COMPUTATION TOOLS 2019

The Tenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking

ISBN: 978-1-61208-709-2

May 5 - 9, 2019

Venice, Italy

COMPUTATION TOOLS 2019 Editors

Claus-Peter Rückemann, Leibniz Universität Hannover / Westfälische Wilhelms-Universität Münster / North-German Supercomputing Alliance (HLRN), Germany

COMPUTATION TOOLS 2019

Forward

The Tenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2019), held between May 5 - 9, 2019 - Venice, Italy, continued a series of events dealing with logics, algebras, advanced computation techniques, specialized programming languages, and tools for distributed computation. Mainly, the event targeted those aspects supporting context-oriented systems, adaptive systems, service computing, patterns and content-oriented features, temporal and ubiquitous aspects, and many facets of computational benchmarking.

The conference had the following tracks:

- Advanced computation techniques
- Tools for distributed computation

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2019 technical program committee, as well as the numerous reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to COMPUTATION TOOLS 2019. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the COMPUTATION TOOLS 2019 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope COMPUTATION TOOLS 2019 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of computational logics, algebras, programming, tools, and benchmarking. We also hope that Venice provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city.

COMPUTATION TOOLS 2019

Committee

COMPUTATION TOOLS 2019 Steering Committee
Ricardo Rocha, University of Porto, Portugal
Cristian Stanciu, University Politehnica of Bucharest, Romania
Ralph Müller-Pfefferkorn, Technische Universität Dresden, Germany
Laura Carnevali, University of Florence, Italy

COMPUTATIONAL TOOLS 2019 Industry/Research Advisory Committee
Miroslav Velev, Aries Design Automation, USA
Cornel Klein, Siemens AG, Germany
Laura Nenzi, TU Wien, Austria
Cecilia Esti Nugraheni, Parahyangan Catholic University, Indonesia
Azahara Camacho, Carbures Defense, Spain
Keiko Nakata, SAP SE - Potsdam, Germany

COMPUTATION TOOLS 2019 Technical Program Committee

Lorenzo Bettini, DISIA - Università di Firenze, Italy
Ateet Bhalla, Independent Consultant, India
Narhimene Boustia, University Saad Dahlab, Blida 1, Algeria
Azahara Camacho, Carbures Defense, Spain
Laura Carnevali, University of Florence, Italy
Emanuele Covino, Università degli Studi di Bari Aldo Moro, Italy
Marc Denecker, KU Leuven, Belgium
David Doukhan, Institut national de l'audiovisuel (Ina), France
António Dourado, University of Coimbra, Portugal
Andreas Fischer, Technische Hochschule Deggendorf, Germany
Tommaso Flaminio, DiSTA - University of Insubria, Italy
Khalil Ghorbal, INRIA, Rennes, France
George A. Gravvanis, Democritus University of Thrace, Greece
Fikret Gurgen, Bogazici University - Istanbul, Turkey
Hani Hamdan, Université de Paris-Saclay, France
Cornel Klein, Siemens AG, Germany
Jianwen Li, Iowa State University, USA
Roderick Melnik, Wilfrid Laurier University, Canada
Ralph Müller-Pfefferkorn, Technische Universität Dresden, Germany
Keiko Nakata, SAP SE, Germany
Adam Naumowicz, University of Bialystok, Poland
Laura Nenzi, TU Wien, Austria
Cecilia Esti Nugraheni, Parahyangan Catholic University, Indonesia
Javier Panadero, Open University of Catalonia, Spain

Table of Contents

Code-level Optimization for Program Energy Consumption

Cuijiao Fu, Depei Qian, Tianming Huang, Zhongzhi Luan
School of Computer Science and Engineering
Beihang University
Beijing, China
e-mail: {fucuijiao, depeiq, tianminghuang, luan.zhongzhi}@buaa.edu.cn

***Abstract*—A lot of time is spent on Central Processing Unit (CPU) waiting for memory accesses to complete during the program is being executed, which would be longer because of data structure choice, lack of design for performance, and ineffective compiler optimization. Longer execution time means more energy consumption. To save energy, avoiding unnecessary memory accesses operations is desirable. In this paper, we optimize program energy consumption by detecting and modifying the dead write, which is a common inefficient memory access. Our analysis of the Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmarks shows that the reduction of the program running energy consumption is significant after the dead write in the code was modified. For example, the SPEC CPU2006 gcc benchmark had reduced energy consumption by up to 26.7% in some inputs and 13.5% on average. We think this energy optimization approach has tremendous benefits for the developer to develop more energy-efficient software.**

Keywords-Energy Optimization; Ineffective Memory Access; Energy-efficient Software

I. INTRODUCTION

As power and energy consumption are becoming one of the key challenges in the system and software design, several researchers have focused on the energy efficiency of hardware and embedded systems [1][2], the role of application software in Information Technology (IT) energy consumption still needs investigation. On modern computer architectures, memory accesses are costly. For many programs, exposed memory latency accounts for a significant fraction of execution time. Unnecessary memory accesses, whether cache hits or misses, which lead to poor resource utilization and have a high energy cost as well [3]. In the era where processor to memory gap is widening [4][5], gratuitous accesses to memory are a cause of inefficiency, wasting so much energy, especially in large data centers or High Performance Computer (HPC) running complex scientific calculations. Therefore, the optimization of program memory access can bring about significant effects on energy consumption reduction.

Prior work about on the optimization of energy consumption in computer systems mostly focused on the scheduling of system resources, such as the research and attempt of load balancing in clusters [6]. Due to the complexity of the computer system when the program is running and the uneven level of the developer, it is difficult to modify the program code for energy optimization. Our analysis found that there are a lot of redundant memory accesses in common programs, and the energy waste they cause cannot be eliminated by resource allocation and scheduling. It is very necessary to analyze and optimize the source code of the program.

Fortunately, we found it conveniently to analyze and record the memory accesses during program execution by using Pin [7]. Pin is a dynamic binary instrumentation tool powered by Intel, which provides a rich set of high-level Application Programming Interfaces (APIs) to instrument a program with analysis routines at different granularities including module, function, trace, basic block and instruction. With this tool, we can instrument every read and write instruction, which helps us find out the redundant memory access clips in the program source code.

In this paper, we focused on the impact of dead write on program energy consumption. A 'dead write' occurs when there are two successive writes to a memory location without an intervening read. Our work mainly focuses on the following three aspects. 1) Locating dead writes exactly to the line in the source code of programs. 2) Analyzing and modifying the source code fragments found in 1). 3) Measuring and comparing energy consumption of programs before and after modification of dead writes.

The rest of the paper is organized as follows. Section 2 presents detailed decision process of dead write and sketches the methodology for positioning dead writes in programs' source lines. Section 3 analyses two codes to explore the causes of dead writes and the energy optimization benefits of dead write elimination. Finally, conclusions are drawn in Section 4.

II. METHODOLOGY

Chabbi et al. [8] described a type of redundant memory access and named it *dead write*, which means *two writes to the same memory location without an intervening read operation make the first write to that memory location dead.* This definition gives us a way to reduce energy consumption of programs by optimizing programs' memory access codes.

In the following subsections, we first describe in detail the conditions and scenarios of the formation of dead write. Then, we introduce our methodology to find out the dead writes in programs' source codes.

A. Dead Write

For every used memory address, building a state machine based on the access instructions. The state machine state is changed to initial mark **V** (Virgin) for each used memory address, indicating that no access operation is performed, and when an access operation is performed, the state is set to **R** (Read) according to the type of operation. Or **W** (Write). According to access to the same address, the state machine implements state transitions. The following two cases will be judged to be dead write:

1) A state transition from **W** to **W** corresponds to a dead write.
2) At the end of the program, the memory address in the **W** state, meaning that the program did not read it until the end of the operation.

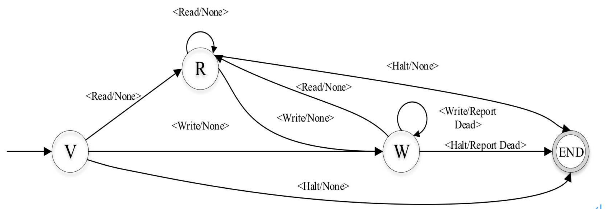


Figure 1 State transition of dead write diagram

A halt instruction transitions the automaton to the terminating state. The *Report Dead* behavior indicates that an invalid write is detected and can be reported.

Because the state machine records every memory access operation from the beginning to the end of the program, false positive or false negative situations can be avoided, and the judgment result is reliable.

B. Finding dead writes in source lines

Developing a tool based on CCTLib, a library uses Pin to track each program instruction, and builds dynamic Calling Context Tree (CCT) [9] with the information of memory access instructions. Each interior node in our CCT represents a function invocation; and each leaf node represents a write instruction. After the program is executed, each dead write will be presented to the user as a pair of CCT branches.

Specifically implemented on our tool is the use of shadow memory [10] on the Linux platform to save the state of each memory location. In order to trace dead writes, each memory access instruction to address M is updated according to the state machine of Figure 1 with the state STATE (M), while saving pointers to restore its calling context and reporting dead writes when encountered. When the node in the created call tree reaches the state needs to report dead write according to the transition state of the state machine in Figure 1, our tool will record this context and output all contexts at the end of the entire analysis. By adding the *-g* option to the gcc compiler when compiling the program to be analyzed, the debugging information is obtained so that the contexts is mapped to the source codes.

III. OPTIMIZATION FOR DEAD WRITES

In this section, we discuss the optimal solution for dead write that has been found in programs. There are many causes of dead writing. For example, Figure 2 is the simplest scenario because of the repeated initialization of an array. The Figure 2 shows the function Bar () and function Foo () initializes the array a separately before the function Foo1 () reads it. In the following, we analyze two complex situations of the gcc benchmark in SPEC CPU2006 [11].

```
#define N (0xfffff)
int a[N]
void Foo() {
  int i;
  for ( i=0; i<N; i++ )  a[i] = 0;
}
void Bar() {
   int i;
   for ( i=0; i<N; i++ )  a[i] = 0;
}
void Foo1() {
  int i;
  for ( i=0; i<N; i++ )  a[i] = a[i];
       +1;
}
int main() {
  Foo();
   Bar();
  Foo1();
return 0;
}
```

Figure 2 A simple example for dead write

For 403.gcc, after testing each input, it was found that for the input **c-typeck.i**, the dead write is very large, accounting for 73% of the total amount of memory accesses. For gcc with the input **c-typeck.i**, do the following analysis and optimization.

```
void  loop_regs_scan (struct  loop * loop, ...)
{...
  last set=(rtx *) xcalloc (-regs>num,
sizeof (rtx));
/*register used in the loop*/
for (each instr in loop) {...
    if(MATCH(ATTERN (insn))==SET || ...)
     count_one_set ...(, last_set, ...);
     ...
if(block is end)
   memset (last_set, 0, regs->num
   *sizeof(rtx));
      }...
}
```

Figure 3 Dead writes in gcc due to an inappropriate data structure

The code snippet shown in Figure 3 is refined in a frequently-called function named **loop_regs_scan ()** in the file **loop.c**. The function of this part of the code fragment is as follows:

- On line 3, 132KB of space is allocated to the array **last_se**t, with a total of 16937 elements, each element occupying 8KB.
- On Lines 6-14, iterating through each instruction in the incoming parameter loop.
- On line 8-9, if the instruction matches a pattern, the **count_one_set** function is called. The function is to update **last_set** with the last instruction that sets the virtual register.
- On lines 11-12, if the previous module completes, reset the entire **last_set** by calling the **memset ()** in the next loop.

This piece of code will produce a large number of dead writes, because the program spends a lot of time to reset the **last_set** to zero. In the module, only a very small number of elements of the array would be used in one cycle. However, at the beginning of the allocation, the largest array size possible for **last_set** is used. It means there are a large number of elements that were repeatedly reseted and cleared when they have not been accessed. It was found through sampling that in the 99.6% case, only 22 different elements per cycle would be written with a new value. Thus, a simple optimization scheme is: we maintain an array of 22 elements to record the index of the modified element of the **last_set.** Reseting only the elements of the subscript stored in the array when the reset is cleared. Reseting the entire 132KB array if the encounter array is overflow, then call **memset ()** at the end of the period to reset the entire array.

Another dead write context was found in **cselib_init ()**. As shown in Figure 4, the macro **VARRY_ELT_LIST_ INIT ()** allocates an array and initializes to 0. Then, the function **clear_table ()** initializes the array to 0 again, apparently resulting in a dead write. By reading the source code, there is a more lightweight implementation for **clear_table ()**. This implementation does not initialize the array **reg_values**, so this dead write could be eliminated by changing the interface.

```
void cselib_init () {
  ...
  cselib nregs = max reg num();
  /*initializ reg_values to 0 */
  VARRY_ELT_LIST_INIT (reg_values,
   cselib_nregs, ...);
  ...
   clear_table (1);
}
void clear_table (int clear_all) {
  /*reset all elements of reg_values to 0 */
  for (int i = 0; i < cselib_nregs; i++)
    REG_VALUES (i) = 0;
 ...
}
```

Figure 4 Dead writes in gcc due to excessive reset

IV. Experiment

In this section, we actually take the readings of the hardware performance counters by sampling them while the program is running. Those readings are the input of the Power Model [12] we had published in 2016. The output of the model is the power of the whole system. Obviously, time-based integration of power is energy consumption.

A. *Experiment environment*

We used PAPI [13] to get the readings of the hardware performance counters and gcc to compile the programs with option -g before they are analyzed by dead write analysis tool. Detailed hardware configuration of the experiment platform is shown in Table I.

TABLE I. Hardware Configuration

Component	Description($)
CPU	2.93GHz Intel Core i3
Memory	4GB DDR3 1333HZ
Hard Disk	Seagate Barracuda 7200.12
Net	1000Mb/s Ethernet

B. *Calculation method*

In our prior work [12], we have presented a full system energy consumption model based on performance events, and its accuracy had been verified. We use it in our work this time.

In the model, we calculated full system power as the linear regression of three kinds of readings of the hardware performance counters according to performance Events. As shown in Formula 1. The three kinds of performance Events are **Active Cycles** ({Cycles *in which processor are active*.), **Instruction Retired** (*The instruction* (*micro-operation*) *leaves the "Retirement Unit".)* and **Last-Level Cache (LLC) Misses** (Count *each cache miss condition for references to the last level cache.*).

$$P_{system} = 23.834 + ActiveCycles + 2.093 \times InstructionRetired + 72.113 \times LLC_{Misses} + 47.675 \quad (1)$$

When the host computer does not run the test program, it also has background programs running, and the components are also consuming power. Therefore, the energy consumption, when the host computer is not running the test program, should be removed to see more obvious contrast. Firstly, reading the host hardware performance counters' value when the test program is not running. Then, using Formula 1 to calculate the long-term power average value $\overline{P}_2$ which is taken as the background power of the host. The energy consumption of this part can be calculated as $\overline{P}_2$ multiple the running time (which is T_{end} - T_{start}). The final energy consumption will be energy caused by P_1 subtract that from $\overline{P}_2$. Therefore, the energy consumption of the test program can be calculated using Formula 2 since energy is the integral of power over time.

$$E_{result} = \int_{T_{start}}^{T_{end}} \overline{P_1 - P_2} \times (T_{end} - T_{start}) \quad (2)$$

When the test program is running, the three hardware performance counters in Formula 1 are sampled every 5 seconds. The calculated system power is connected to each sampling point using a Bezier curve. Then, the energy consumption is calculated by integrating the time with Formula 2.

C. *Result*

Since the same benchmark is running on different inputs, the functions in it invoked are different, so optimization tests are performed for different inputs. For some benchmarks, such as **bzip2**, because the program execution time is too short to sample an accurate reading, which are not suitable for energy consumption measurement. According to 403.gcc, it has a long execution time so that we can observe the changes in energy consumption before and after dead write optimization under different inputs. The results are shown in Table II. All the energy consumptions were calculated by using the methods described in previous parts.

TABLE II. CHANGES IN ENERGY CONSUMPTION FOR GCC

Input	Energy consumption (J)		%Reduction
	before	after	
166.i	141.65	128.48	9.3
200.i	207.34	203.2	2
c-typeck.i	182.37	137.69	24.5
cp-decl.i	133.36	115.76	13.2
expr.i	153.13	127.4	16.8
expr2.i	197.48	169.64	14.1
scilab.i	98.46	97.8	0.8
g23.i	254.07	219.26	13.7
s04.i	227.0	166.39	26.7
% Average		13.46	

The average energy consumption is reduced by 13.46%, which has a significant effect. The result shows that finding and the dead writes in the program code can significantly reduce the energy consumption of the programs.

V. CONCLUSIONS

This paper proposes an optimization method for program energy consumption. The method is based on the optimization of dead write, a widely-existing redundant memory access in the source code. Finding out and eliminating the dead writes in programs, which could increase system efficiency and reduce energy consumption. From the experimental results, the effect is significant. Subsequent work should be focused on developing the tools based this paper, which allow more developers to use simple operations to optimize energy consumption of written program code.

ACKNOWLEDGMENT

This research is supported by the National Key R&D Program (Grant No.2017YFB0202202).

