measured with the Linux `/usr/bin/time` program. The time for loading the data and for executing the query are measured by functions of each system (as far as possible).

Most tests were run ten times and the time average values were calculated. We also checked for outliers, where the real time was much higher than the average. A few times in more than 5000 measurements, it seemed that the system was locked up for two minutes. The CPU time was normal, but the real time was unusally high. This occurred all on the same day. We repeated the measurements of that day.

In the comparison table (Fig. 2 below), we use real time. We believe that parallel execution should be honoured (there, CPU time is higher than real time). Apache Jena used on average 215% CPU (i.e. two parallel threads), DataScript used 133% CPU. For all other systems, the average CPU utilization was 97% to 100%, i.e. there was no big difference between real time and CPU time. However, there were a few measurements with only 50% CPU for XSB. These were very small benchmarks (waiting on the disk might explain the low CPU utilization).

4 Test Graphs

4.1 Random Graphs from the OpenRuleBench collection

The OpenRuleBench collection [6] uses a particular algorithm to generate random graphs that avoids a duplicate check for already selected edges. The domain $d = \{1, \dots, n\}$ is copied into separate lists a and b which are both extended with a special symbol $\circ$ so that their lengths are coprime, and then shuffled. A random graph contains edges from $(a_{(i \mod |a|)}, b_{(i \mod |b|)})$ where neither component is $\circ$. Selecting e tuples in order with increasing i (no larger than $|a| * |b|$) leads to all vertexes having nearly identical degrees, compared to just selecting edges at random. Cyclic graphs generated in this manner are usually strongly connected, even with low average node degree (about 1.5).

Additionally, the OpenRuleBench scripts generate graphs that are marked "nocyc" (non-cyclic). This is done by ordering the edges towards the greater node. Since this happens after the edges have already been selected, it leads to duplicate edges in the generated non-cyclic graphs. Furthermore, reflexive edges (loops) were not excluded in the settings of the distributed scripts.

We have rewritten the graph generation in the same spirit but for acyclic graphs we skipped reflexive edges and added a duplicate check in order to generate the correct amount of unique edges.

The cyclic and non-cyclic graphs are named $U_{n,e}$ and $F_{n,e}$ respectively (where n is the number of nodes, and e is the number of edges). The motivation for the letter U is the quite "uniform" degree, and F is the same with only "forward" edges. The parameter values for the test graphs are listed in Figure 1.

4.2 Additional Deterministic Graphs

Additionally, we created generators for graphs with properties that are easily analyzed in the context of the transitive closure problem. The number of edges in the graph is the input size of the problem. The transitive closure than computes the connected node pairs, i.e. the "reachability" relation. The diameter of the connected subgraphs is the maximum over the length of the shortest paths between any two connected nodes. This is the number of iterations until the fixed point of the T_p operator is found. We label nodes incrementally starting from 1.

- The complete graph K_n has n vertices and the diameter 1. The graphs are their own transitive closure with n^2 edges.
- The maximum acyclic graph A_n has n vertices and is a subset of K_n without the edges (a, b) with $a \geq b$. The family of graphs have the diameter 1 and the graphs themselves are their own transitive closure with $\frac{n(n-1)}{2}$ edges.
- The cycle graph C_n has n vertices and the diameter n. Its transitive closure is the complete graph K_n.
- The directed short-circuited cycle graph is a directed graph $S_{n,m}$, where n is divisible by $m + 1$, that is a superset of the directed cycle graph C_n where $m > 0$ additional edges per vertex are added, skipping $\frac{n}{m+1}$ nodes in the cycle. The diameter of the graph is $\frac{n}{m+1}$. The transitive closure of that family of graphs is the complete graph K_n. We give the set of edges $E = \{(i, (i + \frac{nt}{m+1}) \mod n) | t = 1, \ldots, m \wedge i = 1, \ldots, n\} \cup C_n$. One additional edge per vertex allows skipping $\frac{1}{2}$ of the circle, while two additional edges would allow skipping $\frac{1}{3}$ or $\frac{2}{3}$ of the circle.
- The graph P_n is a directed graph with n vertices that is the path containing the edges $E = \{(i, i + 1) | i = 1, \ldots, n - 1\}$. The diameter is $n - 1$. The transitive closure of the path P_n is A_n.
- The multi-path graph $M_{n,m}$ is a directed graph that contains n vertices and m pairwise vertex-disjoint paths of length $\frac{n}{m} - 1$. This is also the diameter of the connected subgraphs. The set of edges is $E = \{(i, i+m) | i = 1, \ldots, n-m\}$. The transitive closure of that family of graphs has $\frac{n}{2}(\frac{n}{m} - 1)$ edges.
- The binary tree graph B_h of height $h > 0$ is a directed graph that contains $2^h - 1$ vertices and $2^h - 2$ edges. The diameter of that graph is $h - 1$. The transitive closure of that family of graphs has $\sum_{i=1}^{h-1} i2^i = (h - 2) * 2^h + 2$ edges. The set of edges is $E = \{(i, 2i), (i, 2i + 1) | i = 1, \ldots, 2^{h-2} - 1\}$.

4.3 Cost Measure for Processing Time

Calculating the transitive closure of a graph takes time. We want to associate a given problem instance with some sort of cost in order to estimate how much

Graph	Nodes	Edges	In-Degree	Out-Deg.	Cyc.	TC Size	Iter.	Comp. Cost
k_1k	1000	1000000	1000–1000	1000–1000	yes	1000000	1	1001000000
k_2k	2000	4000000	2000–2000	2000–2000	yes	4000000	1	8004000000
a_1k	1000	499500	0–999	0–999	no	499500	1	166666500
a_2k	2000	1999000	0–1999	0–1999	no	1999000	1	1333333000
c_1k	1000	1000	1–1	1–1	yes	1000000	1000	1001000
c_2k	2000	2000	1–1	1–1	yes	4000000	2000	4002000
c_4k	4000	4000	1–1	1–1	yes	16000000	4000	16004000
s_2k_1	2000	4000	2–2	2–2	yes	4000000	1000	8004000
s_2k_2	2000	6000	3–3	3–3	yes	4000000	288	12006000
s_2k_3	2000	8000	4–4	4–4	yes	4000000	500	16008000
s_2k_4	2000	10000	5–5	5–5	yes	4000000	400	20010000
p_1k	1000	999	0–1	0–1	no	499500	999	499500
p_2k	2000	1999	0–1	0–1	no	1999000	1999	1999000
p_4k	4000	3999	0–1	0–1	no	7998000	3999	7998000
m_4ki_2	8192	8190	0–1	0–1	no	16773120	4095	16773120
m_1ki_8	8192	8184	0–1	0–1	no	4190208	1023	4190208
m_256_32	8192	8160	0–1	0–1	no	1044480	255	1044480
m_64_128	8192	8064	0–1	0–1	no	258048	63	258048
m_16_512	8192	7680	0–1	0–1	no	61440	15	61440
m_4_2ki	8192	6144	0–1	0–1	no	12288	3	12288
b_17	131071	131070	0–1	0–2	no	1966082	16	1966082
b_18	262143	262142	0–1	0–2	no	4194306	17	4194306
b_19	524287	524286	0–1	0–2	no	8912898	18	8912898
u_1k_50k	1000	50000	47–51	46–51	yes	1000000	3	50050000
u_1k_125k	1000	125000	122–127	122–127	yes	1000000	2	125125000
u_1k_250k	1000	250000	248–251	248–251	yes	1000000	2	250250000
u_2k_250k	2000	200000	98–101	98–101	yes	4000000	3	400200000
u_2k_500k	2000	500000	248–251	248–251	yes	4000000	2	1000500000
u_2k_1m	2000	1000000	499–502	499–502	yes	4000000	2	2001000000
f_1k_50k	1000	50000	0–101	0–101	no	472863	8	15421338
f_1k_125k	1000	125000	0–251	0–251	no	492170	5	40705777
f_1k_250k	1000	250000	0–500	0–501	no	497810	5	83073458
f_2k_250k	2000	200000	0–201	0–201	no	1946015	8	128081155
f_2k_500k	2000	500000	0–500	0–500	no	1985377	6	330379789
f_2k_1m	2000	1000000	0–1000	0–1001	no	1995412	4	665127921

Fig. 1. Data of the Test Graphs

time solving the problem will roughly take. The cost measure is independent of any actual implementation but corresponds to an abstract evaluation scheme.

Such a cost measure is an independent reference point for the problem size. Without such a cost measure, the runtimes of a system for different graphs would just be single numbers.

As a simple first try, we take the number of applicable rule instances, i.e. the number of rule instances where all body literals are contained in the minimal Herbrand model. This corresponds to the number of rule instances that the T_P operator will apply. Seminaive evaluation requires that each such rule instance is considered only once.

As an example, consider the complete graph K_{1000} with 1000 vertices and 1000 000 edges. The (non-recursive) starting rule

```
tc(X,Y) :- par(X,Y)
```

is applicable 1000 000 times (once for each of the input facts). For the rule

```
tc(X,Z) :- par(X,Y), tc(Y,Z)
```

there are 1000 000 facts in the `par` relation and each one has 1000 join partners in the `tc` relation. Thus the total size of the join `par(X,Y), tc(Y,Z)` is 1000 000 000. Of course, in this example, the second rule generates only duplicates, but they must be computed and eliminated. The associated costs for the rules are added and we get a final cost of 1001 000 000.

Note that this cost measure is purely declarative, and does not look at the sequence in which facts are computed. For the applicable rule instances, we only need the final version of the `tc`-relation in the minimal model.

This simple cost measure does not include non-successful join operations, or the actual number or size of arguments. Also the reading of the input data files is not explicitly considered, but for the transitive closure program, the cost of the first rule is the input size. However, one could consider different weights. Furthermore, many data structures for looking up tuples have runtime $O(\log(n))$.

For the practical approach we used PostgreSQL to check the test graphs by means of SQL queries on the `par` relation. The results are shown in Figure 1. The associated computation cost measure for the problem of the given size is listed in the last column.

Contrary to previous work [7], the number of iterations seems to have little or even no impact on the cost compared to the other factors in our cost measure. As we can see from the examples, a single large join can be much more costly than a large number of smaller joins. This is also reflected in the actual runtimes for most systems (comparing C_{1k} to K_{1k}).

In Section 5, we will investigate the relationship between this cost measure and the actual runtimes of various systems.

4.4 Cost Measure for Memory Usage

For memory usage, we simply take the number of facts in the minimal model as cost measure. For the transitive closure, this is the sum of the size of the `par` relation and the size of the `tc` relation. For the complete graph K_{1000}, the result is 2000 000. Simple solutions for duplicate elimination will keep all derived facts in memory, so this cost measure seems quite reasonable.

5 Analysis of the Results

5.1 Runtime Results

The average runtimes of the main systems for the various test graphs are shown in Fig. 2. We also give the factor compared to our own BAM prototype. E.g. a factor of 2 would mean that the system needs double as much time for computing the transitive closure of the graph.

Figure 3 shows the relation between the runtime and the cost measure for the graphs. Basically, for many systems the relationship between our cost measure and the runtime seems to be more or less linear. We were very positively surprised when we saw the graphs e.g. for XSB for the first time. At least, this shows that the cost measure is quite well correlated with the runtime.

However, in pure numbers it does not look quite as nice. We define the average processing speed of a system as the average processed cost over time. If a database system solves a problem that has an associated cost of 1000 in 10 seconds, the speed is $100\frac{cost}{s}$ (i.e. rule instances per second). To estimate runtimes we would like our cost measure to have a linear relation to the actual runtimes. For a perfect cost measure, the ratio of cost and runtime would be constant. However, we see already from Figure 2 that the factors between different systems vary between the graphs. So there cannot be a cost measure that uses only properties of the graph and completely accurately predicts the runtime with a fixed "processing speed" for each system. Figure 4 shows the resulting speed value for a selection of the input graphs.

For lower runtimes and cost measures the results vary widely, as might be expected. The speed is in general lower because it takes some time to start up, analyze and optimize the query, create caches, and so on. In particular, for Jena even a small graph takes 2 seconds. For larger inputs, the speed values do not vary so much. If one wants to reduce all the measurements to a single number for each system, an average speed still might be useful first indicator. Of course, this is a "lossy compression". The average over all graphs is shown in Figure 5.

From the speed and the cost measure of the input graph, one can compute an estimate for the runtime (cost divided by speed). If one tries to minimize the number of cases where the error is more than 2 s and more than 20% of the estimate, the average time might not be the best choice. We used the speed values shown in the second row of Figure 5 (the result of "manual optimization" with the total error of the limit violations as secondary optimization goal). For XSB, the estimate was outside the limits only for 4 (out of 35) graphs (and not very far outside). Thus, the XSB runtime is quite well predictable from our cost measure. For BAM, there are 6 errors. For other programs, the estimation does not work well. In case of Jena, the estimation formula probably should consider the large startup time of the JVM.

Graph	BAM	XSB	[Factor]	YAP	[Factor]	Jena	[Factor]	SQLite	[Factor]
k_1k	32.291	103.074	3.2	70.120	2.2	180.557	5.6	595.849	18.5
k_2k	363.309	810.679	2.2	552.711	1.5	1477.700	4.1	4983.400	13.7
a_1k	10.595	16.272	1.5	12.660	1.2	34.982	3.3	95.443	9.0
a_2k	130.801	127.694	1.0	122.537	0.9	255.690	2.0	811.345	6.2
c_1k	0.137	0.574	4.2	0.496	3.6	3.738	27.3	1.999	14.6
c_2k	0.449	1.514	3.4	2.110	4.7	8.098	18.0	7.816	17.4
c_4k	2.948	5.511	1.9	9.628	3.3	23.023	7.8	31.396	10.6
s_2k_1	0.554	1.765	3.2	3.090	5.6	9.347	16.9	9.900	17.9
s_2k_2	0.660	2.054	3.1	4.137	6.3	10.071	15.3	12.311	18.7
s_2k_3	0.804	2.337	2.9	4.607	5.7	10.615	13.2	14.030	17.5
s_2k_4	0.947	2.595	2.7	6.108	6.4	11.367	12.0	15.947	16.8
p_1k	0.081	0.344	4.2	0.184	2.3	2.885	35.6	0.998	12.3
p_2k	0.188	0.939	5.0	1.015	5.4	5.370	28.6	3.740	19.9
p_4k	0.758	3.055	4.0	4.341	5.7	13.797	18.2	15.016	19.8
m_4ki_2	3.656	6.167	1.7	9.240	2.5	24.628	6.7	32.829	9.0
m_1ki_8	0.331	1.833	5.5	2.149	6.5	8.954	27.1	7.688	23.2
m_256_32	0.126	0.726	5.8	0.691	5.5	3.978	31.6	1.920	15.2
m_64_128	0.110	0.240	2.2	0.193	1.8	2.663	24.2	0.629	5.7
m_16_512	0.010	0.130	13.0	0.057	5.7	2.290	229.0	0.206	20.6
m_4_2ki	0.000	0.106		0.047		2.065		0.113	
b_17	0.471	1.144	2.4	1.058	2.2	8.082	17.2	3.957	8.4
b_18	1.832	1.989	1.1	2.496	1.4	14.817	8.1	8.119	4.4
b_19	8.590	4.016	0.5	4.587	0.5	31.808	3.7	17.402	2.0
u_1k_50k	1.914	5.230	2.7	12.394	6.5	13.711	7.2	33.188	17.3
u_1k_125k	5.284	12.440	2.4	30.228	5.7	26.976	5.1	79.150	15.0
u_1k_250k	11.025	24.609	2.2	53.183	4.8	50.019	4.5	155.578	14.1
u_2k_250k	17.155	40.852	2.4	114.367	6.7	85.051	5.0	286.996	16.7
u_2k_500k	39.972	98.343	2.5	291.672	7.3	197.237	4.9	676.669	16.9
u_2k_1m	83.816	194.181	2.3	515.968	6.2	371.938	4.4	1303.429	15.6
f_1k_50k	0.710	1.825	2.6	2.442	3.4	6.631	9.3	9.727	13.7
f_1k_125k	1.867	4.118	2.2	7.084	3.8	11.791	6.3	24.343	13.0
f_1k_250k	3.623	8.390	2.3	17.066	4.7	20.468	5.6	48.579	13.4
f_2k_250k	6.536	12.854	2.0	17.031	2.6	31.899	4.9	85.702	13.1
f_2k_500k	16.727	32.402	1.9	55.599	3.3	72.170	4.3	213.121	12.7
f_2k_1m	34.841	65.082	1.9	142.702	4.1	138.465	4.0	417.273	12.0

Fig. 2. Run Time (Real Time in s) for TCFF Benchmark, Factor compared with BAM

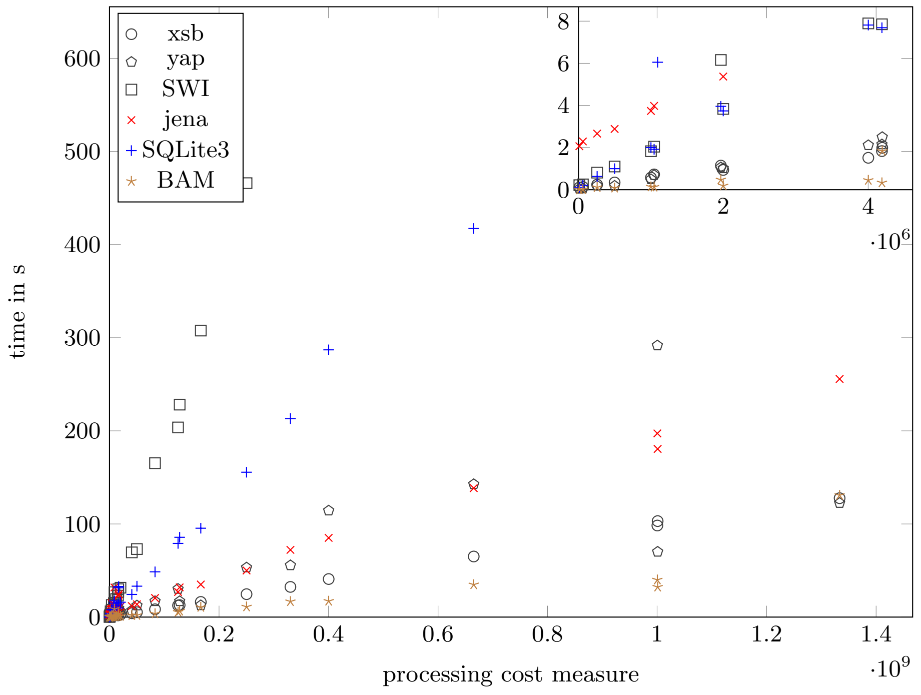


Fig. 3. Time vs cost measure

instance	p_2k	c_2k	b_18	s_ 2k_2	m_ 4ki_2	u_1k _50k	f_2k_ 250k	a_1k	f_2k_ 500k	u_2k_ 250k	f_2k _1m	k_1k	k_2k
cost in 10^6	2	4	4	12	17	50	128	167	330	400	665	1001	8004
BAM	10.63	8.91	2.29	18.19	4.59	26.15	19.60	15.73	19.75	23.33	19.09	31.00	22.03
XSB	2.13	2.64	2.11	5.85	2.72	9.57	9.96	10.24	10.20	9.80	10.22	9.71	9.87
YAP	1.97	1.90	1.68	2.90	1.82	4.04	7.52	13.17	5.94	3.50	4.66	14.28	14.48
SWI	0.52	0.51	0.32	0.62		0.69	0.56	0.54	0.47	0.59	0.47	0.51	0.50
Jena	0.37	0.49	0.28	1.19	0.68	3.65	4.02	4.76	4.58	4.71	4.80	5.54	5.42
SQLite3	0.53	0.51	0.52	0.98	0.51	1.51	1.49	1.75	1.55	1.39	1.59	1.68	1.61
DataScript	0.0004	0.0004	0.0259										

Fig. 4. System speed for selected graphs (10^6 rule instances/s)

System	BAM	XSB	YAP	Jena	SQLite
Avg. Speed (10^6 rule inst./s)	15.08	6.04	4.27	2.37	1.05
Speed for runtime estimation	19.90	8.31	4.13	4.41	1.39
Estimation errors (from 35)	6	4	11	22	12

Fig. 5. Average speed over all measured graphs, Runtime estimation

5.2 Memory Estimation

As we can see from Figure 6, our memory cost measure (number of facts) seems to have a relatively good correlation to the actually used memory for some systems.

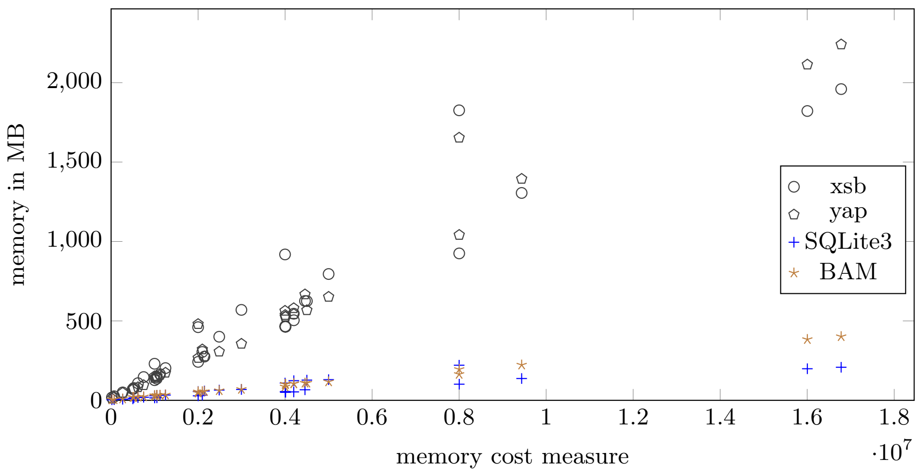


Fig. 6. Memory vs Memory Cost Measure

For Jena and SWI-Prolog, the memory cost measure does not seem to work well, therefore we removed them from the graph. In case of Jena, the memory management of the JVM with a garbage collector running at undefined times could explain the problems.

5.3 Additional Observations

The two problems K_{1k} (complete graph with 1000 edges) and $U_{2k,500k}$ (random graph with 2000 vertices and 500000 edges) have nearly identical cost (about 10^9) but for all systems the complete graph, with fewer vertices but more edges, was a bit faster to calculate than the cost measure would have suggested.

While the SPARQL query for Jena was quite expressive, the implementation for Property Paths was a problem for us. With the default settings we quickly had a stack overflow in the Java Virtual Machine for graphs that had a diameter of more than 2000. We changed some memory settings of the JVM to try to mitigate this problem.

DataScript was incredibly slow and often ran out of memory. Some benchmarks ran up to 40 hours when we terminated them. In the future we want to try smaller graphs to see whether our cost measure holds for problem sizes that are easier to handle for DataScript.

6 Conclusions

The source code, data files and single benchmarking results are available at

[http://dbs.informatik.uni-halle.de/rbench/]

We are working on integrating other systems (e.g., PostgreSQL and Soufflé) and more benchmarks into our scripts.

References

1. Brass, S., Stephan, H.: Bottom-up evaluation of Datalog: Preliminary report. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16/WFLP'16. pp. 13–26. No. 234 in EPTCS, Open Publishing Association (2017), https://arxiv.org/abs/1701.00623
2. Brass, S., Stephan, H.: Experiences with some benchmarks for deductive databases and implementations of bottom-up evaluation. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16/WFLP'16. pp. 57–72. No. 234 in EPTCS, Open Publishing Association (2017), https://arxiv.org/abs/1701.00627
3. Brass, S., Stephan, H.: Pipelined bottom-up evaluation of Datalog: The Push method. In: Petrenko, A.K., Voronkov, A. (eds.) Perspectives of System Informatics (PSI'17). LNCS, vol. 10742, pp. 43–58. Springer (2018), http://www.informatik.uni-halle.de/~brass/push/publ/psi17.pdf
4. Brass, S., Wenzel, M.: An abstract machine for Push bottom-up evaluation of Datalog. In: Hartmann, S., Ma, H., Hameurlain, A., Pernul, G., Wagner, R.R. (eds.) Database and Expert Systems Applications, 29th International Conference, DEXA 2018, Proceedings, Part II. LNCS, vol. 11030, pp. 270–280. Springer (2018)
5. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming 12(1–2), 5–34 (2012), https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2012-TPLP.pdf
6. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: An analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web (WWW'09). pp. 601–610. ACM (2009), http://rulebench.projects.semwebcentral.org/
7. Przymus, P., Boniewicz, A., Burzańska, M., Stencel, K.: Recursive query facilities in relational databases: A survey. In: Zhang, Y., Cuzzocrea, A., Ma, J., Chung, K., Arslan, T., Song, X. (eds.) Database Theory and Application, Bio-Science and Bio-Technology (DTA/BSBT 2010). pp. 89–99. No. 118 in Communications in Computer and Information Science, Springer (2010), http://www-users.mat.umk.pl/~eror/papers/dta-2010.pdf
8. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. In: Arenas, M., et al. (eds.) The Semantic Web — ISWC 2015, 14th International Semantic Web Conference, Proceedings, Part I. LNCS, vol. 9366, pp. 19–35. Springer (2015)
9. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Snodgrass, R.T., Winslett, M. (eds.) Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94). pp. 442–453 (1994), http://user.it.uu.se/~kostis/Papers/xsbddb.html
10. Schütz, H.: Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs). Ph.D. thesis, TU München (1993)
11. SPARQL 1.1 Query Language, W3C Recommendation (March 2013), http://www.w3.org/TR/sparql11-query/

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

Detecting Highly Overlapping Parts in Constraint Satisfaction Problems For Speed-Up

Sven Löffler, Ke Liu, and Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Germany
Sven.Loeffler@b-tu.de

Abstract. This paper describes a new approach on splitting constraint satisfaction problems (CSPs) by means of an auxiliary constraint satisfaction optimization problem (CSOP) that detects sub-CSPs with a potentially high number of conflicts. The purpose of this approach is to find sub-CSPs maximal in size which can be substituted by constraints with a higher consistency level or to be solved in parallel. This, eventually, often allows to significantly speed-up the solution process of a CSP.

Keywords: Constraint Programming · CSP · Refinement · Optimizations

1 Introduction

Constraint programming (CP) is a powerful method to model and solve NP-complete problems in a declarative way. Typical research problems in CP are rostering, graph coloring, optimization and satisfiability (SAT) problems [6].

