Proceedings of the MycroftFest Symposium

Celebrating the career of Professor Alan Mycroft

December 1st 2023

UNIVERSITY OF CAMBRIDGE

Editors: Dominic Orchard, Tomas Petricek, Jeremy Singer

Introduction

If "academia is a pie eating contest where the prize is more pie", then what better way to celebrate the distinguished career of a retiring professor than by giving talks and writing papers! Thus, in Autumn 2023, with Professor Alan Mycroft formally retiring from the University of Cambridge, we thought it fitting to celebrate his career with a one-day symposium and a formal Festschrift publication to appear in the new year.

This informal proceedings gathers the abstracts of contributed talks which capture the wide-ranging nature of Alan's career, interests, and expertise, as well as the scope of his influence through a large number of collaborators, students, and their descendants. Included are a number of draft papers associated with some of the talks.

Alan is well-known for pioneering contributions to programming language theory and applications, covering both design and implementation. His work ranges from compilation and optimization techniques, through semantics, static analysis, and type systems, to parallel, concurrent and dataflow programming.

Following a BA in Mathematics at the University of Cambridge (1977), Alan completed the Diploma in Computer Science (1978). He then moved to Edinburgh for a PhD on the topic of "Abstract Interpretation and Optimising Transformations for Applicative Programs"[1] which was completed in 1981 under the supervision of Rod Burstall and Robin Milner. After an EPSRC post-doctoral fellowship at Edinburgh from 1981-83 and a stint as a *Forskarassistent* ('Research Assistant') at Chalmers in Gothenburg, Sweden, Alan returned to the Computer Laboratory at Cambridge in 1984. He has remained here ever since, metamorphosing from Lecturer to Senior Lecturer, then Reader in Programming Language Implementation,[2] before becoming Professor of Computing in 2004.[3] Since 1987 he has also been a fellow at Robinson College. Along the way he co-created the Norcroft C compiler, co-authored the book *Java 8 in Action: Lambdas, streams, and functional-style programming*, and co-founded the Raspberry Pi Foundation, helping to "put the fun back into learning computing" and encouraging programming back into the homes of families worldwide.

To those who have worked with Alan, or have been taught by him, he is renowned for his unstoppable enthusiasm about any challenging problem related to programming and computers, making him an inspiring teacher, mentor, and collaborator.

Thank you Alan for your enthusiasm, encouragement, and much scribbled feedback in red pen.

Dominic Orchard
Tomas Petricek
Jeremy Singer

[1] `https://era.ed.ac.uk/handle/1842/6602`
[2] `https://www.admin.cam.ac.uk/reporter/2001-02/weekly/5877/5.html`
[3] `https://www.admin.cam.ac.uk/reporter/2004-05/weekly/5992/12.html`

Contents

Talk abstracts

On the NorCroft Compiler

Jeremy Singer, University of Glasgow

Back in the mid 1980s, a 'couple of academics wearing startup-company hats' were developing a retargetable C compiler. The most prolific target for this compiler was the Arm processor, newly designed by Acorn. In this talk, we will review 1980s compiler technology and see how the NorCroft compiler significantly advanced the state of the art, particularly highlighting the friendly error messages and graph-colouring register allocation. We will conclude by considering the legacy of the NorCroft compiler.

Static Analysis for Hardware Design

Mads Rosendahl, Roskilde University
Maja Kirkeby, Roskilde University

Implementing algorithms in hardware can be a substantial engineering challenge. Hardware accelerators for some algorithms may be a way to achieve better time and energy efficiency of the computational problems. We explore some possible applications of static analysis in the design phase of construction hardware design for algorithms targeting field-programmable gate arrays (FPGA).

Drawing inspiration from Alan Mycroft's 2007 invited talk on static analysis and subsequent articles discussing the connection between hardware evolution, language design, and static analysis, we explore the usage of static analysis as a tool to facilitate the realization of hardware accelerators for algorithms. We examine methodologies for analyzing communication and data flows within the hardware design, thereby enhancing our understanding of these aspects in the pursuit of efficient FPGA-based algorithm implementations.

When Obfuscations Preserve Constant-Time

Matteo Busi, Ca' Foscari University of Venice
Pierpaolo Degano, Dipartimento di Informatica - Universita' di Pisa
Letterio Galletta, IMT School for Advanced Studies Lucca

Obfuscating compilers are designed to protect a program by obscuring its meaning and impeding the reconstruction of its original source code. Usually, the main concern with such compilers is their robustness against reverse engineering. On the contrary, little attention is paid to ensure that obfuscation introduces no attacks in the transformed program that where not present in the original one — in the style of secure compilation.

We are interested in checking whether a given obfuscation technique preserves the constant-time property. Cryptographic libraries often resort to this property to guarantee that no attackers can learn any secret values by monitoring and analysing program execution time.

Here, we propose a sufficient condition to check if a given obfuscation preserves constant-time. Checking this condition amounts to a simple and efficient static analysis that can be easily implemented.

We consider several obfuscating transformations implemented in popular obfuscating compilers (e.g. the `Tigress` C compiler and `O-MVLL`). By relying on our condition we prove that some of them preserve constant-time, while others do not. When a transformation breaks constant-time, we propose a translation validation that applies our condition case by case.

No Need to Imply Anything

Paulo Torrens, University of Kent

The need to reason about source programs has led to the development of intermediate representation languages whose semantics are well-suited for analyises and optimization techniques employed within compilers. The monadic nature of such languages traces back to the double-negation translation found in logic, where the use of continuations not only expose details about a program's control flow, but it also allows for a very natural imperative interpretation of functional programs by dropping the basic notion of a function, seeing lambda terms through the lenses of labels and jumps. In this presentation we aim to cover some ground into this imperative interpretation of functional calculi, such as the case of Thielecke's CPS-calculus, arising both from practical interest within compiler development and from a theoretical viewpoint, as these languages correspond to systems of logic that reject implication as a primitive and take negation as a more basic notion instead.