REFERENCES

[1] E. Capra, C. Francalanci, and S.A. Slaughter, “Is software green? Application development environments and energy efficiency in open source applications”, Information & Software Technology, vol. 54, no. 1, pp. 60–71, 2012.

[2] I. Manotas, L. Pollock, and J.Clause, “Seeds: a software engineer’s energy-optimization decision support framework”, Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 503–514.

[3] P. Hicks, M. Walnock, and R. M.Owens, “Analysis of power consumption in memory hierarchies”, International Symposium on Low Power Electronics and Design, 1997, pp. 239–242.

[4] B. Jacob, “The memory system: you can’t avoid it, you can’t ignore it, you can’t fake it”, Synthesis Lectures on Computer Architecture, vol.4, no. 1, 2009, pp.1-15.

[5] S. A. Mckee, “Reflections on the memory wall”, in Conference on Computing Frontiers, 2004, p. 162.

[6] R. Azimi, M.Badiei, X. Zhan, N. Li, and S. Reda, “Fast decentralized power capping for server clusters”, in IEEE International Symposium on High Performance Computer Architecture, 2017, pp. 181–192.

[7] C.K.Luk et.al, “Pin: building customized program analysis tools with dynamic instrumentation”, 2005, pp.190–200.

[8] M.Chabbi, and J. Mellor-Crummey, “Deadspy: a tool to pinpoint program inefficiencies”, Proceedings of the Tenth International Symposium on Code Generation and Optimization(CGO’12), pp. 124-134.

[9] M. Chabbi, X. Liu, and J. Mellor-Crummey, “Call paths for pin tools”, IEEE/ACM International Symposium on Code Generation and Optimization, 2014, pp. 76–86.

[10] N. Nethercote and J. Seward, “How to shadow every byte of memory used by a program”, International Conference on Virtual Execution Environments, 2007, pp. 65–74.

[11] J. L. Henning, “Spec cpu2006 benchmark descriptions”, ACM SIGARCH Computer Architecture News, vol. 34, no. 4, pp. 1–17, 2006.

[12] S. Yang, Z. Luan, B. Li, G. Zhang, T. Huang, and D. Qian, “Performance events based full system estimation on application power consumption”, IEEE International Conference on High Performance Computing and Communications, 2017, pp.749–756.

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “Papi: A portable interface to hardware performance counters”, DoD Hpcmp Users Group Conference, 1999, pp.7–10.

Image Reconstruction using Partial Fuzzy Transform

Martina Daňková*, Viktor Pavliska†
National Supercomputing Centre IT4Innovations
Institute for Research and Applications of Fuzzy Modeling,
University of Ostrava
30. dubna 22, 701 03 Ostrava 1,
Czech Republic
Email: *martina.dankova@osu.cz, †viktor.pavliska@osu.cz

***Abstract*—We will introduce various extensions of ordinary binary operations to a specially chosen single dummy element representing an undefined or missing value. We will apply these extensions to the Perfilieva's fuzzy transform technique and extend it appropriately to operate with this dummy element too. The result of this extension will be called partial fuzzy transform. Moreover, we will demonstrate the usefulness of our approach using partial fuzzy transform technique on an image reconstruction problem.**

***Keywords*–Undefined; Missing data; Error propagation; Fuzzy transform; Image processing.**

I. Introduction

Undefined or Missing (U/M) data as a source of various bugs are a common problem in most scientific research domains. Causes are various, e.g., computation exceptions, non-terminating computation, mishandling of samples, measurement error, non-response or division by zero. Such data are needed to be represented in order to be correctly handled. For this purpose, there exist various markers in programming languages, e.g., `Null`, `NaN`, `Inf`. In this paper, we propose to use a single dummy element denoted by $*$ to represent U/M data, and we extend binary functions to operate with $*$ also [1]. As an output, we obtain several families of binary functions due to variability of treating U/M data. Remark that these extensions are analogous to the connectives of a partial fuzzy logic [2].

Our main aim is to demonstrate the usefulness of the above-mentioned extensions. For this purpose, we chose the ordinary Fuzzy Transform (FT) technique introduced in [3] and its application to an image reconstruction problem introduced in [4]. A brief description of FT, algorithms and their implementations for Image Reconstruction (IR) using FT can be found in [5].

We have observed that algorithms for IR using FT in all available sources lack an explanation of what happens in case of exceptions in a computation process. We refer mainly to the problem of the division by 0 when computing components of the direct FT which often occurs in damaged parts of an image. Moreover, the description of Multi-Step Reconstruction Algorithm (MSRA) from [4] does not correspond with experimental results presented in [4]. It is because U/M parts of an input image are considered as having value 0 which influences the final infilling of U/M areas of images in each iteration step. The main problem lies in Step 5 of MSRA which states the following: "Update the mask by deleting pixels whose reconstructed values are strictly greater than zero". Fulfilling this requirement leads to gradual falling (with a growing number of iterations) of infilled values to 0 and consequently to the darkening of the centers of infilled areas (see Figure 5). However, this is not the case of the experimental results presented in [4]. Surprisingly, their results are almost identical with ours (see Figure 6). A discussion on differences between algorithms from [4] and our approach will be explained in Section VI.

In this paper, we will present algorithms for an image reconstruction based on FT which will be able to handle U/M parts of images represented by $*$. For this purpose, we will choose suitable extensions of elementary operations to $*$ according to our intuitive expectations on the behavior of FT on U/M parts of an image which will result in a definition of the so-called partial FT. In our approach, we encode U/M areas of an image by $*$ and reconstructed parts are computed only from the real image values.

The paper is organized as follows. In Section II, we recall three basic extensions of binary functions from [1]. Perfilieva's FT and partial FT are presented in Section III and IV, respectively. Next, in Section V, we introduce algorithms for image reconstruction using partial FT together with illustrative examples. Finally, features of the used formalism are summarized in Section VI.

II. Extensions of operations to U/M data

Extensions of connectives to undefined truth values in partial fuzzy logic [2] can be analogously carried for an arbitrary binary function $o\colon X^2 \to X$ as follows:

$$\begin{array}{c|cc} o^{\mathrm{B}} & y & * \\ \hline x & o(x,y) & * \\ * & * & * \end{array} \qquad \begin{array}{c|cc} o^{\mathrm{S}} & y & * \\ \hline x & o(x,y) & x \\ * & y & * \end{array} \tag{1}$$

where $* \notin X$. We call o^{B} the *Bochvar-extension* of o, o^{S} the *Sobociński-extension* of o

Moreover, if $a \in X$ is an absorbing element of o, i.e., $o(a,x) = o(x,a) = a$ then, we introduce the *Kleene-extension* of o:

$$\begin{array}{c|ccc} o^{\mathrm{K}} & a & y & * \\ \hline a & a & a & a \\ x & a & o(x,y) & * \\ * & a & * & * \end{array} \tag{2}$$

Note that

- $o^{\mathrm{B}}, o^{\mathrm{S}}, o^{\mathrm{K}}$ operate on $(X \cup \{*\})^2$ and take values from $X \cup \{*\}$.
- These extensions are motivated by classical four valued logics.

- Sobociński-style operations treat U/M inputs encoded by $*$ as irrelevant and ignore them. In the case of Bochvar-style operations, $*$ represents a fatal error, and it terminates a computation. Kleene-style operations treat $*$ as a vincible error where $*$ is treated as a fatal error up to the case of an absorbing element which overwrites $*$.
- An extension should be chosen due to a required behavior of o in the case of U/M data.

III. Ordinary Fuzzy Transform

Recall that FT is a well-established technique in the image processing domain. Let us recall some applications, e.g., [6] introduces an algorithm for pattern matching and provides a comparison with existing methods, an image fusion was elaborated in [7], and an image contrast enhancement can be found in [8]. A higher order ordinary FT [9] was used in an edge detection problem [10], as well as in an improvement of image reconstruction method [11].

FT uses weighted arithmetic mean to compute transformation components and invert them as linear combination of components with their weights. We recall a discrete FT from [3] adjusted for 2-D case, where $X = [a, b] \times [c, d] \neq \emptyset \subset \mathbb{R}^2$ and a continuous function $f\colon X \to \mathbb{R}$ is given only at some non-empty finite set of points $D \neq \emptyset \subseteq X$.

Let $m, n \geq 2$, $\{x_i\}_{i=0}^{m+1}$ and $\{y_j\}_{j=0}^{n+1}$ be nodes such that $a = x_0 = x_1 < \ldots < x_m = x_{m+1} = b$ and $c = y_0 = y_1 < \ldots < y_n = y_{n+1} = d$. Consider two finite sets of fuzzy sets $\mathbf{A} = \{A_1, \ldots, A_m\}$ and $\mathbf{B} = \{B_1, \ldots, B_n\}$, where each A_k and B_l are identified with their membership functions $A_k\colon [a, b] \to [0, 1]$, $B_l\colon [c, d] \to [0, 1]$ for each $k = 1, \ldots, m$ and $l = 1, \ldots, n$.

Let $\mathbf{A}$ fulfill the following conditions, for each $k = 1, \ldots, m$:

1) Locality: $A_k(x) = 0$ if $x \in [a, x_{k-1}] \cup [x_{k+1}, b]$.
2) Continuity: A_k is continuous on $[a, b]$.
3) Normality: $A_k(x) = 1$ for some $x \in (x_{k-1}, x_{k+1})$.
4) Ruspini's condition: $\sum_{i=1}^{n} A_i(x) = 1$ for all $x \in [a, b]$.

We shall say that $\mathbf{A}$ form a fuzzy partition of $[a, b]$.

Often, we deal with the so called h-uniform fuzzy partition $\mathbf{A}$ (see [4]) determined by the generating function $A : [-1, 1] \mapsto [0, 1]$, which is assumed to be even, continuous, bell-shaped and fulfill $A(0) = 1$. Fuzzy sets from a h-uniform fuzzy partition are created as a shifted copy of a generating function A.

Definition 1 *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *form fuzzy partitions of* $[a, b]$ *and* $[c, d]$*, respectively. Let* $m = |\mathbf{A}|$*,* $n = |\mathbf{B}|$*, and moreover, let* $\mathbf{A}$ *and* $\mathbf{B}$ cover D*, i.e., for each* $A \in \mathbf{A}$*,* $B \in \mathbf{B}$ *there exist* $(x, y) \in D$ *such that* $A(x) > 0$ *and* $B(y) > 0$*. Then, a (direct discrete)* fuzzy transform *of* f *w.r.t.* $\mathbf{A}$*,* $\mathbf{B}$ *and* D *is defined as a* $m \times n$ *matrix*

$$F_{m,n}[f] = \begin{bmatrix} F_{11} & F_{12} & \ldots & F_{1n} \\ F_{21} & F_{22} & \ldots & F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ F_{m1} & F_{m2} & \ldots & F_{mn} \end{bmatrix}$$

where the (k, l)*-th* component $F_{k,l}$ *is equal to*

$$F_{k,l} = \frac{\sum_{(c,d) \in D} f(c, d) A_k(c) B_l(d)}{\sum_{(c,d) \in D} A_k(c) B_l(d)} \tag{3}$$

for each $k = 1, \ldots, m$ *and* $l = 1, \ldots, n$.

Observe that the matrix of components consists of weighted arithmetic means with weights given by the values of the respective fuzzy sets from $\mathbf{A}$ and $\mathbf{B}$. A linear combination of components from $F_{m,n}[f]$ and fuzzy sets from $\mathbf{A}$ and $\mathbf{B}$ returns a continuous function on X called inverse fuzzy transform.

Definition 2 *The* inverse discrete fuzzy transform *of* f *w.r.t.* $\mathbf{A}$*,* $\mathbf{B}$ *and* D *is a function* $f_{F,m,n}\colon X \to \mathbb{R}$ *defined as*

$$f_{F,m,n}(x, y) = \sum_{k=1}^{m} \sum_{l=1}^{n} F_{k,l} \cdot A_k(x) \cdot B_l(y), \quad (x, y) \in X. \tag{4}$$

We refer to the direct and inverse discrete FT from the above definitions as the *ordinary* FT.

IV. Partial Fuzzy Transform

Since $f\colon X \to \mathbb{R}$ is given only at some non-empty finite set of points, hence, it is total on D and partial on X. Therefore, it can be considered undefined on $X \setminus D$. This fact can be formalized by extension of real-line by a dummy element $*$ (represents U/M value) to $\mathbb{R}_* = \mathbb{R} \cup \{*\}$ and f to $f^*\colon X \to \mathbb{R}_*$ as follows:

$$f^*(x) = \begin{cases} f(x) & \text{for } x \in D; \\ * & \text{otherwise.} \end{cases} \tag{5}$$

Now, we can introduce FT components for f^* as extended weighted arithmetic means.

Definition 3 *Let* $\mathbf{A}$*,* $\mathbf{B}$ *and* m, n *be as in Definition 1 and*

$$P_{kl} = \{(x, y) \in X | A_k(x) \cdot B_l(y) > 0\}.$$

Then, a (direct discrete) partial fuzzy transform *of* f^* *w.r.t.* $\mathbf{A}$*,* $\mathbf{B}$ *and* D *is defined as a matrix of* components

$$F^*_{m,n}[f^*] = \begin{bmatrix} F^*_{11} & F^*_{12} & \ldots & F^*_{1n} \\ F^*_{21} & F^*_{22} & \ldots & F^*_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ F^*_{m1} & F^*_{m2} & \ldots & F^*_{mn} \end{bmatrix}$$

where

$$F^*_{k,l} = \frac{\sum^{\mathrm{S}}_{(x,y) \in P_{kl}} f^*(x, y) \cdot^{\mathrm{B}} (A_k(x) \cdot B_l(y))}{\sum_{(c,d) \in D \cap P_{kl}} A_k(c) \cdot B_l(d)}^{\mathrm{B}} \tag{6}$$

for $k = 1, \ldots, n$.

In this definition, we do not consider $\mathbf{A}$ and $\mathbf{B}$ to cover D. It follows that $F^*_{k,l} \neq *$ if and only if there exists $(x, y) \in D$ such that $x \in [x_{k-1}, x_{k+1}]$ and $y \in [y_{l-1}, y_{l+1}]$.

Observe that (6) can be equivalently written as

$$F^*_{k,l} = \frac{\sum^{\mathrm{S}}_{(x,y)\in P_{kl}} f^*(x,y) \cdot^{\mathrm{B}} (A_k(x)\cdot B_l(y))}{\sum_{(x,y)\in P_{kl}} A_k(x)\cdot B_l(y)\cdot \chi_D(x,y)}^{\mathrm{B}} \quad (7)$$

where χ_D denotes characteristic function of D.

An inverse transformation of $F^*_{m,n}[f^*]$ to the space of *partial* functions is defined as follows:

Definition 4 *The* inverse discrete partial fuzzy transform *of f^* w.r.t.* $\mathbf{A}$, $\mathbf{B}$ *and D is a function $f^*_{F,m,n}\colon X \to \mathbb{R}_*$ such that*

$$f^*_{F,m,n}(x,y) = \sum_{k=1}^{m}{}^{\mathrm{B}} \sum_{l=1}^{n}{}^{\mathrm{B}} (F^*_{k,l} \cdot^{\mathrm{K}} (A_k(x)\cdot B_l(y))) \quad (8)$$

for all $(x,y)\in X$.

Convention 5 *We denote a partial fuzzy transform briefly by FT*.*

This definition of FT* that operates on $*$-extended reals allows us to fill in "small" gaps on X between the given data D by means of real values given by the inverse FT* while "big" gaps are filled only on the edges and remain undefined otherwise. A meaning of "small" and "big" gaps is captured by the width of supports of fuzzy sets in $\mathbf{A}$ and $\mathbf{B}$.

V. Image Reconstruction using Partial Fuzzy Transform

In this contribution IR problem does not mean a fusion of several images, but a problem of reconstruction of undefined, missing or corrupted part of a single input image. Methods for solving this problem are often called image inpainting (or image interpolation) methods. There are several approaches to the problem of IR [12]. The F-transform based method falls to the class of the interpolation techniques. For an overview and a comparison of interpolation techniques, we refer to [13].

In this section, we will provide IR algorithms based on FT* which fill in U/M regions of the input image I. We will assume that information about these regions (denoted by ω) is available in the form of the input mask S having value $*$ for U/M data and 1 otherwise. A goal of IR algorithms is to replace ω by values gained through computation on undamaged regions of an input image. Here, we use FT* to do so.

Remark that inverse FT and FT* transform a matrix of components to the space of 2-D continuous functions having values in $\mathbb{R}$. Therefore, we round the received real values of the inverse FT* to the closest natural number from the interval $[0,255]$ and $*$ remains untouched.

Recall that gray-scale digital images are represented as functions on $X \subset \mathbb{N}^2$ with values in $[0,255] \subset \mathbb{N}$, i.e., as mappings $I : X \mapsto [0,255]$, where $X = [1,M]\times[1,N]$ and $[1,M],[1,N]$ are closed intervals on $\mathbb{N}$.

In the following, let us consider a non-empty set $\omega \subset X$ and parameters $s,h\in\mathbb{N}$.