Since the search space of CSPs is very big and the solution process often needs an extremely high amount of time we are always interested in improvement and optimization of the solution process. In general, a CSP is solved by interleaved backtrack search (often depth-first search) and propagation. If there is a fail in the search tree then a backtrack returns to the last step where another value assignment was possible. This means unnecessary work was done and all value assignments and propagations must be withdrawn by backtracking. With our approach we try to find constraints which are the origin of such fails so that we can remove or separate them (with other algorithms).

In this paper, we propose a way to model an auxiliary constraint satisfaction optimization problem (CSOP) to detect areas of a given CSP with a large number of overlapping constraints that are likely to cause fails during backtrack search. We see this approach as a preparation step for the substitution of sub-CSPs by new CSPs for speeding-up the solution process as described in [4] or for problem splitting (section 2.3.4 in [5]) in parallel constraint solving.

2 Preliminaries

In this section we introduce necessary definitions, methods and theoretical considerations which are the basic of our approach. We consider constraint satisfaction problems (CSPs), constraint satisfaction optimization problems (CSOP) and sub-CSPs which are defined as follows.

CSP [3] A constraint satisfaction problem (CSP) is defined as a 3-tuple $P = (X, D, C)$ with $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where D_i is the domain of x_i and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of primitive or global constraints covering between one and all variables in X.

CSOP [9,10] A constraint satisfaction optimization problem (CSOP) $P_{opt} = (X, D, C, f)$ is defined as a CSP with an optimization function f that maps each solution to a numerical value. The task in a CSOP is to find the solution tuple with the optimal value with regard to the optimization function f.

Sub-CSP Let $P = (X, D, C)$ be a CSP. For $C' \subseteq C$ we define $P_{sub} = (X', D', C')$ such that $X' = \bigcup_{c \in C'} vars(c)$ with corresponding domains $D' = \{D_i \mid x_i \in X'\} \subseteq D$.

2.1 Definitions and Methods

For the further concepts and methods we assume that a CSP $P = (X, D, C)$ is given.

In this paper, it is important to distinguish between local [2] and global consistency [3]. Local consistency guarantees that each value of a variable in the scope of a certain constraint is at least part of one of its solutions. A CSP is locally consistent if all of its constraints are locally consistent. In contrast, global consistency implies that each value of the variable of the CSP can be extended to at least one solution of the entire CSP. Therefore, global consistency is a much stronger enforcement of consistency. In particular, search interleaved with global consistency is backtrack free.

In our description we use the two functions $vars(c)$ and $cons(x)$, where the method *vars* has a constraint $c \in C$ as input and returns all variables $X' \subseteq X$ which are covered by this constraint. Analogously, the method *cons* has a variable $x \in X$ as input and returns all constraints $C' \subseteq C$ which cover this variable.

We say two constraints $c_i, c_j \in C, i \neq j$ *overlap* if the intersection of variables $vars(c_i)$ and $vars(c_j)$ is not empty.

We define the maximal size $size(P)$ of a CSP $P = (X, D, C)$ as the product of all cardinalities of the domains of the CSP P.

$$size(P) = \prod_{i=1}^{|X|} |D_i| \tag{1}$$

The size of a variable and of a constraint can be defined similar, i.e. $size(x_i) = |D_i|$ and $size(c) = \prod_{x_i \in vars(c)} |D_i|$.

2.2 Theoretical Consideration

For a given CSP P, we can separate the propagators of the constraints into two sets: the one set ensure local consistency for the constraints and the other one not (e.g. bound consistency). Examples for both categories based on their implementation in Choco Solver [7] are for locally consistent constraints: *arithm*, *count* or *regular* and for not locally consistent constraints: *cumulative* or *sum*.

Consider a CSP P with only one constraint c_1, which has a propagator that ensures local consistency, this implies that P must be backtracking free. The question is what leads to a not backtracking free CSP? If we add another constraint c_2, which has a propagator that ensures local consistency, to P then there are two possibilities.

Case 1: The two constraints c_1 and c_2 do not overlap. It is clear that this cannot lead to a fail because we can decompose such a CSP into to two separate CSPs P_1 and P_2, solving them individually and merge the results together. Hence, each solution of P_1 contains no value assignment for a variable of P_2 and vice versa, every element of the cross product of the solutions of P_1 and P_2 is a solution of P.

Case 2: The two constraints c_1 and c_2 overlap. Depending on different things such as the types of constraints c_1 and c_2, search strategy, propagation order, variable order etc. the CSP has a fail or not.

Thus, the origin for fails in a CSP P with only locally consistent constraints are overlapping constraints. This leads to the consideration that areas of the CSP P with a high number of overlapping constraints have the potential to be responsible for a large number of fails.

3 The Detection of Highly Overlapping Sub-CSPs

In this section, we present an approach to detect highly overlapping areas in a CSP P using a CSOP P_{opt}.

3.1 A CSOP to Find Maximal Overlapping Sub-CSPs

The idea behind this approach is to find sub-CSPs in P which are responsible for a large number of fails inside the backtrack search and small enough a) to solve them in parallel or b) to replace them by another CSP with less conflicts inside. We assume that a sub-CSP with a large number of overlapping constraints is likely to incur fails during backtrack search.

Solving the following CSOP P_{opt} leads to a sub-CSP P_{sub} of $P = (X, D, C)$ with $size(P_{sub})$ smaller or equal to a given value $subCspSize$ and a maximum number of overlapping variables.

Let $P_{opt} = (X', D', C', f)$ be a CSOP with:

$$X' = \{x_{ds,i}, x_{nc,i}, x_{ac,j} \mid \forall i \in \{1, ..., |X|\}, j \in \{1, ..., |C|\}\},$$
$$D' = \{D_{ds,i}, D_{nc,i}, D_{ac,j} \mid \forall i \in \{1, ..., |X|\}, j \in \{1, ..., |C|\},$$
$$\quad D_{ds,i} = \{1, |D_i|\},\ D_{nc,i} = \{0, ..., |cons(x_i)|\}, D_{ac,j} = \{0, 1\}\}$$

$C' = C_1 \cup C_2 \cup C_3 \cup C_4$ where
$C_1 = \{(x_{ds,i} = 1 \iff x_{nc,i} = 0) \mid \forall i \in \{1, ..., |X|\}\}$
$C_2 = \{(x_{nc,i} = \sum_{\forall j \in \{1,...,|C|\}, c_j \in cons(x_i)} x_{ac,j}) \mid \forall i \in \{1, ..., |X|\}\}$
$C_3 = \{(x_{ac,j} = 1 \iff x_{ds,i} = |D_i|) \mid \forall i \in \{1, ..., |X|\}, j \in \{1, ..., |C|\}$ where $x_i \in vars(c_j)\}$
$C_4 = \{(\prod_{i=1}^{|X|} x_{ds,i} \leq subCspSize)\}$
$\max f = (\sum_{i=1}^{|X|} x_{nc,i}) - |\{x \mid x \in \{x_{nc,1}, ..., x_{nc,|x|}\}, x \neq 0\}|$

There are three types of variables in P_{opt}. For each variable $x_i \in X$, two variables $x_{ds,i}$ and $x_{nc,i}$ are created and for each constraint $c_j \in C$ a binary variable $x_{ac,j}$ is created. The abbreviations ds, nc and ac are only names and stay for domain size, number of constraints and active constraint.

If the variable x_i is part of the sub-CSP then the size of the sub-CSP is multiplied with the domain size of x_i, otherwise the size of P_{sub} is unchanged. Thus, the domain $D_{ds,i}$ of variable $x_{ds,i}$ contains the values 1 and $|D_i|$ (1 as a neutral element of the multiplication). The product of all $x_{ds,i}, i \in \{1, ..., |X|\}$ is then the size of P_{sub}. The constraint in C_4 guarantees that the resulting sub-CSP P_{sub} has a size $size(P_{sub})$ smaller or equal to a given value $subCspSize$.

The variable $x_{nc,i}$ represents the number of constraints which are part of the sub-CSP P_{sub} and cover the corresponding variable x_i. Hence the possible values of $x_{nc,i}$ are between 0 and the number of constraints over x_i. Only variables $x \in X$ which are covered by a constraint of P_{sub} are part of the sub-CSP. The constraints in C_1 ensure this. If the number of constraints of a variable x_i in P_{sub} is null ($x_{nc,i} = 0$) then it has no influence on the size of P_{sub} (so $x_{ds,i}$ must be the neutral element of the multiplication) and vice versa.

Each variable $x_{ac,j}$ corresponds to a constraint $c_j \in C$ in a way, that if the constraint c_j is part of P_{sub} then the value of $x_{ac,j}$ is 1, else it is 0. From the descriptions of the variables $x_{nc,i}$ and $x_{ac,j}$ follows that the sum of all $x_{ac,j}$ where $x_{ac,j}$ corresponds to a constraint c_j which is part of P_{sub} and covers x_i must be equal to $x_{nc,i}$, which is realized by the constraints in C_2.

The constraints in C_3 guarantee that if a constraint $c_j \in C$ is part of P_{sub} (corresponding to $x_{ac,j} = 1$) then also all variables which are covered by this constraint ($x_i \in vars(c_j)$, corresponding to $x_{ds,i} = |D_i|$) must be part of the sub-CSP and vice versa.

The goal of P_{opt} is to find the sub-CSP with the most overlaps. This is expressed by the objective function f. The variable $x_{nc,i} \forall i \in \{1, ..., |x|\}$ represents the number of constraints of the variable $x_i \in X$ in P_{sub}. The number of overlapping constraints for x_i in P_{sub} is the value of $x_{nc,i}$ reduced by one if the value is greater than null or null otherwise. To count the number of overlaps correctly we must subtract one for each variable $x_{nc,i}$ which is not null. The objective function f is modeled exact like this.

A solution of P_{opt} can be transformed into a sub-CSP in the following way. All variables $x_i \in X$, where the corresponding variable $x_{ds,i}$ is not set to value 1 are element of the sub-CSP. Analogously all constraints $c_j \in C$ are part of the sub-CSP, where the corresponding variable $c_{ac,j}$ is set to 1.

3.2 Remarks on the CSOP

Using our approach to solve the CSP P, we do the following: 1) generating and solving CSOP P_{opt} to find highly overlapping sub-CSPs, 2) replacing there by equivalent sub-CSPs, which can be solved more efficiently (e.g. with less fails in the search), 3) solving the newly generated CSP P'. Thus, solving the COSP P_{opt} should not be too complex or time consuming computed to the original CSP P. Consider the following remarks:

1. The depth of the search tree can be much smaller then the number of variables of P_{opt}. The reason for this is that after setting all binary variables $c_{ac,j} | \forall j \in \{1, ..., |C|\}$ to a value (1 or 0) and propagating them, all other variables are also instantiated. Each variable $x_{nc,i}$ is instantiated by the constraints of C_2 and each variable $x_{ds,i}$ is then instantiated by the constraints in C_1. So it makes sense to use a search strategy which searches only variables in the set of $c_{ac,j} | \forall j \in \{1, ..., |C|\}$.
2. A first possible (not perfect) solution of P_{opt} can be given directly by setting all $c_{ac,j} | \forall j \in \{1, ..., |C|\}$ to null. While the size of the sub-CSP is smaller or equal *subCspSize* one constraint after the other can be added to P_{sub}. This solution can be used as a starting point for local search.
3. Portfolios like explained in [1] can be used which often leads to good speed-ups in optimization problems.
4. It is not necessary to find a perfect solution. Often a very good solution can be found in a fraction of the time needed to find the best solution.
5. For solving sub-CSPs in parallel more than one sub-CSP is necessary. For this we can extend our CSOP in a way that we add for each sub-CSP we want to find again a set of variables $X'' = \{x_{ds,i}, x_{nc,i}, x_{ac,j} \mid \forall i \in \{1, ..., |X|\}, j \in \{1, ..., |C|\}\}$ with domains D'' and constraints C'' analogously to D' and C'. Then we only need to add *count* constraints over the variables $x_{ac,j}$ of all sub-CSPs which only allows one value one in all variables $x_{ac,j}$ with index j from the different sub-CSPs. Such constraints guarantee that each constraint of the given CSP P is only associated to one sub-CSP. The new objective function is then the sum of the objective functions of all sub-CSPs.

Currently, the approach is specialized to find sub-CSPs with a high density of overlapping constraints which are responsible for a high number of fails in the search tree. This is especially useful for the substitution of the constraints of a sub-CSP with only one constraint which has a propagator that ensure local consistency like the `table` or `regular` constraint. Doing this leads to a high reduction of fails in the search tree of the original CSP.

A second approach is to use the detected sub-CSPs as work load balancing decompositions for parallel consistency [8] techniques. Parallel consistency enforcement encloses the problem of combining partial solutions of sub-CSPs into one solution of the complete CSP. Using our approach to find sub-CSPs with a high number of overlapping constraints inside each sub-CSP but with a low number of constraints between different sub-CSPs can be a promising approach. There we could apply a different objective function, which searches for the lowest number of overlaps between different sub-CSPs instead.

4 Conclusion

Summary and Conclusion. We presented a new approach for the decomposition of CSPs into sub-CSPs using a CSOP. It was presented how a CSOP can be modeled to find decompositions with a high number of constraint overlaps inside. We explained two possibilities how the decomposed sub-CSPs can be used to improve the solving process of the original CSP. One posibility is the substitution of the constraints of a sub-CSP by a locally consistent constraint to reduce the number of fails in the original CSP significantly. The other posibility is using the detected sub-CSPs as work balancing decomposition for parallel consistency methods.

Future work. In the future we will do benchmark tests to verify the influence of our optimizations on real problems. Besides, we want to investigate the idea of transformations of sub-CSPs into locally consistent regular constraints as discussed in [4]. Finally, we want to find a criterion, when to apply the approach.

References

1. Amadini, R.: Portfolio approaches in constraint programming. Constraints **20**(4), 483 (2015). https://doi.org/10.1007/s10601-015-9215-9, https://doi.org/10.1007/s10601-015-9215-9
2. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
3. Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)
4. Löffler, S., Liu, K., Hofstedt, P.: The power of regular constraints in csps. In: Eibl, M., Gaedke, M. (eds.) INFORMATIK 2017. pp. 603–614. Gesellschaft für Informatik, Bonn (2017)
5. Malapert, A., Régin, J., Rezgui, M.: Embarrassingly parallel search in constraint programming. J. Artif. Intell. Res. **57**, 421–464 (2016). https://doi.org/10.1613/jair.5247, https://doi.org/10.1613/jair.5247
6. Marriott, K.: Programming with Constraints - An Introduction. MIT Press, Cambridge (1998)
7. Prud'homme, C., Fages, J.G., Lorca, X.: Choco documentation (2018), http://www.choco-solver.org, last visited 2018-05-04
8. Rolf, C.C., Kuchcinski, K.: Parallel consistency in constraint programming. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes. pp. 638–644 (2009)
9. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam, First edn. (2006)
10. Tsang, E.P.K.: Foundations of constraint satisfaction. Computation in cognitive science, Academic Press (1993)

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

Toward a concept of derivation for Prolog

Marija Kulaš

FernUniversität Hagen, Wissensbasierte Systeme, 58084 Hagen, Germany
kulas.marija@online.de

Abstract. In logic programming, derivations are dependant on two kinds of implicit parameters: *history* of a derivation step, and (in case of implementation) also *algorithms*, which restrict the freedom of derivation and generate backtracking. On that basis, we propose a concept of derivation including backtracking and aggregation of steps.

Keywords: backtracking · big-step derivation · compositionality · built-in predicate · cut

1 Introduction

This work is motivated by the wish to prove that an operational model of (slightly enriched) pure Prolog is adequate, in the sense of reflecting pure Prolog computation in a sound and complete way. Faced with such a task, the authors of an aspiring operational model X for (a subset of) Prolog have three ready options. They may deem X so straightforward that its adequacy seems obvious (e.g. for pure Prolog with cut [8], for full Prolog both [3] and our previous approach [14]). If they feel uncomfortable with that, they can prove something, but what would be "enough"? For example, it may be proved that any logical consequence of the program, and nothing else, can be derived in X [25]; this leaves out operational aspects such as the order of answers. Since the advent of ISO Standard Prolog, there is a third option: to brave its complex definition [7] and hopefully show correspondence with X, hereby contributing X to the pool of trusted models [24]. What we now suggest is a fourth option: to try adapting the traditional concept of SLD-derivation to the basic restrictions of an implementation. From there, correspondence with the model X may be easier to prove.

SLD-derivation in the context of pure Prolog: too much freedom, too little structure? An SLD-derivation represents a proof attempt in Horn clause logic (*HCL*). Thus, it is a suitable basis for claims about HCL, but less so if implementations like Prolog are concerned. Yet, it is still used as a de facto model of Prolog computation. This poses two problems: first, backtracking cannot be expressed. Second, a major impediment to proving claims about SLD-derivations is the abundance of freedom, both in resolution order (choice of an atom from a query, and of a program clause from a predicate definition), and in variable substitution (choice of mgu and choice of renaming).

Fixing the order of resolutions as in Prolog means that a derivation now must represent a depth-first traversal of "an" ordered SLD-tree for the given query.

Which leaves the other half of freedom: it is not just one tree but infinitely many, obtained by choosing different mgus and renamings of program clauses. This has been approached by complex notions like "equivalence modulo enhanced variance" [4]. In case of HCL, they may be necessary, but in case of implementations we suggest a more pragmatic approach: to acknowledge that those choices are *also* fixed by algorithms. So we still do not know how the variables are going to be substituted, but we know that it will be in one of a fairly small number of practical ways, obeying certain properties. For example, "reasonable" unification algorithms are compatible with renaming and produce "relevant" mgus, which allows constructive versions of the *variant lemma* [17]. Such formal claims about logic programming systems or their compilation are still few and far between, a notable exception being [22].

From the standpoint of meta-reasoning, SLD-derivation appears homogeneous, it lacks grouping of steps ("big step"). Here we argue that by observing another somewhat neglected parameter of implemented HCL (history of "spent" variables), as well as the "residual" query, a kind of modularity emerges.

Adding utilities to pure Prolog: Another important aspect of implemented HCL are *utilities*, usually called *built-in predicates*. It would be good to have a way of accomodating them in a relatively intuitive yet formal way. Obvious candidates are disjunction, explicit unification and truth values. Also, utilities like "cut" could profit from a concise definition.

1.1 Overview of the paper

For the most part we concentrate on a slight enhancement of pure Prolog called Pure*. We start with an instance of HCL called HCL*, its syntax and proof method (Section 2). In Section 3 we consider restrictions due to implementation, turning HCL* into a programming language, Pure*, and propose a small-step concept of top-level derivation for Pure*, including backtracking.

Next we attempt a big-step concept, based on *query context* (Section 4). In Subsection 4.1, backtracking is captured by means of *residual*, enabling a definition of the n-th answer. In Subsection 4.2, the lack of transitivity for the path relation and the lack of compositionality are compensated using *preface* of "spent" variables, so that computing a conjunction can be done piecewise (Theorem 28). A variant lemma for big-step Pure* derivations is shown (Theorem 30).

Finally, Section 5 outlines how further utilities like "cut" may be added.

1.2 Notation

In Prolog, everything is a "term", and so shall term be here the topmost syntactic concept. *Term*s are built in the usual way starting from two disjoint sets: a countably infinite set $\boldsymbol{V}$ of *variable*s and a set $\boldsymbol{Fun}$ of *functor*s. Associated with every functor f is at least one n denoting its possible number of arguments, *arity*. For disambiguation, the notation f/n may be used, which is also the *outline* of the term $f(t_1, ..., t_n)$. If s occurs within t, we write $s \mathrel{\mathring{\in}} t$. The set of variables in t

is *Vars*(t). A sequence of terms enclosed in brackets like $[t_1, ..., t_n]$ is a *list*. If r is obtained from s by replacing its variables in order of appearance with $t_1, ..., t_n$, we write $r = s[t_1, ..., t_n]$.

A *substitution* σ is a mapping of variables on terms with finitely many non-identity pairs, which provide its *core representation*: $\sigma = \begin{pmatrix} x_1 & ... & x_n \\ \sigma(x_1) & ... & \sigma(x_n) \end{pmatrix}$. It is extended in the functor-preserving way on all terms, and $\sigma(t)$ is called an *instance* of t. The identity mapping () is also denoted ε. A *renaming* ρ is a substitution represented by a permutation, and $\rho(t)$ is a *variant* of t. A substitution θ is *at least as general* as σ, if there is δ with $\sigma = \delta \cdot \theta$.

If there is a substitution θ with $\theta(s) = \theta(t)$, then θ is a *unifier* of the equation $E := (s{=}t)$. It is *relevant*, if $\mathit{Vars}(\theta) \subseteq \mathit{Vars}(E)$. It is a *most general unifier* (*mgu*) of E, if it is at least as general as any other; the set of all mgus of E is *MguSet*(E).

Limit cases of term: *nothing* or *void*, written as $\square$, *anything*, written as _, and *impossibility*, written as $\bot$. In a few places we make reference to *(logical) formula*, taken in the sense of classical (first-order) logic as in e.g. [16].

2 Modifying HCL toward HCL* and beyond

2.1 The syntax

In the course of enriching HCL with ever more "utilities" (or "system procedures"), like explicit disjunction or unification, the original syntax of HCL was stretched to accomodate those as well. As a result, mathematical meaning of "predicate" (logical relation) and "predication" (atomic formula, i.e. a formula not structured by "logical connectives") has been blurred.

Some pitfalls and dilemmas in the process of merging the terminology of mathematical logic ("predicate/connective") with that of programming languages ("procedure/control") are reflected in Prolog standardization documents as well. For example, looking up the notion of "predication" reveals: in the main text of [7] and in [5, p. 9], "predication" may be any term but variable or number; in the appendix of [7] this is further restricted by excluding conjunction, disjunction and implication (`'->'`); reverse implication (`':-'`) is not excluded, so a whole clause is *also* a predication. Regarding types of utilities, [7] differentiates between "control construct" (ten utilities including the logical connectives of conjunction, disjunction and implication, but also `true` and `fail`) and "built-in predicate"; [5] regards all of the 112 listed utilities as "built-in predicates".

Indeed, there does not seem to be an easy way to merge utilities into logic. As a compromise, we suggest to consider only conjunction and clause-building as connectives, and all other reserved functors as "built-in predicates". More formally, assume a set $\boldsymbol{P} \subseteq \boldsymbol{Fun}$ whose members shall be called *predicate*s. A term whose outline is a predicate shall be called a *predication*. Apart from predicates, ***Fun*** includes two *logical connective*s, conjunction and reversed implication, ***Str*** $:= \{$`','`$/2,$ `':-'`$/2\}$. A term with outline in ***Str*** is a *(logically) structured term*. Hence, a predication is (logically) unstructured, so it is also called a *(logical) atom*. With this understanding of connectives, any utility we might add, like disjunction, shall be seen merely as *restricting* the choice of definable atoms.

Definition 1 (HCL*). *The set of* utilities *(or* built-in predicates*) of HCL* is* $\boldsymbol{BiP}^* := \{$`true`/0, `fail`/0, `true`/1, `'='`/2, `';'`/2$\}$. *A* utility atom *(*u-atom*) is an atom whose outline is in* $\boldsymbol{BiP}^*$*. Any other atom is a* definable atom *(*d-atom*). A* HCL* program *is a conjunction of program clauses from Figure 1.*

$$
\begin{aligned}
\langle \mathit{HCL}^* \ \mathit{clause} \rangle &::= \langle \mathit{program\ clause} \rangle \mid \langle \mathit{query} \rangle \\
\langle \mathit{program\ clause} \rangle &::= \langle \mathit{d\text{-}atom} \rangle\texttt{.} \mid \langle \mathit{d\text{-}atom} \rangle \ \texttt{:-} \ \langle \mathit{qf} \rangle\texttt{.} \\
\langle \mathit{query} \rangle &::= \langle \mathit{qf} \rangle \\
\langle \mathit{qf} \rangle &::= \langle \mathit{atom} \rangle \mid \langle \mathit{atom} \rangle\texttt{,} \langle \mathit{qf} \rangle \\
\langle \mathit{atom} \rangle &::= \langle \mathit{d\text{-}atom} \rangle \mid \texttt{true} \mid \texttt{fail} \mid \texttt{true(}\langle \mathit{qf} \rangle\texttt{)} \mid \langle \mathit{term} \rangle \texttt{=} \langle \mathit{term} \rangle \mid \langle \mathit{qf} \rangle\texttt{;} \langle \mathit{qf} \rangle
\end{aligned}
$$

Fig. 1. Syntax of HCL*

Alongside the standard predicate `true`/0, we include a version with one dummy argument, `true`/1 (from an old Quintus Prolog package `true.pl` [23]). Our use for `true(t)` is to hold the variables of the term t, similarly to the *epsoid* ε_t in [12].

Following [2], for the sake of readability we define SLD-derivation in a "positive" manner, using "queries" and not their negations ("goal clauses"). In a pragmatical deviation from the original meaning of clauses in HCL, we treat program clauses as *fixed terms*, having their variables chosen by the programmer. As a visual help, program clauses shall be written with a hat, like $\hat{\mathcal{K}}$.

2.2 The proof method

Can utilities be handled with the inference rule of HCL, known as *SLD-resolution* [10]? This inference rule corresponds to a simple procedural reading of a clause P `:-` $Q, R, ..., S$ in a logic program $\mathit{\Pi}$: To answer a query G that is unifiable with P by an mgu σ, it suffices to answer $\sigma(Q)$, $\sigma(R)$, ..., and $\sigma(S)$. If the premiss is void, G is immediately answered, by substituting its variables according to σ (an "answer"). It can be shown (Subsection A.1) that, in case there is nothing left to answer, the mgus $\sigma_1, ..., \sigma_n$ employed satisfy[1] $\models \forall[\mathit{\Pi}] \to \forall[\sigma_n \cdot ... \cdot \sigma_1(P)]$.

Such procedural reading of program clauses is still in effect in Standard Prolog, but its correspondence with SLD-resolution is broken, because some of the utilities go beyond the classical logic. For example, the utility `var`/1 treats variables not as placeholders (as it should, in logic) but as terms in their own right, so from the successful computation of `var(X)`, whose answer substitution just renames `X`, cannot be deduced $\forall x$ `var(`x`)`. But many utilities, including the ones in HCL*, can be seen as relations in the sense of classical logic.

To resolve utilities, two questions must be dealt with. First, can utilities (as possibly structured formulas, like disjunction or implication) participate in a resolution? Second, how would they be defined?