The Contributions of Alan Mycroft to Abstract Interpretation

Patrick Cousot, Courant Institute of Mathematical Sciences, New York University

We briefly summarize the contributions of Alan Mycroft to abstract interpretation which he pioneered by inventing strictness analysis of higher-order functional programming languages in the early 80's. His work originated a lot of research on strictness analysis and more generally the static analysis of functional programs, a crucial contribution to the theory, diffusion, and application of abstract interpretation.

Comonadic notions of computation revisited

Tarmo Uustalu, Reykjavik University

Comonadic and graded comonadic notions of computation became the subject of the work of Alan and his students on static analysis with type-and-coeffect systems.

I want to revisit the issue that it only makes good sense to pair contextual values (like values with their pasts in dataflow computation) if they are of the same shape. How to control shapes? We considered one solution in our CMCS 2008 paper. What can we say in 2023?

How to construct graded monads

Dylan McDermott, Reykjavik University

Models of computational effects based on graded monads are a useful tool for reasoning about programs. Each grade can be viewed as an abstraction of a set of computations that we can reason about, a perspective that is not available for models based on (ungraded) monads. Given this perspective, it is natural to wonder whether we can construct graded monads from monads by choosing sets of computations for grades to represent. We show that this is usually the case. If the chosen sets of computations satisfy some closure conditions, we can extract a graded monad almost for free. It turns out that many of the graded monads appearing in the literature can be seen as instances of this construction. We obtain a technique for constructing graded models from ungraded models, in such a way that the graded model can actually be used for program reasoning.

A Tale of Two Graded Calculi: The Marriage of Coeffects and Graded Comonads

Vilem Liepelt, University of Kent
Daniel Marshall, University of Kent
Dominic Orchard, University of Kent and University of Cambridge

The notion of graded types is an overarching paradigm for type systems that embed additional information for reasoning about the underlying structure of programs. Examples include a wide range of effect systems and coeffect systems, which capture respectively how a program changes its context or depends upon it. In the literature, two styles of coeffect system have emerged in the last decade: those in which coeffects annotations are pervasive (requiring annotations on function types), and those in which coeffects are added by way of a graded modal type operator atop some existing base language. In this work, we show how the two styles of coeffect systems relate and in what circumstances they have equivalent power. This parallels the two styles of effect reasoning in the literature: effect systems or (graded) monads, which have also been studied and found to be equivalent. Our work thus serves to unify the literature on coeffect systems to enable transfer of results and ideas in the future.

Linearity, Uniqueness, Ownership: An Entente Cordiale

Daniel Marshall, University of Kent

Ten years ago, Mycroft and Voigt surveyed notions of aliasing and ownership in programming languages, and argued for the view that many existing forms of aliasing control represent limited facets of the higher-level structured view corresponding to ownership. More recently, concepts of ownership and borrowing have become relevant to practical programmers as well as researchers through the medium of Rust, which aims to enforce safe memory management while bringing some of the guarantees offered by pure functional programming into the realm of performant systems code. Models like RustBelt and Oxide aim to formalise Rust's ownership system in detail, but there is less of a focus on integrating the basic ideas directly into more traditional functional type systems. We explain Granule's approach towards this, which builds on existing mechanisms for linearity and uniqueness through a form of grading akin to Boyland's fractional permissions, and discuss how this mechanism fits in to the framework presented by Mycroft and Voigt. We then look to the future, and briefly consider how a graded setting such as Granule's type system might be well suited for a more unified theory such as the one they hoped to uncover.

Sustainable software development - new challenges for programming, language design and program analysis.

Bent Thomsen, Aalborg University
Lone Leth Thomsen, Aalborg University
Thomas Bøgholm, Aalborg University

Energy consumption and the associated CO2 footprint of Information and Communication Technology (ICT) has become a major concern. Some estimates suggest that 4-6% of global energy consumption in 2020 was spent on ICT and, although the ICT industry is very good at using green energy, CO2 emissions from ICT are at par with CO2 emissions from Aviation. Pessimistic forecasts suggest that energy consumption from ICT may rise to 20% in 2030.

Clearly software as such does not emit CO2, but software is executed on hardware, and hardware consumes energy when executing software. In recent years there has been a huge effort in understanding the relationship between software and energy consumption of the underlying hardware. There is now evidence that the structure of the software, the program constructs used in the software and even the programming languages and compilers used for developing the software influence the energy consumption when the software is executed. There is a huge global effort on raising awareness of sustainable software development and there is a growing body of knowledge of many aspects.

However, the literature on how programming language design and analysis can impact energy consumption of the underlying hardware is sparse.

In a seminal presentation at SAS'2007 Alan gave an overview of the changes going on in hardware and outlined his view on the implications of this on programming language design and analysis research. In this paper we will try to follow in Alan's footsteps and outline our view on the implications of energy consumption on programming language design and analysis research.

A Symbolic Computing Perspective on Software Systems

Arthur Norman, Trinity College Cambridge
Stephen Watt, University of Waterloo

Symbolic mathematical computing systems have served as a canary in the coal mine of software systems for more than sixty years. They have introduced or have been early adopters of programming language ideas such ideas as dynamic memory management, arbitrary precision arithmetic and dependent types. These systems have the feature of being highly complex while at the same time operating in a domain where results are well-defined and clearly verifiable. These software systems span multiple layers of abstraction with concerns ranging from instruction scheduling and cache pressure up to algorithmic complexity of constructions in algebraic geometry. All of the major symbolic mathematical computing systems include low-level code for arithmetic, memory management and other primitives, a compiler or interpreter for a bespoke programming language, a library of high level mathematical algorithms, and some form of user interface. Each of these parts invokes multiple deep issues.