A. A Simple IR Algorithm Based on Partial FT

Let us briefly sketch steps of the simple IR algorithm based on FT* with the inputs I, ω, $h>1$ and a generating function A:

- Rewrite values of I on ω by $*$ and denote the result by I^*.
- Compute m,n from h and generate h-uniform partitions $\mathbf{A}$ and $\mathbf{B}$ using A.
- Compute the direct and inverse FT* of I^* w.r.t. $\mathbf{A}$, $\mathbf{B}$ and D.
- Reconstruct a part of ω in I^* using the inverse FT*.

Inputs: Image I, damaged areas ω, width $h>1$, generator A.

1) Set $I^*(x,y) = \begin{cases} *, & \forall (x,y)\in\omega; \\ I(x,y), & \text{otherwise.}\end{cases}$
2) Set $D = X\setminus\omega$.
3) Compute m,n from h and generate h-uniform partitions $\mathbf{A}$ and $\mathbf{B}$ using A.
4) Compute the direct FT* of I^* w.r.t. $\mathbf{A}$, $\mathbf{B}$ and D, i.e., $F^*_{m,n}[I^*]$ by (6).
5) Compute the inverse FT* of I^* w.r.t. $\mathbf{A}$, $\mathbf{B}$ and D, i.e., $I^*_{F,m,n}$ by (8).
6) Set $I^*(x,y) = I^*_{F,m,n}(x,y)$, for all $(x,y)\in\omega$.
7) Rewrite all $*$ by 0 in I^*.

Output: Image I^*.

B. An Iterative IR Algorithm Based on Partial FT

In this algorithm, we repeat the simple IR algorithm described above with an increasing width h (determined by the step s) until the whole ω is reconstructed. A detailed description follows.

Inputs: Image I, damaged areas ω, width $h>1$, generator A, step $s>0$.

1) Set $I^*(x,y) = \begin{cases} *, & \forall (x,y)\in\omega; \\ I(x,y), & \text{otherwise.}\end{cases}$
2) Set $D = X\setminus\omega$.
3) Compute m,n from h and generate h-uniform partitions $\mathbf{A}$ and $\mathbf{B}$ using A.
4) Compute the direct FT* of I^* w.r.t. $\mathbf{A}$, $\mathbf{B}$ and D, i.e., $F^*_{m,n}[I^*]$ by (6).
5) Compute the inverse FT* of I^* w.r.t. $\mathbf{A}$, $\mathbf{B}$ and D, i.e., $I^*_{F,m,n}$ by (8).
6) Set $I^*(x,y) = I^*_{F,m,n}(x,y)$, for all $(x,y)\in\omega$.
7) Set $h=h+s$ and $\omega = \{(x,y)\in I^* \mid I^*(x,y) = *\}$.
8) If $\omega\neq\emptyset$ then go to 2) else output I^*.

Output: Image I^*.

Remark 6 *In the case of RGB (where $I\colon X\mapsto[0,255]^3$) or another image representation model, we run the selected algorithm in each channel separately with the same settings of the input parameters.*

Example 7 *Consider $A(x) = 1-|x|$. We apply the simple and iterative IR algorithms based on FT* to an image with small, as well as large U/M areas visualized in Figure 1. Figures 2 and 3 show an effect of the simple algorithm with $h=2$ and $h=3$, respectively. A suitable choice of h (the parameter determining fuzzy partitions) leads to an infilling of U/M areas*

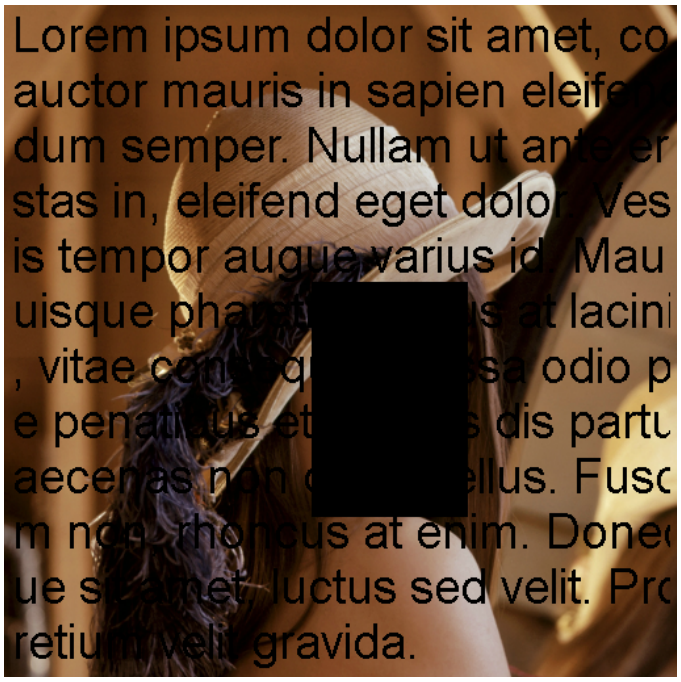

Lorem ipsum dolor sit amet, co
auctor mauris in sapien eleif
dum semper. Nullam ut ant
stas in, eleifend eget dolor. Ves
is tempor augue varius id. Mau
uisque pha s at lacini
, vitae con sa odio p
e penatibus et s dis partu
aecenas non ellus. Fusc
m non, rhoncus at enim. Donec
ue sit amet luctus sed velit. Pr
retium velit gravida.

Figure 1. An original image with U/M areas.

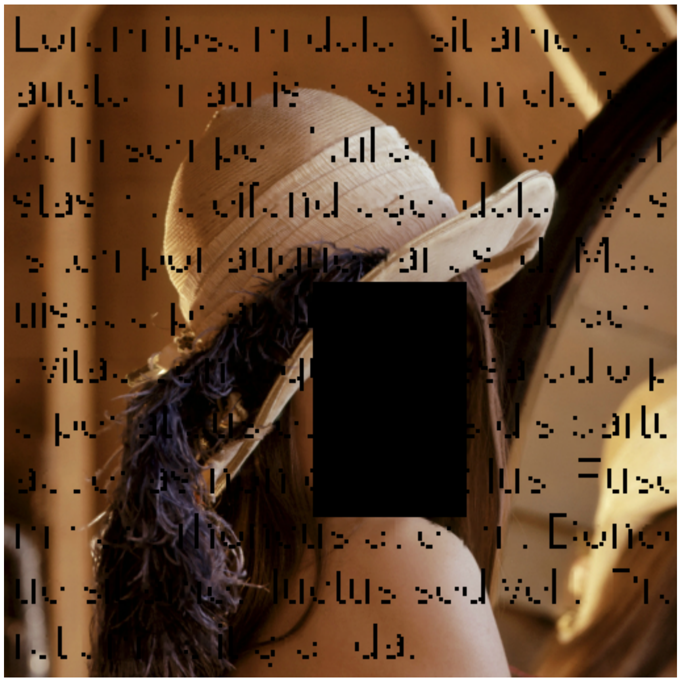


Figure 2. Application of the simple IR algorithm based on FT* to the image from Figure 1 with $h = 2$.

of a maximal vertical and horizontal width smaller than h. Finally, an output of the iterative IR algorithm based on FT with $h = 2$ and $s = 1$ is depicted in Figure 4. In this particular example, both algorithms produce fine reconstructions of small U/M areas. In the case of the big U/M area in Figure 1, which covers Lena's face, some context based IR algorithm would be more applicable.*

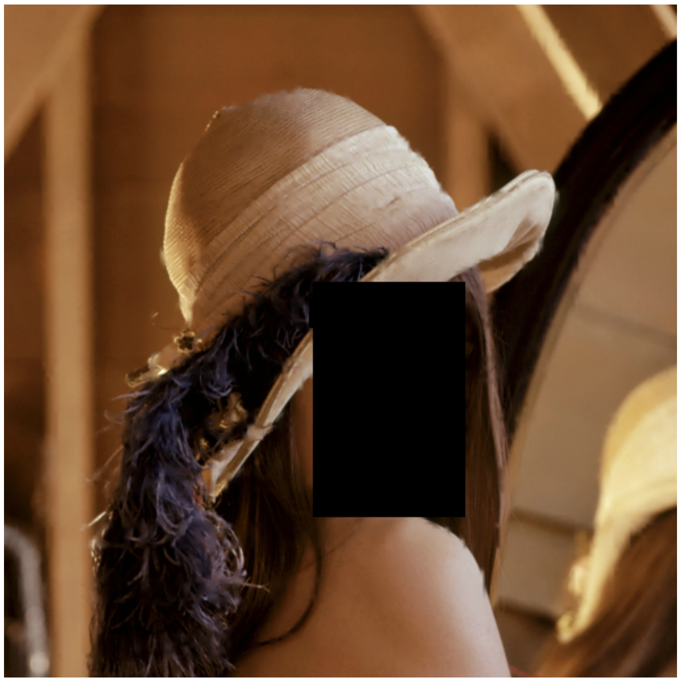


Figure 3. Application of the simple IR algorithm based on FT* to the image from Figure 1 with $h = 3$.

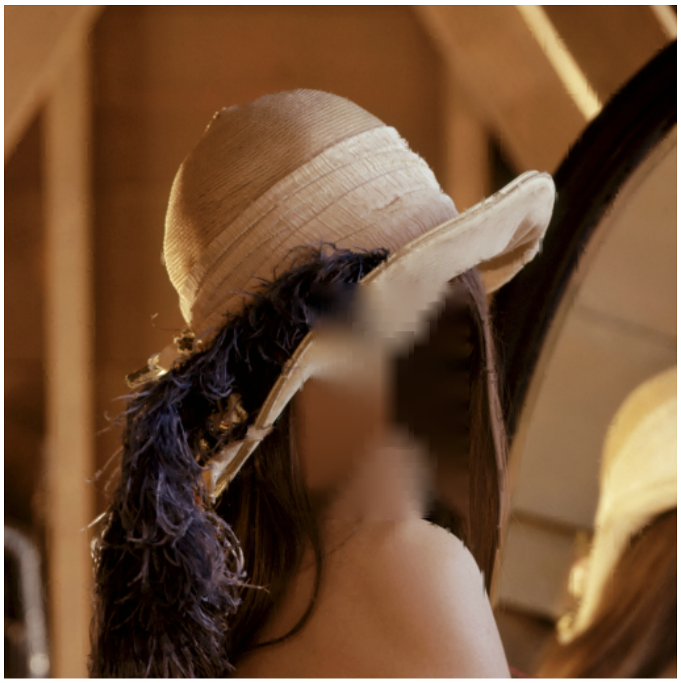


Figure 4. Application of the iterative IR algorithm based on FT* to the image from Figure 4.

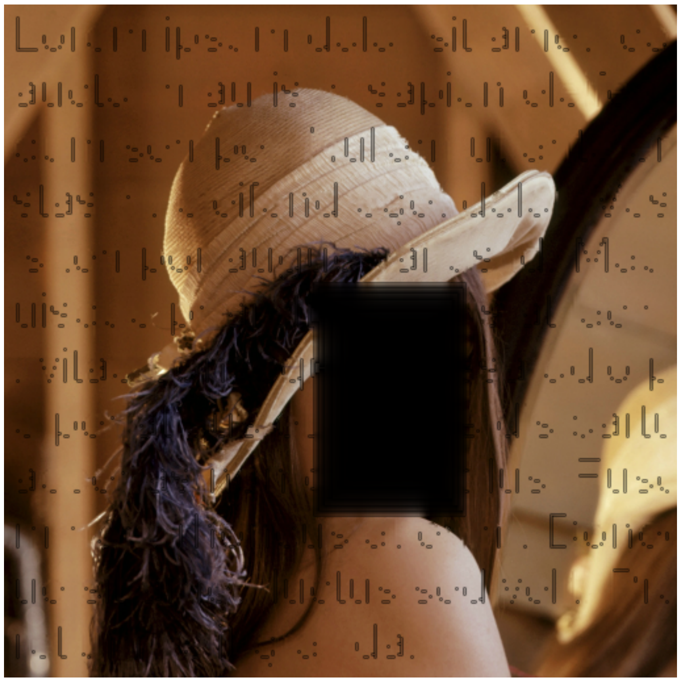
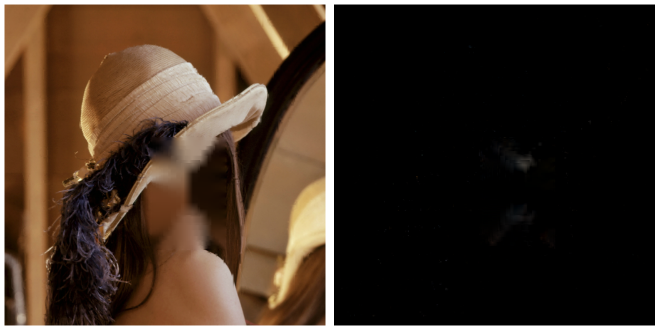


Figure 5. MSRA (due to the description in [4]) applied to Figure 1.



(a) IMSRA applied to Figure 4. (b) Difference between Figure 4 and IMSRA.

Figure 6. An example of an output of the Implementation of MSRA (IMSRA) provided in [5].

VI. Conclusions

The most important features of FT* is a significant simplification and correction of MSRA from [4] (and also other later sources). Indeed, experimental results of MSRA are almost identical with outputs of our iterative IR algorithm based on FT* (see Figure 6). Observe that FT* handles automatically U/M parts and they do not need to be explicitly encoded in (6) and (8). Moreover, this particular application shows the usefulness of the introduced formalism that deals with one error code $*$ for an arbitrary exception and various extensions of operations to $*$. Hence, we can choose a suitable extension of the user operation due to our requirements on the behaviour of $*$ and consequently, we do not need to take care of any exception because it is correctly handled using extended operations. In our opinion, this approach may be useful also in other research domains and problem-solving techniques where U/M data are present.

Acknowledgment

The work is supported by projects 16–19170S of GA ČR and LQ1602 of MŠMT ČR (NPU II).

References

[1] L. Běhounek and M. Daňková, “Extending aggregation functions for undefined inputs,” in International Symposium on Aggregation and Structures, 2018, pp. 15–16.

[2] L. Běhounek and V. Novák, “Towards fuzzy partial logic,” in Proceedings of the IEEE 45th International Symposium on Multiple-Valued Logics (ISMVL 2015), 2015, pp. 139–144.

[3] I. Perfilieva, “Fuzzy transforms: Theory and applications,” Fuzzy Sets and Systems, vol. 157, 2006, pp. 992–1023.

[4] I. Perfilieva and P. Vlašánek, “Image reconstruction by means of F-transform,” Knowledge-Based Systems, vol. 70, 2014, pp. 55–63.

[5] P. Vlašánek, “Fuzzy image processing tutorials,” https://docs.opencv.org/4.0.1/d7/d36/tutorial_fuzzy.html, December 2018.

[6] P. Hurtik and P. Števuliáková, “Pattern matching: overview, benchmark and comparison with f-transform general matching algorithm,” Soft Computing, vol. 21, no. 13, 2017, pp. 3525–3536.

[7] F. Di Martino and S. Sessa, “Complete image fusion method based on fuzzy transforms,” Soft Computing, 2017, pp. 1–11.

[8] C. Reshmalakshmi, M. Sasikumar, and G. Shiny, “Fuzzy transform for low-contrast image enhancement,” International Journal of Applied Engineering Research, vol. 13, no. 11, 2018, pp. 9103–9108.

[9] I. Perfilieva, M. Daňková, and B. Barnabas, “Towards a higher degree f-transform,” Fuzzy Sets and Systems, vol. 180, no. 1, 2011, pp. 3–19.

[10] I. Perfilieva, P. Hodáková, and P. Hurtík, “Differentiation by the f-transform and application to edge detection,” Fuzzy Sets and Systems, vol. 288, 2016, pp. 96–114.

[11] P. Vlašánek and I. Perfilieva, “Patch based inpainting inspired by the f1-transform,” International Journal of Hybrid Intelligent Systems, vol. 13, no. 1, 2016, pp. 39–48.

[12] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester, “Image inpainting,” in Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 417–424.

[13] S. Fadnavis, “Image interpolation techniques in digital image processing: An overview,” International Journal of Engineering Research and Applications, vol. 4, 2014, pp. 70–73.