[1] Assumed is some basic knowledge of "valid" implications $\models A \to B$ in classical logic (see e.g. [18, Ch. 2]). The *universal closure* $\forall[F]$ of a formula F consists of F prefixed with $\forall x$ for every x "free" in F. Similarly for $\exists[F]$.

Participation would be less of a problem: SLD-resolution rule actually holds in a more general form than the one employed in HCL. It can derive from $\forall[\neg(A' \wedge B)]$ and $\forall[A'' \vee \neg C]$ the formula $\forall[\neg(\sigma(C) \wedge \sigma(B))]$, if A', A'', B, C are quantifier-free formulas and exists $\sigma \in \mathit{MguSet}(A'{=}A'')$. Recall that in HCL A', A'' are d-atoms and B, C are conjunctions of d-atoms, but there is nothing to preclude u-atoms (like disjunction).

Regarding utility definition, any computable function over any Herbrand universe is definable by Horn clauses over the same universe [1]. So at least in theory many experiments are imaginable. Some of them are easy and well-known, like emulating truth values, unification and disjunction (Listing 1.1).

```
true.   true(_).   X=X.   X;Y :- X.   X;Y :- Y.
```

Listing 1.1. Embedding HCL* in HCL

To accomodate utilities, we separate the two parts of resolution: the replacing of a query atom A' with $\sigma(C)$, and the joining of the replacement with the (now suitably instantiated) rest of the query, $\sigma(B)$. Let us call the first part *reduction*. For d-atoms, it will be defined as expected:

Definition 2 (reduction of definable atom). *Let A be a d-atom. It can be* reduced *to $\sigma(B)$ with* score *$\mathcal{K}{:}\sigma$, written as $A \triangleright_{\mathcal{K}:\sigma} \sigma(B)$, if $\mathcal{K} = (H\,\texttt{:-}B)$ is a variant of a program clause $\hat{\mathcal{K}}$, such that $\sigma \in \mathit{MguSet}(H{=}A)$. A score can be applied on a term by $\_{:}\sigma(t) := \sigma(t)$.*

For u-atoms we *pre-compile* program clauses into dedicated reduction rules, using a unique variable-free term *placebo clause*, denoted κ, as well as the *placebo score*, $\phi := \kappa{:}\varepsilon$.

Definition 3 (reduction of utility atom). *Dedicated rules of HCL* are*

- $\texttt{true} \triangleright_\phi \Box$, *as well as* $\texttt{true(\_)} \triangleright_\phi \Box$
- $X\texttt{=}Y \triangleright_{\kappa:\sigma} \Box$, *if* $\sigma \in \mathit{MguSet}(X{=}Y)$
- $X\texttt{;}Y \triangleright_\phi X$ *and* $X\texttt{;}Y \triangleright_\phi Y$

Based on reduction, we now define SLD*-resolution as an extension of SLD-resolution. The main difference is using the full potential of the SLD-resolution rule, i.e. dropping the restriction on d-atoms. Syntactically, reduction of an atom is now an explicit operation. Also, we do not restrict the composition of mgus on query variables, unlike [16,2] whose "(computed) answer (substitution)" we denote $\mathit{cas}_{\mathcal{L}}$. As usual, assumed is a rule for *selecting* an atom from a sequence of atoms, possibly relying on the history of the computation.

Definition 4 (SLD*-resolution and derivation). *Let A be an atom and M, N be queries. If A is selected from M, A, N and for some s, B holds $A \triangleright_{\mathsf{s}} B$, we say that $\mathsf{s}(M), B, \mathsf{s}(N)$ is an* SLD*-resolvent *of M, A, N with score s, written as $M, A, N \hookrightarrow\!\!\triangleright_{\mathsf{s}} \mathsf{s}(M), B, \mathsf{s}(N)$.*

Given a query G, an SLD*-derivation *$\mathcal{D}$ of G is a chain of SLD*-resolutions $G \hookrightarrow\!\!\triangleright_{\mathsf{s}_1} G_1 \hookrightarrow\!\!\triangleright_{\mathsf{s}_2} \ldots$. In each $\mathsf{s}_n = \mathcal{K}_n{:}\sigma_n$ must $\mathcal{K}_n$ be variable-disjoint (i.e.*

standardized apart*) from the* current history*,* $\mathcal{D}_{[n-1]} := G \hookrightarrow\!\!\!\triangleright_{\mathsf{s}_1} \ldots \hookrightarrow\!\!\!\triangleright_{\mathsf{s}_{n-1}} G_{n-1}$*, whose variables* $Vars((G, \mathsf{s}_1, ..., \mathsf{s}_{n-1}))$ *are regarded as* spent *at step n.*

If $\mathcal{D}$ *has n steps, it can be abbreviated as* $G \hookrightarrow\!\!\!\triangleright^n_{\mathsf{S}} G_n$*. Here* $\mathsf{S} := \mathsf{s}_1 + ... + \mathsf{s}_n$ *is the* (cumulative) score *of* $\mathcal{D}$*, from which* input clause*s,* $CList(\mathsf{S}) := [\mathcal{K}_n, ..., \mathcal{K}_1]$*, and mgus,* $SList(\mathsf{S}) := [\sigma_n, ..., \sigma_1]$*, can be extracted.*

By composing the mgus in the score of $\mathcal{D}$ *we obtain the* partial answer *for G by* $\mathcal{D}$*,* $Subst(\mathsf{S}) := \sigma_n \cdot ... \cdot \sigma_1$*. A final partial answer, whenever* $G_n = \square$*, is an (*complete*)* answer *for G. In that case G is said to* succeed*. The effect of a score* S *on a term t is given by* $\mathsf{S}(t) := Subst(\mathsf{S})(t)$*.*

Remark 5 (dual meaning of "derivation"). "Derivation" is here understood both as a *relation* between queries (logical consequence w.r.t. program in the sense of Subsection A.1), and as a *syntactic object* i.e. *term* (a sequence of queries linked with scores). Hence, we may say "let $A \hookrightarrow\!\!\!\triangleright^* B$ hold" as well as "let $\mathcal{D} := A \hookrightarrow\!\!\!\triangleright^* B$ with no C within". Similarly for all later kinds of derivation.

To visualize the search for a successful SLD*-derivation of a query, *SLD*-tree*s are used, defined analogously to SLD-trees. Each point with more than one descendant is a *choice point.* Any point labeled $\square$ is a *success point.* If one SLD-tree for a query is successful, then any one is [2, p. 70]. Hence, if one finite SLD*-tree for G is unsuccessful, then any one is, so we say G has *(finitely) failed.*

To systematically search for a success point, *backtracking* is used: If the search reaches a point without descendants, then it goes back to the last choice point, where a new choice shall be tried.

3 From HCL* to Pure*: Restrictions by implementation

Despite Prolog standard, there seems to be no consensus on a definition of pure Prolog [9], but since we need some kind of understanding, let us try this:

Definition 6 (pure Prolog, Pure*). *Pure Prolog is a programming language implementing HCL and obeying the following restrictions:*

(R-var) *variable substitution is fixed: mgu is calculated by a fixed algorithm* ***U****, and standardizing apart is calculated by a fixed algorithm* ***S***

(R-ord) *reduction order is fixed: it respects the ordering of atoms in a query, and the ordering of program clauses within a predicate definition.*

Similarly, Pure* implements HCL* and obeys the same restrictions, plus:

(R-ord*) *reduction rules for each utility are ordered.*

Every implementation of HCL* obeying (R-var), hence any Prolog, is parametrized by the two algorithms, ***U*** and ***S***.

3.1 Two "reasonable" algorithms

The examples in this text use $\boldsymbol{U} := \mathsf{MM}$ and $\boldsymbol{S} := \mathsf{NT}$ [13]. The algorithm MM is the deterministic version of Martelli-Montanari algorithm that uses sequences instead of sets and picks the leftmost equation eligible for a rule application. It produces relevant mgus, i.e. MM is *relevant*. Let us now define NT. Standardizing-apart is a special case of (relatively) fresh renaming:

Definition 7 (fresh renaming). *A binary function New is a* fresh-renaming algorithm*, if for any s, t holds: $New_s(t)$ is a variant of t variable-disjoint with s. To enable accumulation of spent variables, we allow an optional parameter in superscript, so New^{+r} is defined as $New_s^{+r}(t) := New_{r,s}(t)$.*

In theoretical work, the existence of such algorithms is usually enough, established by the well-known renaming device of Lloyd [16, p. 41]: For the n-th derivation step, the original program clause variables are indexed with n, assuming that top-level queries may not contain indices. This assumption, however, rules out *resuming* of a derivation, i.e. starting from a resolvent, which is needed for proofs involving modularity. As a remedy, instead of the whole $\mathcal{D}$ solely the *variables* of $\mathcal{D}$ can be taken as a parameter of the algorithm.

So we shall say that a fresh-renaming algorithm *New* is *flat*, if it depends only on the spent variables, i.e. $New_s(t) = New_{Vars(s)}(t)$. This is satisfied e.g. if we rename only where necessary:

Definition 8 (thrifty fresh renaming). *The algorithm NT renames a variable x apart from s as follows. If x appears in s, together with its indexed versions $x_1, ..., x_k$, but x_{k+1} does not, then $\mathsf{NT}_s(x) := x_{k+1}$. Otherwise, $\mathsf{NT}_s(x) := x$.*

3.2 Ordered reduction and a single tree

Adding the restrictions (R-ord) and (R-ord*) means that the order of resolvents for a query is determined. To formalize this, we need a new concept of derivation step, based on *ordered reduction*, which shall take over from "$\hookrightarrow\!\!\triangleright$".

Definition 9 (n-th program clause, n-th utility reduction). *If a predicate p/k is defined by m program clauses, we enumerate them in the order of appearance as $Clause(p/k, n)$, $n = 1, ..., m$. We also write $Clause(G, n)$, where G may be any atom with outline p/k. Similarly for utilities: in case of more than one reduction rule for a utility, they are ordered as $\triangleright^1, \triangleright^2, \ldots$.*

So for disjunction we would have $X;Y \triangleright_\phi{}^1\, X$ and $X;Y \triangleright_\phi{}^2\, Y$.

To reduce a d-atom, we may need some fresh variables, therefore we must avoid re-using already spent variables, contained within a term P.

Definition 10 (n-th reduction). *Let n be a number, P a term or void, and A an atom. In case A is a d-atom, it has the n-*th reduction *to C with score* s apart from *P, written as $A \triangleright_{\mathsf{s}}{}^{n,P}\, C$, if $\mathcal{K} = (H\,{:}\!-B) := \boldsymbol{S}_P(Clause(A, n))$ and if exists $\sigma := \boldsymbol{U}(H{=}A)$, giving $\mathsf{s} := \mathcal{K}{:}\sigma$ and $C := \sigma(B)$. In case A is a u-atom, no fresh variables are needed, so we set $\triangleright^{n,P} := \triangleright^n$.*

In terms of SLD*-trees, the restrictions amount to having just one SLD*-tree per query and P, and searching it depth-first. A single tree deserves a name:

Definition 11 (*LD**-tree). *Let G be a query or void, and P be a term or void. The* derivation tree *for G* apart from *P, denoted as $LD^*{}_P(G)$, is the SLD*-tree for G determined by (R-var) with $\boldsymbol{S}^{+P}$, (R-ord) and (R-ord*).*

Observe that here again we allow some variables (in P) to be already spent. Clearly, G is finitely failed iff $LD^*{}_\_(G)$ is finite and without a success point.

Example 12. Assume $\boldsymbol{U} := \mathsf{MM}$, $\boldsymbol{S} := \mathsf{NT}$ and the program

```
alt(X) :- p(X), q(X).            % $\hat{\mathcal{K}}_{a1}$
alt(X) :- s(X).                  % $\hat{\mathcal{K}}_{a2}$
p(a). p(b). q(b). t(1). t(3).    % $\hat{\mathcal{K}}_{p1}$, $\hat{\mathcal{K}}_{p2}$, $\hat{\mathcal{K}}_{q}$, $\hat{\mathcal{K}}_{t1}$, $\hat{\mathcal{K}}_{t2}$
```

The tree $LD^*{}_\Box$((`alt(X),r(X)`)) is shown in Figure 2. Here $\mathsf{s}_{a1} = \hat{\mathcal{K}}_{a1}[X_1]\colon \binom{X_1}{X}$, $\mathsf{s}_{a2} = \hat{\mathcal{K}}_{a2}[X_2]\colon \binom{X_2}{X}$, $\mathsf{s}_{p1} = \hat{\mathcal{K}}_{p1}\colon \binom{X}{\mathtt{a}}$, $\mathsf{s}_{p2} = \hat{\mathcal{K}}_{p2}\colon \binom{X}{\mathtt{b}}$ and $\mathsf{s}_q = \hat{\mathcal{K}}_q\colon\varepsilon$.

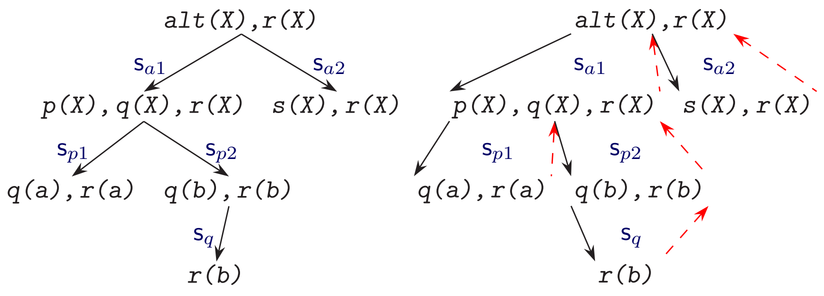


Fig. 2. *LD**-tree, on the right with backtracking

3.3 Top-level computation

Based on n-th reduction, we now define top-level Pure* computation of G. Intuitively, it is the left-to-right, depth-first path from the root of $LD^*(G)$ until possibly the first success point is encountered. Choices for G now being ordered, this computation must be deterministic. Its linear rendering as a sequence of "nodes" shall be achieved with additional devices:

– negative scores, where $-\mathsf{s}$ means to cancel a choice with score s
– meta-functor *Back*, where *Back* G means to look for the next choice for G
– meta-functor *Top*, as the parent of every top-level query

A derivation for G shall start as $Top \rightharpoondown_{\Phi} G$. As it proceeds, reduction scores accumulate in S, some of them in cancelled form. Even those are important if we want to know which variables are already spent, hence $CList(\mathsf{S})$ is again defined to gather all clauses from S, disregarding cancellation.

However, for the partial answer only *non-cancelled* scores in S are important, so we need something like *net-value* of a score, and *zero score.*

Definition 13 (zero score, net-value). *The symbol Φ is called* zero score. *It is neutral for addition: $SList(\Phi) := []$, $CList(\Phi) := []$, $Subst(\Phi) := \bot$.*

The net-value *of a (cumulative) score S, $Net(\mathsf{S})$, is computed by leaving out cancelled scores, $Net(\mathsf{T}+\mathsf{s}+(-\mathsf{s})+\mathsf{U}) := Net(\mathsf{T}+\mathsf{U})$, where s is a reduction score and T, U are (possibly void) sub-sums of S. It also obeys $Net(\Box) = Net(\Phi) := \Phi$.*

The partial answer by S is thus $\overline{\mathsf{S}} := Subst(Net(\mathsf{S}))$. The list of promising clause*s in S, i.e. clauses from non-cancelled scores, is $CList(Net(\mathsf{S}))$.*

Definition 14 (node). *If G is a query, then $Back\ G$ is a* revisited query. *A* node *is a query or a revisited query. To provide for the limits, a node can also be $\Box$, Top or $Back\ Top$. We abbreviate "G or $Back\ G$" as $(Back)G$.*

Definition 15 (Pure* derivation, top level). *Let G be a Pure* query. The* (top-level) Pure* derivation *for G in 0 steps is $Top \multimap_{\Phi} G$. Assume $n \geq 1$ and let $\mathcal{D}$ be the derivation for G in $n-1$ steps ending with G_{n-1}, with $G_0 := G$. If G_{n-1} is of the form $\Box$ or $Back\ Top$, no further step is possible.*

*Otherwise, G_{n-1} is of the form $(Back)A, R$ where A is an atom and R may be void. Assume the most recent reduction of A as the starting atom of $(Back)A, R$ was m-th (in case of none, set $m := 0$). The n-*th step *is determined as follows:*

1. *If there is a next reduction for A, i.e. for some minimal $k > 0$ exists $m+k$-th reduction apart from $\mathcal{D}$ as $A \rhd_{\mathsf{s}}{}^{m+k,\mathcal{D}}\ C$, then $(Back)A, R \multimap_{\mathsf{s}} C, \mathsf{s}(R)$.*
2. *Else, $(Back)A, R \multimap_{-\mathsf{s}} Back\ B$, where $(Back)B \multimap_{\mathsf{s}} A, R$ is the most recent step with non-negative score leading to A, R.*

   We call B the parent query *of this occurrence of A, R: $Parent_{n-1}\{A, R\} := B$, and the node $(Back)B$ the* parent node *of $G_{n-1} = (Back)A, R$.*

Remark 16 (occurrence-based). Mostly we drop n in $Parent_n\{G\}$ for better readability. Observe, however, that $Parent\{G\}$ always maps a *fixed occurrence* of G (determined in the text) to its unique parent in the derivation.

The *(top-level) Pure* computation* for G is the maximal Pure* derivation for G, as usual. Figure 3 shows the computation of `alt(X),r(X)` for the program in Example 12. It is a linear rendering of $LD^*{}_{\Box}($`alt(X),r(X)`$)$ from Figure 2.

Assuming the computation of G is finite, how could it have ended? By definition, if $G \multimap^* \Box$, the computation must stop. Also, if there is no choice for resolving G, then we readily obtain $G \multimap Back\ Top$. But it can also happen that there are choices for G, yet none leads to $\Box$. Here we obtain $G \multimap_{\mathsf{s}} H \multimap \ldots \multimap_{-\mathsf{s}} Back\ G \multimap^* Back\ Top$. The pattern $\multimap G \multimap^* Back\ Parent\{G\}$ we call a *dead-end branch*[2] [3], i.e. a chain of derivation steps that clearly signifies finite failure of G. In Figure 3, steps 2-3, 5-6, 4-7, 1-8, 9-10 and 0-11 are such branches.

The next claim states that backtracking happens stackwise: before we backtrack on a parent, we backtrack on all of its children.

[2] Here we exclude the starting node $(Back)Parent\{G\}$, to enable neat discarding of dead-end branches.

[3] By Remark 16, for a given occurrence of G is $G \multimap^+ Back\ Parent\{G\}$ unique.

derivation step			aspects of cumulative score S:	
nr.	score	query	promising input clauses	part. answer
		Top	$CList(Net(\mathsf{S}))$	$\overline{\mathsf{S}}$
0.	$-\!\!\triangleright_{\Phi}$	`alt(X),r(X)`	$[]$	$\perp$
1.	$-\!\!\triangleright_{\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}}$	`p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
2.	$-\!\!\triangleright_{\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	`q(a),r(a)`	$[\hat{\mathcal{K}}_{p1}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{a} \cdot \binom{X_1}{X}$
3.	$-\!\!\triangleright_{-\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	*Back* `p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
4.	$-\!\!\triangleright_{\hat{\mathcal{K}}_{p2}:\binom{X}{b}}$	`q(b),r(b)`	$[\hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{b} \cdot \binom{X_1}{X}$
5.	$-\!\!\triangleright_{\hat{\mathcal{K}}_{q}:\varepsilon}$	`r(b)`	$[\hat{\mathcal{K}}_{q}, \hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\varepsilon \cdot \binom{X}{b} \cdot \binom{X_1}{X}$
6.	$-\!\!\triangleright_{-\hat{\mathcal{K}}_{q}:\varepsilon}$	*Back* `q(b),r(b)`	$[\hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{b} \cdot \binom{X_1}{X}$
7.	$-\!\!\triangleright_{-\hat{\mathcal{K}}_{p2}:\binom{X}{b}}$	*Back* `p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
8.	$-\!\!\triangleright_{-\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}}$	*Back* `alt(X),r(X)`	$[]$	$\perp$
9.	$-\!\!\triangleright_{\hat{\mathcal{K}}_{a2}[X_2]:\binom{X_2}{X}}$	`s(X),r(X)`	$[\hat{\mathcal{K}}_{a2}[X_2]]$	$\binom{X_2}{X}$
10.	$-\!\!\triangleright_{-\hat{\mathcal{K}}_{a2}[X_2]:\binom{X_2}{X}}$	*Back* `alt(X),r(X)`	$[]$	$\perp$
11.	$-\!\!\triangleright_{-\Phi}$	*Back Top*	$[]$	$\perp$

Fig. 3. Linear rendering of the LD*-tree in Figure 2

Lemma 17 (failure, zero sum). *For a query G, if $-\!\!\triangleright_{\mathsf{s}} G -\!\!\triangleright^*_{\mathsf{T}}$ Back Parent$\{G\}$, then Net$(\mathsf{s} + \mathsf{T}) = \Phi$ and G fails finitely in HCL*.*

Knowing the boundaries of a computation enables us to define subnodes: If $A -\!\!\triangleright^* B$ such that no *Back Parent*$\{A\}$ is within, we say B is a *subnode* of A.

Now let us see how to recognize success (as defined in HCL*) in a Pure* derivation as well. By discarding of dead-end branches, we obtain

Lemma 18 (success). *For a query G, if $G -\!\!\triangleright^*_{\mathsf{S}} \square$, then $G \hookrightarrow\!\!\triangleright^*_{Net(\mathsf{S})} \square$, i.e. G succeeds in HCL*.*

Remark 19 (past is now included). In contrast to SLD*-derivations, a Pure* derivation does not consist only of "promising" resolvents, but also of "dead-end" resolvents: anything that was tried (and possibly failed) along the way to the current query. All of it determines the length of derivation and spent variables, necessary for proofs involving Pure* derivations.

A variant lemma for Pure* can be stated even more simply than the one for HCL$_{\boldsymbol{U}}$ from [12, Sec. 6]: Pure* derivations of the same length which start from variant queries have variant nodes and variant cumulative scores.

4 Toward big-step derivation

As seen in Section 3, implementing a logic programming language removes many sources of non-determinism. Must we now see Pure* computation of a query as unique ("the computation"), or still as one of many? We advocate the latter view, justified by a somewhat neglected further source of non-determinism: the *context*.

At first glance, computing a Pure* query G should be "the same" independent of what preceded it and what is about to follow. Yet this is not quite so:

- Without queries to follow G, no backtracking on G shall be attempted. Otherwise, more than one answer for G is possible, i.e. shorter or longer computations of G.
- Standardizing apart depends on a pool of available variables, so if there were queries preceding (or following) G, their variables are not available: they are "spent". Thus, computations of G may differ in fresh variables.

4.1 Backtracking cause: residual

The top-level computation of G (Definition 15) stops in case of finite failure, or the first answer. To try to capture the search for the "next" answer, i.e. backtracking on G, we have to consider G not alone but as part of some conjunction G, R. In other words, we need a "residual" query R coming after G. Without any such, there would be no need to ever[4] go back on G.

Normally, a *residual* is just a query, but to include top-level computation we allow it to be void. Term-or-void shall be abbreviated as *term*$^\circ$, and query-or-void as *query*$^\circ$.

Unlike the (first and only) answer for G in a top-level derivation, the n-th answer for G upon R is computed intermittently, in a sequence of "spells": after answering G, the residual is going to be computed, and if it fails, another answer for G shall be sought, and so on.

Remark 20 (border condition). On a technical level, the precise moment in which G succeeded (important for the score) is now not so obvious: $G, R \mathrel{-\!\!\triangleright}^* \sigma(R)$ would not be a sufficient criterion, because of possible loops $\theta(R) \mathrel{-\!\!\triangleright}^* \sigma(R)$. One solution is to require *maximal* derivation *not* containing an instance of R inside.

Definition 21 (initial spell). *Let G be a query and R be a query$^\circ$. The* initial spell *of G upon R is the maximal derivation starting with G, R and not including as inner node an instance of R, or Back Parent$\{G, R\}$.*

Assume the initial spell of G upon R was finite, with score S. By Definition 4 and Lemma 18, if the spell ends with an instance of R, then it must be $\mathsf{S}(R)$ and *Net*$(\mathsf{S}) \neq \Phi$. Otherwise the spell ends in *Back Parent*$\{G, R\}$ and by Lemma 17 holds *Net*$(\mathsf{S}) = \Phi$.

Definition 22 ($n+1$-th spell). *Let $n \geq 1$. Assume the n-th spell of G upon R was finite, with score S_n and Net$(\mathsf{S}_n) \neq \Phi$. If $\mathsf{S}_n(R)$ fails, then $G, R \mathrel{-\!\!\triangleright}^* \mathsf{S}_n(R) \mathrel{-\!\!\triangleright}^*$ Back Parent$\{\mathsf{S}_n(R)\}$. In that case, there is $n+1$-*th spell *of G upon R: the maximal subderivation starting from Back Parent$\{\mathsf{S}_n(R)\}$ and not including an instance of R, or Back Parent$\{G, R\}$. The* score *of a finite spell is the cumulative score from Top.*

[4] In a Prolog top-level loop, the user's reaction can be thought of as the residual.

A Pure* *computation* of G is a chronological sequence of all spells of G upon R, for some residual R. A *derivation* is an initial fragment of a computation, as usual. In Figure 3, there are two success spells of *`p(X)`* upon *`q(X),r(X)`* followed by exhaustion: step 2, steps 3-4 and steps 7-8.

Definition 23 (n-th answer). *Assume the n-th spell of G upon R is finite with score S_n. If $Net(\mathsf{S}_n) \neq \Phi$, we say G* succeeded upon *residual R with the* n-th score S_n*, written as $G \setminus^n_{\mathsf{S}_n} R$, and the* n-th (complete) answer $\overline{\mathsf{S}}_n$*.*

The justification for calling $\overline{\mathsf{S}}_n$ an answer for G is given by

Lemma 24 (all answers). *If $G \setminus^n_{\mathsf{S}}$ `fail`, then $G \hookrightarrow^*_{Net(\mathsf{S})} \square$. Conversely, if $G \hookrightarrow^*_{\mathsf{T}} \square$, then for some n and S holds $G \setminus^n_{\mathsf{S}}$ `fail` and $\mathsf{T} = Net(\mathsf{S})$.*

For different R, just the fresh variables may vary: If $G \setminus^n_{\mathsf{S}_1} R_1$ and $G \setminus^n_{\mathsf{S}_2} R_2$, then $\mathsf{S}_1(G) = \mathsf{S}_2(G)$.