We present some lessons learned from this environment and free flowing opinions on topics including:

- Portability of software across architectures and decades,
- Infrastructure to embrace and infrastructure to avoid,
- Choosing base abstractions upon which to build,
- How to get the most out of a small code base,
- How developments in compilers both to optimise and to validate code have always been and remain of critical importance, with plenty of remaining challenges,
- The way in which individuals like Alan Mycroft who has been able to span from hand-crafting Z80 machine code up to the most abstruse high level code analysis techniques are needed, and
- Why it is important to teach full-stack thinking to the next generation.

Programming systems deserve a theory too!

Tomas Petricek, Charles University
Joel Jakubovic, University of Kent

Making programming easier and more accessible to non-experts has been a dream since the dawn of computing. In 1957, a catchy advertising brochure for FLOW-MATIC promised that the system will "virtually eliminate your coding load". We have certainly learned how to build more complex and reliable systems since then, but programming largely remains accessible only to experts and even a simple change requires navigating through hundreds of thousands of lines of code.

What would it take to make programming simpler and more accessible? In this somewhat philosophical talk, we will reflect on a number of issues that we encountered along the way when thinking about the problem. What would the notation for a simple programming system have to look like? Have visions around Free Software or Smalltalk taken us closer to this aim? And can we overcome Fred Brooks' tenet that "there is no silver bullet"?

The starting point for our investigation is the belief that we need to think less about programming languages, in which textual programs are written, and start to think about programming systems, interactive and stateful environments with which we interact when creating and using programs.

Air quality big data analytics using low-cost sensors

Eleftheria Katsiri, Democritus University of Thrace, Department of Electrical and Computer Engineering

Air pollution is the fourth most important risk of death factor in the world with a proven burden of disease that includes billions of deaths and thousands of DALYS. Furthermore, 4000 industries and 3000 diagnostic labs, in Greece only, are required by law to monitor air quality. Recently, both the covid-19 pandemic and climate change are changing current legislation worldwide, strengthening the need to monitor air quality.

On the other hand, the emergence of the low-cost sensor technology has changed the pollution monitoring paradigm, by enabling the monitoring of pollutants close to the source, with high temporal and spatial granularity. This make it possible to answer new questions about the underlying causes of poor air quality, ensure more accurate modelling and prediction at local scales, improve the ability to identify the links between air quality and human health or environmental degradation, identify potential air pollution "hot spots", enhance the ability to quantify the impacts of pollutant mitigation techniques and promote savings through on-demand ventilation. However, the exploitation of low-cost sensors requires in-depth knowledge of sensing principles, low-noise electronics, calibration in the lab and in he field, real-time edge processing and device-cloud communication, analytics and AI. Our team has been engaged in the development of reliable air quality sensing devices using low-cost sensors, custom sensor boards, embedded software and cloud services. The SibaIoT PM device used in this work measures ambient particulate matter concentrations in ug/m3 of three classes of particles, namely PM1.0, PM2.5, PM10, humidity and temperature. The device has very good accuracy, response time and sensitivity in indoor pollution levels.

With respect to the methodology, we have conducted a pilot application in a state-of-the art industrial space that is sensitive to infection caused by particulate matter such as dust. Fifteen PM devices were installed in three different production areas with varying air quality sensitivity. Indicative visual analytics are presented in the paper such as descriptive analytics, histograms, density, delay and line plots, as well as outcomes from the application of analytic functions such as Pearson correlation, k-means clustering, classification and aggregated exposure to pollution on the sensor data. More specifically, both Production-Area analytics, i.e., analysis of multiple time-series generated by multiple devices deployed at a specific production area and Intra-Area analytics, i.e., analysis of multiple time-series generated by multiple devices deployed at logically connected production areas, are discussed. For example, we have calculated annual max, min, average values per day-of-the-week, shift, day of the month, hour-of-the-day, poor, good and fair quality clusters.

Preliminary results show that the above analytics are promising in providing useful insights on the both the level of pollution and the intensity of industrial activity in the dairy. For example we have found that air quality is aggravated indoors during weekdays, that annual average particulate matter concentration follows humidity and temperature with a fixed lag.

Parallel Multiprecision Arithmetic the Easy Way

Cosmin Oancea, University of Copenhagen
Stephen Watt, University of Waterloo

In today's world, computation on multiple precision integers is a cornerstone of privacy and other applications. At the same time, every device, from telephones to desktop computers, has a GPU, often underutilized. Programming these using vendor libraries or low-level APIs is tedious and error prone and not at all suitable for writing high-level mathematical code.

We look at applying a functional data parallel language aimed at GPU computation to building big integer functions. These functions can be expressed in terms of natural arithmetic

operations that compose just as those using primitive types. We examine how functional primitives can be used to give elegant and efficient GPU arithmetic for multiple precision integers of the sizes needed for practical applications.

Computational thinking bridging the conceptual gap between idiosyncratic architectures and high-level abstractions is one of the hallmarks of Alan Mycroft's work from which we have taken inspiration.

axs: a workflow automation language for omni-benchmarking and optimization

Anton Lokhmotov, KRAI
Leo Gordon, KRAI
Alastair Donaldson, Imperial College London

We present axs (pronounced "access"), a workflow automation language developed at KRAI with the primary purpose of benchmarking and optimization of Computer Systems for Machine Learning applications. We discuss the most salient features of the language in the context of preparing highly competitive and fully compliant submissions to MLPerf, an industry competition often referred to as "The Olympics of ML Benchmarking". axs has enabled hundreds of such submissions from KRAI's partners including Qualcomm, HPE, Dell and Lenovo, sometimes helping them to win "Gold" and "Silver" medals against the entrenched competition represented by NVIDIA.