A Sequent Based On-the-fly Procedure to Get Hilbert Proofs in Classical Propositional Logic

Mauro Ferrari
DiSTA, Univ. degli Studi dell'Insubria,
Via Mazzini, 5, 21100, Varese, Italy
mauro.ferrari@uninsubria.it

Camillo Fiorentini
DI, Univ. degli Studi di Milano,
Via Celoria 18, 20133 Milano, Italy
fiorentini@di.unimi.it

Guido Fiorino
DISCO, Univ. degli Studi di Milano-Bicocca,
Viale Sarca 336, 20126 Milano, Italy
guido.fiorino@unimib.it

***Abstract*—In this paper, we present a preliminary result on the generation of Hilbert proofs for classical propositional logic. This is part of our ongoing research on the comparison of proof-search in different proof-systems. Exploiting the notion of evaluation function, we define a fully deterministic terminating decision procedure that returns either a derivation in the Hilbert calculus of the given goal or a counter model witnessing its unprovability.**

***Keywords*–Automated Theorem Proving; Hilbert calculi.**

I. Introduction

It is well-known [1] that the standard formalizations of classical and intuitionistic logic based on Hilbert calculi, sequent calculi and natural deduction are equivalent. In spite of this, proof-search has been mainly developed around the notion of sequent calculi, almost neglecting the cases of natural deduction and Hilbert calculi. This is primarily motivated by the fact that the latters lack the *structural properties* of sequent calculi, like cut-elimination and subformula property, that can be immediately exploited to define a goal-oriented proof-search procedure (see, e.g., [2]-[3] for an accurate discussion). In [4][5] it is shown that in the case of natural deduction it is possible to design proof-search procedures with structural and complexity properties comparable with those based on sequent calculi. This is obtained by imposing a strict discipline on the backward application of the rules and by exploiting some internal features of the goal via evaluation functions. In this paper, we begin an investigation concerning Hilbert calculi with the aim to apply in this context the ideas developed in the above cited papers. We take Classical Propositional Logic (CPL) as the entry point of our investigation. We consider the Hilbert system **Hc** for CPL as defined in [6] and we introduce a proof-search strategy for it. The procedure builds Hilbert proofs during the proof-search phase. Since the proof-search phase is based on a strategy for a sequent calculus, the procedure can be seen as building Hilbert proofs in one-pass, that is, by translating on-the-fly sequent proofs. The procedure relies on a sequent calculus with at most one formula on the right, thus to get termination and completeness we need to introduce the machinery of *evaluations*.
As regards related works, as far as we know, the only two papers addressing this issue are [7][8], and both are scarcely related with our approach. The former deals with backward application of modus ponens, introducing metavariables to represents cut-formulas. In [8], an implementation of a prover for CPL is presented, that relies on a semantic method exploiting Kalmar completeness theorem.

The system **Hc** consists of some axioms and only one rule, *modus ponens* (from $A \to B$ and A infer B). The main problem in defining a proof-search procedure for **Hc** is to control the application of modus ponens. If we backward apply modus ponens to prove a goal formula B, we have to guess a cut-formula A, so that next goals are A and $A \to B$. To avoid considering useless cut-formulas, we adapt to the classical case the notion of evaluation function introduced in [9] for intuitionistic propositional logic. Essentially, an evaluation function is a lightweight computational mechanism that drives the backward application of the rules, only analyzing the current goal of the proof search. The evaluation function for **Hc** is introduced in Section III. The proof-search procedure described in Section IV takes as input a goal G and a set of assumptions Γ and returns either a deduction of G in **Hc** from assumptions Γ or a classical model of Γ falsifying G. As proved in Section IV the proof-search procedure is terminating and it does not require backtracking.

II. The Hilbert calculus Hc

We consider the propositional language $\mathcal{L}$ of CPL based on a denumerable set of propositional variables Pv and the connectives $\wedge$, $\vee$, $\to$ and $\neg$. We write $A \leftrightarrow B$ as a shorthand for $(A \to B) \wedge (B \to A)$. A *literal* is a formula of the form p or $\neg p$ with $p \in \mathrm{Pv}$; the set of literals is denoted by Lit. The size of a formula A, denoted by $|A|$, is the number of logical symbols occurring in A. A *model* $\mathcal{M}$ is a subset of Pv; we write $\mathcal{M} \models A$ to denote that the formula A is *valid in* $\mathcal{M}$, according to the usual definition. Let Γ be a set of formulas; $\mathcal{M} \models \Gamma$ iff, for every $A \in \Gamma$, $\mathcal{M} \models A$. By $\Gamma \models A$ we mean that A is a *classical consequence* of Γ, namely: for every model $\mathcal{M}$, $\mathcal{M} \models \Gamma$ implies $\mathcal{M} \models A$.

We call **Hc** the Hilbert calculus for CPL introduced in [6]. Logical axioms of **Hc** are:

(Ax1)	$A \to (B \to A)$		
(Ax2)	$(A \to B) \to ((A \to (B \to C)) \to (A \to C))$		
(Ax3)	$A \to (B \to (A \wedge B))$		
(Ax4a)	$(A \wedge B) \to A$	(Ax4b)	$(A \wedge B) \to B$
(Ax5a)	$A \to (A \vee B)$	(Ax5b)	$B \to (A \vee B)$
(Ax6)	$(A \to C) \to ((B \to C) \to ((A \vee B) \to C))$		
(Ax7)	$(A \to B) \to ((A \to \neg B) \to \neg A)$		
(Ax8)	$\neg\neg A \to A$		

The only rule of **Hc** is MP (*Modus Ponens*): from $A \to B$ and A infer B.

Let Γ be a set of formulas and A a formula. A *deduction* of A from assumptions Γ is a finite sequence of formulas $\mathcal{D} = \langle A_1, \ldots, A_n \rangle$ (possibly with repetitions) such that $A_n = A$ and, for every A_i in the sequence, one of the following conditions holds:

(a) $A_i \in \Gamma$ (namely, A_i is an assumption);
(b) A_i is an instance of a logical axiom;
(c) A_i is obtained by applying MP to A_j and A_k, with $j < i$ and $k < i$.

We denote with $\mathrm{Fm}(\mathcal{D}) = \{A_1, \ldots, A_n\}$ the set of formulas occurring in $\mathcal{D}$. By $\mathcal{D} : \Gamma \vdash A$ we mean that $\mathcal{D}$ is a deduction of A from assumptions in Γ. Given two sets of formulas $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$, we write $\mathcal{D} : A_1, \ldots, A_n, B_1, \ldots, B_m \vdash A$ instead of $\mathcal{D} : \{A_1, \ldots, A_n\} \cup \{B_1, \ldots, B_m\} \vdash A$.

Given two deductions $\mathcal{D}_1 : \Gamma_1 \vdash A_1$ and $\mathcal{D}_2 : \Gamma_2 \vdash A_2$, we denote by $\mathcal{D}_1 \circ \mathcal{D}_2$ the deduction obtained by concatenating the sequences $\mathcal{D}_1$ and $\mathcal{D}_2$. One can easily check that $\mathcal{D}_1 \circ \mathcal{D}_2 : \Gamma \vdash A_2$, where $\Gamma \subseteq \Gamma_1 \cup \Gamma_2$. Note that Γ can be a proper subset of $\Gamma_1 \cup \Gamma_2$, since some of assumptions in Γ_2 could be obtained by applying MP in $\mathcal{D}_1 \circ \mathcal{D}_2$ (see the next example).

Example 1: Let us consider the derivations

$$\mathcal{D}_1 = \langle p_1, p_1 \to p_2 \rangle \qquad \mathcal{D}_2 = \langle p_2, p_2 \to (p_2 \vee q), p_2 \vee q \rangle$$

We have $\mathcal{D}_1 : p_1, p_1 \to p_2 \vdash p_1 \to p_2$. In $\mathcal{D}_2$, the formula $p_2 \to (p_2 \vee q)$ is an instance of axiom (Ax5*a*) *and $p_2 \vee q$ is obtained by applying* MP *to $p_2 \to (p_2 \vee q)$ and p_2, hence $\mathcal{D}_2 : p_2 \vdash p_2 \vee q$. In the concatenation $\mathcal{D}_1 \circ \mathcal{D}_2$ the formula p_2 can be obtained by applying* MP *to $p_1 \to p_2$ and p_1, hence $\mathcal{D}_1 \circ \mathcal{D}_2 : p_1, p_1 \to p_2 \vdash p_2 \vee q$.*

In the following proposition, we introduce some deduction schemas we use later on.

Lemma 1: For all formulas A, B and C, the following deductions can be constructed:

(i) $\mathcal{D}_{\mathrm{MP}}(A \to B, A) : A \to B, A \vdash B$;

(ii) $\mathcal{D}_{\neg\neg\mathrm{E}}(A) : \neg\neg A \vdash A$;

(iii) $\mathcal{D}_{\mathrm{EF}}(A, B) : A, \neg A \vdash B$;

(iv) $\mathcal{D}_{\mathrm{EF}\to}(A, B) : \neg A \to B, \neg A \to \neg B \vdash A$;

(v) $\mathcal{D}_{\mathrm{EF}\to\neg}(A, B) : A \to B, A \to \neg B \vdash \neg A$;

(vi) $\mathcal{D}_{\vee\mathrm{E}}(A, B, C) : A \to C, B \to C, A \vee B \vdash C$.

Proof: The following sequence of formulas proves Point (vi):

(1)	$A \to C$	Assumption
(2)	$B \to C$	Assumption
(3)	$A \vee B$	Assumption
(4)	$(A \to C) \to$ $((B \to C) \to ((A \vee B) \to C))$	(Ax6)
(5)	$(B \to C) \to ((A \vee B) \to C)$	MP (4) (1)
(6)	$(A \vee B) \to C$	MP (5) (2)
(7)	C	MP (6) (3)

■

A distinguishing feature of Hilbert calculi is that there are no rules to close assumptions. Thus, to prove the Deduction Lemma for **Hc** we have to rearrange a deduction $\mathcal{D}$ of $A, \Gamma \vdash B$ to get a deduction of $\Gamma \vdash A \to B$. An analogous issue holds for negation elimination and negation introduction. The following lemma provides the schemas of derivation transformations treating such cases:

Lemma 2 (Closing assumption lemma):

(i) Let $\mathcal{D} : A, \Gamma \vdash B$. Then, there exists a deduction $\mathcal{E}_{\mathrm{DL}}(\mathcal{D}, A) : \Gamma \vdash A \to B$ such that, for every $C \in \mathrm{Fm}(\mathcal{D})$, $A \to C \in \mathrm{Fm}(\mathcal{E}_{\mathrm{DL}}(\mathcal{D}))$.

(ii) Let $\mathcal{D} : \neg A, \Gamma \vdash K$ such that $\neg K \in \mathrm{Fm}(\mathcal{D})$. Then, there exists a deduction $\mathcal{E}_{\neg\mathrm{E}}(\mathcal{D}, \neg A) : \Gamma \vdash A$.

(iii) Let $\mathcal{D} : A, \Gamma \vdash K$ such that $\neg K \in \mathrm{Fm}(\mathcal{D})$. Then, there exists a deduction $\mathcal{E}_{\neg\mathrm{I}}(\mathcal{D}, A) : \Gamma \vdash \neg A$.

Proof: For Point (i) see the proof of the Deduction Lemma in [6]. As for Point (ii), by Point (i) there exists a deduction $\mathcal{E}_{\mathrm{DL}}(\mathcal{D}, \neg A) : \Gamma \vdash \neg A \to K$ such that $\neg A \to \neg K \in \mathrm{Fm}(\mathcal{E}_{\mathrm{DL}}(\mathcal{D}, \neg A))$. Let us consider the derivation $\mathcal{D}_{\mathrm{EF}\to}(A, K) : \neg A \to K, \neg A \to \neg K \vdash A$ defined in Lemma 1.(iv). We set $\mathcal{E}_{\neg\mathrm{E}}(\mathcal{D}, \neg A) : \Gamma \vdash A$ as the deduction $\mathcal{E}_{\mathrm{DL}}(\mathcal{D}, \neg A) \circ \mathcal{D}_{\mathrm{EF}\to}(A, K)$. The deduction $\mathcal{E}_{\neg\mathrm{I}}(\mathcal{D}, A)$ of Point (iii) is built in a similarly using $\mathcal{D}_{\mathrm{EF}\to\neg}(A, K)$ instead of $\mathcal{D}_{\mathrm{EF}\to}(A, K)$. ■

III. The evaluation mechanism

Evaluation functions have been introduced in [9] to get a terminating proof-search procedure for Gentzen sequent calculus **LJ** for propositional intuitionistic logic. Here, we adapt this mechanism to the case of classical propositional logic.

Let $\mathcal{L}_{\mathtt{t},\mathtt{f}}$ be the language obtained by adding to $\mathcal{L}$ the constants $\mathtt{t}$ (true) and $\mathtt{f}$ (false), with the usual interpretation; let H be a formula of $\mathcal{L}_{\mathtt{t},\mathtt{f}}$ and let Γ be a set of formulas of $\mathcal{L}$. The *evaluation of H in* Γ, denoted by $\mathbf{eval}(H, \Gamma)$, is the formula of $\mathcal{L}_{\mathtt{t},\mathtt{f}}$ obtained by replacing every subformula K of H such that K is an axiom or $K \in \Gamma$ with $\mathtt{t}$ and then by performing some truth preserving boolean simplifications (for instance, any subformula of the kind $\mathtt{t} \vee K$ is replaced by $\mathtt{t}$, while $\mathtt{f} \vee K$ is replaced by K). The function **eval** and the auxiliary function **simpl** are defined by mutual induction in Figure 1. We point out that $\mathbf{eval}(H, \Gamma)$ is either $\mathtt{t}$ or $\mathtt{f}$ or a formula of $\mathcal{L}$ (not containing the constants $\mathtt{t}$, $\mathtt{f}$). Moreover, $|\mathbf{eval}(H, \Gamma)| \leq |H|$.

One can easily prove that, assuming Γ, the formulas H, $\mathbf{eval}(H, \Gamma)$ and $\mathbf{simpl}(H, \Gamma)$ are classically equivalent, as stated in the next lemma

Lemma 3: Let Γ be a set of formulas of $\mathcal{L}$ and H a formula of $\mathcal{L}_{\mathtt{t},\mathtt{f}}$. Then:

1) $\Gamma \vDash H \leftrightarrow \mathbf{eval}(H, \Gamma)$;

2) $\Gamma \vDash H \leftrightarrow \mathbf{simpl}(H, \Gamma)$.

By Lemma 3 we immediately get:

Theorem 1: Let Γ be a set of formulas of $\mathcal{L}$, let A be a formula of $\mathcal{L}$ and $\mathcal{M}$ a model such that $\mathcal{M} \vDash \Gamma$.

1) $\mathbf{eval}(A, \Gamma) = \mathtt{t}$ implies $\mathcal{M} \vDash A$.

$$\mathbf{eval}(H,\Gamma) = \begin{cases} \mathtt{t} & \text{if } H \text{ is an axiom or } H \in \Gamma \\ \mathtt{f} & \text{if } \neg H \in \Gamma \\ H & \text{if } H \in (\mathrm{Pv} \setminus \Gamma) \cup \{\mathtt{t},\mathtt{f}\} \\ \mathbf{simpl}(H,\Gamma) & \text{otherwise} \end{cases}$$

in the following $K_1 = \mathbf{eval}(H_1,\Gamma)$ and $K_2 = \mathbf{eval}(H_2,\Gamma)$

$$\mathbf{simpl}(\neg H_1,\Gamma) = \begin{cases} \mathtt{t} & \text{if } K_1 = \mathtt{f} \\ \mathtt{f} & \text{if } K_1 = \mathtt{t} \\ \neg K_1 & \text{otherwise} \end{cases} \qquad \mathbf{simpl}(H_1 \wedge H_2,\Gamma) = \begin{cases} \mathtt{t} & \text{if } K_1 = \mathtt{t} \text{ and } K_2 = \mathtt{t} \\ \mathtt{f} & \text{if } K_1 = \mathtt{f} \text{ or } K_2 = \mathtt{f} \\ K_1 \wedge K_2 & \text{otherwise} \end{cases}$$

$$\mathbf{simpl}(H_1 \vee H_2,\Gamma) = \begin{cases} \mathtt{t} & \text{if } K_1 = \mathtt{t} \text{ or } K_2 = \mathtt{t} \\ \mathtt{f} & \text{if } K_1 = \mathtt{f} \text{ and } K_2 = \mathtt{f} \\ K_1 \vee K_2 & \text{otherwise} \end{cases} \qquad \mathbf{simpl}(H_1 \to H_2,\Gamma) = \begin{cases} \mathtt{t} & \text{if } K_1 = \mathtt{f} \text{ or } K_2 = \mathtt{t} \\ \mathtt{f} & \text{if } K_1 = \mathtt{t} \text{ and } K_2 = \mathtt{f} \\ K_1 \to K_2 & \text{otherwise} \end{cases}$$

Figure 1. Definition of **eval**

2) $\mathbf{eval}(A,\Gamma) = \mathtt{f}$ implies $\mathcal{M} \models \neg A$.

By $\mathcal{L}^{\neg}$ we denote the set of formulas H such that H is a literal or $H = \neg(A \vee B)$. A set of formulas Γ is *reduced* iff the following properties hold:

- $\Gamma \subseteq \mathcal{L}^{\neg}$;
- for every $p \in \mathrm{Pv}$, $p \in \Gamma$ implies $\neg p \notin \Gamma$;
- for every $H = \neg(A \vee B) \in \Gamma$, $\mathbf{eval}(H, \Gamma \setminus \{H\}) = \mathtt{t}$.

By Theorem 1, we get:

Theorem 2: Let Γ be a reduced set of formulas. Then, $\Gamma \cap \mathrm{Pv}$ is a model of Γ.

Proof: Let Θ be the subset of Γ containing all the formulas of the kind $\neg(A \vee B)$ of Γ. We prove the theorem by induction on the cardinality of Θ. If Θ is empty, then Γ only contains propositional variables or formulas $\neg p$ such that $p \in \mathrm{Pv} \setminus \Gamma$; this implies that $\Gamma \cap \mathrm{Pv}$ is a model of Γ. Let $\neg(A \vee B) \in \Theta$, let $\Gamma_1 = \Gamma \setminus \{\neg(A \vee B)\}$ and let $\mathcal{M}$ be the model $\Gamma \cap \mathrm{Pv}$. By induction hypothesis, $\mathcal{M}$ is a model of Γ_1. Since Γ is reduced, we have $\mathbf{eval}(\neg(A \vee B), \Gamma_1) = \mathtt{t}$; by Theorem 1 we get $\mathcal{M} \models \neg(A \vee B)$, hence $\mathcal{M}$ is a model of Γ. ■

A set of formulas Γ is *contradictory* iff there exists a formula X such that $\{X, \neg X\} \subseteq \Gamma$. In the proof of termination of the proof-search procedure **Hp**, we need the following properties of **eval**.