Traditionally, universal termination of G is defined as finiteness of $LD^*(G)$. An equivalent definition would be: G terminates universally if and only if G, *`fail`* terminates [19]. This is another natural use for residuals.

Definition 25 (termination, n-finiteness). *If G (finitely) fails or succeeds, we say it* terminates*. If there is a maximal $n \geq 1$ such that $G \setminus^n R$ for some R, we say G is* n-finite*. If G fails, it shall be called* 0-finite*. Any n-finite query is said to* terminate universally*, or to be* exhaustible*.*

In Example 12, *`alt(X),r(X)`* finitely failed (0-finite), but *`alt(X)`* upon *`r(X)`* is 1-finite with the complete answer $\binom{X_1\ X}{b\ \ b}$, and *`p(X)`* upon *`q(X),r(X)`* is 2-finite with the complete answers $\binom{X}{a}$ and $\binom{X}{b}$.

4.2 Modularity: impact of spent variables

For proving properties of a formal model of Pure*, it would be good to have a kind of modularity of Pure* derivations. In particular, two aspects are of interest to us: first, could a derivation be resumed, and second, could a conjunction be computed in a piecemeal fashion?

Regarding resumability, it may come as a surprise that, even in HCL* with relevant mgus, the path relation is not transitive: If $A \hookrightarrow^*_{\mathsf{S}} B$ and $B \hookrightarrow^*_{\mathsf{T}} C$, it would be expected that also $A \hookrightarrow^*_{\mathsf{S}+\mathsf{T}} C$, or at least $A \hookrightarrow^* C$. However, this does not always hold [5], as the following program shows:

```
p(X) :- q(Y).   % K̂1
q(Y) :- r(X).   % K̂2
```

Here *`p(X)`* $\hookrightarrow_{\hat{\mathcal{K}}_1[X_1,Y]:\binom{X_1}{X}}$ *`q(Y)`* and *`q(Y)`* $\hookrightarrow_{\hat{\mathcal{K}}_2[X,Y_1]:\binom{Y_1}{Y}}$ *`r(X)`*, although *`p(X)`* $\hookrightarrow^*$ *`r(X)`* is not possible, due to standardizing-apart and assumed relevance of mgus.

Clearly, the reason for non-transitivity is that a derivation step has an *implicit parameter*, the current history. By making it explicit, and choosing $\boldsymbol{S}$ to be flat, we obtain a replacement for transitivity:

[5] The variant lemma [17] ensures only that a variant of C must be reached.

Lemma 26 (resumability). *Assume $\boldsymbol{S}$ to be flat. For any queries A, B, C and any term$^\circ$ P satisfying $Vars((P, B)) = Vars((A, \mathsf{S}))$ holds: If $A \rightarrowtriangle^*_{\mathsf{S}} B$ and* `true(P)`*$, B \rightarrowtriangle^+_{\phi+\mathsf{T}} C$, then $A \rightarrowtriangle^*_{\mathsf{S}+\mathsf{T}} C$.*

Regarding compositionality, in case of equations there is an iteration property [2, p. 39] enabling us to compute an mgu ϕ of E', E'' by computing an mgu σ for E', followed by an mgu θ for $\sigma(E'')$, and setting $\phi := \theta \cdot \sigma$ (this also holds for "the" mgu computed by MM).

In case of arbitrary queries, such an iteration property does not hold for either complete answer or $cas_{\mathcal{L}}$. Assuming relevance of $\boldsymbol{U}$, it holds for $cas_{\mathcal{L}}$ (which is a consequence of [2, Theorem 3.25]). Assuming flatness of $\boldsymbol{S}$ and accomodating some history again, it holds for complete answer, as stated in Theorem 28.

Definition 27 (preface). *Let P be a term$^\circ$. We abbreviate* `true(P)`*$, G \setminus^n_{\phi+\mathsf{S}} R$ as $P \mathbin{\backslash\!\!\backslash} G \setminus^n_{\mathsf{S}} R$, and say that P is a* preface *for computing G.*

Theorem 28 (iteration for complete answer). *Let $\boldsymbol{S}$ be flat. Let A be a query, B, R be queries$^\circ$, and P be a term$^\circ$. Then:*

1. *If $P \mathbin{\backslash\!\!\backslash} A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$, then also $P \mathbin{\backslash\!\!\backslash} A, B \setminus_{\mathsf{S}+\mathsf{T}} R$.*
2. *Conversely, if $P \mathbin{\backslash\!\!\backslash} A, B \setminus_{\mathsf{U}} R$, then there are S, T such that $P \mathbin{\backslash\!\!\backslash} A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$ and $\mathsf{U} = \mathsf{S} + \mathsf{T}$.*

Using big-step and partitions, even the hitherto neglected order of answers for a conjunction can be expressed (without proof):

Lemma 29 (n-th success of conjunction). *Let $\boldsymbol{S}$ be flat. Upon R, the query A, B has the n-th answer $\overline{\mathsf{U}}_n$, in other words $A, B \setminus^n_{\mathsf{U}_n} R$, iff there are natural numbers p and $m_1, ..., m_p$ and $n_1, ..., n_p$ such that: $0 = m_0 < m_1 < ... < m_p$, $n_1 + ... + n_p = n$, and also*

1. *A has at least m_p answers: For $j = 1, ..., m_p$ and some S_j holds $A \setminus^j_{\mathsf{S}_j} B, R$.*
2. *Some of them suit B: For $i = 1, ..., p$, $\mathsf{S}_j(B)$ fails whenever $m_{i-1} < j < m_i$, but for $k = 1, ..., n_i$ and some $\mathsf{T}_{i,k}$ holds $A, \mathsf{S}_{m_i} \mathbin{\backslash\!\!\backslash} \mathsf{S}_{m_i}(B) \setminus^k_{\mathsf{T}_{i,k}} \mathsf{S}_{m_i}(R)$.*
3. *All but last n_i must be maximal, i.e. $\mathsf{S}_{m_i}(B)$ is n_i-finite for $i = 1, ..., p-1$.*

Finally, for $i = 1, ..., p$ and $k = 1, ..., n_i$ holds $\mathsf{U}_{n_1+...+n_{i-1}+k} = \mathsf{S}_{m_i} + \mathsf{T}_{i,k}$.

We conclude with a variant lemma for Pure* derivations, outlined at the close of the previous section (p. 10), now rephrased in big-step manner.

Theorem 30 (variant, big-step). *Assume a renaming-compatible and relevant $\boldsymbol{U}$. Let α be a prenaming[6] with $D(\alpha) = Vars((P, Q, R))$. If $P \mathbin{\backslash\!\!\backslash} Q \setminus_{\mathsf{S}} R$, then $\alpha(P) \mathbin{\backslash\!\!\backslash} \alpha(Q) \setminus_{(\alpha \uplus \lambda)(\mathsf{S})} \alpha(R)$, where λ is the prenaming of input clauses.*

[6] A *prenaming* α is a variable-pure substitution mapping elements of a "relaxed" core $D(\alpha) \supseteq Dom(\alpha)$ to mutually distinct variables [12].

5 Adding utilities that modify backtracking

Tampering with backtracking, as in the case of "cut" utility *'!'*/0 or the utility pair *catch*/3 and *throw*/1, can also be interpreted using customized reduction, plus one or two special-case rules for *Back*. Here we suggest[7] a way to emulate cut using backtracking steps, and in Subsection A.2 catch/throw is addressed.

Upon execution, the cut succeeds like *true*/0, but upon re-execution it prunes the LD*-tree. This can be emulated by its reduction to void and an additional rule for *Back*. The rule handles the special case of revisiting a query starting with cut, and shall in that case replace the default rules 1 and 2 from Definition 15.

$$\begin{aligned} \texttt{!} &\rhd_\phi \square \\ \mathit{Back}\ \texttt{!,}R &\rightarrowtriangle_{-\mathsf{S}} \mathit{Back}\ \mathit{Parent}\{\mathit{CutParent}\{\texttt{!,}R\}\} \end{aligned}$$

The *cut-parent* of a node containing cut is the most recent[8] node whose resolution produced this occurrence of cut in a "transparent" way, i.e. not embedded in *call*/1 or some other "opaque" non-symbolic functor, as opposed to symbols like *','*/2 or *';'*/2. This corresponds to a heuristics by O'Keefe [21] ("cuts can't see through letters"). To provide for cuts on top level, we set $\mathit{Parent}\{\mathit{Top}\} := \mathit{Top}$. Finally, S is the score of the subderivation from the parent node of $\mathit{CutParent}\{\texttt{!,}R\}$ to the current $\mathit{Back}\ \texttt{!,}R$.

Example 31 (cut). If we change $\hat{\mathcal{K}}_{a1}$ in Example 12 to

```
alt(X) :- p(X), !, q(X).  % new K̂a1
```

we obtain the computation in Figure 4, with $\mathsf{S} = \hat{\mathcal{K}}_{t1}\colon\binom{Y}{1} + \hat{\mathcal{K}}_{a1}[X_1]\colon\binom{X_1}{X} + \hat{\mathcal{K}}_{p1}\colon\binom{X}{\mathsf{a}} + \phi - \phi$. Notice that the second clause for *alt*/1 shall not be tried.

$$\begin{array}{ll}
 & \texttt{t(Y), alt(X), r(X)} \\
\rightarrowtriangle_{\hat{\mathcal{K}}_{t1}:\binom{Y}{1}} & \texttt{alt(X), r(X)} \\
\rightarrowtriangle_{\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}} & \texttt{p(X), !, q(X), r(X)} \\
\rightarrowtriangle_{\hat{\mathcal{K}}_{p1}:\binom{X}{a}} & \texttt{!, q(a), r(a)} \\
\rightarrowtriangle_{\phi} & \texttt{q(a), r(a)} \\
\rightarrowtriangle_{-\phi} & \mathit{Back}\ \texttt{!, q(a), r(a)} \quad \textit{\% cut-parent: } \texttt{alt(X), r(X)} \\
\rightarrowtriangle_{-\mathsf{S}} & \mathit{Back}\ \texttt{t(Y), alt(X), r(X)} \\
\rightarrowtriangle_{\hat{\mathcal{K}}_{t2}:\binom{Y}{3}} & \dots
\end{array}$$

Fig. 4. Emulating cut

Having cut enables us to emulate several more utilities:

$$\begin{aligned} \texttt{once(}G\texttt{)} &\rhd_\phi G\texttt{,!} \\ \mathit{If}\texttt{->}\mathit{Then}\texttt{;}\mathit{Else} &\rhd_\phi \texttt{call(}\mathit{If}\texttt{),!,}\mathit{Then}\texttt{;}\mathit{Else} \\ \mathit{If}\texttt{->}\mathit{Then} &\rhd_\phi \texttt{call(}\mathit{If}\texttt{),!,}\mathit{Then}\texttt{;fail} \\ \backslash\texttt{+}G &\rhd_\phi \texttt{call(}G\texttt{),!,fail;true} \end{aligned}$$

[7] These ideas are still somewhat speculative, not having been run through a prototype.
[8] "the most recent" is needed for *If*->*Then*;*Else*

Observe that we do not regard *If*`->`*Then*`;`*Else* as a special case of disjunction. Rather, we regard it as a ternary functor. This should ensure proper backtracking (i.e., no fallback on default rules for disjunction).

Standard-abiding scoping of cut for *If*`->`*Then*`;`*Else* [5] disturbs the heuristics of O'Keefe in that *If* must be opaque for cut, unlike *Then* or *Else*. In other words, for cut in *If* the cut-parent is more recent, `call(`*If*`),!,`*Then*`,`*R*. This is ensured by enveloping *If* in `call`/1, emulated simply by `call(`*G*`)` $\triangleright_\phi$ *G*.

6 Related work

It would appear that adapting the concept of SLD-derivation to suit Prolog has not been tried much. Using *Back* in linear rendering of backtracking may be reminiscent of *decision literal* in transition rules for SAT solving [20, Sec. 2]. The impression of not much previous interest extends to "big-step" concepts, like "prefix" of spent variables, and their suitable representation[9]. As to "residual" of remaining queries, it is nothing else than *success continuation* [15], and hence a natural part of a Prolog semantics. Mostly it occurs in literature without a specific name, e.g. while explaining control flow [5, p. 86]. In [4], residual is modeled by "conjunction of two derivations". Unlike concepts of Prolog derivation, semantics of cut has been an active topic. More recent approaches include an operational semantics of pruning LD-trees [6] and an approximative denotational semantics that treats cut in a contextual manner [11].

7 Summary

A way to add logical utilities to pure Prolog and still remain in resolution logic is shown. To this aim, the concept of *logical structure* is suitably restricted, so that "atom" (logically unstructured term whose outline is a predicate) can be a utility. The programming language obtained by adding truth values, explicit unification and explicit disjunction to pure Prolog is called *Pure**.

To represent Pure* computation, we start from the traditional concept of SLD-derivation and add backtracking steps necessary for implemented logic. A big-step concept of derivation is also suggested, revealing a kind of inherent modularity. The concept is based on *context of the query*, which is two-fold, consisting of the prequel to the query (*preface*) and its sequel (*residual*). Residual is the cause of backtracking. Preface provides history of computation. With context, not only *top-level* but also *sub*-computation can be expressed. This bridges the gap between SLD-derivation, seemingly devoid of structure, and compositional formal models. As an aside, it is outlined how the concept of backtracking steps can accomodate "cut" and other utilities that affect backtracking.

Acknowledgement

Many thanks are due to anonymous referees for their helpful critique.

[9] The notation $A \setminus B$ is already used for lambda-abstraction (λProlog by Nadathur and Miller, αProlog by J. Cheney).

References

1. Andréka, H., Németi, I.: The generalised completeness of Horn predicate-logic as a programming language. Acta Cybernetica **4**(1), 3–10 (1978)
2. Apt, K.R.: From logic programming to Prolog. Prentice Hall (1997)
3. Börger, E., Rosenzweig, D.: A mathematical definition of full Prolog. Sci. of Computer Programming **24**(3), 249–286 (1995)
4. Comini, M., Meo, M.C.: Compositionality properties of SLD-derivations. Theor. Comp. Sci. **211**, 275–309 (1999)
5. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog: The standard (reference manual). Springer-Verlag (1996)
6. Drabent, W.: Proving completeness of logic programs with the cut. Formal Aspects of Computing **29**, 155–172 (2017)
7. ISO/IEC JTC 1/SC 22: ISO/IEC 13211-1-1995. Information technology - Programming languages - Prolog - Part 1: General core (1995), `https://www.iso.org/standard/21413.html`
8. Jones, N.D., Mycroft, A.: Stepwise development of operational and denotational semantics for Prolog. In: Proc. of the ISLP'84. pp. 281–288. Atlantic City (1984)
9. Kifer, M.: What is "pure Prolog" (2005), `https://www.w3.org/2005/rules/wg/wiki/Pure_Prolog.html`
10. Kowalski, R.A.: Predicate logic as programming language. In: Rosenfeld, J. (ed.) Information Processing 74. pp. 569–574. North-Holland (1974)
11. Kriener, J., King, A.: Semantics for Prolog with cut – revisited. In: Codish, M., Sumii, E. (eds.) Proc. FLOPS'14, LNCS, vol. 8475, pp. 270–284. Springer (2014)
12. Kulaš, M.: A practical view on renaming. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16 and WFLP'16. EPTCS, vol. 234, pp. 27–41 (2017)
13. Kulaš, M.: On separation, conservation and unification. Tech. Rep. IB 378-06/2019, FernUniversität in Hagen (2019). https://doi.org/10.18445/20190618-134958-0
14. Kulaš, M., Beierle, C.: Defining Standard Prolog in rewriting logic. In: Futatsugi, K. (ed.) Proc. of WRLA'00. ENTCS, vol. 36, pp. 158–174. Elsevier (2001)
15. Lindgren, T.: A continuation-passing style for Prolog. In: Proc. SLP'94. pp. 603–617 (1994)
16. Lloyd, J.W.: Foundations of logic programming. Springer-Verlag, 2. edn. (1987)
17. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Logic Programming **11**(3-4), 217–242 (1991)
18. Manna, Z.: Mathematical theory of computation. McGraw-Hill (1974)
19. Neumerkel, U.: Teaching Prolog and CLP. In: Tutorial notes of ICLP'97 (1997)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories. Journal of the ACM **53**(6), 937–977 (2006)
21. O'Keefe, R.A.: The Craft of Prolog. The MIT Press (1990)
22. Pusch, C.: Verification of compiler correctness for the WAM. In: Proc. TPHOLs. LNCS, vol. 1125, pp. 347–361. Springer Berlin Heidelberg (1996)
23. Quintus Corp., Palo Alto, CA: Quintus Prolog language and library (1991), release 3.1. Also on `http://quintus.sics.se/isl/quintus/html/quintus`
24. Ströder, T., et al.: A linear operational semantics for termination and complexity analysis of ISO Prolog. In: Proc. LOPSTR'11. LNCS, vol. 7225, pp. 237–252 (2012)
25. Tobermann, G., Beckstein, C.: What's in a trace: The box model revisited. In: Proc. of AADEBUG'93. LNCS, vol. 749, pp. 171–187. Springer-Verlag (1993)

A Remarks

A.1 SLD-derivation and logical consequence

The arrow "$\hookrightarrow\!\triangleright$" in an SLD-derivation has to do with logical consequence "$\rightarrow$"[10], albeit when furnished with negation and an implicit conjunct at each step, as in (A.1). To see this, assume a logic program Π and recall that an SLD-derivation

$$G \hookrightarrow\!\triangleright_{\mathcal{K}_1:\sigma_1} G_1 \hookrightarrow\!\triangleright_{\mathcal{K}_2:\sigma_2} \ldots \hookrightarrow\!\triangleright_{\mathcal{K}_n:\sigma_n} G_n$$

abbreviates $\forall[\neg G] \wedge \forall[\mathcal{K}_1] \rightarrow \forall[\neg G_1]$, ..., $\forall[\neg G_{n-1}] \wedge \forall[\mathcal{K}_n] \rightarrow \forall[\neg G_n]$, where $G, G_1, \ldots$ are queries and $\mathcal{K}_1, \mathcal{K}_2, \ldots$ are variants of some clauses from Π.

Due to $A \wedge \forall[\Pi] \rightarrow A \wedge \forall[\mathcal{K}_i]$ and $A \wedge B \rightarrow C \leftrightarrow A \wedge B \rightarrow C \wedge B$, the chain of resolutions implies

$$\forall[\neg G] \wedge \forall[\Pi] \rightarrow \forall[\neg G_1] \wedge \forall[\Pi] \rightarrow \ldots \rightarrow \forall[\neg G_n] \wedge \forall[\Pi] \qquad (A.1)$$

thus justifying the above intuition of arrows.

In case of $G_n = \square$, interpreted as $\top$, from (A.1) follows $\forall[\neg G] \wedge \forall[\Pi] \rightarrow \bot$, which is equivalent to $\forall[\Pi] \rightarrow \exists[G]$, i.e. $\exists[G]$ is a logical consequence of $\forall[\Pi]$. Hence, SLD-resolution does what it should, as a proof method of HCL.

It does even more: it supplies appropriate values, $\forall[\Pi] \rightarrow \forall[\sigma_n \cdot \ldots \cdot \sigma_1(G)]$. This can be seen from interim deductions, exposing mgus:

$$\forall[\neg G] \wedge \forall[\Pi] \rightarrow \neg\sigma_1(G) \wedge \forall[\Pi] \rightarrow \neg G_1 \wedge \forall[\Pi]$$

and similarly $\neg\sigma_2(G_1) \wedge \forall[\Pi] \rightarrow \neg G_2 \wedge \forall[\Pi]$, etc. Since $A \rightarrow \sigma(A)$, this further gives $\neg\sigma_2(\sigma_1(G)) \wedge \forall[\Pi] \rightarrow \neg\sigma_2(G_1) \wedge \forall[\Pi] \rightarrow \neg G_2 \wedge \forall[\Pi]$ etc.,[11] and finally $\neg\sigma_n(\ldots(\sigma_1(G))\ldots) \wedge \forall[\Pi] \rightarrow \ldots \rightarrow \neg\top$, hence $\forall[\Pi] \rightarrow \sigma_n \cdot \ldots \cdot \sigma_1(G)$, so by generalizing $\forall[\Pi] \rightarrow \forall[\sigma_n \cdot \ldots \cdot \sigma_1(G)]$, or written in full: $\models \forall[\Pi] \rightarrow \forall[\sigma_n \cdot \ldots \cdot \sigma_1(G)]$.

A.2 Emulating catch/throw

For `catch`/3 and `throw`/1, the following reduction rules can be used:

$$\texttt{catch(}G\texttt{,}\,Catcher\texttt{,}\,Recovery\texttt{)} \triangleright_{\phi} G$$
$$\texttt{throw(}Ball\texttt{)} \triangleright_{\phi} \texttt{fail}$$

Reducing `throw`/1 to `fail`/0 is not the same as default failure by lack of program clauses, because it enables backtracking on `throw`/1. On backtracking, `throw`/1 springs to life:

$$Back\ \texttt{throw(}B\texttt{)}, R \multimap\!\triangleright_{\phi} \sigma((Rec, R'))$$
$$Back\ \texttt{throw(}B\texttt{)}, R \multimap\!\triangleright_{-\mathsf{S}} Back\ Parent\{\texttt{catch(}G\texttt{,}\,C\texttt{,}\,Rec\texttt{)}, R'\}$$

[10] To save some space, valid implications $\models A \rightarrow B$ ("logical consequence") we write here without $\models$.

[11] With a bit of rearranging, this implies $\models \forall[\Pi] \rightarrow \forall[G_i \rightarrow \sigma_i \cdot \ldots \cdot \sigma_1(G)]$, which justifies the concept of *resultant* from [17].

Here we have not one but two *Back* rules for revisiting query `throw(`B`)`$, R$, to replace respectively the default rules 1 ("next choice") and 2 ("no more choices") from Definition 15. The latter serves to discard any choices between the current `throw` and the parent of the issuing `catch`.

To fill in the hitherto unspecified variables G, C, Rec, R' and σ, observe: If the utility pair `catch-throw` is used in standard-abiding way [5], then `throw(`B`)` was issued by G from some `catch(`G`,`C`,`Rec`)`, and B is unifiable with C. In that case, `throw(`B`),`R must be a subnode of the most recent `catch(`G`,`C`,`Rec`),`R', with existing $\sigma := \boldsymbol{U}(\boldsymbol{S}_C(B){=}C)$. Also, it must be a subnode of G, R'.

Finally, S is the score to be discarded, belonging to the subderivation between the parent node of `catch(`G`,`C`,`Rec`),`R' and the current *Back* `throw(`B`),`R.

Example 32 (catch/throw). By changing $\hat{\mathcal{K}}_q$ in Example 12 to

```
q(b) :- throw(xb).   % new K̂q
```

the computation in Figure 5 is obtained, with $\mathsf{S} = \hat{\mathcal{K}}_{t1}\colon\binom{Y}{1} + \ \dots\ - \phi$.

$$
\begin{array}{ll}
 & \texttt{t(Y), catch(alt(X),B,rx(B)), r(X)} \\
\rightarrow_{\hat{\mathcal{K}}_{t1}:\binom{Y}{1}} & \texttt{catch(alt(X),B,rx(B)), r(X)} \\
\rightarrow_{\phi} & \texttt{alt(X), r(X)} \\
\rightarrow_{\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}} & \texttt{p(X), q(X), r(X)} \\
\rightarrow_{\hat{\mathcal{K}}_{p1}:\binom{X}{a}} & \texttt{q(a), r(a)} \\
\rightarrow_{-\hat{\mathcal{K}}_{p1}:\binom{X}{a}} & \textit{Back}\ \texttt{p(X), q(X), r(X)} \\
\rightarrow_{\hat{\mathcal{K}}_{p2}:\binom{X}{b}} & \texttt{q(b), r(b)} \\
\rightarrow_{\hat{\mathcal{K}}_{q}:\varepsilon} & \texttt{throw(xb), r(b)} \\
\rightarrow_{\phi} & \texttt{fail, r(b)} \\
\rightarrow_{-\phi} & \textit{Back}\ \texttt{throw(xb), r(b)} \\
\rightarrow_{\phi} & \texttt{rx(xb), r(X)} \quad \% = \sigma((\texttt{rx(B),r(X)}))\ \textit{with}\ \sigma = \binom{B}{xb} \\
\rightarrow_{-\phi} & \textit{Back}\ \texttt{throw(xb), r(b)} \quad \%\ \textit{due to lack of clauses for}\ \texttt{rx}/1 \\
\rightarrow_{-\mathsf{S}} & \textit{Back}\ \texttt{t(Y), catch(alt(X),B,rx(B)), r(X)} \\
\rightarrow_{\hat{\mathcal{K}}_{t2}:\binom{Y}{3}} & \dots
\end{array}
$$

Fig. 5. Emulating catch/throw

B Proofs

B.1 Auxiliary claims

Let us start with a simple variable-conservation claim. By applying a substitution on a term, the term may lose or win some variables, but any changes are contained within the substitution.

Lemma 33 (no change). *For any term t and any substitution σ holds*

$$Vars(t) \cup Vars(\sigma) = Vars(\sigma(t)) \cup Vars(\sigma)$$

Proof. Let $x \mathrel{\mathring{\in}} t$. If $x \notin \mathit{Dom}(\sigma)$, then $x = \sigma(x) \mathrel{\mathring{\in}} \sigma(t)$. In other words, any variables from t that are missing in $\sigma(t)$ can be found in $\mathit{Dom}(\sigma)$. Analogously for a possible win. ♢

In the following we will need a symbol for a reduction of A that is actually *reachable* with the given residual R. We also assume a preface P. The reduction can happen after a lot of backtracking, but, due to zero sum of dead-end branches, the score of the reduction is equal to (the net-value of) the score of the derivation leading to it.

Definition 34 (reduction in context). *Given is an atom A, a term° P and a query° R. If for some C, S holds $\mathtt{true}(P), A, R \multimap\!\!\triangleright^{+}_{\phi+\mathsf{S}} C, \mathsf{S}(R)$ and $A \triangleright_{\mathit{Net}(\mathsf{S})} C$, that shall be shorter written as $P \mathbin{\backslash\!\!\backslash} A \triangleright C \mathbin{\backslash_{\mathsf{S}}} R$.*

In Theorem 28, we show an iteration property for complete answers to a conjunctive query. It rests upon the base case of one resolution step. The base case is divided in two parts. In each, as expected, the preface P is a term°, and the residual R is a query°.