Triemaps that match

Simon Peyton Jones, Epic Games
Sebastian Graf, Karlsruhe Institute of Technology

The trie data structure is a good choice for finite maps whose keys are data structures (trees) rather than atomic values. But what if we want the keys to be *patterns*, each of which matches many lookup keys? Efficient matching of this kind is well studied in the theorem prover community, but much less so in the context of statically typed functional programming. Doing so yields an interesting new viewpoint — and a practically useful design pattern, with good runtime performance.

This paper grew out of work on compilers and functional programming, both of which are among Alan's long term interests

Draft papers

Sustainable software development:

new challenges for programming, languages design and analysis

(Preliminary version)

Bent Thomsen[1], Lone Leth Thomsen[1], and Thomas Bøgholm[1]

Aalborg University, Aalborg, Denmark
{bt, lone, boegholm}@cs.aau.dk

Abstract. Energy consumption and the associated CO2 footprint of Information and Communication Technology (ICT) has become a major concern. Some estimates suggest that 4-6% of global energy consumption in 2020 was spent on ICT and, although the ICT industry is very good at using green energy, CO2 emissions from ICT are at par with CO2 emissions from aviation. Pessimistic forecasts suggest that energy consumption from ICT may rise to 20% in 2030.
Clearly software does not emit CO2, but software is executed on hardware, and hardware consumes energy when executing software. In recent years there has been a huge effort in understanding the relationship between software and energy consumption of the underlying hardware. There is now evidence that the structure of the software, the program constructs used in the software and even the programming languages and compilers used for developing the software influence the energy consumption when the software is executed. There is a huge global effort on raising awareness of sustainable software development and there is a growing body of knowledge of many aspects.
However, the literature on how programming language design and analysis can impact energy consumption of the underlying hardware is sparse. In a seminal presentation at SAS'2007 Alan gave an overview of the changes going on in hardware and outlined his view on the implications of this on programming language design and analysis research. In this paper we follow in Alan's footsteps and outline our view on the implications of energy consumption on programming language design and analysis research.

1 Introduction

The World Economic Forum estimates that digital technologies can reduce global CO2 emissions by 15% in sectors such as energy, manufacturing, agriculture, construction and transport [1, 10]. Despite the fact that ICT often is able to exploit green energy, CO2 emissions from ICT are at par with CO2 emissions from aviation. Some pessimistic forecasts suggest that ICT in 2030 may consume as much as 20% of the global energy production [14].

Developers of ICT systems have a strong desire to help reduce global CO2 emissions in general and from ICT systems in particular. Tremendous advances

in hardware technology have to a large extent kept a lid on an explosion of energy consumption from ICT despite a rise in demand for computations. Considering the bleak forecast in [14] it seems that the energy consumption from the worlds data centers has been kept at around 2% despite at six fold rise in throughput since 2010.

The impressive achievements of hardware engineers have, however, not been matched by software engineers, who to a large degree, are unaware of the opportunities and possibilities that they have regarding reducing energy consumption caused by ICT. Software engineers are not entirely to blame for this situation as they have been shielded from the development in hardware that still, as Alan pointed out in his SAS'07 talk, tries to give the software engineers the illusion that hardware operates according to a computational model aligned more or less with the i486 from 1985. Even when software engineers understand that hardware has changed dramatically they often lack the tools to help them even address the issue [18].

From a research perspective, there has been a huge effort in understanding the relationship between software and energy consumption of the underlying hardware. There is now evidence that the structure of the software, program constructs used in the software and even the programming language and compiler used for developing the software influence the energy consumption when the software is executed. There are various tools able to identify energy hungry program patterns and replace these with less energy demanding patterns. There are also recommendations that software engineers integrate an energy testing practice into their software development process to identify energy hotspots as early as possible.

In this paper we will follow in Alan's footsteps and first outline changes in technology, then discuss programming implications (for programmers and for languages) and finally outline opportunities for static analysis and type systems, before listing our conclusions.

2 Changes in technology

As Alan explained in his SAS'07 talk [20], hardware has been on a tremendous journey from the invention of the integrated circuit in 1958 where a few transistors could be placed together to a situation today in 2023 where over 100 billion transistors can be fitted on a chip. Early CPU's had a few thousand transistors, e.g. in 1971 the Intel 4004 chip had 2300 transistors, in 2007 the Intel Itanium 2 dual core processors had more than 1.7 billion transistors, and in 2023 the Intel Core i9-11900K has 17 billion transistors and the Apple M1 Ultra has 114 billion transistors. The development has followed Moore's law: doubling the number of transistors per unit area in an integrated circuit every 18 months (or every two years), although a slight slowdown seems to have materialized in recent years since we have not quite reached the 256 billion transistors on a chip that Alan predicted!

The huge number of available transistors has implications for the architecture of CPUs. In the early years, the seventies to mid-eighties, more and more complex CPUs with more and more advanced instructions were constructed, however in the late seventies and early eighties the x86 instruction set emerged as a dominant player in the PC market. With shrinking feature size it was possible to increase clock speed and thus speed up execution time, leading to the misquoted version of Moore's law that processor speed doubles every 18 months.

Although feature size has gone down uniformly, the speed of circuits has not developed uniformly. In the early years there was a 1-1 relationship between execution of instructions on the CPU and memory access to RAM. However, from the mid-eighties this correspondence started to drift with RAM becoming relatively slower and slower. A large part of the increased number of transistors was in the period from 1985 to 2005 used to hide this fact by introducing on-chip caches, pipelines and super scalar processing, thus hiding the fact that the relative speed between CPU and RAM by 2005 was a factor 1-200. With the shrinking feature size comes an additional challenge, namely that accessing on chip memory is also getting slower as a round trip on a chip may take as much as 75 clock cycles. Therefore several layers of caches, referred to as L1, L2, L3 etc., were introduced - this means that a large part of the chip is used for memory.