Lemma 4: Let Γ be a non-contradictory set of formulas, let H be a formula and let us assume that $\mathbf{eval}(H,\Gamma) = \tau$, where $\tau \in \{\mathtt{t},\mathtt{f}\}$. Let $K \in \Gamma$ and $\Delta = \Gamma \setminus \{K\}$. Then:

1) If $K = \neg\neg A$ and $\Delta \cup \{A\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{A\}) = \tau$.
2) If $K = A \wedge B$ and $\Delta \cup \{A, B\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{A, B\}) = \tau$.
3) If $A_0 \vee A_1 \in \Gamma$ and $, \Delta \cup \{A_k\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{A_k\}) = \tau$.
4) If $K = A \to B$ and $\Delta \cup \{\neg A\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{\neg A\}) = \tau$.
5) If $K = A \to B$ and $, \Delta \cup \{B\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{B\}) = \tau$.
6) If $K = \neg(A_0 \wedge A_1)$ and $k \in \{0,1\}$ and $\Delta \cup \{\neg A_k\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{\neg A_k\}) = \tau$.
7) If $K = \neg(A_0 \vee A_1)$ and $\mathbf{eval}(K, \Delta) = \mathtt{t}$ and $k \in \{0,1\}$ and $\Delta \cup \{\neg A_k\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{\neg A_k\}) = \tau$.
8) If $K = \neg(A_0 \vee A_1)$ and $\mathbf{eval}(A_0, \Delta) = \mathbf{eval}(A_1, \Delta) = \mathtt{f}$, then $\mathbf{eval}(H,\, \Delta) = \tau$.
9) If $\neg(A \to B) \in \Gamma$ and $\Delta \cup \{A, \neg B\}$ is not contradictory, then $\mathbf{eval}(H,\, \Delta \cup \{A, \neg B\}) = \tau$.

IV. PROCEDURE Hp

We present the procedure **Hp** to search for a deduction in **Hc**. Let Γ be a set of formulas and let G be the goal formula or the special symbol $\Box$ representing the empty goal. We define the procedure **Hp** satisfying the following properties:

(H1) If $G \in \mathcal{L}$, $\mathbf{Hp}(\Gamma, G)$ returns either a deduction $\mathcal{D} : \Gamma \vdash G$ or a model of $\Gamma \cup \{\neg G\}$.
(H2) $\mathbf{Hp}(\Gamma, \Box)$ returns either a deduction $\mathcal{D} : \Gamma \vdash K$ such that $\neg K \in \mathrm{Fm}(\mathcal{D})$ or a model of Γ.

```
Procedure Hp (Γ, G)
(1) if $G$ is an axiom or $G \in \Gamma$
      return $\langle G \rangle$
(2) if there is $\{\neg A, A\} \subseteq \Gamma$
      if $G \neq \Box$, then return the proof $\mathcal{D}_{\mathrm{EF}}(A, G)$
      else return $\langle \neg A, A \rangle$
(3) if there is $\neg\neg A \in \Gamma$
      let $\mathcal{D} = \mathbf{Hp}(\,(\Gamma \setminus \{\neg\neg A\}) \cup \{A\},\, G\,)$.
      if $\mathcal{D}$ is a model, then return $\mathcal{D}$
      else return $\langle \neg\neg A, \neg\neg A \to A \rangle \circ \mathcal{D}$
(4) if there is $A \wedge B \in \Gamma$
```

```
     let $\mathcal{D} = \mathbf{Hp}(\,(\Gamma \setminus \{A \wedge B\}) \cup \{A, B\},\, G\,)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else return $\langle A \wedge B, (A \wedge B) \to A, (A \wedge B) \to B\rangle \circ \mathcal{D}$
(5)  if there is $A_0 \vee A_1 \in \Gamma$ and $G = \Box$
     for $i = 0, 1$, let $\mathcal{D}_i = \mathbf{Hp}(\,(\Gamma \setminus \{A_0 \vee A_1\}) \cup \{A_i\},\, \Box\,)$
     if there exists $i \in \{0,1\}$ such that $\mathcal{D}_i$ is a model, then return $\mathcal{D}_i$
     else for $i = 0, 1$, let $K_i$ the last formula of $\mathcal{D}_i$
         let $\mathcal{D}'_1 = \mathcal{D}_1 \circ \mathcal{D}_{\mathrm{EF}}(K_1, \neg K_0) \circ \mathcal{D}_{\mathrm{EF}}(K_1, K_0)$
         let $\mathcal{E}_0 = \mathcal{E}_{\mathrm{DL}}(\mathcal{D}_0, A_0)$
         let $\mathcal{E}_1 = \mathcal{E}_{\mathrm{DL}}(\mathcal{D}'_1, A_1)$
         return $\mathcal{E}_0 \circ \mathcal{E}_1 \circ \mathcal{D}_{\vee\mathrm{E}}(A_0, A_1, \neg K_0) \circ \mathcal{D}_{\vee\mathrm{E}}(A_0, A_1, K_0)$
(6)  if there is $A_0 \vee A_1 \in \Gamma$ and $G \neq \Box$
     for $i = 0, 1$, let $\mathcal{D}_i = \mathbf{Hp}((\Gamma \setminus \{A_0 \vee A_1\}) \cup \{A_i\}, G)$
     if there exists $i \in \{0,1\}$, such that $\mathcal{D}_i$ is a model, then return $\mathcal{D}_i$
     else for $i = 0, 1$, let $\mathcal{D}'_i = \mathcal{E}_{\mathrm{DL}}(\mathcal{D}_i, A_i)$
         return $\mathcal{D}'_0 \circ \mathcal{D}'_1 \circ \mathcal{D}_{\vee\mathrm{E}}(A_0, A_1, G)$
(7)  if ($G = A_0 \vee A_1$ or $G \in \mathrm{Pv}$) and $\mathbf{eval}(G, \Gamma) \neq \mathtt{f}$
     let $\mathcal{D} = \mathbf{Hp}(\,\Gamma \cup \{\neg G\}, \Box\,)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else return $\mathcal{E}_{\neg\mathrm{E}}(\mathcal{D}, \neg G)$
(8)  if $G = A_0 \wedge A_1$
     for $i = 0, 1$, let $\mathcal{D}_i = \mathbf{Hp}(\Gamma, A_i)$
     if there exists $i \in \{0,1\}$, such that $\mathcal{D}_i$ is a model, then return $\mathcal{D}_i$
     else let  $\mathcal{D}_2 = \langle\; A_0 \to (A_1 \to (A_0 \wedge A_1)),$
                              $A_1 \to (A_0 \wedge A_1), A_0 \wedge A_1\rangle$
         return $\mathcal{D}_0 \circ \mathcal{D}_1 \circ \mathcal{D}_2$
(9)  if $G = A \to B$
     let $\mathcal{D} = \mathbf{Hp}(\Gamma \cup \{A\}, B)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$, else return $\mathcal{E}_{\mathrm{DL}}(\mathcal{D}, A)$
(10) if $G = \neg A$
     let $\mathcal{D} = \mathbf{Hp}(\Gamma \cup \{A\}, \Box)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$, else return $\mathcal{E}_{\neg\mathrm{I}}(\mathcal{D}, A)$
(11) if $G = A_0 \vee A_1$
     // *here* $\mathbf{eval}(A_0 \vee A_1, \Gamma) = \mathtt{f}$
     if $\mathbf{eval}(A_0, \Gamma) \neq \mathtt{f}$
         let $\mathcal{D} = \mathbf{Hp}(\Gamma, A_0)$
         if $\mathcal{D}$ is a model, then return $\mathcal{D}$
         else return $\mathcal{D} \circ \langle A_0 \to (A_0 \vee A_1), A_0 \vee A_1\rangle$
     if $\mathbf{eval}(A_1, \Gamma) \neq \mathtt{f}$
         let $\mathcal{D} = \mathbf{Hp}(\Gamma, A_1)$
         if $\mathcal{D}$ is a model, then return $\mathcal{D}$
         else return $\mathcal{D} \circ \langle A_1 \to (A_0 \vee A_1), A_0 \vee A_1\rangle$
     // *here* $\mathbf{eval}(A_0, \Gamma) = \mathbf{eval}(A_1, \Gamma) = \mathtt{f}$
     let $\mathcal{D} = \mathbf{Hp}(\Gamma, \Box)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else let $K$ be the last formula of $\mathcal{D}$
         return $\mathcal{D} \circ \mathcal{D}_{\mathrm{EF}}(K, G)$
(12) if $G \in \mathrm{Pv}$ and $\neg G \in \Gamma$
     let $\mathcal{D} = \mathbf{Hp}(\Gamma, \Box)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else let $K$ be the last formula of $\mathcal{D}$
         return $\mathcal{D} \circ \mathcal{D}_{\mathrm{EF}}(K, G)$
     // *here* $G = \Box$
(13) if there is $\neg A \in \Gamma$ such that $A = B_0 \vee B_1$
     and $\mathbf{eval}(\neg A, \Gamma \setminus \{\neg A\}) \neq \mathtt{t}$
     let $\mathcal{D} = \mathbf{Hp}(\Gamma, A)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else return $\langle \neg A\rangle \circ \mathcal{D}$
(14) if there is $\neg A \in \Gamma$ such that $A \notin \mathrm{Pv}$ and $A \neq B_0 \vee B_1$
     let $\mathcal{D} = \mathbf{Hp}(\,\Gamma \setminus \{\neg A\},\, A\,)$
     if $\mathcal{D}$ is a model, then return $\mathcal{D}$
     else return $\langle \neg A\rangle \circ \mathcal{D}$
(15) if there is $A \to B \in \Gamma$
     let $\mathcal{D}_0 = \mathbf{Hp}(\,\Gamma \setminus \{A \to B\},\, A\,)$
     let $\mathcal{D}_1 = \mathbf{Hp}(\,(\Gamma \setminus \{A \to B\}) \cup \{B\},\, G\,)$
     if, for some $i \in \{0,1\}$, $\mathcal{D}_i$ is a model, then return $\mathcal{D}_i$
     else return $\mathcal{D}_0 \circ \langle A \to B\rangle \circ \mathcal{D}_1$
     // *here* $\Gamma$ *is reduced*
(16) return the model $\Gamma \cap \mathrm{Pv}$
```

Before discussing the properties of $\mathbf{Hp}$, we give a high-level overview of the proof-search procedure. Steps (1), (2) and (16) are the base cases of $\mathbf{Hp}$ and immediately return a value. In particular, Step (1) returns a derivation of G when it is either an axiom or an assumption and Step (2) returns the derivation of G when Γ is contradictory. As for Step (16), it is executed when steps (1)-(15) cannot be applied; thus, Γ is reduced and G is the empty goal $\Box$. As we discuss later (see Theorem 4), the returned value $\Gamma \cap \mathrm{Pv}$ is a model of Γ. Steps (3)-(6) act on an assumption $K \in \Gamma$; according with the form of K, these steps reduce the problem to prove $\Gamma \vdash G$ to the problem to prove $\Gamma' \vdash G$, where Γ' is obtained by replacing K with its relevant subformulas. These steps exploit axioms (Ax8), (Ax4a), (Ax4b), (Ax6) to decompose the assumption. Steps (7)-(12) act on the goal formula G. Step (7) essentially corresponds to an application of *reductio ad absurdum*. Note that this case only applies when G is an atomic or a disjunctive formula; how we discuss later (see Point (P6) below), the side condition $\mathbf{eval}(G, \Gamma) \neq \mathtt{f}$ has an essential role to guarantee termination. Steps (8)-(12) exploit axioms (Ax3), (Ax5a) and the closing assumption schemas of Lemma 2 to decompose the goal formula. Steps (13) and (14) handle the case of negated assumptions. We note that in Step (13) the treated assumption $\neg(B_0 \vee B_1)$ is retained in the recursive call; also in this case the firing condition involving the evaluation function plays an essential role to guarantee termination (see Point (P8) below). Finally, Step (15) handles the case of implicative assumptions.

Termination and completeness

To search for a derivation of a goal G_0 from assumptions Γ_0, we have to compute $\mathbf{Hp}(\Gamma_0, G_0)$. We show that the call $\mathbf{Hp}(\Gamma_0, G_0)$ terminates. The stack of recursive calls involved in the computation of $\mathbf{Hp}(\Gamma_0, G_0)$ can be represented by a chain of the form $\Gamma_0 \vdash G_0 \;\mapsto\; \Gamma_1 \vdash G_1 \;\mapsto \cdots \mapsto\; \Gamma_n \vdash G_n$ where, for every $k \geq 0$, Γ_k is a set of formulas and the goal G_k is a formula or $\Box$. Each sequent $\sigma_k = \Gamma_k \vdash G_k$ in the chain represents a call $\mathbf{Hp}(\Gamma_k, G_k)$ performed along the computation of $\mathbf{Hp}(\Gamma_0, G_0)$. By $\mathcal{G}_k$ we denote the set $\Gamma_k \cup \{\neg G_k\}$ if $G_k \neq \Box$ and the set Γ_k if $G_k = \Box$.

We have to prove that, for every set of formulas Γ_0 and goal G_0, every chain starting from $\sigma_0 = \Gamma_0 \vdash G_0$ is finite (namely, the chain contains finitely many sequents). Let us assume, by absurd, that there exists an infinite chain $\mathcal{C}(\sigma_0) = \sigma_0 \mapsto \sigma_1 \mapsto \sigma_2 \ldots$; we show that we get a contradiction. To this aim, we state some properties of $\mathcal{C}(\sigma_0)$ (the related proofs are only sketched).

(P1) For every $k \geq 0$, G_k is not an axiom.
(P2) For every $k \geq 0$, the set $\mathcal{G}_k$ is not contradictory.

Let us assume, by contradiction, that there exists $k \geq 0$ such that G_k is an axiom. Then the call $\mathbf{Hp}(\Gamma_k, G_k)$ immediately ends at step (1), hence σ_k would be the last sequent of $\mathcal{C}(\sigma_0)$, against the assumption that $\mathcal{C}(\sigma_0)$ is infinite. This proves (P1).

To prove (P2), let us assume that there exists $k \geq 0$ such that $\mathcal{G}_k$ is contradictory. Then, there is a formula X such that $\{X, \neg X\} \subseteq \mathcal{G}_k$. If $\{X, \neg X\} \subseteq \Gamma_k$ or $G_k = \Box$, then the call $\mathbf{Hp}(\Gamma_k, G_k)$ immediately ends at step (2), a contradiction. Thus, let us assume $G_k \neq \Box$ and either $G_k \in \Gamma_k$ or $\neg\neg G_k \in \Gamma_k$. In the first case, the call $\mathbf{Hp}(\Gamma_k, G_k)$ immediately ends at step (1). If $\neg\neg G_k \in \Gamma_k$, by inspecting $\mathbf{Hp}$ one can check that there exists $j > k$ such that $G_k \in \Gamma_j$ and $G_j = G_k$ (see step (3)), hence σ_j would be the last sequent of $\mathcal{C}(\sigma_0)$, a contradiction. This concludes the proof of (P2).

By (P2) and the properties of $\mathbf{eval}$, we can prove that the evaluation of a formula is preserved along the chain, as stated in the next property:

(P3) Let $\mathbf{eval}(A, \mathcal{G}_k) = \tau$, with $\tau \in \{\mathtt{t}, \mathtt{f}\}$. Then, for every $j \geq k$, $\mathbf{eval}(A, \mathcal{G}_j) = \tau$.

To prove (P3), we can exploit Lemma 4, since, by Point (P2), all the sets $\mathcal{G}_k$ are non-contradictory.

By $\mathrm{Sf}^{\neg}(\sigma_0)$ we denote the set of the formulas H and $\neg H$ such that H is a subformula of a formula in σ_0. By inspecting the definition of $\mathbf{Hp}$, one can easily check that:

(P4) For every $k \geq 0$, $\mathcal{G}_k \subseteq \mathrm{Sf}^{\neg}(\sigma_0)$.
(P5) For every $i \leq j$, $\mathcal{G}_i \cap \mathcal{L}^{\neg} \subseteq \mathcal{G}_j \cap \mathcal{L}^{\neg}$.

We now state some crucial properties of disjunctive goals:

(P6) Let $G_k = A \vee B$ and $\mathbf{eval}(\Gamma_k, A) \neq \mathtt{f}$. Then, there exists $j > k$ such that $\mathbf{eval}(\Gamma_j, A) = \mathtt{f}$.
(P7) Let $G_k = A \vee B$ and $\mathbf{eval}(\Gamma_k, A) = \mathtt{f}$ and $\mathbf{eval}(\Gamma_k, B) \neq \mathtt{f}$. Then, there exists $j > k$ such that $\mathbf{eval}(\Gamma_j, A) = \mathbf{eval}(\Gamma_j, B) = \mathtt{f}$.
(P8) Let $H = \neg(A \vee B)$ such that $H \in \Gamma_k$ and $\mathbf{eval}(H, \Gamma_k \setminus \{H\}) \neq \mathtt{t}$. Then, there exists $j > k$ such that $H \in \Gamma_j$ and $\mathbf{eval}(H, \Gamma_j \setminus \{H\}) = \mathtt{t}$.