The direct part is straightforward, it just rephrases a resolution step by means of "\":

Lemma 35 (split). *Let A be an atom and B a query°. If $P \mathbin{\backslash\!\!\backslash} A, B \mathbin{\backslash_{\mathsf{S}}} R$, then for some $C, \mathsf{T}, \mathsf{U}$ holds $P \mathbin{\backslash\!\!\backslash} A \triangleright C \mathbin{\backslash_{\mathsf{T}}} B, R$ and $P, A, \mathsf{T} \mathbin{\backslash\!\!\backslash} C, \mathsf{T}(B) \mathbin{\backslash_{\mathsf{U}}} \mathsf{T}(R)$ and $\mathsf{S} = \mathsf{T} + \mathsf{U}$.*

The interesting part is the converse, claiming that a derivation starting from a resolvent and staying apart from the original query and the resolution score is indistinguishable from the original derivation. In other words, the outcome of a resolution step is determined by the algorithms $\boldsymbol{U}$, $\boldsymbol{S}$ and the spent variables, and by nothing else. For this to hold, $\boldsymbol{S}$ must be flat.

Lemma 36 (assemble). *Let $\boldsymbol{S}$ be flat. Let A be an atom and B a query°. If for some $C, \mathsf{S}, \mathsf{T}$ holds*

$$P \mathbin{\backslash\!\!\backslash} A \triangleright C \mathbin{\backslash_{\mathsf{S}}} B, R \quad \textit{and} \quad P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} C, \mathsf{S}(B) \mathbin{\backslash_{\mathsf{T}}} \mathsf{S}(R)$$

then also $P \mathbin{\backslash\!\!\backslash} A, B \mathbin{\backslash_{\mathsf{S}+\mathsf{T}}} R$.

Proof. From $P \mathbin{\backslash\!\!\backslash} A \triangleright C \mathbin{\backslash_{\mathsf{S}}} B, R$ follows

$$\mathtt{true(}P\mathtt{)}, A, B, R \multimap\!\!\triangleright^{+}_{\phi+\mathsf{S}} C, \mathsf{S}(B), \mathsf{S}(R) \tag{B.2}$$

From $P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} C, \mathsf{S}(B) \mathbin{\backslash_{\mathsf{T}}} \mathsf{S}(R)$ follows

$$\mathtt{true((}P, A, \mathsf{S}\mathtt{))}, C, \mathsf{S}(B), \mathsf{S}(R) \multimap\!\!\triangleright^{+}_{\phi+\mathsf{T}} \mathsf{T}(\mathsf{S}(R)) \tag{B.3}$$

Before Lemma 26 could be applied, the variables of (B.2) and (B.3) must obey

$$\mathit{Vars}((P, A, \mathsf{S}, C, \mathsf{S}(B), \mathsf{S}(R))) = \mathit{Vars}((P, A, B, R, \mathsf{S}))$$

By Lemma 33 we know $\mathit{Vars}((B, \mathsf{S})) = \mathit{Vars}((\mathsf{S}(B), \mathsf{S}))$, and similarly for R. Also, $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{S}))$. Therefore, Lemma 26 may be applied on (B.2) and (B.3), giving

$$\texttt{true(P)}, A, B, R \twoheadrightarrow^{+}_{\phi+\mathsf{S}+\mathsf{T}} \mathsf{T}(\mathsf{S}(R))$$

The border condition (Remark 20) is ensured by the two original derivations. ♢

B.2 Claims from the main text

Lemma 17 (failure, zero sum). For a query G, if $\twoheadrightarrow_{\mathsf{s}} G \twoheadrightarrow^{*}_{\mathsf{T}} \mathit{Back}\ \mathit{Parent}\{G\}$, then $\mathit{Net}(\mathsf{s} + \mathsf{T}) = \Phi$ and G fails finitely in HCL*.

Proof. Towards more clarity, let us include the starting node $(\mathit{Back})\mathit{Parent}\{G\}$, so a dead-end branch looks typically like $(\mathit{Back})F \twoheadrightarrow_{\mathsf{s}} G\ \ \ldots\ \ (\mathit{Back})G \twoheadrightarrow_{-\mathsf{s}} \mathit{Back}\ F$.

We shall use induction on the number n of inner nodes. If $n = 1$, the branch is $(\mathit{Back})F \twoheadrightarrow_{\mathsf{s}} G \twoheadrightarrow_{-\mathsf{s}} \mathit{Back}\ F$, and it satisfies the claim. Assume the claim holds for up to $n \geq 1$ inner nodes and consider a dead-end branch with $n + 1$ inner nodes. To cater for more than one inner node, there must have been choices for G. Then the branch looks like

$$(\mathit{Back})F \twoheadrightarrow_{\mathsf{s}} G \twoheadrightarrow H\ \ldots\ \mathit{Back}\ G \twoheadrightarrow_{-\mathsf{s}} \mathit{Back}\ F$$

By applying the inductive hypothesis on the branch $G \twoheadrightarrow H\ \ldots\ \mathit{Back}\ G$, having less than n inner nodes, the net sum of Φ is obtained. ♢

Lemma 26 (resumability). Assume $\boldsymbol{S}$ to be flat. For any queries A, B, C and any term° P satisfying $\mathit{Vars}((P, B)) = \mathit{Vars}((A, \mathsf{S}))$ holds: If $A \twoheadrightarrow^{*}_{\mathsf{S}} B$ and $\texttt{true(P)}, B \twoheadrightarrow^{+}_{\phi+\mathsf{T}} C$, then $A \twoheadrightarrow^{*}_{\mathsf{S}+\mathsf{T}} C$.

Proof. At the start of the second derivation, the variables of A and S (and no others) are spent, the same situation as at the end of the first derivation. Since $\boldsymbol{S}$ is assumed to be flat, solely the spent variables determine standardizing apart. So the first derivation can be continued indistinguishably from the second one, after its getting rid of $\texttt{true(P)}$, which produces the extra score ϕ. ♢

Theorem 28 (iteration for complete answer). Let $\boldsymbol{S}$ be flat. Let A be a query, B, R be queries°, and P be a term°. Then:

1. If $P \mathbin{\backslash\!\!\backslash} A \mathbin{\backslash}_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} \mathsf{S}(B) \mathbin{\backslash}_{\mathsf{T}} \mathsf{S}(R)$, then also $P \mathbin{\backslash\!\!\backslash} A, B \mathbin{\backslash}_{\mathsf{S}+\mathsf{T}} R$.
2. Conversely, if $P \mathbin{\backslash\!\!\backslash} A, B \mathbin{\backslash}_{\mathsf{U}} R$, then there are S, T such that $P \mathbin{\backslash\!\!\backslash} A \mathbin{\backslash}_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \mathbin{\backslash\!\!\backslash} \mathsf{S}(B) \mathbin{\backslash}_{\mathsf{T}} \mathsf{S}(R)$ and $\mathsf{U} = \mathsf{S} + \mathsf{T}$.

Proof. By induction on the number k of conjuncts in A. We start with the base case $k = 1$, i.e. A is an atom, and need to prove both parts of the theorem.

Base case, direct part. By induction on the length n of derivation of A. For $n = 1$, A can be ***true***, ***true(_)*** or a unification. The claim follows from the definition of resolvent. Now assume the claim holds for derivations of length $\leq n$ and consider a derivation for A of length $n + 1$. Assumptions are

$$P \mathbin{\backslash\!\backslash} A \quad \backslash_{\mathsf{S}}\ B, R \tag{B.4}$$

$$P, A, \mathsf{S} \mathbin{\backslash\!\backslash} \mathsf{S}(B) \backslash_{\mathsf{T}}\ \mathsf{S}(R) \tag{B.5}$$

From Lemma 35 applied on (B.4) (with void sibling), A had to have been resolved, so for some $C, \mathsf{V}, \mathsf{U}$ with $\mathsf{S} = \mathsf{V} + \mathsf{U}$ we have

$$P \mathbin{\backslash\!\backslash} A \triangleright C \backslash_{\mathsf{V}}\ B, R \tag{B.6}$$

$$P, A, \mathsf{V} \mathbin{\backslash\!\backslash} C \quad \backslash_{\mathsf{U}}\ \mathsf{V}(B), \mathsf{V}(R) \tag{B.7}$$

Then $\mathsf{S}(\_) = \mathsf{U}(\mathsf{V}(\_))$, so by (B.5) holds $P, A, \mathsf{V} + \mathsf{U} \mathbin{\backslash\!\backslash} \mathsf{U}(\mathsf{V}(B)) \backslash_{\mathsf{T}} \mathsf{U}(\mathsf{V}(R))$ and hence, by $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{V}))$,

$$P, A, \mathsf{V}, C, \mathsf{U} \mathbin{\backslash\!\backslash} \mathsf{U}(\mathsf{V}(B)) \backslash_{\mathsf{T}} \mathsf{U}(\mathsf{V}(R)) \tag{B.8}$$

Applying the inductive hypothesis on (B.7) and (B.8), we obtain

$$P, A, \mathsf{V} \mathbin{\backslash\!\backslash} C, \mathsf{V}(B) \backslash_{\mathsf{U}+\mathsf{T}} \mathsf{V}(R)$$

This and (B.6) give, by Lemma 36, $P \mathbin{\backslash\!\backslash} A, B \backslash_{\mathsf{V}+\mathsf{U}+\mathsf{T}} R$ with $\mathsf{V} + \mathsf{U} + \mathsf{T} = \mathsf{S} + \mathsf{T}$, which proves the direct part.

Base case, converse part. By induction on the length n of derivation of A, B. For $n = 2$, A and B can be ***true***, ***true(_)*** or a unification. The claim follows from the definition of resolvent. Now assume the claim holds for derivations for A, B of length $\leq n$ and consider a derivation of length $n + 1$.

The assumption is $P \mathbin{\backslash\!\backslash} A, B \backslash_{\mathsf{U}} R$. By Lemma 35, for some $C, \mathsf{W}, \mathsf{V}$

$$P \mathbin{\backslash\!\backslash} A \triangleright C \quad \backslash_{\mathsf{W}}\ B, R \tag{B.9}$$

$$P, A, \mathsf{W} \mathbin{\backslash\!\backslash} C, \mathsf{W}(B) \backslash_{\mathsf{V}}\ \mathsf{W}(R), \quad \text{and} \quad \mathsf{U} = \mathsf{W} + \mathsf{V} \tag{B.10}$$

By induction hypothesis on (B.10), V can be split as $\mathsf{V} = \mathsf{V}_1 + \mathsf{V}_2$ such that

$$P, A, \mathsf{W} \mathbin{\backslash\!\backslash} C \qquad \backslash_{\mathsf{V}_1}\ \mathsf{W}(B), \mathsf{W}(R) \tag{B.11}$$

$$P, A, \mathsf{W}, C, \mathsf{V}_1 \mathbin{\backslash\!\backslash} \mathsf{V}_1(\mathsf{W}(B)) \backslash_{\mathsf{V}_2}\ \mathsf{V}_1(\mathsf{W}(R)) \tag{B.12}$$

By Lemma 36 applied on (B.9) and (B.11),

$$P \mathbin{\backslash\!\backslash} A \backslash_{\mathsf{W}+\mathsf{V}_1} B, R$$

By (B.12) and $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{W}))$, we obtain

$$P, A, \mathsf{W} + \mathsf{V}_1 \mathbin{\backslash\!\backslash} \mathsf{V}_1(\mathsf{W}(B)) \backslash_{\mathsf{V}_2} \mathsf{V}_1(\mathsf{W}(R))$$

So U can be split as $\mathsf{U} = \mathsf{W} + \mathsf{V}_1 + \mathsf{V}_2$ with required properties.

Base case is thus proved. Now assume the claim holds for conjunctions of up to k conjuncts. Let $A := Q, Q'$ where Q is an atom and Q' is a conjunction of length k.

Inductive case, direct part. The assumptions are

$$P \setminus\!\!\setminus Q, Q' \setminus_{\mathsf{S}} B, R \tag{B.13}$$

$$P, Q, Q', \mathsf{S} \setminus\!\!\setminus \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R) \tag{B.14}$$

We wish to obtain $P \setminus\!\!\setminus Q, Q', B \setminus_{\mathsf{S+T}} R$. By applying the converse part of the base case on (B.13), there are U, V with $\mathsf{U} + \mathsf{V} = \mathsf{S}$ and

$$P \setminus\!\!\setminus Q \quad \setminus_{\mathsf{U}} Q', B, R \tag{B.15}$$

$$P, Q, \mathsf{U} \setminus\!\!\setminus \mathsf{U}(Q') \setminus_{\mathsf{V}} \mathsf{U}(B, R) \tag{B.16}$$

By Lemma 33, $\mathit{Vars}(Q') \cup \mathit{Vars}(\mathsf{U}) = \mathit{Vars}(\mathsf{U}(Q')) \cup \mathit{Vars}(\mathsf{U})$. Hence, (B.14) can be rearranged as

$$P, Q, \mathsf{U}, \mathsf{U}(Q'), \mathsf{V} \setminus\!\!\setminus \mathsf{V}(\mathsf{U}(B)) \setminus_{\mathsf{T}} \mathsf{V}(\mathsf{U}(R)) \tag{B.17}$$

By induction hypothesis, from (B.16) and (B.17) follows

$$P, Q, \mathsf{U} \setminus\!\!\setminus \mathsf{U}(Q'), \mathsf{U}(B) \setminus_{\mathsf{V+T}} \mathsf{U}(R) \tag{B.18}$$

By applying the direct part of the base case on (B.15) and (B.18),

$$P \setminus\!\!\setminus Q, Q', B \setminus_{\mathsf{U+V+T}} R$$

Inductive case, converse part. This we leave to the reader. ♢

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

PyPlC – Towards a Prolog Database Connectivity for Python

Stefan Bodenlos[1], Daniel Weidner[1], and Dietmar Seipel[2]

University of Würzburg, Department of Computer Science,
Am Hubland, D – 97074 Würzburg, Germany

[1]{stefan.bodenlos,daniel.weidner}@stud-mail.uni-wuerzburg.de
[2]dietmar.seipel@uni-wuerzburg.de

Abstract. Integrating languages of different programming paradigms introduces interesting questions and conflicts. In multi–paradigm programming, interfaces and data structures must be specified in a way that is reasonable for both worlds. In this respect, the open database connectivity ODBC is a success story, being today's standard tool of accessing relational databases with applications written in mostly any programming language.
In this work, we extend the concepts of ODBC to the integration of the logic programming language Prolog, and we present a unified approach for the intuitive integration of Prolog queries into Python, a popular imperative programming language. Our tool chain, called PYPLC, provides a similar standard for accessing derived facts from the Prolog database without side effects. PYPLC contains the Python/Prolog query language PYPLQL. Together with a specification language based on XML Schema, this allows to specify and generate structures in Python corresponding to facts from Prolog.

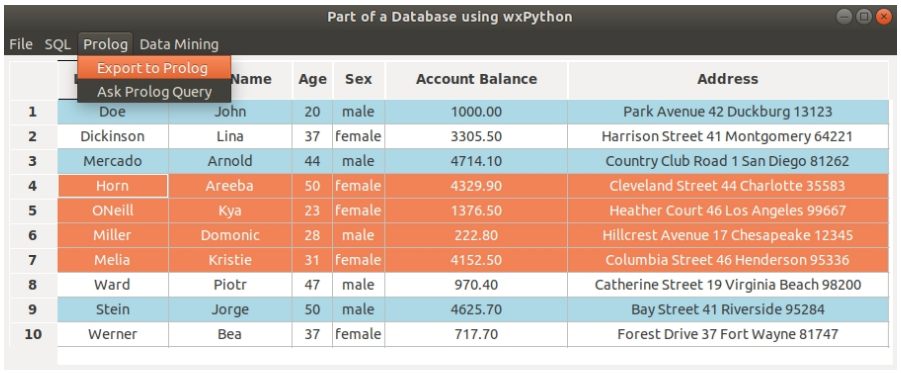
Keywords: Prolog· Python· Multi–paradigm Programming · Integration.

1 Introduction

Integrating different technologies is a common issue in the process of software development. Often, the program's logic and data tiers are driven by different programming paradigms. Either two separate teams design and maintain both components, or a single team does: the first causes expenditure for coordinating the teams, the latter complicates the planning process instead of simplifying it.

Python is a universal, multi–paradigm programming language, which is well known for its simple but nevertheless powerful syntax [11]. Although it is multi–paradigm, object–orientation can be viewed as one of its characteristics, and Python lacks a support for logic programming. Prolog provides powerful mechanisms and tools for logic programming related to deductive databases, some applications are easier to implement with logic programming, e.g., natural language parsing [9], and in some areas, like artificial intelligence [3], many solutions have been developed. Python has many powerful libraries for data mining and for accessing databases, and also graphical user interfaces can be built nicely with Python. The following Figure 1 shows a GUI frontier for accessing a relation of a University database from Python using Prolog, SQL, and ODBC; we are also planning to call data mining algorithms.

Fig. 1. Python GUI for Accessing a Relation of a University Database.



		Name	Age	Sex	Account Balance	Address
1	Doe	John	20	male	1000.00	Park Avenue 42 Duckburg 13123
2	Dickinson	Lina	37	female	3305.50	Harrison Street 41 Montgomery 64221
3	Mercado	Arnold	44	male	4714.10	Country Club Road 1 San Diego 81262
4	Horn	Areeba	50	female	4329.90	Cleveland Street 44 Charlotte 35583
5	ONeill	Kya	23	female	1376.50	Heather Court 46 Los Angeles 99667
6	Miller	Domonic	28	male	222.80	Hillcrest Avenue 17 Chesapeake 12345
7	Melia	Kristie	31	female	4152.50	Columbia Street 46 Henderson 95336
8	Ward	Piotr	47	male	970.40	Catherine Street 19 Virginia Beach 98200
9	Stein	Jorge	50	male	4625.70	Bay Street 41 Riverside 95284
10	Werner	Bea	37	female	717.70	Forest Drive 37 Fort Wayne 81747

Because Python is a widespread and easy–to–learn language, the integration of Prolog offers an access to a powerful technology for a large part of software developers. One way to integrate Prolog in Python is to imitate a Prolog interface in Python. Lager and Wielemaker have proposed a library called Pengines [7], which is an infrastructure providing general mechanisms for converting Prolog data and handling Prolog non–determinism; it is included in the standard package for SWI–Prolog 7. Like in ODBC, a client can send a Prolog program to a thread, that provides the data exchange. After that, the client can send Prolog queries to the thread, which will respond with a set of answer tuples. Based on Pengines, the Python module PythonPengines has been proposed [1]. A user can construct Prolog programs and query terms in Python. Here, a Python user has to comprehend larger parts of the Prolog syntax and semantics. Moreover, the instructions are usually not compact, because they need some lines of code to declare a query step by step.

There is a broad field of applications for Prolog. Here, we focus on its strength for database management and provide a framework for this specific application. Many solutions have been proposed, that can establish a connection between Prolog and object–oriented programming languages like Python. Often they mainly aim to establish a connection at all, and the user must have a deep understanding of the language to integrate (in this case Prolog), just as ODBC requires some knowledge of SQL. Here, we want to go one step further and integrate Prolog into Python in greater depth. In this new approach, a user can understand and query a Prolog database using Python concepts only. Thus, knowledge about Prolog is no prerequisite, nevertheless Prolog can be employed.

The fundamental structure of an object–oriented programming language is an object. It contains data describing the object, called attributes, as well as operations that manipulate the data and serve as an interface. Each object belongs to exactly one class, that can be viewed as a description for objects. All objects of a class share the same properties, except the concrete values of the attributes. A class can be inherited by another class, meaning that the class extends the original description. In general, Python follows these principles in a way that ensures a high level of dynamicity. Especially, the set of attributes of an object can be dynamically extended even at run time. Since the object–oriented and the relational paradigm differ, issues can occur when an object is to be saved in a relation. This is called object–relational (impedance) mismatch [5]. A solution for this is an object–relational mapping; that is, an object–oriented program can access a relational database in an object–oriented manner.

In the present paper we propose a framework, which is more intuitively usable in terms of the Python philosophy. We introduce a new approach of such database connectivities extending the Odbc approach, namely a tool chain called PyPlC (Python/Prolog Database Connectivity) [2]. It aims to convert Prolog structures into corresponding, object–oriented structures, which can then easily be processed in Python. A new intuitive object–oriented query language allows for a simple way of querying a Prolog database in a Python–like fashion. PyPlC translates such requests into Prolog syntax and maps the result back into Python objects. Instead of realizing such procedures manually by a user, all of these processes are automatized and customizable by PyPlC. This simplifies Prolog database integration for Python. PyPlC is inspired by the architecture of the uniform and object–oriented integration framework CAPJa for Java and Prolog introduced in [8], but it is a new implementation from scratch with some new features. The Java tool CAPJa supports most Prolog systems, and provides semi–automatized and completely customizable mechanisms for the integration and a fully automatized mapping from Prolog functors to Java objects; an object–oriented query language enables queries based on Java syntax and semantics.

The rest of this paper is structured as follows: Section 2 describes how Prolog facts should be mapped to corresponding Python structures. Section 3 presents the query language for posing queries to Prolog in Python. Our approach is discussed shortly in Section 4. Conclusions and future work are given in Section 5.

2 Corresponding Structures in Prolog and Python

As a core for the intuitive access to a Prolog database, PyPlC generates corresponding structures in Python, or more specifically, it maps Prolog functors into corresponding classes. Thus, PyPlC eases the work for programmers, because it automatizes this integration process, whereas customization is completely supported.

The University Database in Prolog. For illustration, we introduce a simple Prolog database representing a university. Figure 1 has already shown the student relation of the database. Listing 1.1 shows the Prolog schema and a single fact of the student relation. It contains a functor `student/6` for facts composed of constants and a compound term with the functor `address/4`. The object–oriented representation of this model is a class structure, in which every functor is represented by a class, whose attributes represent the arguments and whose name is derived by the functor's name (see Figure 2).

The individual facts of the database are represented by objects of these classes. The mapping definition of PyPlC is bijective; therefore, mapping and remapping of facts and objects can be done any number of times without loss of information.

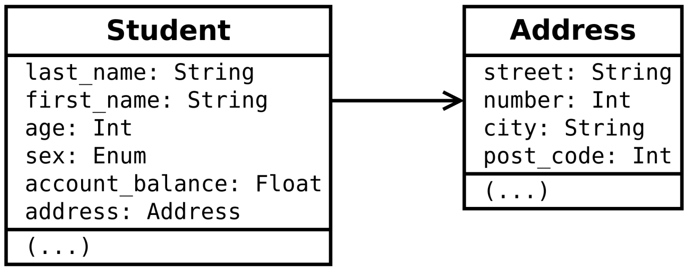
Listing 1.1. Part of a Prolog Database Representing a University.

```
% student( Last_Name, First_Name, Age, Sex,
%     Account_Balance,
%     address(Street, Number, City, Post_Code) ).

student( 'Doe', 'John', 20, male, 1000.00,
    address('Park Avenue', 42, 'Duckburg', 12345) )
```

In Python, attributes can be addressed by their names, and in Prolog, arguments can be addressed by their position in the compound term. Because of that, a mapper defines the association between both components. For each built–in type, a static mapping is implemented. A mapper transforms the components: it either uses such a mapping definition for built–in types, or it calls a mapper (possibly recursively). Thus, the type of the attribute *address* is the class *Address*.

Fig. 2. Class Structure Representing the University Database.

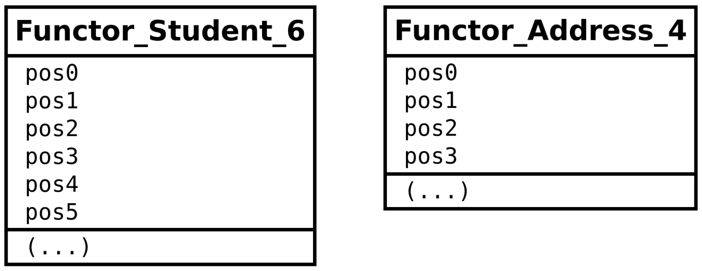


Static and Dynamic Mapping. Our framework offers two types of mappings: static and dynamic mappings, where these names allude how they are intended to be used rather than how they operate.

The dynamic mapping aims at providing a straightforward integration of Prolog into Python without any preparatory work. It avoids major changes in the Python module of PyPlC and only stores necessary information needed for the mapping process. When using the dynamic mode, the mapping process is not customizable at all. Thus, every time a Prolog database is integrated into Python, the representation remains unchanged. This is helpful if the user is very familiar with the queried Prolog database.

PyPlC dynamically maps the university sample database of Listing 1.1 into the class structure given in Figure 3. Here, the name is extended by the arity of the predicate, because Prolog makes a distinction between predicates of different arities. In Python, the use of an attribute can easily be understood by its name, whereas in Prolog there are no such descriptions for components. Hence, PyPlC cannot extract coherent names for the attributes. Also the association between classes cannot be generated from a plain Prolog predicate, since the typing comes from a conversion rather than from the syntax. The created structure is not user–friendly, but it offers a simple access to a Prolog database in Python, which the user knows all about.

Fig. 3. Dynamically Created Class Structure Representing the University Database.



The General Object–Oriented Mapping Notation. The static mapping resolves these issues, but it requires a description of the Prolog database. Here, we are planning to use the General Object–Oriented Mapping Notation (Goomn), an Xml–based notation. For each component of a Prolog structure, it provides a meaningful name and a generic description of the type. Once a Prolog database is described in Goomn, it can be mapped in principle to any object–oriented programming language due to its generic nature; see Listing 1.2 for the Goomn notation of the example database. Note that in an Xml–based notation there are no default types float and enum.

Listing 1.2. Xml Schema Notation for the Predicate `student/6`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.db.uni-passau.de/University">
  <xsd:element name="University" type="UniInfoType"/>
  <xsd:complexType name="Student"> ... </xsd:complexType>
</xsd:schema>

<xsd:complexType name="Student">
  <xsd:sequence>
    <xsd:element name="Last Name" type="xsd:string"/>
    <xsd:element name="First Name" type="xsd:string"/>
    <xsd:element name="Age" type="xsd:integer"/>
    <xsd:element name="Sex" type="xsd:string"/>
    <xsd:element name="Number" type="xsd:integer"/>
    <xsd:element name="Address">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Street" type="xsd:string"/>
          <xsd:element name="Number" type="xsd:integer"/>
          <xsd:element name="City" type="xsd:string"/>
          <xsd:element name="Post Code" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The generated class structure in Python is identical with the opening Figure 2, and obviously it is much more user–friendly than the dynamically created class structure of Figure 3.