One problem with the shrinking feature size is that the heat produced as a consequence of switching circuits gets concentrated on a smaller and smaller area. This puts a limit on how much the clock speed can be raised and as a consequence clock speeds since 2005 have been kept relatively stable around 2.5-3.5 GHz. Instead of a faster single core CPU a development towards multi-core architectures, starting with dual core CPUs in commercial use from 2005 onward, has been pursued. Extrapolating based on Moore's law we should by now have chips with more than a thousand cores and indeed there are experimental 1000 core CPUs and even commercial CPUs like the Arm-based CPU "Ampere Altra / Altra Max" with up to 256 cores.

However, mainstream computing has stopped at 4 to 12 cores on consumer platforms, with a few high-end server processors with 40 or more cores. Instead of adding more cores the focus has been on integrating the GPU and providing cores with different energy consumption characteristics.

Koomey's law, which states that the number of computations per joule of energy dissipated doubles about every 1.57 years, indicates that hardware is getting more energy efficient in general. However, this is a very broad trend with huge variations.

Already before the turn of the millennium it had become clear that the increase in processor speed comes with a higher energy cost. As reported in [12] the reason for the increase in power is that the design techniques used in the desktop microprocessors tended to result in much more energy being expended per instruction due to the higher capacitance toggled to process each instruction. [12] reports that the Pentium Pro processor was 1.8 times faster than the Pentium processor but consumed 3.3 times the power. The increase in

speed was entirely a result of the deeper pipeline on the Pentium Pro processor since the process technology was the same for both.

[12] compares energy per instruction (EPI) for a number of intel processors; i486, Pentium, Pentium Pro, two Pentium 4 variants (Willamette, Cedarmill), Pentium M and Core Duo (benchmarked using a single core). Although these processors are manufactured using different feature sizes, have different clock frequencies and different voltage, normalized to the i486 the EPI reported in nJ are: 10, 14, 24, 38, 48, 15 and 11. The energy efficiency of the mobile processors, Pentium M and the Core Duo, is a result of more modest pipeline depths, moderately-sized out-of-order structures, aggressive clock gating, and micro-op fusion. The Pentium M and Core Duo processors deliver higher performance by performing more useful work in each clock cycle. Similarly in 2020 AMD reported that, since 2014, the company has managed to improve the efficiency of its mobile processors by a factor of 31.7. Thus with the advent of mobile computing came a new focus on energy efficiency, which later proliferated to desktop and server hardware.

A simple way of saving energy is to reduce the voltage supplied to the CPU and as higher frequencies require higher voltage the concept of dynamic voltage and frequency scaling (DVFS) has been introduced, sometimes referred to as overvolting and undervolting or turbo boosting. With these techniques the CPU energy consumption, and heating, can be controlled to some extent from software by throttling down when e.g. conservation of battery is important or boosting when high demand jobs are executing.

With the advent of multicore architectures another power saving method has been introduced, where some cores are "turn off" or at least put into reduced power state when not needed. Power management states, P-states and C-states are mechanisms used in modern processors to control and optimize power consumption. P-states are also known as performance states. These are different operating states or frequency/voltage pairs that a processor can switch between to balance performance and power consumption. At higher P-states, the processor runs at higher clock frequencies and voltages, providing better performance but consuming more power. Lower P-states involve lower clock frequencies and voltages, reducing power consumption at the cost of lower processing performance. The operating system and the processor work together to adjust the processor's P-state dynamically based on the current workload. This allows the system to save power during periods of low activity and ramp up performance when needed. C-states are idle or sleep states that a processor enters when there is little to no computational load. These states help reduce power consumption when the CPU is not actively processing tasks. Each C-state corresponds to a different level of idle activity, with higher-numbered C-states indicating deeper levels of sleep and lower power usage. When a processor enters a C-state, it effectively powers down parts of its core or cores to minimize power usage. This can involve halting the clock or even turning off parts of the processor until activity resumes. As soon as the processor detects activity, it exits the C-state and returns to an active state to resume processing tasks. The combination of P-states

and C-states allows modern processors to adapt their power usage in real-time to match the system's workload, optimizing power efficiency while still providing adequate performance when required. This is crucial for laptops and mobile devices to extend battery life and for data centers to reduce energy consumption while maintaining responsiveness.

Another power saving approach was introduced in 2023 with Intel's new Alder Lake chips which come with two sets of CPU cores: P-cores and E-cores. P cores are designed for high-performance tasks. They typically have a higher clock speed and are optimized for single-threaded performance, making them well-suited for tasks that do not benefit from multiple cores. P cores use a more advanced process node and often have a higher power consumption compared to E cores. E cores are designed for power efficiency and less demanding tasks. They have a lower power consumption and are optimized for tasks that can benefit from multi-threading. E cores are typically used for background tasks, lightweight applications, and power-saving scenarios to extend battery life in laptops or reduce power consumption in desktops. Alder Lake processors use a combination of P and E cores to achieve a balance between high performance and power efficiency. This architecture is sometimes referred to as a "big.LITTLE" design, with P cores acting as the "big" high-performance cores and E cores serving as the "LITTLE" power-efficient cores. The operating system and software are responsible for scheduling tasks to run on the appropriate core based on their resource requirements.

Although P and and E cores in consumer PCs are mainly used for running different applications with different execution speed needs, it also makes sense to have different sized processors in a many-core architecture to improve parallel speedup by reducing the time it takes to run the less parallel code. As described in [2] an implication of Amdahl's law is that the less parallel portion of a program can limit performance on a parallel computer. Therefore it makes sense to execute the inherent sequential code of an application on a faster processor with larger caches, a bigger multiplier, deeper pipelines, even if this also implies more power consumption. [2] presents the following argument:

"For example, assume 10% of the time a program gets no speedup on a 100-processor computer. Suppose to run the sequential code twice as fast, a single processor would need 10 times as many resources as a simple core, the comparative speedups of a homogeneous 100 simple processor design and a heterogeneous 91 processor design relative to a single simple processor are: Speedup Homogeneous = 1 / (0.1 – 0.9/100) = 9.2 times faster. Speedup Heterogeneous = 1 / (0.1/2 – 0.9/90) = 16.7 times faster. In this example, even if a single larger processor needed 10 times as many resources to run twice as fast, it would be much more valuable than 10 smaller processors it replaces."