Let us consider Point (P6). If $G_k = A \vee B$ and $\mathbf{eval}(\Gamma_k, A) \neq \mathtt{f}$, then there exists $l \geq k$ such that chain starting from the sequent σ_l has the form

$$\Gamma_l \vdash A \vee B \ \mapsto \Gamma_{l+1} \vdash A \ \mapsto \ \cdots ; \mapsto \Gamma_{l+m} \vdash H$$

where the formula H satisfies one of the following properties:

(a) $H = H_0 \vee H_1$ and $\mathbf{eval}(H_0 \vee H_1, \Gamma_{l+m}) \neq \mathtt{f}$;
(b) H is a literal.

Indeed, after a (possibly empty) initial phase where some formulas in Γ_k are handled, the goal formula $A \vee B$ is eagerly decomposed until a disjunctive formula H satisfying (a) or a literal H is obtained. In this phase, whenever we get a goal $G_j = B \to C$, the formula B is added to the left-hand side and the next sequent of the chain is $\Gamma_j, B \vdash C$ (see step (9)); this might introduce a sequence of steps to decompose the formula B and its subformulas. By exploiting (P3), one can prove that:

(c) $\mathbf{eval}(H, \Gamma_{l+m+1}) = \mathtt{f}$ implies $\mathbf{eval}(A, \Gamma_{l+m+1}) = \mathtt{f}$.

If H satisfies (a), the next sequent of the chain is $\sigma_{l+m+1} = \Gamma_{l+m}, \neg H \vdash \Box$ (see step (7)). Since $\mathbf{eval}(H, \Gamma_{l+m+1}) = \mathtt{f}$, by (c) we get $\mathbf{eval}(A, \Gamma_{l+m+1}) = \mathtt{f}$, hence Point (P6) holds. Similarly, if H is a propositional variable, then $H \notin \Gamma_{l+m}$, otherwise the set $\mathcal{G}_{l+m}$ would be contradictory, against Point (P2). Thus, the next sequent of the chain is $\sigma_{l+m+1} = \Gamma_{l+m}, \neg H \vdash \Box$ (see step (7)), and this proves Point (P6). The case $H = \neg p$, with $p \in \mathrm{Pv}$, is similar (see step (10)). The proofs of points (P7) and (P8) are similar.

By the above points, it follows that we eventually get a sequent σ_m such that Γ_m is reduced and $G_m = \Box$. Indeed, by (P4), the formulas occurring in $\mathcal{C}(\sigma_0)$ are subformulas of $\mathrm{Sf}^{\neg}(\sigma_0)$. Along the chain, formulas H occurring in Γ_k such that $H \notin \mathcal{L}^{\neg}$ are eventually decomposed, formulas $H \in \mathcal{L}^{\neg}$ are preserved (see (P5)). Formulas of the kind $\neg p$, with $p \in \mathrm{Pv}$, do not threaten termination. Let us take a formula $H = \neg(A \vee B)$ occurring in some Γ_k. By (P8), there exists $l > k$ such that $\sigma_l = \Gamma_l \vdash G_l$ and $\mathbf{eval}(H, \Gamma_l \setminus \{H\}) = \mathtt{t}$. By (P3), for every $j \geq l$ it holds that $\mathbf{eval}(H, \Gamma_j \setminus \{H\}) = \mathtt{t}$. Since the formulas of the kind $\neg(A \vee B)$ that can occur in the sets Γ_k are finitely many, it follows that we eventually get a sequent σ_m such that Γ_m is reduced and $G_m = \Box$. Since the only step applicable to the call $\mathbf{Hp}(\Gamma_m, G_m)$ is (16), σ_m should be the last sequent of $\mathcal{C}(\sigma_0)$, a contradiction. This proves that $\mathcal{C}(\sigma_0)$ is finite, thus:

Theorem 3 (Termination): Let Γ be a finite set of formulas of $\mathcal{L}$ and G a formula of $\mathcal{L}$ or $\Box$. Then, $\mathbf{Hp}(\Gamma, G)$ terminates.

Now, we prove that $\mathbf{Hp}$ is correct, namely:

Theorem 4 (Correctness): Let Γ be a finite set of formulas of $\mathcal{L}$ and G a formula of $\mathcal{L}$ or $\Box$. Then, $\mathbf{Hp}(\Gamma, G)$ satisfies properties (H1) and (H2).

Proof: By induction on the length l of $\mathcal{C}(\Gamma \vdash G)$. If $l = 0$, then the computation of $\mathbf{Hp}(\Gamma, G)$ does not require any recursive call. Accordingly, one of the steps (1), (2) or (16) is executed. In the first two cases, a derivation satisfying properties (H1) and (H2) is returned. In the latter case, the model $\mathcal{M} = \Gamma \cap \mathrm{Pv}$ is returned. Since none of the conditions in the if-statements of cases (1)–(15) holds, the set Γ is reduced and $G = \Box$; by Theorem 2, we conclude $\mathcal{M} \models \Gamma$. Let $l > 1$ and let $\mathcal{C}(\Gamma \vdash G) = \Gamma \vdash G \mapsto \Gamma_1 \vdash G_1 \ \mapsto \ \cdots$. Since the length of $\mathcal{C}(\Gamma_1 \vdash G_1)$ is $l - 1$, by induction hypothesis the call $\mathbf{Hp}(\Gamma_1, G_1)$ satisfies properties (H1) and (H2). By a case analysis, one can easily check that $\mathbf{Hp}(\Gamma, G)$ satisfies (H1) and (H2) as well. For instance, let us assume that $G = A_0 \vee A_1$ and that step (11) is executed. If $\mathbf{eval}(A_0, \Gamma) = \mathtt{f}$, the recursive call $\mathbf{Hp}(\Gamma, A_0)$ is executed. If a deduction $\mathcal{D}$ is returned, by induction hypothesis $\mathcal{D}$ is a deduction of $\Gamma \vdash A_0$. This implies that the deduction $\mathcal{D} \circ \langle A_0 \to (A_0 \vee A_1), A_0 \vee A_1 \rangle$ returned by $\mathbf{Hp}(\Gamma, A_0 \vee A_1)$ is a deduction of $\Gamma \vdash A_0 \vee A_1$ (note that $A_0 \to (A_0 \vee A_1)$ is an instance of the axiom $(\mathrm{Ax}5a)$). Let us assume that $\mathbf{Hp}(\Gamma, A_0)$ returns a model $\mathcal{M}$. By induction hypothesis, we have $\mathcal{M} \models \Gamma \cup \{\neg A_0\}$. Moreover, since the if-condition in step (7) is false, it holds that $\mathbf{eval}(A_0 \vee A_1, \Gamma) = \mathtt{f}$. By Theorem 1, we get $\mathcal{M} \models \neg(A_0 \vee A_1)$, hence $\mathcal{M} \models \Gamma \cup \{\neg(A_0 \vee A_1\}$. The proof of the remaining subcases is similar. ■

To conclude this section we provide two examples, the first one showing an example of an execution of the proof-search procedure returning a counter model, the second one returning an $\mathbf{Hc}$-derivation.

Example 2: Let us consider the case where the goal formula is $p \lor q$ and the set of the assumptions is empty. We describe the chain of recursive calls applied to compute the goal, referring to the steps in the definition of $\mathbf{Hp}$.

(c1) $\mathbf{Hp}(\emptyset, p \lor q)$
Since $\mathbf{eval}(p \lor q, \emptyset) = p \lor q$*, case (7) is applied and the recursive call* $\mathbf{Hp}(\{\neg(p \lor q)\}, \Box)$ *is executed.*

(c2) $\mathbf{Hp}(\{\neg(p \lor q)\}, \Box)$
Since $\mathbf{eval}(\neg(p \lor q), \emptyset) = \neg(p \lor q)$*, case (13) is selected and* $\mathbf{Hp}(\{\neg(p \lor q)\}, p \lor q)$ *is executed.*

(c3) $\mathbf{Hp}(\{\neg(p \lor q)\}, p \lor q)$
Since $\mathbf{eval}(p \lor q, \{\neg(p \lor q)\}) = \mathtt{f}$*, case (11) is selected. Moreover, since* $\mathbf{eval}(p, \{\neg(p \lor q)\}) = p$*, the call* $\mathbf{Hp}(\{\neg(p \lor q)\}, p)$ *is executed.*

(c4) $\mathbf{Hp}(\{\neg(p \lor q)\}, p)$
Since $\mathbf{eval}(p, \{\neg(p \lor q)\}) = p$*, case (7) is selected and* $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, \Box)$ *is executed.*

(c5) $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, \Box)$
Since $\mathbf{eval}(\neg(p \lor q), \{\neg p\}) = \neg q$*, case (13) is selected and* $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, p \lor q)$ *is executed.*

(c6) $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, p \lor q)$
Since $\mathbf{eval}(p \lor q, \{\neg(p \lor q), \neg p\}) = \mathtt{f}$*, case (11) is selected. Moreover, since* $\mathbf{eval}(p, \{\neg(p \lor q), \neg p\}) = \mathtt{f}$ *and* $\mathbf{eval}(q, \{\neg(p \lor q), \neg p\}) = q$*,* $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, q)$ *is executed.*

(c7) $\mathbf{Hp}(\{\neg(p \lor q), \neg p\}, q)$
Since $\mathbf{eval}(q, \{\neg(p \lor q), \neg p\}) = q$*, case (7) is selected and* $\mathbf{Hp}(\{\neg(p \lor q), \neg p, \neg q\}, \Box)$ *is executed.*

(c8) $\mathbf{Hp}(\{\neg(p \lor q), \neg p, \neg q\}, \Box)$
Since $\mathbf{eval}(\neg(p \lor q), \{\neg p, \neg q\}) = \mathtt{t}$ *and* $\neg p$ *and* $\neg q$ *are literals, case (16) is executed and the model* $\emptyset$ *is returned.*

Since the recursive call in (c8) returns the model $\emptyset$*, any of the recursive calls in points (c7)-(c1) returns the same model, which indeed is a model of* $\{\neg(p \lor q)\}$.

Example 3: Let us consider the case where the goal formula is $p \lor \neg p$ *and the set of assumptions is empty.*

(c1) $\mathbf{Hp}(\emptyset, p \lor \neg p)$
Since $\mathbf{eval}(p \lor \neg p, \emptyset) = p \lor \neg p$*, case (7) is selected and the recursive call* $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, \Box)$ *is executed.*

(c2) $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, \Box)$
Since $\mathbf{eval}(\neg(p \lor \neg p), \emptyset) = \neg(p \lor \neg p)$*, case (13) is selected and* $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, p \lor \neg p)$ *is executed.*

(c3) $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, p \lor \neg p)$
Since $\mathbf{eval}(p \lor \neg p, \{\neg(p \lor \neg p)\}) = \mathtt{f}$*, case (11) is selected and since* $\mathbf{eval}(p, \{\neg(p \lor \neg p)\}) = p$*, the call* $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, p)$ *is executed.*

(c4) $\mathbf{Hp}(\{\neg(p \lor \neg p)\}, p)$
Since $\mathbf{eval}(p, \{\neg(p \lor \neg p)\}) = p$ *case (7) is selected and* $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, \Box)$ *is executed.*

(c5) $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, \Box)$
Since $\mathbf{eval}(\neg(p \lor \neg p), \{\neg p\}) = \mathtt{f}$*, case (13) is selected and* $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, p \lor \neg p)$ *is executed.*

(c6) $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, p \lor \neg p)$
Since $\mathbf{eval}(p \lor \neg p, \{\neg(p \lor \neg p), \neg p\}) = \mathtt{t}$*, case (11) is selected. Moreover, we have* $\mathbf{eval}(p, \{\neg(p \lor \neg p), \neg p\}) = \mathtt{f}$ *and* $\mathbf{eval}(\neg p, \{\neg(p \lor \neg p), \neg p\}) = \mathtt{t}$*; as a consequence* $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, \neg p)$ *is executed.*

(c7) $\mathbf{Hp}(\{\neg(p \lor \neg p), \neg p\}, \neg p)$
Since $\neg p \in \{\neg(p \lor \neg p), \neg p\}$ *case (1) is selected and the* $\mathbf{Hc}$*-derivation* $\langle \neg p \rangle$ *is returned.*

Since the recursive call in (c7) returns a derivation, any of the calls in points (c6)-(c1) returns the derivation built according with the receipt described in the corresponding case.

V. Conclusions

In this paper, we have presented a deterministic and terminating proof-search procedure $\mathbf{Hp}$ for a Hilbert calculus for classical propositional logic (a Prolog implementation is available[10]). $\mathbf{Hp}$ is a procedure that builds Hilbert proofs, if any, in one-pass, that is during the proof-search phase. Such proofs can be seen as derivations based on a sequent calculus having at most one formula on the right. The machinery of evaluations is used to get completeness and termination.
As regards future works, we plan to provide terminating procedures returning Hilbert proofs for intermediate logics by applying the results in [9][11]. We also aim to extend the proof-search procedure for $\mathbf{Hc}$ to treat some modal logics, as, e.g., $S4$ and $S5$ [12] and conditional logics [13].

References

[1] A. Troelstra and H. Schwichtenberg, Basic Proof Theory, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000, vol. 43.

[2] M. D'Agostino, "Classical natural deduction," in We Will Show Them!, S. A. et al., Ed. College Publications, 2005, pp. 429–468.

[3] W. Sieg and J. Byrnes, "Normal natural deduction proofs (in classical logic)," Studia Logica, vol. 60, no. 1, 1998, pp. 67–106.

[4] M. Ferrari and C. Fiorentini, "Proof-search in natural deduction calculus for classical propositional logic," in TABLEAUX 2015, ser. LNCS, H. D. Nivelle, Ed., vol. 9323. Springer, 2015, pp. 237–252.

[5] ——, "Goal-oriented proof-search in natural deduction for intuitionistic propositional logic," Journal of Automated Reasoning, vol. 62, no. 1, Jan 2019, pp. 127–167. [Online]. Available: https://doi.org/10.1007/s10817-017-9427-3

[6] S. Kleene, Introduction to Metamathematics. Van Nostrand, 1952.

[7] E. Eder, "Automated Deduction in Frege-Hilbert Calculi," in 8th WEAS International Conference on Applied Computer and Applied Computational Science, Chen, SY and Li, Q, Ed. World Scientific and Engineering Acad and Soc, 2009, pp. 21–25.

[8] J. Toriz, I. Ruiz, and J. R. Arrazola-Ramírez, "On automatic theorem proving with ML," in 13th Mexican International Conference on Artificial Intelligence, MICAI 2014, 2014, pp. 231–236.

[9] M. Ferrari, C. Fiorentini, and G. Fiorino, "An evaluation-driven decision procedure for G3i," ACM Transactions on Computational Logic (TOCL), vol. 6, no. 1, 2015, pp. 8:1–8:37.

[10] "Implementation of **Hp** ", 2019, URL: https://drive.google.com/open?id=1nGvqKx_Rj0y2IscL4jldH2jPMWzDB-0X.

[11] A. Avellone, P. Miglioli, U. Moscato, and M. Ornaghi, "Generalized tableau systems for intermediate propositional logics," in TABLEAUX 1997, ser. LNAI, D. Galmiche, Ed., vol. 1227. Springer-Verlag, 1997, pp. 43–61.

[12] A. Chagrov and M. Zakharyaschev, Modal Logic. Oxford University Press, 1997.

[13] D. Lewis, Counterfactuals. John Wiley & Sons, 2013.

Vision on Next Level Quantum Software Tooling

Rob F.M. van den Brink
TNO
ICT Department
The Hague, The Netherlands

Frank Phillipson
TNO
ICT Department
The Hague, The Netherlands
Email: frank.phillipson@tno.nl

Niels M.P. Neumann
TNO
ICT Department
The Hague, The Netherlands

***Abstract*—Software tools for programming gate-based quantum computers are being developed by many parties. These tools should now grow towards a phase where they support quantum devices running realistic algorithms to outperform classical algorithms on digital computers. At this moment, they lack capabilities for generic gates, capabilities for quantum debugging and generic quantum libraries. This paper gives a view on the functionalities needed for such software environments looking at the various layers of the software stack and at the interfaces for quantum cloud computing.**

Keywords–Quantum Software; Quantum Computing; Quantum Compiler.

I. INTRODUCTION

Quantum computers are still in an early stage of development, and experimental quantum processors are getting to support up to a few dozen of qubits [1][2]. That number is growing, and support for a large number of qubits is just a matter of time. Where we speak of quantum computers in this paper, we mean gate based quantum computers and not quantum annealers. Software tools for programming gate-based quantum computers are also in an early stage of development, where the basics, however, were already set 20 years ago [3]–[5]. Currently, these software tools are mainly supporting a set of low-level quantum instructions, embedded in a classical (digital) programming language. They are adequate for running rudimentary quantum applications [6] that can already run on only a few qubits. Quantum computing should now grow towards a phase where the development of quantum software must get more emphasis. This is comparable with the sixties of the previous century, when programming tools like Fortran and Algol brought a much higher abstraction to digital computers than just assembly language.