PyPlC creates a mapper class for each mapper and integrates it into the PyPlC module. By changing the Goomn notation, the mapping process is completely customizable: for instance, a user can decide whether a component should be mapped or not. Since one can compile the mapper classes, the processing speed can be significantly increased.

The Prolog Mapping Notation. Currently, we are still using an alternative to Goomn, the Prolog Mapping Notation (Pmn) inspired by [8]. Similar to Goomn, Pmn aims to provide a description for Prolog functors. Here, the information is saved into Prolog facts, instead of an external Xml file; see Listing 1.3 for the Pmn notation of the example database.

Listing 1.3. The Schema of the Predicate `student/6` in Pmn Notation.

```
functor(address,
    ['street', 'number', 'city', 'post_code'],
    [str, int, str, int]).
functor(student, ['last_name', 'first_name',
    'age', 'male', 'account_balance', 'address'],
    [str, str, int , enum, float, functor(address)]).
```

In this case, the Pmn structure of address has to be created first, because the predicate student references the predicate address. The generated functors consists of the predicate name, the arguments, and the related types. From the Pmn notation, a Python programmer can immediately understand the structure of the original Prolog code.

3 The Python/Prolog Query Language

To keep the benefits of the underlying logic programming system, the query processing and the data storage will be completely done by a Prolog interpreter.

We introduce the new Python/Prolog Query Language (PyPlQL), which is inspired by the query language Jpql of CAPJa. It is an easy–to–learn internal domain–specific language [4] based on corresponding structures mentioned above. In simple words, a PyPlQL query is a Python function, whose parameters declare the required objects and the function body describes the conditions to be fulfilled by a Boolean expression. Such functions are not executed as they are; instead, they get transmitted to PyPlC translating, passing on to a Prolog instance and transformating the result into objects.

The syntax is defined in the following Listing 1.4. `queryName` is the name of the query and `[types]` represent the query types and the special query instructions. A query type is considered to be an object of a generated class according to the Python syntax, whereby a type hint has to be given for each single query type. The apparent objects are no actual objects (thus, they do not exist at run time), instead they declare which objects are required in that query. `constraints` defines the conditions the query types have to fulfill. Conditions can be made by an attribute of any query type or a hard–coded value as well as every built–in relational or Boolean operators of Python, used in the way they are permitted according to Python syntax. Additionally one can use brackets to define the order of evaluation.

Listing 1.4. Definition of a PyPlQL Query

```
def queryName [types]: constraints
```

A special query instruction is, for example, the `omit`–expression. If an attribute `a` of the query type `o` is declared by an `omit={o.a}`–expression, then the attribute `a` will be ignored during the query processing leading to a faster execution (see anonymous variable in Prolog). The value for such an attribute will be set to `None`. Queries, that omit an attribute, which is later accessed in the condition of a query, are invalid.

Other special query instructions are the aggregation expressions `min(o.a)`, `max(o.a)`, `avg(o.a)`, and `sum(o.a)`. They will retrieve the minimum, maximum, average, or summed values, respectively, of the attribute `a` of the query type `o`, for every group specified by `groupBy`.

PyPlC can delete all objects matching the constraint of a PyPlQL query, retrieve all of them, or retrieve just a single object and the other objects gradually, if requested (see backtracking in Prolog). Alternatively, PyPlC can be instructed to retrieve the number of matching objects. If an aggregation or `groupBy` expression is used in a PyPlQL query, then all instances fulfilling the conditions or the number of matching objects can be retrieved. In a `groupBy` case, see, e.g., Listing 1.9, the return value is a (nested) Python dictionary, whose keys are the attributes mentioned in the `groupBy` clause and whose values are the results. Queries with joins can also be implemented in PyPlQL; see, e.g., Listing 1.8. This extends the Odbc approach.

3.1 Example Queries in PyPlQL

Assume that a user wants to determine every student of the Prolog university database whose surname is `'Doe'` and who is at least 18 years old. According to the created, corresponding class structure, this can be expressed as follows: determine every object of the class `Student`, whose attribute value for `last_name` is equal to `'Doe'`, and whose attribute value for `age` is greater than or equal to 18. This is a very natural way of expressing this regarding the object–oriented programming paradigm. Coming from that way of looking at the problem, it is easy to formulate the PyPlQL query, see Listing 1.5. In Python, atoms such as Student should start with capital characters. For a Python programmer, this query is easier to understand than the corresponding native Prolog code of Listing 1.6.

Listing 1.5. Complex Prolog Goal in Python: Student over 18 with the Last Name `'Doe'`

```
def studentQuery1 (s : Student, omit = {s.address}):
    s.last_name == 'Doe' and s.age >= 18
```

PyPlC translates the query `studentQuery1` from Listing 1.5 to the Prolog representation of Listing 1.6, which could also be send to a Prolog engine with Pengines.

Listing 1.6. Generated Prolog Representation of the Student Query

```
Name = 'Doe',
student(Name, Age, C, D, E, _),
Age >= 18.
```

The transformed return value is a set containing the single student of the university database, see Listing 1.7 (where Student and Address have been indented in Prolog style for readability, other than it would be in Python).

Listing 1.7. Result of the Example Query in Python

```
result = [
  Student('Doe', 'John', 20, male, 1000.00,
    Address('Park Avenue', 42, 'Duckburg', 12345))  ]
```

Another example query computes two students living in the same street. This can be expressed like in Listing 1.8.

Listing 1.8. Self Join in Python: Two Students Living in the Same Street

```
def studentQuery2 (s1 : Student, s2 : Student):
   s1.address.street == s2.address.street
```

The number of students grouped by cities can be retrieved by a query like in Listing 1.9. I.e., PyPlQL includes aggregation and grouping.

Listing 1.9. Aggregation in Python: Number of Students Grouped by Cities

```
def studentQuery3
   (s : Student, count(*), groupBy = [s.address.city]):
       pass
```

3.2 Case Study: Querying and Result Processing in Python

In the case study of Listing 1.10, we want to demonstrate how a user can query the university data base step by step in order to print (result processing) information of students living in New York.

Listing 1.10. Case Study in Python: Querying Prolog

```
# Create an instance of PyPlC
pyplc = Pyplc('swipl')

# Import the university Prolog database
# and its scheme
pyplc.import_database('university_db.pl',
  'university_db_goomn.xml')

# After examining the generated classes,
# the user can define the query
def student_query
  (s : Student, omit = {s.account_balance}):
```

```
13      s.address.city == 'Duckburg'
14  # and query the Prolog database
15  res = pyplc.retrieveAll(student_query)
16
17  # Print all retrieved students
18  for s in res:
19    print(
20      'Last Name: ' + s.last_name
21      + '; First Name: ' + s.first_name
22      + '; Street: ' + s.address.street
23      + '; Number: ' + s.address.number
24      + '; City: ' + s.address.city)
25  # Print output
26  # Last Name: Doe; First Name: John; ...
```

Although PyPlC aims to integrate Prolog from the Python point of view, it also supports some methods for Prolog experts. First, the user can use the dynamic mapping module of PyPlC. Second, he can formulate a native Prolog query and bypass the mapping component of PyPlC completely. In that case, the command lines 9–15 in the example above, can be replaced like in Listing 1.11 (where student and address have been indented in Prolog style for readability, other than it would be in Python). Since there are no explicit query types in a Prolog query, the result is not a student object, but a pre–processed Prolog result. Therefore, the user has to predict the structure of the result, in order to process it in Python. The command lines 17–24 in the previous example have to be changed like in Listing 1.12, since the result list will contain assignments for the Prolog variables given in the query, like `Last_name = 'Doe'`. Hence, the result processing becomes much more impractical. However, we consider this variant as an additional feature of PyPlC, if a user wants to have a direct access to Prolog; in this case, one might also prefer to use other tools like PythonPengines.

Listing 1.11. Case Study in Python: Querying Prolog

```
# After examining the generated classes,
# the user can define the query
student_query_prolog =
    'student( Last_Name, First_Name, Age, Sex, _,
        address(Street, Number, "Duckberg", Post_Code) )'
# and query the Prolog database
res = pyplc.retrieveAll(
    student_query_prolog, bypassing = True)
```

Listing 1.12. Case Study in Python: Processing Prolog Result

```
# Print all retrieved students
for r in res:
```

```
for a in r:
  if isinstance(a, PrologAssignment) and
    a.variable = 'Last_name':
    print('Last Name: ' + r.value + '; ')
  elif isinstance(a, PrologAssignment) and
    a.variable = 'First_name':
    print('First Name: ' + r.value + '; ')
  elif isinstance(a, PrologAssignment) and
    a.variable = 'Street':
    print('Street: ' + r.value + '; ')
  elif isinstance(a, PrologAssignment) and
    a.variable = 'Number':
    print('Number: ' + r.value + '; ')
  elif isinstance(a, PrologAssignment) and
    a.variable = 'City':
    print('City: ' + r.value + ';')
```

4 Discussion

Instead of only offering a Prolog–like interface, PyPlC allows intuitive queries to a Prolog database using an object–oriented query language. This approach has a couple of advantages.

PyPlC does not only use Python structures, it also uses them in a natural way. Especially, no prerequisite knowledge of Prolog is needed. PyPlC is compatible with current versions of Swi–Prolog and Python. In principle there is a support for every Prolog implementation, since PyPlC only uses Iso–Prolog syntax [6] and includes an abstract layer offering an easy–to–extend interface for a specific Prolog implementation. But so far, PyPlC only supports Swi–Prolog. Interchangeability of a specific implementation means a gain of quality in software development.

The new Xml–based markup language Goomn enables a standardized notation of Prolog functors. A Goomn–notated database can be integrated in any other programming language, since it only uses a very general syntax. Also non–notated functors can be available in Python through a dynamic mapping, whereas a user has to use another tool to understand the Prolog database, for example the visualization module of the Prolog–based system Declare for data and knowledge engineering [10].

In the Declare, there also exist extended methods for queries to relational and deductive databases, cf., e.g., the generic aggregation operator ddbase_aggregate, which extends the standard Prolog operator findall by grouping and user–defined aggregation functions.

Another way to use PyPlC is to integrate not only deductive databases, but more complex Prolog libraries enabling a powerful way of calculating. For instance, the Prolog library ClioPatria [12] offers a convenient access to Rdf databases; thus, PyPlC also provides an interface to the semantic web.

5 Conclusions and Future Work

The proposed tool PyPlC offers a new unified and intuitive way for the integration of Prolog into Python. Derived Prolog facts can be retrieved to Python; all results for a Prolog query are derived like in deductive databases. Whereas in deductive databases, derivations usually are bottom–up, they are top–down here. But, bottom–up derivations can be implemented on the Prolog side, e.g. using Declare, to ensure termination for more logic programs, such as, e.g., the ones for cyclic transitive closure queries.

By automatically creating a corresponding Python class structure of a Prolog database and by providing a Python–like, object–oriented query language, PyPlC ensures an easy access to Prolog, where not much prior knowledge of Prolog is necessary.

This paper describes work in progress. We will consider improvements of the new approach in our future work. It is impossible to create a Prolog database from Python. Thus, PyPlC is an integration of Prolog into Python, and not vice versa. Creating objects and mapping them into a relational scheme causes many problems, and Python users could face unexpected behaviour in such cases.

The PyPlQL syntax should be extended in a way permitting every built–in Python command to be used for formulating a query. For example, for retrieving whether a substring is contained in a string, the Python command **in** is suitable. A modified version of the PyPlQL query of Listing 1.5 – asking only for surnames containing the substring `'man'` – can be seen in Listing 1.13.

Listing 1.13. PyPlQL Query with Substring Condition

```
def studentQuery1_substr (s : Student):
    'man' in s.last_name and s.age >= 18
```

In some cases, it is possible to translate a PyPlQL query during compilation, which would bring a significant speedup of the execution.

There are many fascinating Prolog applications. However, many programmers do not use these tools, because there are obstacles in terms of different paradigms compared to many wide–spread programming languages. PyPlC aims to spread interest in Prolog. Therefore, it provides a way for Prolog beginners, without getting into Prolog specifics. In order to overcome this entry barrier, PyPlC generates corresponding structures of a Prolog database in Python. By using an easy–to–learn query language, a beginner can use Prolog tools straightforwardly.

Acknowledgements. The authors would like to thank Mirco Lukas and Ludwig Ostermayer for their helpful suggestions during the master's thesis of Stefan Bodenlos, and also Falco Nogatz for his useful comments on an earlier draft of the paper.

References

1. Ian Andrich. PythonPengines. `https://github.com/ian-andrich/PythonPengines`.
2. Stefan Bodenlos. Integration von Prolog und ClioPatria in Python. Master's thesis, University of Würzburg, 2017.
3. Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson Education, 2001.
4. Martin Fowler. *Domain–specific languages*. Pearson Education, 2010.
5. Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object–relational impedance mismatch. pages 36–43, 2009.
6. ISO. `https://www.iso.org/standard/21413.html`.
7. Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming*, 14(4–5):539–552, 2014.
8. Ludwig Ostermayer. *Integration of Prolog and* Java *with the connector architecture* CAPJa. PhD thesis, University of Würzburg, 2017.
9. Fernando Pereira and Stuart Shieber. *Prolog and natural–language analysis*. Microtome Publishing, 2002.
10. Dietmar Seipel. The Declare developers' toolkit (Ddk). `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`.
11. Guido van Rossum and Python Development Team. The Python language reference. `https://docs.python.org/3/archives/python-3.6.3rc1-docs-pdf-a4.zip`.
12. Jan Wielemaker, Wouter Beek, Michiel Hildebrand, and Jacco van Ossenbruggen. ClioPatria: A Swi–Prolog infrastructure for the semantic web. *Semantic Web*, 7(5):529–541, 2016.

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

Logische Programmierung und Constraint Programmierung als Studieninhalte für BWL-Studiengänge - *Motivation, Vorgehen und Erfahrungen*

Ulrich John

Hochschule für Wirtschaft, Technik und Kultur (HWTK)
ulrich.john@hwtk.de

Abstract: Ein belastbares Überblickswissen über die Logische Programmierung und die Constraint-(Logische) Programmierung ist in der heutigen Zeit nach Erfahrung und Meinung des Autors auch für (angehende) Betriebswirte und Wirtschaftsinformatiker wichtig beziehungsweise dringend empfehlenswert, wobei insbesondere Aspekte der deklarativen Programmierung und deren Anwendbarkeit für komplexe betriebswirtschaftliche Problemstellungen eine besondere Rolle spielen. Gerade auch im Kontext der Digitalisierung/ Digitalen Transformation können beide Programmierparadigmen nutzbringend verwendet werden. Im Paper werden einige ausbaufähige Lehrinhalte mit Aspekten zu erprobten Vorgehensweisen vorgestellt und Erweiterungsmöglichkeiten benannt.

Keywords: Constraint Programmierung, Logische Programmierung, Künstliche Intelligenz, BWL- & Wirtschaftsinformatik-Studium, Wissensmanagement, Entscheidungsunterstützungssysteme, Digitalisierung, Digitale Transformation, komplexe Planungsprobleme

1 Einleitung

Studierende der Betriebswirtschaftslehre und der Wirtschaftsinformatik werden im Laufe ihres Berufslebens mit hoher Wahrscheinlichkeit mit Aufgabenstellungen der Digitalisierung und/ oder Digitalen Transformation sowie den damit verbundenen Herausforderungen konfrontiert werden oder werden mindestens im Kontext dieser tätig sein. Obwohl die Digitalisierung von Wirtschaft und Gesellschaft schon seit mehreren Jahrzehnten voranschreitet und entsprechende – teilweise drastische - Transformationen vonstattengingen, ist sie mittlerweile – unter anderem aufgrund der Reife von IT-Technologien und anhaltendem Wettbewerbsdruck - zu einem nahezu „allbestimmenden Themenkomplex“ geworden. Dieser Tatsache muss auch bei der Ausgestaltung von BWL-Studieninhalten sowie bei der Beratung bereits tätiger Manager mit besonderem Augenmerk Rechnung getragen werden. Neben anderen Technologien der Künstlichen Intelligenz spielen aus unserer Sicht – in zunehmendem Maße - die Logische Programmierung und die Constraint Programmierung/ Constraint Logische Programmierung eine entscheidende Rolle, so dass auch diese entsprechend thematisiert werden sollten. Aus der Sicht des Autors kann dies mit Bezug

und in Ergänzung zu Lehrinhalten des Operations Research erfolgen (siehe z.B. [Do15]). Die geeignete Aufnahme der Logischen Programmierung und der Constraint Programmierung/ Constraint Logischen Programmierung in die BWL-Lehre stellt eine besondere Herausforderung dar, da diese beiden Paradigmen ja selbst in verschiedenen Informatik-Studiengängen nicht oder nur rudimentär „Einzug gefunden haben", wobei dieses aufgrund intendierter Spezialisierungen durchaus eine Berechtigung haben kann. Unverständlich ist es unserer Meinung nach jedoch bei Wirtschaftsinformatik-Lehrbüchern. In Standardlehrwerken der Wirtschaftsinformatik (z. B. [Me17], [La16], [La18]) werden zum Beispiel zwar Ausführungen zur objektorientierten Programmierung geboten, Informationen zur Logischen Programmierung und Constraint Programmierung sucht man jedoch derzeit vergeblich.
In den folgenden Abschnitten werden nach einer kurzen Motivationsuntermauerung einige geeignete Beispiele – ohne Anspruch auf Vollständigkeit - für die Inhalts- und Vorgehensgestaltung der LP[1]/ C(L)P[2]-bezogenen Lehrkomponenten von Informatik-/ Wirtschaftsinformatik-Kursen für Studierende der Betriebswirtschaftslehre präsentiert, die an der Hochschule für Wirtschaft, Technik und Kultur seit 5 Jahren erfolgreich verwendet werden[3]. Basierend darauf lassen sich – je nach Zeitrahmen - weitere (betriebswirtschaftliche) Problemstellungen beziehungsweise Lösungsansätze für diese aufbauen, so dass sich mit Anreicherung um Theorieaspekte auch Konzepte für gesamte LP- oder/ und CP-Kurse im Rahmen von Wirtschaftsinformatikstudiengängen o.ä. aufbauen lassen, wie dies zum Beispiel für den HWTK-Studiengang „Informatik und Management" praktiziert wird.

2 Motive

Die Rollenbilder von Betriebswirten haben sich in den vergangenen Jahren deutlich gewandelt und werden sich in Zukunft teilweise drastisch weiter wandeln. Viele menschliche Arbeitsinhalte werden durch Softwarefunktionalitäten substituiert werden (Digitalisierung, Digitale Transformation, vgl. [Jo15], [Jo17]). Dies wird nicht nur „einfache" Verwaltungs- und Beratungsfunktionen betreffen, sondern auch komplexe Managementprozesse, für die es mindestens leistungsstarke Entscheidungsunterstützungssysteme geben wird. Dies wird insbesondere durch die Reifung von IT-Technologien und durch disruptive Technologien ermöglicht, wobei Technologien der Künstlichen Intelligenz eine besondere Rolle spielen werden. Der Erfahrung nach waren in den vergangenen Jahren Studierende der Betriebswirtschaftslehre zu einem erheblichen Anteil „wenig Informatik-affin". Die meisten von Ihnen verfügen zum Beispiel über keinerlei Kenntnisse auf den Gebieten Programmierung und Softwareentwicklung. Eine ablehnende oder ignorierende Haltung gegenüber IT-Inhalten und

[1] Logische Programmierung
[2] Constraint (Logische) Programmierung
[3] davor in Teilen bereits im Bachelor-Studiengang Betriebswirtschaftslehre der Hochschule Lausitz (2011) und im Masterkurs „Algorithms and Optimizations" (2009 – 2011) des Studienganges „Internationale Medieninformatik" der HTW Berlin (in anderem inhaltlichen Kontext und mit anderer didaktischer Einbindung).

IT-gestützten Prozessen wird perspektivisch nicht mehr praktikabel sein, da eine erfolgreiche Betätigung im Berufsleben in aller Regel nur mit entsprechenden Kenntnissen der Informatik/ Wirtschaftsinformatik möglich sein wird. Selbstverständlich muss und soll ein Betriebswirt nicht ein Informatiker werden/ sein, Ausrichtung und Detailierung der IT-spezifischen Kenntnisse müssen auf einem „geeigneten Niveau“ vorliegen. Insbesondere sollte das Wissen aktuelle Kenntnisse bezüglich konzeptioneller und technologischer Möglichkeiten und bezüglich der damit verbundenen Herausforderungen, z. B. Kosten, Entwicklungsdauer und Risiken, umfassen.

3 Mögliche Inhalte & erprobtes Vorgehen

Die LP- und CP/CLP-thematisierenden Lehrkomponenten sind in die Wirtschaftsinformatikkurse thematisch einzubetten und ggf. in inhaltlichen Kontext zum Operations Research zu stellen. Analysiert man die Wirtschaftsinformatik-Curricula von BLW-Studiengängen verschiedener Hochschulen/ Universitäten, so sind deutliche Unterschiede festzustellen. An der HWTK wird neben den klassischen Inhalten insbesondere Wert auf der Vermittlung von Überblickwissen über aktuelle Technologien, Paradigmen und IT-Themenfelder (z.B. Cloud Computing, SOA, BIG Data, IT-Sicherheit & Datenschutz, Digitale Transformation, Industrie 4.0 usw.) gelegt, so dass die Studierenden diesbezüglich ein belastbares und zukunftssicheres Wissen erwerben. Ein separater, zeitlich nachgelagerter, Teilkurs widmet sich derzeit dem Themengebiet Datenbanken.
Umfang und inhaltliche Tiefe der Lehrkomponenten können je nach zur Verfügung stehender Zeit variiert werden. Wir veranschlagen für die Themenkomplexe LP & CP derzeit in der Regel bis zu 4 Unterrichtseinheiten a 45 Minuten.

3.1 Thematische Vorarbeit

Die LP- und CP/CLP-Lehrkomponenten werden in den Themenkomplex Programmierparadigmen eingebettet. Hier stellt sich die Frage, ob die deklarative Programmierung schwerpunktmäßig vor der imperativen Programmierung behandelt werden sollte oder danach, insbesondere da die meisten BWL-Studierenden über keinerlei Programmierkenntnisse verfügen. Bei uns wurden in verschiedenen Semestern beide Reihenfolgevarianten praktiziert, wobei eine Präferenz herausgebildet wurde, die deklarative Programmierung vor der imperativen Programmierung zu behandeln. Zeitlich vorgelagert werden die Studierenden unter anderem mit den Begrifflichkeiten *Berechenbarkeit, Entscheidbarkeit, kombinatorische Komplexität*[4] und mit exemplarischen Problemklassen aus dem Alltag von Unternehmen vertraut gemacht. Untermauert werden kann dies mit „Stories aus der Praxis“, da viele Betriebswirte/ Manager diesbezüglich kein belastbares Wissen besitzen und daher leider Fehleinschätzungen und Managementfehler entstehen, die ggf. zum Scheitern von Projekten führen und/ oder erhebliche, vermeidbare Kosten verursachen.

[4] zum Beispiel anhand des Beispiels der Türme von Hanoi

3.2 Logische Programmierung

Beim Themenkomplex *Logische Programmierung* beginnen wir mit der Vorstellung des grundlegenden Prinzips des Paradigmas der Logischen Programmierung und der damit verbundenen Begrifflichkeiten (Wissensbasis, Fakten, Regeln, Inferenzmaschine, Anfragen, Resultate), ohne ins Detail zu gehen.

Anschließend präsentieren wir verschiedene grundlegende hierarchische Strukturen aus betriebswirtschaftlichem Kontext, z.B. Zuliefernetze, Organigramme, Produkt- und Prozessbäume. Anhand dieser diskutieren wir praktische Fragestellungen, die unter anderem das Themenfeld Informations- und Wissensmanagement betreffen. Anhand eines Beispielproblems präsentieren wir (bei ausreichender Zeit ist eine gemeinsame Entwicklung didaktisch sinnvoll) ein Prologprogramm, dass dieses Problem in Form von Regeln und Fakten spezifiziert und erklären dieses. Für das „Top-Boss-Problem" (Unternehmensorganigramm) kann das folgendermaßen aussehen:

```
vorgesetzter(X, Y):-
        direkterVorgesetzter(X, Y).
vorgesetzter(X, Y):-
        direkterVorgesetzter(X, Z),
        vorgesetzter(Z, Y).
topBoss(X, Y):-
        vorgesetzter(X, Y),
        not(direkterVorgesetzter(_, X)), !.     % ! ist nichtdeklarative Ausnahme
direkterVorgesetzter(heinzMeier, erwinMüller).
direkterVorgesetzter(drDieterZetsche, heinzMeier).
…
```

Anhand von exemplarischen Anfragen wird die Vorgehensweise der Inferenzmaschine einschließlich *Backtracking* (grob) erklärt; Beispielanfrage:

? topBoss(X, karlWichtigtuer).
Ausgabe: *X = drDieterZetsche*

Bei zeitlicher Passung könnte man nun eine Erweiterung der präsentierten Problemstellungen vornehmen, die komplexe Management-Fragestellungen adressieren und die dann Übungsinhalte für die Studierenden bilden. Beispiele hierfür sind unter anderem die Aufnahme von zeitlichen Aspekten/ Historieninformationen in Liefernetze oder in Organigramme. Zusätzlich (mit Hinweis auf Analogien) oder alternativ könnte nicht-betriebswirtschaftliches „Alltagswissen" modelliert werden, z.B. wie klassisch oft getan, Wissen über Verwandtschaftsbeziehungen.
Insgesamt sollte den Studierenden klar werden, welche Potenziale die Logische Programmierung für die Realisierung von effizientem Informations- und Wissensmanagement besitzt. Eine Inkontextsetzung zu weiteren Aspekten/ Inhalten des Wissensmanagements sollte an geeigneter Stelle erfolgen.