Some modern processors facilitate simultaneous multi-threading (SMT), called hyper threading on Intel processors. SMT allows a single physical processor core to simulate two or more logical cores. If a core encounters a stall or waits for data, it can switch to another thread that is ready to execute, thereby reducing idle time and improving overall CPU utilization.

Another parallelisation strategy is Single Instruction, Multiple Data (SIMD), sometimes refered to as vector operations or Streaming SIMD Extensions (SSE) instructions. SSE was first introduced with the Intel Pentium II processor, but they are now found on all modern x86 processors, and are the default floating point interface in 64-bit mode.

To make use of SIMD instructions, software developers often rely on compilers to automatically vectorize their code. The compiler identifies opportunities to apply SIMD instructions and generates the appropriate machine code to take advantage of parallelism. Proper data alignment is crucial when using SIMD instructions. Data elements in SIMD vectors must be aligned to fit the specified width of the SIMD registers. Misaligned data can result in performance penalties.

SIMD instruction were partially introduced to support game programming. Another trend coming from game programming is the General-Purpose Graphics Processing Unit (GPGPU) using the graphics processing units (GPUs) for computational tasks beyond traditional graphics rendering. While GPUs were originally developed to handle graphics-related calculations (such as rendering images, videos, and 3D graphics), their highly parallel architecture has proven valuable for a wide range of general-purpose computational tasks. Modern GPUs are highly parallel processors with hundreds or thousands of cores compared to the relatively fewer cores in CPUs. GPUs are optimized for handling large amounts of data and performing computations on this data quickly due to their high memory bandwidth and parallel processing capabilities. For certain workloads, GPGPUs can be more energy-efficient than CPUs, as they can perform a large number of computations in parallel, potentially saving time and power. Moving data to the GPU can take significant time, but co-locating the GPU and the CPU can significantly reduce this time.

With modern multi-core processors determining the EPI is more difficult. [26] presents an instruction-level energy model for the Intel Xeon Phi processor, identifying how energy per instruction scales with the number of cores, the number of active threads per core, and instruction types and modes of data operand access. Compared to [12] it is now clear that instructions have different energy costs. For example [26] reports that scalar operations with operands in registers cost 0.45 nJ, whereas scalar operations on data in L1, L2 or in memory costs; 0.88 nJ, 7.72 nJ, 52.14 nJ (with prefetch) and 232.62 nJ (without prefetch). For vector instructions the EPI with register operands is 1.00 nJ while the EPI of moving data from memory to the ALU without prefetching is 233.17 nJ.

3 Programming Implications (for Programmers and for Languages)

In his SAS'07 talk Alan identified several implications of the change in hardware for programmers and programming languages. First and foremost he identified that the programmers view of memory and computations being in a 1-1 relationship was no longer accurate. Therefore programmers should write code that as far as possible uses local data or consider re-computing as "Wires are no longer

free, and local re-computation is far better than sharing computation with a distant place".

The implication of this is that the programming model underpinning imperative programming languages like C is no longer accurate and Alan calls for research on programming languages to support non-uniform memory and designs which de-emphasise shared RAM and serialised access.

Alan suggested that "Pointers (should be) considered harmful", at least unrestricted pointers as known from C, Java or ML, because pointer copying is more expensive than integer copying and because pointer copying create aliasing. Alan also suggested that OO-languages (C++ and Java) should be "considered harmful". He highlights the problem with various parameter mechanisms, by value and by reference, whereby the programmer at an early stage of development is forced to freeze constraints about whether a function shares a data address space with its caller. Another concern with OO is that proper OO design fosters a memory layout of data where it is very difficult to achieve locality - just think of how many pointers have to be followed to random places in memory when traversing an AST in a compiler designed following best OO practice.

Clearly functional languages are safe(r) because the compiler can choose to copy or alias data in pure functional languages. But ML has references that cannot be copied and Haskell's laziness may imply that coping becomes rework! Even pure data poses the question of when to distribute and when not?

If we look at what happened in programming language evolution since 2007, functional languages, especially Haskell, but also ML (especially the O'caml variant) had a surge in popularity, in particular in the FinTech industry. However, these languages are still not in widespread use. Instead the major OO languages (C++, C# and Java) were extended with first class functions, also known as lambdas, albeit with various restrictions. The introduction of lambdas exposed mainstream programmers to functional programming concepts such as higher order function and immutable data. A new breed of functional OO languages, like Scala, F#, Kotlin and Swift, have also emerged. However, none of these languages have eliminated the "pointer" problem, although they encourage a value oriented functional programming style. In particular Scala has been popular in Big Data analytics due to its adoption in Spark/Hadoop, which is based on the map/reduce functional idiom.

Clearly multi-core chips brings the possibility of parallel programming into mainstream. However, as Alan predicted, the new model (in 2007) of shared memory multi-core computing would scale to 2, 4 and maybe 8 cores, but questioned that it would scale to 1024, simply because shared memory would become a bottleneck and with shared memory comes the need for locking mechanisms, and as Alan said "Locks considered harmful" and needs to be discouraged along with programming mechanisms which stress it without user awareness. E.g. synchronized objects in Java.