Since it is not obvious what ‘higher abstraction’ means for quantum computing, this paper discusses what functionality is needed in the next level of quantum computing and how this can be implemented in a structured way, by means of a stack of software layers. Important here is that, in this paper, we assume that (1) quantum processors remain bulky devices for a long time (like digital computers in the beginning), (2) that real quantum applications will always be a mix of digital and quantum computing, where only a part of the problem is solved in a quantum manner, and (3) that we want to make this available for a larger public, that has limited knowledge about the underlying quantum layer. For this, we require (1) a strict interface between remote quantum hardware, (2) local software that runs at the user side that can, (a) independent of the used local programming language, interact with the remote hard- and software, (b) independent of the used local programming language, make use of high level quantum algorithms in libraries, and (c) has profound quantum specific debugging capabilities where (small versions of) the desired algorithms can be analysed, while stepping through the program in debug mode, up to the underlying quantum states using simulations.

TABLE I. Shortlist of languages being used for quantum software tools.

C / C++ based (>30)	Matlab / Octave based (12)
F# based (1)	Maxima based (2)
GUI based (>10)	NET based (4)
Java based (>15)	Online service (10)
JavaScript based (1)	Perl / PHP based (3)
Julia based (1)	Python based (>6)
Maple based (3)	Scheme/Haskell/LISP/ML based (8)
Mathematica based (8)	

The remainder of this paper is organised as follows. In Section II, we give an overview of the current state of the art, by making a functional grouping of existing tools and their capabilities. In Section III, we discuss the functionality we expect from a quantum software environment to meet the above requirements. In Section IV, we sketch our desired software environment, by defining layers and discussing the functionality of each of those layers and the placement of the separation between local and cloud, followed by some examples of (later defined) quantum function libraries in Section V. We conclude in Section VI with a summary and conclusions.

II. STATE OF THE ART

Many quantum software tools are freely available via the Internet, and overviews with good summaries of those tools can be found in [7]–[9]. Most of these tools are still in the development phase. The list of available tools is too long to be duplicated here, but a first impression can be gained by organising them according to the used programming language. Table I shows the programming languages that are being used for implementing quantum tools, and the number of each of these quantum tools. What can be observed in this table is the wide range of different programming languages that has been used. There is no clear winner from that, simply because almost any classical programming language is suitable for this job, with each of these languages having its own advantages and disadvantages.

An other way to group the tools is taking a high-level view on their functionality. We recognise the following functional

TABLE II. Examples of quantum compiling tools

Name	Remarks
Scaffold or ScafCC	C-alike, updated version of CLANG. Compiles to LLVM & QASM [10].
Liquid (Microsoft)	F#-alike, updated version of F# (called Q#), with an emulator on board [11].
Quipper	Haskel-alike [12]
QCL	mix of C and Pascal alike, detailed language specification with emulator on board [13].

groups: (a) quantum compilers, (b) quantum function libraries, and (c) back-end quantum simulators. We are aware that some tools may not fully fit into one particular group and these groups may overlap, but it is helpful to identify several high-level functionalities. We will now discuss the characteristics of each of these groups one by one.

A. *Quantum compilers*

The word *compiler* refers in this context to a tool that translates an entire program from higher level statements into some lower level instructions (e.g., binary or assembly). The execution of that program is started thereafter. The purpose of a compiler is to generate low level quantum instructions (in assembly or as native code), from a mix of low and high level quantum statements, of classical control statements and loops, of organised code in reusable routines, etc.

Quantum tools that present themselves as quantum compilers are mainly modifications from an existing (classical) compiler. The result is a changed syntax to have it extended with quantum specific instructions. The present extensions are mainly to offer a syntax for elementary instructions at the level of quantum assembly code. Table II offers a few examples of those tools.

Their main disadvantage is that creating a new language may break access to existing code libraries with classical algorithms. For instance, the development of quantum algorithms for outperforming classical machine learning algorithms will draw significant benefits if classical code libraries with, for example, neural network algorithms are well-accessible from the same software environment.

Next to this, quantum compilers that introduce new languages, or break compatibility with existing languages lack in most cases powerful development and (quantum) debugging tools.

B. *Quantum function libraries*

The term *function library* refers in this context to the use of an existing and well-supported (classical) programming environment, where the quantum programmer can call quantum-specific routines from a library. Their purpose is similar to that of compilers, with the difference that the generation of low level quantum code occurs in run time. Examples of such quantum tools are: Qiskit (IBM), Quantum Learning Machine (QLM, Atos), PyQuil (Rigetti), ProjectQ (ETH Zurich), and OpenQL (TU Delft).

The approach of using function libraries brings hardly any limitation, think of generating thousands of dedicated quantum instructions by a single call to a library function. For instance, a single call to a routine for a Quantum Fourier Transform or a matrix expression, that automatically generates hundreds of quantum instructions operating on tens of qubits. However, the present quantum function libraries have a strong focus on low-level quantum computing, and may only be targeted at a specific quantum processor. Quantum programming with the present tools is mainly a matter of calling routines for generating individual low level quantum instructions (like assembler). By calling several of them with different parameters, one can generate a sequence of quantum instructions to build a quantum circuit. The identification and handling of higher level quantum instructions is still a topic of further research.

Some of these tools already offer powerful quantum debugging capabilities, like the generation of a drawing of the generated quantum circuit and access to a build-in quantum simulator. Such simulator can return intermediate results, like, for instance, a quantum state, that can never be obtained using a real quantum processor. Some of these tools also offer a graphical user interface, for positioning a few quantum gates into a quantum circuit to process a few qubits. That approach mainly serves an educational purpose for those who are setting their first steps in quantum computing. However, as soon as your quantum program grows in size, the use of scripting for calling quantum functions may become more convenient.

C. *Back-end simulators*

The term *back-end simulator* refers in this context to a tool that reads low level quantum assembly and executes them in a manner like a real quantum processor would do. As such, it becomes irrelevant for the quantum programmer in what language such tool is written since that tool is instructed directly via low level assembly instructions. The concepts of high level instructions and function libraries are not applicable here and therefore we consider it as another group.

Back-end simulators contain a very simple translator, to feed low level assembly instructions (usually stored in an ASCII file) to a build-in simulation engine. They are by definition dedicated to low-level functionality only, and can simulate/emulate on a digital computer what a real quantum processor would have returned. These tools aim at simulating a real quantum processor as good as possible on a classical digital computer. They may even try to simulate the physical imperfections of a particular quantum processor as well. For instance, by adding some random mechanism (noise) to mimic the loss of quantum coherence after executing more and more quantum instructions, or by deliberately accounting for topological limitations of a particular quantum processor.

These tools may be offered as a stand-alone tool, or as part of a larger software environment (sometimes referred to as ‘virtual quantum machine’). The translation functionality they possess, may also be used to interface with a real quantum processor, but that is out of scope here. Examples of such quantum tools are (modules inside) QX (QuTech) [14], QLM (Atos) [15], QVM (Rigetti) [16] and Quantum Experience (IBM) [17].

These tools serve purposes other than quantum function libraries and quantum compilers do. They are valuable to study the quantum assembly language, to study how to deal with limitations in the quantum instruction set of the target machine, or to study quantum error correction methods against decoherence. They are also valuable to study the interfacing between a local software environment and a quantum processor somewhere in the cloud.

TABLE III. Example of two different dialects of quantum assembler, both representing the same quantum circuit

Atos-QASM	QX
BEGIN	
qubits 18	qbits 18
cbits 0	.ckt
H q[0]	H q[0]
H q[5]	H q[5]
CTRL(PH[1.570]) q[15],q[16]	CR q[15], q[16], 1.570
CTRL(PH[0.785]) q[14],q[16]	CR q[14], q[16], 0.785
CTRL(PH[0.392]) q[13],q[16]	CR q[13], q[16], 0.392
H q[15]	H q[15]
...	...
H q[16]	H q[16]
END	

III. Desired software functionality

In this section, we describe the functionality we expect in quantum software, such that it supports implementation of more complex, hybrid digital and quantum algorithms and that it will enable a larger public that has limited knowledge about the underlying quantum layer to access quantum computing.

A. Commonly available functionality

In spite of all functional differences between the discussed tools, they all share a common functionality: the concept of quantum circuits for representing a set of quantum instructions via interconnected gates, and the concept of quantum assembler or Quantum Assembly Language (QASM) as a language for describing those circuits as a list of sequential quantum instructions. There are several of these quantum assembly languages, each with their own dialect (syntax), but from a pure conceptual point of view they all are roughly the same. Table III shows an example of two different QASM dialects, both describing the same quantum circuit. The first one shows the syntax used by QLM (from Atos), the second one shows the syntax used by QX (within Quantum Inspire).

A QASM file is typically a sequence of individual QASM instructions, embedded between a header (with declarations) and a footer. Some QASM tools also allow for grouping those instructions into macros (like functions) to simplify the use of the same code multiple times. Each QASM instruction is build-up from (a) an instruction name, (b) optional parameters, and (c) a list of qubit identifiers on which this instruction should operate. Each instruction can change the contents of a ‘quantum register’ (via gates or measurements), and many of them in sequence define a quantum circuit. There is an apparent consensus on the naming of several of these gates, for instance, instruction names like H, X, Y, Z, but this consensus does not hold for all gates. This difference can be confusing, but is not a big issue when well defined. When these names are well specified by means of the corresponding matrix representation, the conversion from one QASM dialect into another is pretty straight forward. And when a particular gate is not available in one QASM, it can always be created in another QASM via a combination of a few other gates. Note that some QASM dialects start their counting of qubits from 0, while others start from 1. This is only a matter of convention and preference, mainly driven by the supporting language, and not a big issue either. In conclusion, one may say that it is relatively easy to translate one QASM dialect into another one.

B. Next level functionality

To reach the desired level of quantum programming that supports the implementation of more complex, hybrid quantum algorithms and to enable a larger public that has limited knowledge about the underlying quantum layer to access quantum computing, we need a next level of software functionalities. Examples of these functionalities are:

Desired capabilities for generic gates

- Define circuit libraries with generic gates, with an arbitrary number of qubits, and callable as if it was a single instruction. OpenQASM [18] does supports the concept of macros and can provide this functionality, but that concept is not available in all QASMs. The desired circuit library should generate, for instance, a Quantum Fourier Transform, or a circuit for modular exponentiation by one instruction/call for an arbitrary large number of qubits.
- Define circuit libraries with generic gates in terms of an unitary matrix or a matrix expression, while the tool translates that into circuits with only basic gates. This capability is currently a weak point for almost all tools. It may be available for one or two qubit gates, but the issues get problematic for more qubits.

Desired capabilities for quantum debugging

- The capabilities to let the software draw a circuit from a QASM specification to debug what has been specified. Tools like Liquid, QLM and ProjectQ give support for that, but it should be available on all tools of interest.
- The capability to read out the full vector (or full matrix) with complex numbers representing the present quantum state or circuit (also at intermediate points). This is impossible with a real quantum processor, and only possible with a simulator. However, it is a very powerful and essential debugging facility. Some simulation tools do support this, but there are very primitive solutions among them.
- The capability to analyse generic quantum states and gates (during simulation), while stepping through the program in debug mode, using sophisticated linear algebra tooling.
- The capability to visualise in an abstract manner relevant aspects of the (full) state vector or (full) matrix in case they are too large for a full numerical inspection. For instance, histograms, magnitude plots by means of colours, etc,

Desired quantum libraries

- Libraries that implement many quantum functions/circuits, callable from your quantum program. Many functions are well known from the literature, but inserting, for instance, a Quantum Fourier Transform operating on an arbitrary number of qubits (in arbitrary order) should be as simple as inserting a single qubit X gate. Section V provides further details.
- Running quantum programs from a programming environment with good access to classic libraries with legacy solutions for the quantum application under study. This may mean (a) a language that is different

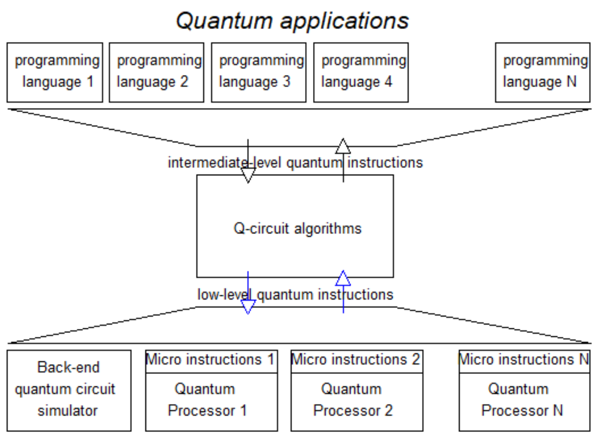


Figure 1. Software stack for quantum tooling with different layers

from the implementation language of the quantum tools, and/or (b) software that prepares the quantum application locally (for debugging reasons) for running it on a remote quantum processor (via a well-supported API, Application Programming Interface), and/or (c) software that uses a different operating system from the one used by the quantum processor. We will also discuss this further in the next chapter.

These desired functionalities are examples only, and the list is not complete. They have currently a strong focus on (quantum) debugging capability, flexibility, and libraries. The present tools may implement fragments of this list, but we have not found a tool that can support them all in a convenient manner.

IV. DESIRED SOFTWARE ENVIRONMENT

If we know what functionality we need in the software to enter the next level of quantum computing, we should define where and how this functionality should be implemented and where the separation between local and cloud can be placed.

A. *Layered software stack*

To define where and how the functionality should be implemented, we use a full software stack with different layers, as shown in Figure 1, covering functionality from quantum applications down to quantum processors. The concept in Figure 1 is very similar to the picture shown in [19][20]. We, however, put more emphasis on the desired functionality within the intermediate levels.

1) Functionalities of bottom Layers: The very bottom of the low-level layers are reserved for the quantum processors, each equipped with dedicated hardware for controlling it via (binary) micro instructions. For instance, to change the quantum state of a quantum processor via dedicated pulses. A software layer directly above this hardware allows for translating a sequence of low-level quantum instructions (e.g., assembly or binary code) into dedicated pulses for the hardware. This software layer should hide the hardware difference among different quantum processors as much as possible, in order to program them with uniform instructions that are more or less hardware-independent. In practice, very different quantum processors will be implemented, based on different physical principles, and each with their own micro instruction sets. The quantum programmer should not be bothered by that.

These instructions are still very low-level, and the use of some QASM dialect is the most obvious choice here. These instructions allow for defining quantum circuits with basic gates, where the word basic refers to a set of predefined gates operating on 1 or 2 qubits only. These instructions may be restricted to the (hardware) instruction range of the target quantum processor, and may also account for topological (hardware) limitations.

Since quantum processors are still under development, the use of a back-end circuit simulator also has its place in the lower layers. Their aim is to simulate a real quantum processor as fast as possible, with as many qubits as the used digital computer platform can handle, and to simulate/emulate all its imperfections and, if applicable, all topology limitations as good as possible.

2) Functionalities of intermediate Layers: A more intermediate level (the Q-circuit algorithms box in Figure 1) gives the quantum programmer access to all kinds of quantum algorithms, in a uniform manner, ideally independent from the used quantum hardware and programming languages. It is, for instance, equipped with all kinds of algorithm libraries (examples can be found in the next section) and quantum debugging capabilities to design the quantum-specific parts of applications.

These instructions are at least capable of defining arbitrary circuits with generic gates. This means within this context that they can operate on an arbitrary number of qubits, far beyond the instruction set of the quantum processor, and they may even be specified via (unitary) matrices or high level expressions. The translation from quantum circuit with a few generic gates into circuits with many basic gates is the proper place to perform gate and qubit optimisation. The best results can be achieved when this translation is partly guided by control parameters representing some of the hardware limitations of the target quantum processor (hardware aware). These control parameters are set only once for a particular target quantum processor, and preferably invisible in the instructions with generic gates (hardware agnostic). As such, this translation is an important intermediate step in quantum programming, applicable to both quantum compilers and tools based on function libraries.

Figure 2 provides an example of a quantum circuit with generic gates. At a first glance, generic may look the same as basic gates, but in this case the gates can also be black boxes operating on many qubits simultaneously specified as matrices, expressions, or standard functionalities.

One may still consider the supported instructions as rather low-level, but the distinction between low and intermediate level brings a significant advantage. If quantum computing is offered via the cloud, for running quantum applications from all over the world, then the interface between the lower and intermediate layer is a natural interface for the cloud based quantum computer. And the use of one or more QASM dialects is a natural component in the interfacing between these layers. It means that software in the intermediate layers may fully run on a local computer, while software in the lower layer should typically run on a remote quantum host.

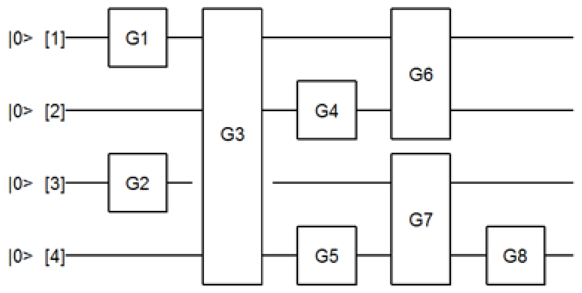


Figure 2. Example of a quantum circuit with generic gates.