3.3 Constraint Programmierung/ Constraint Logische Programmierung

Je nach zur Verfügung stehender Zeit kann man (klassisch) mit der Modellierung eines „Buchstabenrätsels“, wie zum Beispiel SEND + MORE = MONEY (vgl. z.B. [AW07]) starten, die den Studierenden ein Gefühl vermitteln, wie Problemstellungen deklarativ modelliert werden können. Eine Lösungsfindung sollte einerseits durch reine *logische Programmierung* und andererseits durch *constraint-logische Programmierung* realisiert werden. Beide Ansätze sollen hinsichtlich der Laufzeit verglichen werden.
Wir wählen in der Regel initial das „Problemfeld“ der *magischen Quadrate*, wobei sich durch zusätzliche Informationen, etwa aus [He09], das Interesse der Studierenden steigern lässt. Wir starten mit magischen Quadraten in der 3x3-Ausprägung.
Es bietet sich an, ein solches zunächst durch ein rein logisches Programm zu spezifizieren, wobei die Studierenden die einzuordnenden aufeinander folgenden Zahlen (hier 25 bis 33) selbst wählen dürfen, z.B.:

```
start:-
    cputime(Ts),
    % jede Zahl soll genau einmal vertreten sein
    member(25,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(26,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(27,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(28,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(29,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(30,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(31,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(32,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    member(33,[A1,A2,A3,A4,A5,A6,A7,A8,A9]),
    % gleiche Zeilensumme
    Z1 is A1 + A2 + A3,
    Z2 is A4 + A5 + A6,
    Z3 is A7 + A8 + A9,
    Z1 == Z2, Z2 == Z3,
    % gleiche Spaltensummen
    S1 is A1 + A4 + A7,
    S2 is A2 + A5 + A8,
    S3 is A3 + A6 + A9,
    Z3 == S1, S1 == S2, S2 == S3,
    % gleiche Diagonalensummen
    D1 is A1 + A5 + A9,
    D2 is A3 + A5 + A7,
    S3 == D1, D1 == D2,
    cputime(Te), T is Te - Ts,
    write('Loesung: '),nl,
    write([A1,A2,A3]),nl,
    write([A4,A5,A6]),nl,
    write([A7,A8,A9]),nl,
    write('gefunden in '),write(T), write(' msec.'),nl.
```

Nach Aufruf der Prädikates *start* erfolgt relativ schnell die Ausgabe einer Lösung, z.B. mit CHIP unter Windows 7 auf einem i5-3230M, 2,6 GHz mit 12 GB Hauptspeicher in 234 msec (CPU-Zeit):

```
[26, 31, 30]
[33, 29, 25]
[28, 27, 32]
gefunden in 234 msec.
```

Ein analoges (vorbereitetes) Programm für die Berechnung eines 5x5-Magischen Quadrates könnte nun „gestartet“ werden. Da die Lösungsfindung erhebliche Zeit benötigt, könnte man nun entweder erste einfache betriebswirtschaftliche kombinatorische Planungsprobleme diskutieren und modellieren oder aber die prinzipielle Funktionsweise der Constraint (Logischen) Programmierung erläutern. Alternativ, didaktisch unseres Erachtens besser platziert, wird die Funktionsweise nach den „praktischen Übungen“ und den damit verbundenen Erfahrungssammlungen vorgestellt.
Eine Lösung für die Berechnung des 5x5-Magischen Quadrates lässt dann „immer noch auf sich warten“, daher sollte nun die Modellierung des 3x3-Problems *constraint-logisch* erfolgen[5], wobei durch Zuweisung der „Matrix“ zu einer Variable und deren anschließende Verwendung den Programmcode verkürzen, vermutlich aber die „erste Verständlichkeit“ auch erschweren würde:

```
start:-
    [A11,A12,A13,
    A21,A22,A23,
    A31,A32,A33]::25..33,
    S::0..10000,
    S #= A11 + A12 + A13,
    S #= A21 + A22 + A23,
    S #= A31 + A32 + A33,
    S #= A11 + A21 + A31,
    S #= A12 + A22 + A32,
    S #= A13 + A23 + A33,
    % Diagonalen
    S #= A11 + A22 + A33,
    S #= A31 + A22 + A13,
    alldifferent([A11,A12,A13,A21,A22,A23,A31,A32,A33]),
    suchewerte([A11,A12,A13,A21,A22,A23,A31,A32,A33]),
    schreib9([A11,A12,A13,A21,A22,A23,A31,A32,A33]).
```

[5] hier mittels *CHIP* (Constraint Handling in Prolog, www.cosytec.com) dargestellt, alternativ lassen sich unter anderem *ECLiPSe* (www.eclipseclp.org) oder *GNU Prolog* (www.gprolog.org) einsetzen.

```
suchewerte([]).
suchewerte([H|T]):-
    indomain(H),
    suchewerte(T).

schreib9([A11,A12,A13,A21,A22,A23,A31,A32,A33]):-
    write(A11),write('  '),write(A12),write('  '),write(A13),nl,
    write(A21),write('  '),write(A22),write('  '),write(A23),nl,
    write(A31),write('  '),write(A32),write('  '),write(A33),nl.
```

Die unmittelbare Lösungslieferung,

```
26  31  30
33  29  25
28  27  32  (in 15 msec gefunden)
```

sollte bei den Studierenden die Effekte der Constraint Programmierung erkennbar machen. Für die Lösungsfindung wurden 15 msec statt 234 msec benötigt. Man kann die Aufgabenstellung nun erweitern und die Berechnung aller Lösungen verlangen, was durch gezieltes „failen" und dem dadurch ausgelösten Backtracking realisiert und demonstriert werden kann. Nach dieser Demonstration und der Erkenntnis, dass das „reine" Prologprogramm zur Berechnung eines 5x5-Magischen Quadrates immer noch läuft[6], gilt es, die Euphorie hinsichtlich des CP-Ansatzes zu begrenzen und sich dem Thema „*redundante Constraints*" zu nähern. Dafür starten wir nun ein CP-Programm[7] für 5x5-Quadrate (gleiche Zahlen wie im Prologprogramm) [8]:

```
start:-
    [A11, A12, A13, A14, A15,
    A21, A22, A23, A24, A25,
    A31, A32, A33, A34, A35,
    A41, A42, A43, A44, A45,
    A51, A52, A53, A54, A55]::423..447,
    S::0..10000, % Summe
    % Zeilen-Constraints
    S #= A11 + A12 + A13 + A14 + A15,
    S #= A21 + A22 + A23 + A24 + A25,
    S #= A31 + A32 + A33 + A34 + A35,
    S #= A41 + A42 + A43 + A44 + A45,
    S #= A51 + A52 + A53 + A54 + A55,
```

[6] Man kann sich darauf einstellen, dass auch zum Ende der Lehrveranstaltung weder eine Lösung berechnet wurde noch eine Nichtlösbarkeit nachgewiesen wurde.

[7] Das präsentierte Programm ist ein ad-hoc-Programm und könnte natürlich kompakter und „eleganter" formuliert werden.

[8] zur Auflockerung kann man an der Tafel/ am Whiteboard „Zahlenkünstler-Tricks" präsentieren, mit denen sich magische Quadrate mit ungerader Kantenlänge lösen lassen. Es sollte darauf hingewiesen werden, dass es für Magische Quadrate mit ungerader Kantenlänge ein konstruktives Berechnungsverfahren gibt und dass es solche Verfahren für kombinatorisch komplexe Probleme praktischer Natur i.d.R. nicht gibt.

```
    % Spalten-Constraints
    S #= A11 + A21 + A31 + A41 + A51,
    S #= A12 + A22 + A32 + A42 + A52,
    S #= A13 + A23 + A33 + A43 + A53,
    S #= A14 + A24 + A34 + A44 + A54,
    S #= A15 + A25 + A35 + A45 + A55,
    % Diagonal-Constraints
    S #= A11 + A22 + A33 + A44 + A55,
    S #= A51 + A42 + A33 + A24 + A15,
    alldifferent( [A11, A12, A13, A14, A15,
             A21, A22, A23, A24, A25,
              A31, A32, A33, A34, A35,
              A41, A42, A43, A44, A45,
              A51, A52, A53, A54, A55]),
    cputime(T1),
    suchewerte( [A11, A12, A13, A14, A15,
        A21, A22, A23, A24, A25,
         A31, A32, A33, A34, A35,
         A41, A42, A43, A44, A45,
         A51, A52, A53, A54, A55]),
    cputime(T2),
    schreib_25([A11, A12, A13, A14, A15,
        A21, A22, A23, A24, A25,
         A31, A32, A33, A34, A35,
         A41, A42, A43, A44, A45,
         A51, A52, A53, A54, A55]),
        Tg is T2 - T1,
        write('geloest in '), write(Tg), write(' msec').

schreib_25([A11, A12, A13, A14, A15,
        A21, A22, A23, A24, A25,
         A31, A32, A33, A34, A35,
         A41, A42, A43, A44, A45,
         A51, A52, A53, A54, A55]):-
    write(A11),write('  '), write(A12), write('  '),write(A13),write('  '),write(A14),write('  '),write(A15),nl,nl,
    write(A21),write('  '), write(A22), write('  '),write(A23),write('  '),write(A24),write('  '),write(A25),nl,nl,
    write(A31),write('  '), write(A32), write('  '),write(A33),write('  '),write(A34),write('  '),write(A35),nl,nl,
    write(A41),write('  '), write(A42), write('  '),write(A43),write('  '),write(A44),write('  '),write(A45),nl,nl,
    write(A51),write('  '), write(A52), write('  '),write(A53),write('  '),write(A54),write('
'),write(A55),nl,nl,nl.
```

Die Lösungsfindung benötigt trotz CLP-basierter Modellierung erhebliche Zeit. Die Studierenden sollen nun überlegen, welches Wissen man über die Lösung oder die Struktur der Lösung noch generieren kann, ohne eine Lösung zu kennen. In der Regel kommt jemand darauf, dass man die Summe einer Zeile/ einer Spalte/ einer Diagonalen berechnen kann, wenn man die einzuordnenden Zahlen kennt (mit unseren gewählten Werten ist die Summe 2175; es bittet sich an, in diesem Kontext die von Gauß gefundene Berechnungsmethode zu wiederholen). Wenn nicht, muss man diese Idee selbst herleiten. Diese Erkenntnis nutzend wird zusätzlich das redundante Constraint *S #= 2175* eingefügt. Ein erneuter Start der Problemlösung liefert rasch eine Lösung:

```
423  424  435  446  447
425  444  441  428  437
445  438  432  433  427
443  429  431  442  430
439  440  436  426  434
geloest in 125 msec
```

In diesem Kontext wird über die Bedeutung von redundanten Constraints diskutiert. Als didaktisch sinnvolle Programmierübung wird das Programm nun flexibilisiert, d.h., die Berechnung des Summenwertes wird durch das Programm durchgeführt, somit können durch das Programm ohne weiteres verschiedene Probleminstanzen der Problemklasse *5x5-Magische Quadrate* gelöst werden.

Nun widmen wir uns in der Regel einer nächsten, komplexeren Problemklasse (7x7-Quadrate). Hierbei wird klar, dass das Hinzufügen der Zielen-/ Spalten-/ Diagonalensumme als redundantes Constraint nicht ausreichend ist, um in kurzer Zeit eine Lösung zu finden. Bei ausreichend zur Verfügung stehender Zeit oder aber als Inhalt „studentischer Überlegungen" in Begleitung zur Lehrveranstaltungsreihe kann man hier weiterführende Ideen „beauftragen" und testen lassen. Es bietet sich an, hier das Thema *Heuristiken* zu thematisieren.

Wenn nicht schon geschehen (vgl. oben) sollte nun das allgemeine Prinzip der Constraint-Programmierung (einschließlich Propagation[9]) vorgestellt werden. Praktische Problemklassen aus der Betriebswirtschaft, die durch Constraint-Programmierung adressiert werden können, sollten wiederholt/ vorgestellt und diskutiert werden, z.B. *Ablaufplanungsprobleme*, *Fließproduktionsplanung*, *Konfigurationsprobleme* und Multiressourcen-Planungsprobleme, ggf. mit Bezug zum Operations Reasearch, wobei auch hier die Bedeutung der *Deklarativität* der jeweiligen Problemspezifikation thematisiert werden sollte.

Weitere Vorteile (*Unterstützung von flexibler Interaktivität, Anwendbarkeit trotz unvollständigen Wissens* usw.), Potenziale und für praktische Problemlösungen ggf. erforderliche Erweiterungen (z.B. *weiche Constraints*, *Constraint-Hierarchien*, *Präferenzen*) sowie weiter bestehende Herausforderungen der Constraint Programmierung sollten deutlich und einprägsam benannt werden. Erwähnt werden sollten (ohne detaillierte Behandlung) zum Beispiel insbesondere auch *globale Constraints,* z. B. anhand des in den Beispielen verwendeten *alldifferent-Constraints* und deren Bedeutung mit Bezug zum Operations Research. Es sollte darüber hinaus darauf hingewiesen werden, dass für praktische Problemklassen unter Umständen die Notwendigkeit von hybriden Ansätzen/ Modellen erforderlich ist.

[9] zum Beispiel anhand der Intervallgrenzenpropagation

Um die inhaltliche Verfestigung und Wiederholung zum Beginn einer nächsten Wirtschaftsinformatik-Lehrveranstaltung zu unterstützen, rufen wir in der Regel dazu auf, „richtig schwere“ Sudokus „zu ermitteln“. Diese werden dann im Kontext einer inhaltlichen Wiederholung constraint-basiert gelöst. Für ergänzende Informationen sei hierfür auch auf [Si05] verwiesen.

4 Erfahrungen

Wie in Kapitel 2 erwähnt, sind zum Beginn des BWL-Studiums „relativ viele“ Studierende „wenig Informatik-affin“. Der Erfahrung des Autors nach wird bei einem erheblichen Anteil dieser Studierenden durch Integration der beschriebenen Lehrinhalte in der benannten Vorgehensweise ein „gewisses Interesse“ und Verständnis für Informatik/ Wirtschaftsinformatik geweckt/ ermöglicht. Die Studierenden akzeptieren, dass die Beschäftigung mit „aktuellen IT-Themen“ i.d.R. unabdingbar und nutzbringend ist, wenn man ein zukunftssicheres Studium absolvieren will.

Die Tatsache, dass die deklarative Programmierung zeitlich vor der imperativen Programmierung eingeführt wird, begünstigt unseres Erachtens die studentische Erkenntnis, dass Programmieren relativ einfach sein kann und erhöht die Bereitschaft, sich auch mit anderen Themengebieten der Informatik/ Wirtschaftsinformatik zu beschäftigen. Durch die Inkontextsetzung zu betriebswirtschaftlichen Problemstellungen und Managementaufgaben sowie durch die Einbettung in das „big picture“ *Digitales Unternehmen* (vgl. [Joh15], [Joh17]) erkennen die Studierenden die Bedeutung deklarativer Spezifikationen und der deklarativen Programmierung. Sie sind in der Regel in der Lage, abhängig von der Problemklasse zu erkennen, welche Programmierparadigmen potenziell für eine softwaretechnische Unterstützung/ Realisierung in Frage kommen. Neben den potenziellen Möglichkeiten sind sie aber auch hinsichtlich der bestehenden Grenzen, Risiken, Notwendigkeiten und Herausforderungen sensibilisiert, was sie dann insgesamt von einer Vielzahl heutiger Manager unterscheidet[10].

Selbstverständlich stellt die für die skizzierten Lehrinhalte zur Verfügung stehendende Zeit eine Limitierung dar, insbesondere da im Rahmen der Wirtschaftsinformatik-Lehrveranstaltungen ein breiter, belastbarer Überblick über aktuelle Paradigmen, Technologien etc. vermittelt werden soll. Perspektivisch wäre zu überlegen, ob im Rahmen von BWL-Studiengängen mehr Zeit für Fächer, wie Wirtschaftsinformatik, Digitale Transformation u.ä. zur Verfügung gestellt werden. Könnte dadurch das Zeitvolumen für eigene (ggf. angeleitete) studentische Programmierübungen vergrößert werden, würden sich u.E. weitere positive Effekte, wie z.B. höhere Akzeptanz, besseres Verständnis und Förderung von Kreativität ergeben.

[10] ebenfalls von den BWL-Studierenden, die auch heute noch inhaltlich stark eingeschränkte Wirtschaftsinformatik-Lehrveranstaltungen absolvieren.

5 Fazit und Ausblick

Das Berufsbild des Betriebswirts unterliegt seit einiger Zeit einem drastischen Wandel. Insbesondere durch die allgegenwärtige, „existenzentscheidende" Digitalisierung und die Notwendigkeit vieler Unternehmen und anderer Gesellschaftsbereiche zur Digitalen Transformation, rücken IT-Themen und –aufgaben unweigerlich in den Arbeitsfokus von Betriebswirten. Einerseits sind sie unter Umständen von der fortschreitenden Digitalisierung selbst betroffen, was zur Notwendigkeit einer berufsinhaltlichen Umorientierung führt, andererseits sind sie potenzielle Entscheidungsträger, Projektmitarbeiter, Projektbegleiter oder Projektmanager in Digitalisierungsprojekten oder in Projekten im Digitalisierungskontext. Dieser Umstand erfordert, dass Betriebswirte ein breites belastbares – in geeigneter Granularität und Tiefe - Wissen auf dem Gebiet der Wirtschaftsinformatik besitzen beziehungsweise erwerben.
Neben dem Wissen um einschlägige aktuelle und zukunftsträchtige IT-Technologien, wie zum Beispiel dem Cloud Computing, IoT, BIG Data usw. sollten Betriebswirte einen belastbaren Überblick über Paradigmen besitzen, die bei der Realisierung IT-gestützter, effizienter und agiler Unternehmensprozesse zum Einsatz kommen können. Zu diesem Wissen gehört auch Wissen über relevante Programmierparadigmen und deren Anwendbarkeit für die Realisierung von Softwarekomponenten, z.B. für die Lösung (oder mindestens Lösungsunterstützung/ Entscheidungsunterstützung) von komplexen betriebswirtschaftlichen Aufgabenstellungen, wobei diese insbesondere im Kontext des Digitalen Unternehmens (vgl. [Jo15], [Jo17]) zusätzlich eine entscheidende Bedeutung besitzen werden. Hierbei bilden Vertreter der deklarativen Programmierung, z.B. die Logische Programmierung und die Constraint (Logische) Programmierung, eine besondere Klasse von Programmierparadigmen. Umso unverständlicher ist es, dass diese bisher nur in wenigen Curricula für die Wirtschaftsinformatik-Lehrveranstaltungen von BWL-Studiengängen Einzug gefunden haben und auch in Standardwirtschaftsinformatik-Lehrbüchern i.d.R. keine nennenswerte Erwähnung finden. Ein Grund könnte sein, dass viele Wirtschaftsinformatikprofessoren und Lehrbuchautoren selbst keine oder nur wenig Erfahrungen und Kenntnisse auf den Gebieten der Logischen Programmierung und Constraint (Logischen) Programmierung besitzen. Das Paper motiviert die Aufnahme entsprechender Lehrinhalte in die BWL-Wirtschaftsinformatik-Lehrveranstaltungen, stellt einige gut geeignete und erprobte Komponenten für die diesbezügliche Lehre vor, nennt Ausbau- und Fortführungsmöglichkeiten und skizziert ein jahrelang praktiziertes Vorgehen, das natürlich dynamisch weiterentwickelt wird/ werden sollte.
Sicherlich ist es in „reinen" Wirtschaftsinformatik-Studiengängen und wirtschaftsinformatiknahen Studiengängen sehr empfehlenswert, den adressierten Themengebieten einen größeren zeitlichen und inhaltlichen Umfang einzuräumen. Einen besonderen Stellenwert sollten im Kontext der Constraint Programmierung *globale Constraints* und deren Anwendung bei Problemmodellierungen bilden. Ergänzend zur Constraint Logischen Programmierung könnten auch Constraint-Bibliotheken für imperative Programmiersprachen (z.B. CHOCO, JaCoP oder Gecode) und Modellierungssprachen, wie OPL und MiniZinc thematisiert werden. Um den identifizierten Anforderungen gerecht zu werden, gibt es zum Beispiel im Curriculum des HWTK-

Studienganges „Informatik und Management“ gesonderte Kurse, die sich der *Logischen Programmierung*, der *Constraint Programmierung* und der *Künstlichen Intelligenz* widmen.

Literatur

[AW07] Apt, K., Wallace, M.: Constraint Logic Programming using ECLIPSE, Cambridge University Press, 2007.

[Do15] Domschke, W. et al: Einführung in Operations Research. 9. Auflage, Springer Gabler, 2015.

[He09] Heinrich, W.: Geschichte und Anwendung von Magischen Quadraten in Zeit und Raum, Bohmeier, 2009.

[Jo15] John, U.: Digitales Unternehmen – Bausteine für Effizienz, Agilität und Transparenz. In (Cunningham, D. et al (Ed.)) Proc. INFORMATIK 2015, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, 2015.

[Jo17] John, U.: Deklarative Unternehmensmodelle – essentielle Bausteine Digitaler Unternehmen für optimierende Reorganisationen. INFORMATIK 2017 (Maximilian Eibl, Martin Gaedke (Ed.)), Lecture Notes in Informatics (LNI), Gesellschaft für Informatik 2017.

[La16] Laudon, K. C., Laudon, J. P., Schoder, D.: Wirtschaftsinformatik. 3. Auflage, Pearson, 2016.

[La18] Laudon, K. C., Laudon, J. P.: Management Information Systems - Managing the Digital Firm. 15. Auflage, Pearson, 2018.

[Me17] Mertens, P. et al: Grundzüge der Wirtschaftsinformatik. 12. Auflage, Springer Gabler, 2017.

[Si05] Simonis, H.: Sudoku as Constraint Problem. Working Notes of the CP 2005 Workshop on Modeling and Reformulating Constraint Satisfaction Problems.

Ulrich John, Petra Hofstedt, Armin Wolf (Hrsg.): Deklarative KI-Ansätze – Post-Proceedings des W(C)LP 2018, Hochschule für Wirtschaft, Technik und Kultur, Berlin 2019.

Künstliche Intelligenz und Digitalisierung – sind wir auf dem richtigen Weg?

Ulrich Geske

Universität Potsdam
ulrich.geske@uni-potsdam.de

Abstract: Es ist das Jahr 2018 und der Begriff „Künstliche Intelligenz“ (KI) ist nach fast zwei Jahrzehnten Verdrängung aus dem allgemeinen Bewusstsein ein Diskussionsthema in der Öffentlichkeit, stärker als jemals zuvor aber auch kontroverser als früher. Weltweit äußerst erfolgreich agierende amerikanische Unternehmen haben eine Innovation auf dem Gebiet der künstlichen neuronalen Netze, die im Jahr 2009 durch die Computerentwicklung möglich wurde, aufgegriffen und durch intensive Forschungstätigkeit für Anwendungen verfügbar gemacht. Hier soll analysiert werden, inwieweit einerseits Verklärungen von KI zu sich selbst programmierenden Systeme zutreffend sind und andererseits die Forderung nach Transparenz dieser Systeme ein sinnvoller Ansatz ist, um sicher zu sein, dass ethische Normen eingehalten werden.

Keywords: Digitalisierung, Künstliche Intelligenz, KI-Initiativen, selbstlernende Systeme, selbstprogrammierende Systeme, Transparenz, Ethik

1 Einleitung

Zugegeben, es überwiegen im Folgenden die Fragezeichen. Fatalerweise lässt die Entwicklungsgeschwindigkeit der Digitalisierung und Künstlichen Intelligenz kaum Zeit, richtige Antworten zu finden. Der Begriff und das Forschungsgebiet Künstliche Intelligenz existieren seit 1956. Verglichen mit der Mathematik handelt es sich bei der Künstlichen Intelligenz um eine äußerst junge Wissenschaft. Ein weiterer gravierender Unterschied zur Mathematik und anderen Wissenschaften kann darin gesehen werden, dass Mathematik die Welt vor allem erklären will, während Forschung zur Künstliche Intelligenz die Welt vor allem verändern wird, wenn der Aspekt einer möglichen Erklärbarkeit menschlicher Intelligenz außeracht gelassen wird. Zum 50-jährigen Jubiläum der Etablierung der Künstlichen Intelligenz, das in Deutschland im Jahr 2006 während der 29. Jahrestagung Künstliche Intelligenz an der Universität Bremen mit Marvin Minsky und Joseph Weizenbaum, Pionieren der Arbeiten zur Künstlichen Intelligenz, feierlich begangen wurde, war dieser Anlass nur für Insider von Bedeutung. Es war gerade mal wieder KI-Winter. Damit werden die Zeiträume bezeichnet, in denen nach nicht eingetroffenen Erwartungen das Interesse an Künstlicher Intelligenz fast erlosch und Forschungsförderung so gut wie ausblieb. Es war nicht der erste KI-Winter aber rückblickend der schwerwiegendste. Viele grundlagenorientierte deutsche Informatik- und KI-Forschungszentrum, das ECRC in München, das FAW in Ulm, die Gesellschaft für Mathematik und Datenverarbeitung (GMD) wurden innerhalb weniger Jahre um die Jahrtausendwende geschlossen oder wurden umorientiert. Ein erneuter Aufschwung setzte erst ein paar Jahre später, außerhalb Deutschlands und Europas, durch die Entwicklung bei neuronalen Netzen ein.

2 Geht die Entwicklung hin zu selbstprogrammierenden Systemen?

Die Technikentwicklung in den vorangegangenen industriellen Revolutionen hatte ähnliche verändernde Auswirkungen wie sie durch die Künstlichen Intelligenz bewirkt werden könnten und doch gibt es Unterschiede. Sie liegen in der Geschwindigkeit des Fortschritts der Entwicklung und in der Art der Ersetzung menschlicher Arbeit durch maschinelle Arbeit. Erste, zweite und dritte industrielle Revolutionen haben ca. 160 Jahre, 80 Jahre und 40 Jahre gedauert. Werden die Umwälzungen durch die vierte industrielle Revolution, geprägt von Künstlichen Intelligenzen, tatsächlich nach nur etwa 20 Jahre im Wesentlichen vollzogen sein oder werden sie dann bereits von einer noch weiter gehendenden Nachfolgeentwicklung abgelöst werden?