Around 2010 the idea of software transactional memomry (STM) became a hot topic and it was seen as a solution to the problem of getting mainstream programmers to deal with parallel execution and locking on shared memory ar-

chitectures. Languages like Haskell and Scala were extended with STM libraries and the emergence of the Closure language, a LISP variant, designed around STM. However, it turns out that programmers find STM just as difficult as locking. It also turns out that STM performance degrades with the number of cores, as the likelihood of conflicting transactions raises with the number of truly parallel tasks executing.

Instead of a shared memory parallel programming model, Alan suggests a move to a message-passing programming model, as the Actor model has disjoint memory spaces per process. He reminds us of the Needham-Lauer "Duality of operating structures" which showed that shared memory and message-passing are duals and observes that message passing and shared memory are directly equivalent with linear uses of buffers.

Alan mentions the Occam language which has a message passing model based on the CSP calculus. The Occam programming model was designed to be very close to the system architecture of the Transputer. In recent years the Erlang language, based on the Actor model, has risen in popularity, in part due to its use in the popular WattsApp, and also the GO system programming language from Google has a message passing model based on CSP.

In 2004 [4] Microsoft Research presented Polyphonic C# with synchronous and asynchronous methods, based on ideas from the join-calculus. A restricted version of asynchronous methods from polyphonic C# was adopted in the C# language in the form of async/await constructs leading to a concurrent programming style based on futures and promises. This programming style is now in widespread use in many modern programming languages, replacing the need for explicit programming with threads and locks.

The rise of the GPGPU led to a flurry of extensions and libraries in high level languages, like Scala, but these extensions were not widely adopted. Instead the proprietary language CUDA from NVIDIA and the OpenMP API came to dominate GPGPU programming. The rise of AI is to a large extent powered by GPGPU, but the programming is mainly reserved for specialist programmers writing powerful libraries in C/C++ and CUDA/OpenMP with such libraries being accessed by mainstream programmers through the use of Python libraries such as TesorFlow, NumPy and SciPy.

So we are still lacking a programming model and programming languages supporting such a model that align better with modern hardware architectures.

Equally challenging, or perhaps closely related, is the lack of programming and programming language support for programmers wishing to write software that minimizes energy consumption by the underlying hardware (but still achieve needed functional and performance goals).

There is a growing number of studies of program structures, program patterns, program constructs and even the programming language and compiler used for developing the software showing how these subjects influence the energy consumption when the software is executed. The majority of these studies are experimental measuring energy consumption of computers while running benchmarks illuminating a given subject. There are basically two ways of mea-

suring energy consumption from running software, either by an external device, often referred to as a wall-plug measurements, or through on chip sensors using the Running Average Power Limit (RAPL) interfaces, supported on Intel processors, for reporting the accumulated energy consumption of various power domains (for example, PP01 or Package). Wall-plug measurements are usually sampled once every second, whereas RAPL updates the energy counters approximately once every millisecond. Studies show that wall-plug measurements and RAPL measurements are well aligned [15].

An example of such as study is [16] which for Java code shows (not surprisingly) that primitive data types are more energy-efficient than wrapper classes, static variables (somewhat surprisingly) causing a 50% increase in energy consumption compared to local variables, static variables causing 60% increase in execution time compared to local variables. The study also shows that short circuit operators have better energy efficiency if the first case is the most common one, string concatenation being more energy efficient than StringBuilder and StringBuffer append methods, method calls in statements like for loops do not always have increased overhead, try-catch blocks have no cost if no exceptions are thrown. Another study, [7], showed that inheritance proved to be more energy efficient than delegation, with a reduction in run time of 77% and a reduction in average power consumption of 4%. [25] presents at tool to analyse Java collections and replace collections by others with a positive impact on the energy consumption as well as on the execution time. Using jStanley [25] shows energy gains between 2% and 17%, and a reduction in execution time between 2% and 13%.

There are similar studies for constructs in C++, C# and Haskell. There are studies that compare various compilers and compiler options, studies comparing various implementations of the JVM and studies comparing concurrency constructs.

Studies show that it can be of great importance in which programming language code is written, i.e. the same logic, with the same workload, can be more or less power-intensive [8, 24]. Overall, these studies show that code written in C is the most energy efficient, that code in Java is approximately 2 times and code written in C# is more than 3 times more power intensive than C. Code written in Ruby is 45 times more power intensive than C. At the same time, some of these studies show that there is not always a correlation between speed and power consumption, i.e. sometimes fast code is more power consuming than slow code. The results reported in [8, 24] have been used to argue that in the future all code should be written in the RUST programming language [19] and that code written in Python "is Destroying the Planet" [3]. Before drawing such dire conclusions one should consider that the benchmarks used in [8, 24] are all rather small, micro benchmark like, and often well suited for compiled C-like languages.

The majority of studies are not looking into the causes of their results. One notable exception is [26] which show that on Linpack, across different input sizes,

around 10% of the energy is spent on redundant software prefetch operations that fetch data already in the target cache.

All in all the current state of sustainable software development is characterized by a growing number of experiments illuminating various aspects of programming and programming languages suggesting that programmers should be careful in their programming practice and integrate energy consumption testing in the continuous integration practice.

There are few, if any, suggestions to align programming practice or programming languages more closely to the hardware.

One possible exception is the emerging new programming paradigm: data oriented programming (DOP), which is an approach to software development that prioritizes the efficient organization and manipulation of data to improve performance and scalability [27]. It focuses on structuring and processing data in a way that maximizes system throughput and minimizes memory access bottlenecks. DOP emphasizes designing software around how data is accessed and manipulated rather than focusing solely on object-oriented or procedural programming paradigms. It aims to optimize data layout and processing to improve performance. In DOP specialized data structures and layouts that enhance memory access patterns, minimize cache misses, and facilitate efficient processing of large volumes of data, are used. Often parallel processing capabilities (multithreading) and Single Instruction, Multiple Data (SIMD) operations are used to leverage the full potential of modern hardware architectures. DOP is commonly associated with game development due to the need for real-time performance. However, its principles are also applicable to various high-performance computing scenarios, such as scientific simulations, data processing, and more.