3) Functionalities of higher layers: The higher layers (above the Q-circuit algorithms box in Figure 1) are typically reserved for classical programming environments that are extended with quantum capabilities for solving dedicated subproblems. Here the new group of users, with limited knowledge of underlying quantum techniques should be able to play around. It would be a waist of effort if each programming environment has to develop its own collection of libraries with quantum-specific algorithms. Therefore it is far more efficient to equip them only with language-specific interfaces, wrapped around common quantum libraries and debugging capabilities from the intermediate layer.

Quantum applications will most likely be a hybrid mix of classical programming concepts and quantum-specific algorithms. This means that the classical programming languages call a quantum algorithm only when needed for solving particular subproblems. These classical programming languages should offer the following capabilities:

- Good access to classical software libraries, with dedicated algorithms for the problem area. Think of libraries for artificial intelligence applications. But think also of access to quantum circuit libraries with dedicated quantum-specific algorithms like quantum solutions for dealing with decoherence errors or for decomposing a large number into its prime factors.
- Good access to quantum debugging capabilities, also for the quantum specific aspects. Think of inspecting quantum states and matrices of quantum circuits via a build-in simulator and drawing quantum circuits from instructions. These debugging capabilities should easily interact with the higher layers, all in a very interactive and flexible manner.

As such, the universal programming language for everybody does not exist, and therefore the best solution for that is that the intermediate layers offer access to quantum-specific libraries for any programming language of interest. It supports many quantum circuit algorithms as well as quantum-specific debugging capabilities.

The use of languages with build-in support for linear algebra expressions, that are also available as interpreter, give the user powerful extra capabilities for quantum debugging. These tools allow for inspecting and manipulating intermediate results of circuit matrices and state vectors in a very interactive manner during simulation. Linear algebra languages like Matlab/Octave, and to some extend Python with Numpy, are examples of languages offering the desired linear algebra capabilities in an interactive environment and offer access to a broad spectrum of classical code libraries.

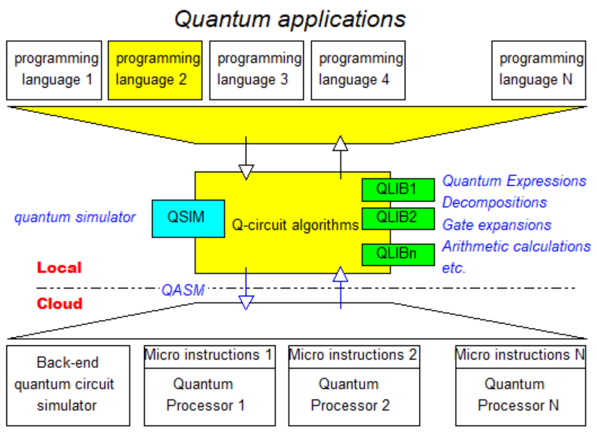


Figure 3. Quantum software stack overview.

B. *Separation between Local and Cloud*

It is assumed that quantum computers remain big installations with bulky refrigeration equipment for a long time. Commercial deployment of quantum computing will then mean a quantum computer hosted in a remote building, offering access via the cloud to many users all over the world. Today, experimental quantum computers already give access via the cloud, but mainly in a restricted manner; end users have to setup a remote terminal session with the hosting computer and they should develop and run everything on that host. This is quite inconvenient as it limits rapid interaction with local software, like exchanging intermediate results with local debugging software. Moreover, commercial users may not be willing to share their source code with the hosting organisation. Exchange of low-level code (binary or assembly) gives then a similar protection as is common today for distributing programs as a binary executable. In that case, end users develop, test and debug on a small scale (locally) at an intermediate level, and then send low level quantum instructions to a remote host in order to run at full quantum speed and size.

The most convenient way of implementing that is therefore not by opening remote terminal sessions, but by accessing the remote quantum computer via an API. An API allows the user to program his (quantum) application in a language that differs from the programming language being used by the remote host. The intermediate layers will then send QASM-alike quantum instructions to a remote host, while the end-user experiences it as if it runs locally. Figure 3 illustrates this interaction model, where the interface between local and cloud is situated between the intermediate and lower layers. The intermediate layers are equipped with all kinds of libraries for quantum specific calculations, as well as debugging capabilities via a local quantum simulator. These libraries generate the required low level QASM instructions and can forward them through a language-independent interface to the lower layers running in the cloud.

V. Example libraries

To get an idea of the type of algorithms that can be implemented via libraries, we will discuss a few examples: libraries that convert a mathematical expression into a quantum circuit, libraries that decompose a generic gate or generic matrix description of such gates into circuits with basic gates and libraries that convert arithmetic calculations into a quantum circuit.

A. *Quantum expression libraries*

The first example is dedicated to generic gates described by mathematical functions for generating special unitary matrices. Think, for instance, of the following matrix expressions:

$$
\begin{aligned}
G_1(a) &= e^{j*a*Z} \\
G_2(a) &= e^{j*a*ZZ} \\
G_3(a) &= e^{j*a*ZZZZ} \\
G_4(a,b,c,d) &= e^{j*(a*XX+b*YY+c*ZZ+d*II)}
\end{aligned}
$$

Where

- a, b, c, d are parameters with arbitrary real values;
- X, Y, Z are the Pauli matrices;
- I is the unity matrix;
- XX, YY, ZZ, II are the Kronecker products of X, Y, Z and I with themselves;
- $ZZZZ$ is the Kronecker product of ZZ and ZZ.
- j indicating the imaginary number $\sqrt{-1}$.

The solution for the first two examples is quite easy, and can be found in almost any basic text book on quantum computing. But finding a good solution for the last one is less obvious. Fortunately, it can be represented by a relative simple circuit [21]. One can easily imagine that the list of such expressions is virtually unlimited, which illustrates the value of bringing them all together into a well-organised quantum expression library.

B. *Quantum decomposition libraries*

Another example occurs when mathematical functions are not available in the expression libraries, as discussed in the previous section. In those cases a solution may be to use a numerical evaluation of that function into a unitary matrix with complex numbers. For instance, to produce an 8×8 matrix representing a 3 qubit gate. It is not that difficult to decompose an arbitrary matrix into the product of much simpler matrices, but a simpler matrix does not automatically mean a simpler quantum circuit. Examples of useful matrix decompositions are singular value decompositions, sine-cosine decompositions [5] or QR-decompositions with Givens rotations [22]. However, these decompositions can easily result in large quantum circuits with an exponential number of basic gates, and can also produce quite inefficient solutions. Decomposition is still an important topic for further research, because we still need algorithms that convert arbitrary matrices into quantum circuits, such that it is fully automatic and produces an efficient circuit as well.

When the matrix is not fully arbitrary, dedicated solutions may yield far more efficient solutions then the generic approach. Examples are

- U is a 1-qubit gate specified by a 2×2 unitary matrix with arbitrary complex numbers;
- cU is a controlled version of U, representing a 4×4 matrix;
- ccU is a double controlled version of U, representing a 8×8 matrix.

Such generic gates can be decomposed into multiple basic gates and the simplest one can be found in almost any basic text book on quantum computing. But finding solutions for multiple controlled gates is not obvious, and should be generated automatically by a single function call, for an arbitrary matrix U and with an arbitrary number of control inputs. One can easily imagine that the list of different decompositions is virtually unlimited, which illustrates the value of bringing them all together into a well-organised 'quantum decomposition library'.

C. *Quantum arithmetic libraries*

Several quantum applications make use of algorithms where discrete numbers are represented by distinct quantum states. For instance, algorithms that make use of modular additions, modular multiplications or modular exponentiations. In those cases it is not obvious what the most efficient way is to implement these on many qubits. We will show three example circuits of how to calculate something 'simple' like the modular increment of a discrete number encoded in 4 qubits. The first one (Figure 4a) can be found in any textbook, looks quite simple, however, requires gates with many control inputs. The second one (Figure 4b) with Peres gates is also known [23], looks more complicated, but requires only single qubits gates and single controlled qubit gates. The used gates are explained in Table IV.

The third example (Figure 4c) using quantum Fourier transforms [24], however, appeared to be the most simple one of these three, in the sense that only single qubit gates and controlled phases are used. This may illustrate that generating an algorithm with the most efficient circuit is not obvious even for a very simple problem.

There are many more of these modular arithmetic calculations, each of them with multiple implementations. Their details are out of scope here, but it may again illustrate the value of bringing all these implementations together into a well-organised 'quantum arithmetic library'.

TABLE IV. Explanation of used gates in Figures 4c and 4b.

Peres approach	QFT approach
$G1 = c([1+q4, 1-q4; 1-q4, 1+q4]/2)$	$G1 = cR(\pi/2)$
$G2 = c([1+q2, 1-q2; 1-q2, 1+q2]/2)$	$G2 = cR(\pi/4)$
$G3 = G2'$ (conjugated transpose of G2)	$G3 = cR(\pi/8)$
$G4 = G1'$ (conjugated transpose of G1)	$G4 = Rs(\pi/8)$
	$G5 = Rs(\pi/4)$
where	$G6 = Rs(\pi/2)$
$q2 = (-j)$	$G7 = Rs(\pi)$
$q4 = \sqrt{(-j)}$	$G8 = cR(-\pi/2)$
$c([g_{11}, g_{12}; g_{21}, g_{22}]) =$	$G9 = cR(-\pi/4)$
$[1, 0, 0, 0;$	$G10 = cR(-\pi/8)$
$0, 1, 0, 0;$	
$0, 0, g_{11}, g_{12};$	where
$0, 0, g_{21}, g_{22}]$	$Rs(\phi) = [1, 0; 0, e^{(j*\phi)}]$
	$cR(\phi) = c(Rs(\phi))$

```
1 :--cccX-----------
      |
2 :---o-ccX---------
      |  |
3 :---o--o--cX------
      |  |  |
4 :---o--o--o--X----
```

(a) Basic circuit of evaluating the modular increment of a number, encoded in 4 qubits.

```
1 :---G1-G1-G2-------------G4-G3------
      |  |  |              |  |
2 :-- |  | -o--G2-G2----G3 | -o-------
      |  |     |  |     |  |
3 :-- | -o---- | -o--cX-o--o----------
      |        |     |
4 :---o--------o-----o-----------X----
```

(b) Peres approach of modular increment: looks more complicated than the basic circuit, however, all gates are operating on only one or two qubits.

```
1 :---H--G1----G2-------G3----------G4---------------------------G10-G9-G8-H--
         |     |        |                                         |  |  |
2 :------o--H- | -G1--- | -G2----------G5----------------G9-G8-H- |  | -o-----
               |  |     |  |                             |  |     |  |
3 :------------o--o--H- |  | -G1----------G6-------G8-H- | -o---- | -o--------
                        |  |  |                    |     |        |
4 :---------------------o--o--o--H-----------G7-H--o-----o--------o-----------
```

(c) QFT approach of modular increment: offers the simplest implementation, although looks the most complicated.

Figure 4. Three examples of evaluating the modular increment of a number, encoded in 4 qubits.

D. *Other quantum libraries*

The list of useful libraries is unlimited. Think of circuits to generate a Quantum Fourier Transform, or to step through a quantum random walk. The same applies for the best way to deal with error corrections, to deal with the topology limitations of a particular quantum computer, or to build standard circuits by using only the native gates of a particular quantum computer (those that can be made with a single pulse). Note that implementations of basic gates like X, Y, Z, may require more than one pulse (this depends on the physical implementation being used). Existing tools have some of those algorithms implemented. However, a common library that can be used by different quantum tools is currently only in an early phase of development.

VI. Summary and conclusions

This paper identified what functionality is needed in software that enables the next level of quantum computing and proposes a way to implement this throughout the software stack. This next level quantum computing makes it possible to run more complicated algorithms on quantum computers in the cloud by a larger public.

The needed functionalities were categorised in functionalities for generic gates, for quantum debugging capabilities and quantum libraries.

A layered quantum software stack was discussed with extra attention to the functionality of the intermediate layers. Important is a clear separation between the software that runs locally and software that runs on a remote host computer that controls a quantum processor.

Looking at the layered stack, the lower layers contain one or more quantum processors and/or back-end simulators and are typically at the remote host. The intermediate and higher layers are typically running locally. This enables also a clear separation between the (classical) programming languages being used for the quantum application, and software that implement quantum-specific algorithms and quantum-specific debugging capabilities. The intermediate layers contain all kinds of libraries, and a local simulator to offer this to the higher layers.

The thoughts discussed in this paper are to provide input to a bigger research agenda on software development for quantum computing. A first step to make the desired functionality happen is increasing the effort on software development for the intermediate layers as well. Activities that deserve more attention are: (a) Interfacing between a local computer and quantum processor at a remote host. This should not only be defined in a language-independent manner, but also be defined for different quantum processors (at different remote hosts) in a common manner. (b) Collecting a wide variety of quantum circuit algorithms into libraries, in a uniform manner that can be used on any quantum processor. This may require an automated translation from an abstract QASM syntax tree (generated by the libraries) into the various QASM dialects of different processors.

This paper has shown a few examples of those libraries. All kinds of good algorithms are scattered around in literature, and paying more attention on bringing them all together into common libraries is a good start.

References

[1] G. W. Dueck, A. Pathak, M. M. Rahman, A. Shukla, and A. Banerjee, “Optimization of circuits for ibm’s five-qubit quantum computers,” in 2018 21st Euromicro Conference on Digital System Design (DSD). IEEE, 2018, pp. 680–684.

[2] X. Fu et al., “A microarchitecture for a superconducting quantum processor,” IEEE Micro, vol. 38, no. 3, 2018, pp. 40–47.

[3] S. Bettelli, T. Calarco, and L. Serafini, “Toward an architecture for quantum programming,” The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics, vol. 25, no. 2, 2003, pp. 181–200.

[4] P. Selinger, “A brief survey of quantum programming languages,” in International Symposium on Functional and Logic Programming. Springer, 2004, pp. 1–6.

[5] R. R. Tucci, “A rudimentary quantum compiler (2cnd ed.),” arXiv preprint quant-ph/9902062, 1999.

[6] P. J. Coles et al., “Quantum algorithm implementations for beginners,” arXiv preprint arXiv:1804.03719, 2018.

[7] R. LaRose, “Overview and comparison of gate level quantum software platforms,” arXiv preprint arXiv:1807.02500, 2018.

[8] “Quantum Computing Report,” [Online] URL: https://quantumcomputingreport.com/resources/tools/ [accessed: 2019-03-04].

[9] “Quantiki,” [Online] URL: https://www.quantiki.org/wiki/list-qc-simulators [accessed: 2019-03-04].

[10] “Scaffold,” [Online] URL: https://github.com/epiqc/ScaffCC [accessed: 2019-03-04].

[11] “Liquid,” [Online] URL: https://www.microsoft.com/en-us/quantum/development-kit [accessed: 2019-03-04].

[12] “Quipper,” [Online] URL: https://www.mathstat.dal.ca/~selinger/quipper/ [accessed: 2019-03-04].

[13] “QCL,” [Online] URL: http://tph.tuwien.ac.at/~oemer/qcl.html [accessed: 2019-03-04].

[14] “Quantum Inspire,” [Online] URL: https://www.quantum-inspire.com [accessed: 2019-03-04].

[15] “Atos,” [Online] URL: https://atos.net/en/insights-and-innovation/quantum-computing/atos-quantum [accessed: 2019-03-04].

[16] “Rigetti,” [Online] URL: https://www.rigetti.com/products [accessed: 2019-03-04].

[17] “IBM,” [Online] URL: https://quantumexperience.ng.bluemix.net/qx/experience [accessed: 2019-03-04].

[18] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, “Open quantum assembly language,” arXiv preprint arXiv:1707.03429, 2017.

[19] Koen Bertels. From qubits to a quantum computer architecture. Intel on the First International Workshop on Quantum Computer Architecture. [Online]. Available: http://www.ce.ewi.tudelft.nl/fileadmin/ce/files/quantum-computer-architecture/QCA_2017_-_Koen_Bertels.pdf

[20] C. G. Almudever et al., “Towards a scalable quantum computer,” in 2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS). IEEE, 2018, pp. 1–1.

[21] P. B. Sousa and R. V. Ramos, “Universal quantum circuit for n-qubit quantum gate: A programmable quantum gate,” arXiv preprint quant-ph/0602174, 2006.

[22] J. J. Vartiainen, M. Möttönen, and M. M. Salomaa, “Efficient decomposition of quantum gates,” Physical review letters, vol. 92, no. 17, 2004, p. 177902.

[23] M. Szyprowski and P. Kerntopf, “Low quantum cost realization of generalized peres and toffoli gates with multiple-control signals,” in Nanotechnology (IEEE-NANO), 2013 13th IEEE Conference on. IEEE, 2013, pp. 802–807.

[24] L. Ruiz-Perez and J. C. Garcia-Escartin, “Quantum arithmetic with the quantum fourier transform,” Quantum Information Processing, vol. 16, no. 6, 2017, p. 152.