Während bisher die Überlegenheit der Maschinen gegenüber dem Menschen durch Ausdauer und Kraft gegeben war, ist Künstliche Intelligenz durch die Ablösung kognitiver menschlicher Leistungen, die wir als „intelligent“ bezeichnen, mithilfe von Computern, Robotern und Software geprägt. Derartige intelligente Leistungen bestehen heute im deduktiven und induktiven Schlussfolgern, auch unter Einschluss von unscharfem oder unsicherem Wissen, in Bilderkennung und maschinellen Lernen. Inwieweit dabei „maschinelle Intelligenz“ auch Kreativität einschließen kann, bleibt abzuwarten. Dieser Aspekt spielt aber eine wichtige Rolle bei den Diskussionen sowohl bei den optimistischen als auch bei den pessimistischen Beobachtern der Entwicklung. Gleichgültig, wie dieser Punkt zu bewerten ist, wird die Ablösung intelligenter menschlicher Leistungen, eventuell vorerst auch nur einzelner davon, zu erheblichen Veränderungen im Arbeits- und gesellschaftlichen Leben führen.

Wie auch in vorangegangenen industriellen Revolutionen werden menschliche Arbeiten nicht nur unterstützt oder erleichtert oder überhaupt erst ermöglicht, sondern auch gänzlich abgelöst werden (Disruption) und dadurch sowohl als Fortschritt empfunden werden als auch Spannungen nach sich ziehen. Durch das Aufkommen des Autos wurden alsbald Kutscher nicht mehr gebraucht, durch autonome Autos könnten demnächst Taxi- und Lkw-Fahrer, Fahrschulen überflüssig werden. Andererseits wird Autofahren, und zwar autonomes, stressfreier und sicherer werden. Bereits heute sind digitale Techniken aus dem täglichen Leben nicht mehr wegzudenken: Briefe schreiben ist weitestgehend ersetzt durch **Email** und ähnliche digitale Verfahren, Bankkonten **online** führen, Tickets buchen per **Internet**, Nachschlagewerke sind ersetzt durch "**Google**"-Suche, fotografieren erfolgt fast ausschließlich digital, das **Navigationsgerät** ersetzt die Landkarte.

Künstliche Intelligenz kann bereits oder wird demnächst unter Verzicht auf menschliche Bearbeiter schwierige und rational aufwändige Prozesse erledigen, wie **Kreditwürdigkeit** ermitteln, Personenidentifikation durch **Gesichtserkennung** vollziehen, **Werbung** gezielt platzieren, Teams zusammenstellen, Auswahl von Stellenbewerbern vornehmen, **Blockchain-Technik** zur sicheren und transparenten Buchführung etablieren, **3D-Drucker** zur Herstellung jeglicher Teile weiter vervollkommnen, **Smart Meter** zur Überwachung des Elektroverbrauchs selbstverständlich werden lassen, Anfertigung von **Berichten** über Texte oder visuelle Ereignisse erlauben, durch Entscheidungsfindungssysteme die Verwaltung effizienter machen, **bargeldlosen** Warenverkehr bis zum Endverbraucher verbreiten. Software, digitale Prozesse, Künstliche Intelligenz sollen Lösungen finden und Entscheidungen treffen, die diskriminierungs- und vorurteilsfrei sind. Auch wegen der enormen Leistungsfähigkeit der Computer können maschinelle Systeme dabei schneller und besser als menschliche Bearbeiter agieren. Besonders hilfreich sind derartige Systeme bei medizinischen Anwendungen, wie z.B. Exoskelette, die in der Rehabilitation erneutes Laufen lernen nach Verletzungen erleichtern und den Trainingsassistenten spürbar entlasten.

In der bisherigen Entwicklung der Künstlichen Intelligenz war es vorwiegend so, dass es die KI-Forscher waren, die große Erfolge innerhalb kürzester Fristen proklamiert hatten: z.B. einen general problem solver, Formelmanipulationssysteme, Expertensysteme, sprachverstehende und -übersetzende Systeme, die Fünfte Rechnergeneration Parallelrechnern, Robotik und mit intelligenten sozialen Anwendungen, Artificial Life. Nunmehr ist es genau anders herum. Von außen, von Unternehmen, Medien, Politik werden Erwartungen an „selbstlernende" und sich „selbst programmierende" Systeme geweckt, die vermutlich erneut nicht zu erfüllen sein werden. Diese Mystifizierung der Künstlichen Intelligenz durch das Hauptaugenmerk auf „sich selbst programmierende und selbst lernende" Systeme, insbesondere aber ihres Teilgebiets der künstlichen neuronalen Netze, enthält die Gefahr einer breiten Verunsicherung gegenüber darauf gründenden Verfahren und Produkten. Vor allem, es ist nicht zu erkennen, dass Programmierung überflüssig wird. Iterations- oder Rekursionsverfahren in der Mathematik, Backtracking bei Suchverfahren, Backpropagation bei künstlichen neuronalen Netzen sind Algorithmen mit klar definiertem Ablauf und mit dem Ziel, durch fortwährende Wiederholung von Vorschriften zu einer Lösung zu kommen. Die im Allg. unglaublich hohe Anzahl dieser Iterationen bei Backpropagation- und anderen Verfahren künstlicher neuronaler Netze ist zwar bemerkenswert macht aber keine Abstriche daran, dass lediglich versucht wird, durch strenge Befolgung des programmierten Algorithmus eine Lösung zu finden. In der Trainingsphase künstlicher neuronaler Netze besteht die Lösung eben darin, die Gewichte von Netzverbindungen zu bestimmen oder zu „erlernen". Beim Newton- oder Heron-Verfahren zur Berechnung der Quadratwurzel einer Zahl reichen im Allg. ganz wenige – auch selbst durchzuführende - Schritte aus, um das Ergebnis für die iterative Berechnungsformel $x_{n+1}=(x_n+a/x_n)/2$ zu erhalten, wobei a das Quadrat der gesuchten Zahl ist. Der Begriff „Lernen" für die iterative Berechnung von x_{n+1} erscheint hier als eine unangemessene Bezeichnung. Anders verhält es sich offenbar mit dem Sprachgebrauch, wenn nicht nur drei, sondern Millionen von Iterationen und auch auf parallelen Zweigen ausgeführt werden, um die Gewichte der Knotenverbindungen zu berechnen. Beim Heron-Verfahren kann ein qualifizierter Anfangswert x_0, der sich mit a ändert, als „Gewicht" angesehen werden. Es ist nicht offensichtlich wie dieses Gewicht vom System selbst gelernt werden könnte. Entweder ist der theoretisch ermittelte mathematische Standard x_0 =(a+1)/2 verwendbar oder durch Programmierung und vergleichende Anwendung der genannten iterativen Berechnungsformel wird ein sogar besseres „Gewicht“ als der Standard erschlossen. Eine vernünftige Verwendung der Begriffe hilft vielleicht, den nächsten KI-Winter zu vermeiden, weil überzogenen Erwartungen nicht geweckt und später nicht erfüllt werden können.

3 Initiativen zur Künstlichen Intelligenz

Als Beginn der verstärkten Beachtung des Potentials der Künstlichen Intelligenz in Deutschland – es ist das Jahr 2018 – können das Strategiepapier der Bundesregierung „Eckpunkte der Bundesregierung für eine Strategie Künstliche Intelligenz“ [BR18] und – weil in zeitlicher Nähe entstanden und wahrscheinlich dafür teilweise ausgewertet und inhaltlich verwertet – die Studie der Fraunhofer-Gesellschaft „Maschinelles Lernen – Kompetenzen, Anwendungen, Forschungsbedarf“ [FhG18], das Positionspapier des Industriezusammenschlusses „KI Bundesverband e.V.“ [BKI18] und Initiativen der Gesellschaft für Informatik e.V. betrachtet werden.

Der umfangreiche Fraunhofer-Bericht enthält eine Reihe von beachtenswerten Aussagen, die sich so nicht im Strategiepapier wiederfinden. Von Ansatz her beschäftigt sich der Bericht ausschließlich mit Lernverfahren, enthält aber immerhin die kritische Bemerkung, dass „Maschinelles Lernen und Künstliche Intelligenz insbesondere im wirtschaftlichen Kontext oftmals vereinfacht synonym verwendet“ werden. Kritisch wird ebenfalls der Ruf nach Transparenz eingeschätzt: „Der Wunsch nach Transparenz, also die Möglichkeit, das Verhalten des Systems vollständig nachvollziehen zu können, ist häufig nur schwierig zu erfüllen, da viele Modelle sehr komplex sein müssen, …“ und „… genau umgekehrt verhält es sich mit symbolischem Wissen, also Regeln und Fakten …“ (Hinweis: gemeint sind logisch-deduktive bzw. logisch-induktive Verfahren). Forderungen, die dem Bericht zu entnehmen sind, betreffen die Stärkung der Grundlagenforschung zur KI und die Sicherung der Langfristigkeit der Forschungsförderung. Der Bericht enthält eine umfangreiche Analyse der Art und des Umfangs der Veröffentlichungstätigkeit deutscher Forschungseinrichtungen zum maschinellen Lernen und stellt letztlich eine Rangfolge auf, in der sechs Einrichtungen hervorgehoben werden. Es sei aus externer Sicht zu dieser Rangfolge vermerkt, dass es sich keinesfalls um die Kompetenzen zum Gesamtgebiet der Künstlichen Intelligenz, sondern um eine Einschätzung zum Teilgebiet maschinelles Lernen innerhalb der Künstlichen Intelligenz handelt. Die Bewertung ethischer Fragen im Zusammenhang mit der breiten Verwendung von KI-Methoden basieren offenbar nicht auf eigenen Analysen der Berichterstatter, sondern es werden vor allem extern Quellen zitiert. Die Vorschläge sind relativ allgemein gehalten, wie „sie (aus dem Kontext: Öffentliche Foren) sollen dabei helfen, mögliche Vorbehalte der Endverbraucher von Beginn an zu berücksichtigen und so langfristig zur Akzeptanz ML-basierter Produkte und Dienstleistungen beizutragen“.

Der 9-Punkte-Plan in „Künstliche Intelligenz. Situation und Maßnahmenkatalog“ des „KI Bundesverbands e.V.“, einer Vereinigung, die mehr als 50 Firmen vertritt, geht unkritischer mit ethischen Fragen um, indem bedingungslos postuliert wird, dass „KI unser Leben in Zukunft positiv beeinflussen kann“ und „eine laufenden Anpassung unseres Gesellschaftssystems“ gefordert wird, wobei darunter offensichtlich die Überwindung von „Hype, Hysterie oder Gemeinplätzen“ bei mit KI-Produkten konfrontierten Menschen verstanden wird. Mögliche Adaptierungen des Wirtschaftssystems zur Vermeidung möglicher negativer Wirkungen von KI-Techniken und mögliche Anpassungen von Technik an das bewährte Gesellschaftssystem werden nicht erwähnt. Im Gegenteil, es wird gefordert, dass gewisse gesellschaftlich erforderliche Regulierungen „ohne Einschränkungen der internationalen Wettbewerbsfähigkeit geschaffen werden müssen“. Andererseits wird eine massive finanzielle Unterstützung der Gesellschaft bei der industriellen Nutzung der Künstlichen Intelligenz, einseitig verstanden als maschinelles Lernen, gefordert.

Demgegenüber erweisen sich die Ethischen Leitlinien der Gesellschaft für Informatik e.V., insbesondere durch Art. 9, in prägnanter Weise der Gesellschaft verpflichtet*: „Das GI-Mitglied tritt mit Mut für den Schutz und die Wahrung der Menschenwürde ein, selbst wenn Gesetze, Verträge oder andere Normen dies nicht explizit fordern oder dem gar entgegenstehen. ...“*

Der Aufruf, seinem Gewissen zu folgen, mag zwar in Kontrast zum realen Leben mit seinen Notwendigkeiten zu stehen und wird auch nicht ausreichend durch Art. 12, *„In Konfliktfällen versucht die GI zwischen den Beteiligten zu vermitteln.“*, unterstützt, ist aber ein Signal an jeden Einzelnen, über die Folgen seiner Arbeit und seines Handelns mit dieser Rückendeckung nachzudenken.

In einer Umfrage des Marktforschungsinstituts YouGov vom August 2018 wird nicht nur die überwiegend kritische Haltung, bzw. sogar Ablehnung der Gesellschaft gegenüber der maschinellen Intelligenz (gemeint sind vor allem Systeme mit maschinellem Lernen) deutlich, sondern sie zeigt auch, dass besonders hohe Ängste in Bezug auf die Arbeitsplätze vorhanden sind. Es ist davon auszugehen, dass, obwohl weder die umstrittene Studie von Frey und Osborne „The Future of Employment: How susceptible are Jobs to Computerisation?“ [FO13] noch die Diskussion dazu in der Öffentlichkeit besonders bekannt sind, dennoch in dieser zu ähnlichen Vermutungen gekommen wird. Die vom BMAS in Auftrag gegebene Kurzexpertise [ZEW15] des Zentrums für Europäische Wirtschaftsforschung versucht dem entgegenzusteuern.

Die „Eckpunkte der Bundesregierung für eine Strategie Künstliche Intelligenz“ greift endlich das lange Zeit vernachlässigte (oder sogar ignorierte) Thema Künstliche Intelligenz unter dem Druck der Erfolge amerikanischer Firmen wie Google, Amazon, Apple und chinesischer Unternehmen auf, reduziert es aber auch auf maschinelles Lernen und Big Data. Wie im Fraunhofer-Bericht nahegelegt und vom KI-Bundesverband gefordert, sollen offenbar (nur einige) Kompetenzzentren (zum maschinellen Lernen) gefördert werden. Die Aussage, dass (nur einige) Zusatz-KI-Lehrstühle an ausgewählten Standorten hinzukommen sollen, verträgt sich schlecht mit Aussagen, dass zu wenig Fachkräfte und zu wenig KI-Kenntnisse vorhanden sind. Die Aussage des Eckpunktepapiers, dass Deutschland *"... weltweit führenden Standort für KI werden, insbesondere durch einen umfassenden und schnellen Transfer von*

Forschungsergebnissen in Anwendungen sowie die Modernisierung der Verwaltung…" verkennt, die nahegelegten Forderungen und naheliegende Notwendigkeit nach zunächst vor allem mehr Grundlagenforschung.
Das Eckpunktepapier scheint, auch wenn zuweilen „*ethische Standards*" und „*Ordnungsrahmen*", aber ohne besonderen Nachdruck, angesprochen werden, vor allem industriegetrieben zu sein. „*KI-Observatorien*" sollen Folgeentwicklungen von KI auf Beschäftigte und Arbeitswelt begutachten. Gesellschaft und Individuen werden in diesem Zusammenhang nicht erwähnt. Es wird der Eindruck eines gewaltigen wirtschaftlichen Effektes durch den Einsatz von Künstlicher Intelligenz (hier: des maschinellen Lernens) erweckt, ohne dass relevante soziale, gesellschaftliche und persönliche Auswirkungen entstehen – und wenn, dann sind sie beherrschbar (Grundgesetz, DSGVO regeln alles).
Als eine wenig vertrauensbildende Maßnahme für die Gesellschaft erscheint die Neu-Etablierung eines Digitalrates, obwohl ein Ethikrat und eine Enquete-Kommission Internet und digitale Gesellschaft bereits existieren. Im Unterschied zu letzterem fehlen im Digitalrat Vertreter der Zivilgesellschaft und aus Deutschland kommen offensichtlich lediglich zwei Experten. In dieser Vorgehensweise wird ein hohes, rein industriell getriebenes Nachahmer-Potential ohne ausreichende Technikfolgenabschätzung gesehen.

4 Ist die Forderung nach Transparenz sinnvoll?

Von vielen Seiten wird die Forderung erhoben, dass „*Entscheidungen die von KI-Systemen vorgeschlagen oder getroffen würden, ... für den Anwender plausibel und transparent*" sein müssen, z.B. [VDI18]. Auch das Eckpunktepapier der Bundesregierung zur Künstlichen Intelligenz enthält die Zielvorstellung oder das Versprechen der „*Sicherstellung von Transparenz, Nachvollziehbarkeit und Überprüfbarkeit der KI-Systeme*" (z.B. Systeme, die versteckte Muster maschinell erkennen und „erlernen"). Wie verfahren wird, wenn dieses Ziel nicht eingehalten wird oder eingehalten werden kann, wird nicht ausgeführt. Die Ernsthaftigkeit dieser Zusicherungen von Transparenz, Nachvollziehbarkeit und Überprüfbarkeit muss in Zweifel gezogen werden. Schon heute können sich Unternehmen bei weniger bedeutenden Aspekten ihres Handelns einer Überprüfung durch Verweis auf ein „Betriebsgeheimnis" entziehen.

„Nachvollziehbarkeit" und „Transparenz" haben mindestens zwei Aspekte: erstens, welche Muster-Daten werden bei der Entwicklung eines Verfahrens verwendet (oder nicht verwendet) und zweitens, nach welcher Methode arbeitet ein Verfahren und behandelt es dabei die Daten formal unterschiedslos? Die praktische Schwierigkeit, Hundertausende von Daten und Millionen von Schleifendurchläufen bei künstlich neuronalen Netzen mit rein menschlichen Fähigkeiten so analysieren zu können, dass ein Nachweis eines „durchschaubaren" oder „transparenten" Ansatzes und Ablaufes möglich ist, wird verkannt oder nicht offengelegt. Die Expertensysteme der achtziger Jahre des letzten Jahrhunderts waren im Allg. rein regelbasiert. Durch eine einprogrammierte Erklärungskomponente konnte der Ablauf der Regelverarbeitung schrittweise und eindeutig dargestellt werden. Schon nach wenigen Regelanwendungen geht aber dem menschlichen Prüfer trotz der klaren Struktur die Übersicht verloren. Das Vertrauen in die Richtigkeit der vom System gefundenen Lösung kann auf diese Art und Weise nicht hergestellt werden. Fraglich ist, ob die genannten Begriffe überhaupt geeignet sind, um vertrauenswürdige Software oder KI zu beschreiben. Existieren überhaupt verbindliche Definitionen für die genannten Begriffe „Überprüfbarkeit", „Nachvollziehbarkeit", Transparenz", die eine Bewertung möglich machen? Wird mit der Verwendung dieser Begriffe mehr gefordert als es in der Softwaretechnik Standard ist? Hier wird durch Verifikation mittels logischer und mathematischer Methoden versucht zu beweisen, dass die Software genau das leistet, was eine formale Spezifikation beschreibt. Damit ist aber nicht validiert, dass die Software genau das leistet, was für die anvisierte Anwendung, für die die Software entwickelt wird, erforderlich ist. Durch Validierung wird getestet, ob die (verifizierte) Software erwartete Ergebnisse liefert. Validierung lässt damit Fehler erkennen, kann aber wegen der im Allgemeinen gegebenen Undurchführbarkeit vollständiger Feldtests nicht die Fehlerfreiheit der Software in Bezug auf das zu lösende Anwendungsproblem zusichern. Die genannten Begriffe haben offensichtlich wenig mit den eingeführten Begriffen Verifikation und Validierung zu tun.

Es kann vermutet werden, dass es geradezu eine Eigenschaft komplexer Systeme ist, nicht oder kaum durchschaubar zu sein und nur mit Hilfe ausgefeilter logischer und mathematischer Verfahren Lösungen zu erzielen sind, die akzeptiert oder nicht akzeptiert werden können aber nicht über das gewählte und beschreibbare Berechnungs- oder Abarbeitungsverfahren hinaus „transparent" sind. Um komplexe Probleme überhaupt lösen zu können werden häufig vereinfachende Annahmen eingeführt (nur ganzzahlige Lösungen; irgendeine, nicht unbedingt die optimale Lösung wird erwartet; Lösung, die in einer vorgegebenen akzeptablen Zeitspanne gefunden wird; Verwendung von Zufallszahlen, um Anfangswerte zuzuweisen oder um zu versuchen, zeitaufwendige Abarbeitungen zufallsgetrieben abzukürzen). Auch dadurch können andere Lösungen oder andere Lösungsfolgen als vielleicht erwartet, entstehen. Wie schwierig es ist, eine Transparenz bei komplexen Abläufen zu wahren, kann beispielhaft die Frage nach der Ursache der Verspätung eines Fernzuges gegenüber dem Fahrplan aufzeigen. Die Verspätung könnte zwar durch die Blockierung des Gleisabschnittes durch einen anderen Zug vordergründig beantwortet werden, jedoch hat auch dieser Vorrang des zweiten Zuges seine Ursache, die wiederum durch ein Ereignis verursacht wurde. Auch ein Übergang in die Metaebene, in der die Gesamtverspätungen eines Fahrnetzes optimiert werden, ist nicht hilfreich, da auch hier nur der Aussage vertraut werden kann, dass es sich um die optimale Lösung handelt. In diesem Zusammenhang geforderte Eigenschaften wie „Überprüfbarkeit", „Durchschaubarkeit", „Transparenz" deuten mehr auf die Befürchtungen hin, dass die Problemlösungs-Software unerlaubt vom ausgewiesenen Verfahren abweichen und dadurch Lösungen zum Vorteil oder Nachteil Einzelner oder Gruppen von Menschen liefern könnte. Genauso wie bei wichtigen menschlichen Entscheidungen ein „Vier-Augen-Prinzip" gilt, sind parallele Bearbeitungen wichtiger maschineller Entscheidungen durch sogar möglichst unterschiedliche KI-Softwaresysteme vorstellbar, die im Fall unterschiedlicher Ergebnisse zur Überprüfung Anlass geben. Wer soll dann entscheiden? Ein Mensch nach formalen Gesichtspunkten, um Grenzfälle zu behandeln, eine künstliche Intelligenz mit der Fähigkeit, Härtefälle zu entscheiden? Eine Festlegung, Zweifelsfälle oder sogar alle maschinellen Entscheidungen einer Künstlichen Intelligenz durch menschliche Bearbeiter zu überprüfen, erscheint sehr von Skepsis geprägt und wäre durch fundierte Untersuchungen auf wissenschaftlicher Grundlage zu untermauern. Eventuell sind „Verlässlichkeit" einer Software – in gleichen oder ähnlichen Situationen, ein gleiches oder ähnliche Resultat zu liefern – und „Akzeptanz" einer KI-Software geeignetere Bestimmungsgrößen.

5 Fehlen noch Antworten auf ethische Fragen?

Die Autoren einer Standardliteratur zur Künstlichen Intelligenz [RN04], Stuart Russel und Peter Norvig, haben schon vor längerer Zeit auf die enge Verbindung von Fortschritt auf diesem Gebiet und gesellschaftlichen Auswirkungen hingewiesen:

„Wir können erwarten, dass mittlere Erfolge in der künstlichen Intelligenz alle Menschen in Ihrem Alltagsleben beeinflussen. … Schließlich scheint es wahrscheinlich, dass großer Erfolg in der künstlichen Intelligenz – die Herstellung von Intelligenz auf der Ebene des Menschen und darüber hinaus – das Leben der Menschheit als Ganzes verändern würde. … Aus diesem Grund können wir die KI-Forschung nicht von den ethischen Konsequenzen trennen."

Schon heute ist zu erkennen, dass Firmen und Unternehmen, die von Anfang an konsequent auf wirtschaftliche Ausnutzung von Ergebnissen der Künstlichen Intelligenz gesetzt haben, Veränderungen in der Gesellschaft bis in den persönlichen Bereich bewirkt haben, ohne dass sie für die Folgekosten aufkommen werden. Allein der Strombedarf aller kleinen persönlichen Computern, von Servern und Höchstleistungsrechner wird zum gesellschaftlichen Problem werden. Eine Strategie, die darin bestehen sollte, schnellstmöglich aufzuholen und eiligst KI-basierte Produkte und Verfahren auf den Markt zu bringen, sollte gleichzeitig auch schnellstmöglich Folgen, Chancen und Risiken abschätzen und ethische Konsequenzen untersuchen und realisieren. Bildung, Qualifizierung, Umschulung, erweiterte personelle und materielle Schul- und Hochschulausstattung, verstärkte Förderung von Grundlagenforschung, vorausschauender Strukturwandel sind nur wesentliche erste Schritte. Reicht die Zeit, um ethische Vorstellungen, technische Entwicklung, wirtschaftliche Nutzung, gesellschaftliches Selbstverständnis mit den abzusehenden Möglichkeiten der Künstlichen Intelligenz zu einer Einheit zu verbinden?

Missbrauch bei der Programmierung und Anwendung der Systeme (durch Menschen) kann nicht ausgeschlossen werden. Wenn nicht gewollt wird, dass zur Leistungsoptimierung **Arbeitnehmer-Tracking**, **Sozialkreditsysteme** oder andere als unethisch empfundene Prozesse etabliert werden, muss sich dafür eingesetzt werden. So haben sich die deutsche Gesellschaft für Informatik und ihr Fachbereich „Künstliche Intelligenz", wie tausende Wissenschaftler weltweit, dem Aufruf des Instituts „Future of Life" angeschlossen und fordern die völkerrechtliche Ächtung tödlicher autonomer (künstlich intelligenter) Waffensysteme.

Literatur

[BKI18] KI Bundesvorstand e.V.: „Künstliche Intelligenz. Situation und Maßnahmenkatalog“, Berlin, 25. 06. 2018.

[BR18] Die Bundesregierung: „Eckpunkte der Bundesregierung für eine Strategie Künstliche Intelligenz“. Stand 18. 07. 2018.

[FhG18] Döbel, I.; Leis, M.; Vogelsang, M.M.; Neustroev, D.; Pretzka, H.; Reimer, A.; Rüping, St.; Voss, A.; Wegele, M.; Welz, J.: Maschinelles Lernen. Eine Analyse zu Kompetenzen, Forschung und Anwendung. Fraunhofer-Gesellschaft, München, 2018.

[FO13] Frey, C.B.; Osborne, M.A.: The future of employment: How susceptible are jobs to computerisation? Working paper, University of Oxford, 2013. [https://www.oxfordmartin.ox.ac.uk/publications/view/1314]

[RN04] S. Russel, P. Norvig: „Künstliche Intelligenz – Ein moderner Ansatz. Pearson Studium, Prentice Hall, 2004.

[VDI18] VDI-Pressemitteilung, 13. 04. 2018. [https://www.vdi.de/presse/artikel/kuenstliche-intelligenz-steckt-noch-in-den-kinderschuhen]

[YG18] YouGov, Umfrage: Künstliche Intelligenz: Deutsche sehen eher die Risiken als den Nutzen. Sept. 2018. [https://yougov.de/news/2018/09/11/kunstliche-intelligenz-deutsche-sehen-eher-die-ris]

[ZEW15] Übertragung der Studie von Frey/Osborne (2013) auf Deutschland. Zentrum für Europäische Wirtschaftsforschung (ZEW), FB-455 des BMAS, 2015. [https://www.bmas.de/SharedDocs/Downloads/DE/PDF-Publikationen/ Forschungsberichte/fb-455.pdf]