4 Opportunities for Static Analysis and Type Systems

Although Alan in his SAS'07 talk mainly focused on programming language design and type systems, there may be some interesting work to be done on programming analysis for energy efficiency. Most compiler optimizations are targeting speed-up and clearly there is a correlation between speed and energy consumption. However, as shown in [26], there may be up to 10% energy savings by removing prefetch instructions inserted to gain speed. Also, as Alan pointed out, it may be cheaper to recompute some expression rather that fetch their results from RAM, thus eliminating the need for the common sub-expression elimination optimization found in many compilers. [24] shows that often the most energy efficient implementation is not the fastest implementation. Hence there is a need for tools that can help find the best compromise between speed and energy.

In general there is a lack of tools to (statically) analyse programs in high level languages regarding their potential energy consumption. A few tools are based on linter principles, identifying syntactic patterns and suggesting replacements. [22] shows that recommendations for exchange of Java collections are very hardware

dependent. Another study shows that sometimes two energy optimizations may counteract each other and therefore lead to higher energy consumption [9].

In his SAS'07 talk Alan presented ideas for "Structured programming for pointers". His idea of introducing a call-by-either-value-or-reference parameter mechanism and checking if a function can tell the difference is essentially linear or quasi-linear types. He also suggest that ownership types from the OO community may be useful. He suggested that pointers are extended to know which memory region they apply to, essentially regions as introduced by Tofte and Talpin [30]. However, Alan also questioned whether such mechanisms will be lightweight enough for ordinary programmers.

An attempt to introduce regions in a C like language was presented in the Cyclone language [13]. Unfortunately, the development of Cyclone has been discontinued. The idea of ownership types has to some extend been introduced in the RUST programming language with its focus on reference lifetimes and borrowing. Quasi-linear types have been applied in attempts to formalize the RUST type system [23]. It seems that (some) programmers actually find these types understandable enough.

[6] introduces a type system for energy management based on the observation that energy management is often based on discrete phases and modes. A phase characterizes a distinct pattern of program workload, and a mode represents an energy state the program is expected to execute in. This is akin to modes in real-time systems [28] which has been used to handle energy management programmatically in the Safety Critical Java profile allowing the program to manage DFVS [17]. To take the dynamic behaviour of the program including parallel tasks into account, it may be worth revisiting [21, 29] which extend region types with processes describing the dynamic communication behaviour of the underlying CML, respectively Facile, programs. Akin to this work, a huge body of work on behavioural types exists [11]. We imagine that our joint work with Alan on schedulability abstractions [5] could be extended to take energy consumption into account using prised timed-automata. One could even imagine such energy abstractions be used to inform the OS and hardware in advance so schedules for e.g. migration of tasks could be planned to minimize heat from a particular core or movement to a location able to utilize green electricity.

5 Conclusion

The development of processor architectures towards more energy efficient computing is impressive, however, the development has not been matched by developments in software, although software engineers are keen to play their part in reducing energy consumption, and thereby CO2 emission, from the global ICT infrastructure. Huge potentials are being demonstrated by a growing list of experiments showing examples of how energy consumption can be reduced by changing the structure of software, e.g. more or less energy demanding program patterns, by choosing the most energy efficient data structures and algorithms, by using the most energy efficient program constructs and even by choosing the

most energy efficient programming languages and compilers. However, inspired by Alan's thoughts there are ample opportunities to make these efforts easier and more accessible to software engineers by designing new programming languages to better reflect new hardware, new type systems and new programming analysis taking the complex hardware infrastructure more into account. Clearly it would also be interesting to see innovative hardware ways to use all the extra transistors to make programming closer to the hardware easier for software developers. We may, just as Alan did in his SAS'07 talk, conclude that "what is a computer" is now changing quite quickly and this gives opportunities for new research areas.

References

1. kommisionen 4.0, S.: Digitalisering af klimakampen, https://ida.dk/om-ida/siri-kommissionen/digitalisering-af-klimakampen
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al.: The landscape of parallel computing research: A view from berkeley (2006)
3. Ayar, M.: Python is destroying the planet, https://levelup.gitconnected.com/python-is-destroying-the-planet-951e83f22748
4. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for c#. ACM Transactions on Programming Languages and Systems (TOPLAS) **26**(5), 769–804 (2004)
5. Bogholm, T., Thomsen, B., Larsen, K.G., Mycroft, A.: Schedulability analysis abstractions for safety critical java. In: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. pp. 71–78. IEEE (2012)
6. Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 831–850 (2012)
7. Connolly Bree, D., Cinnéide, M.Ó.: Inheritance versus delegation: which is more energy efficient? In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 323–329 (2020)
8. Couto, M., Pereira, R., Ribeiro, F., Rua, R., Saraiva, J.: Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian Symposium on Programming Languages. pp. 1–8 (2017)
9. Couto, M., Saraiva, J., Fernandes, J.P.: Energy refactorings for android in the large and in the wild. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 217–228. IEEE (2020)
10. Forum, W.E.: Digital technology can cut global emissions by 15
11. Gay, S., Vasconcelos, V.T., Wadler, P., Yoshida, N.: Theory and applications of behavioural types (dagstuhl seminar 17051). In: Dagstuhl Reports. vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
12. Grochowski, E., Annavaram, M.: Energy per instruction trends in intel microprocessors. Technology@ Intel Magazine **4**(3), 1–8 (2006)
13. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 282–293 (2002)

14. Jones, N., et al.: How to stop data centres from gobbling up the world's electricity. Nature **561**(7722), 163–166 (2018)
15. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: Rapl in action: Experiences in using rapl for power measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) **3**(2), 1–26 (2018)
16. Kumar, M., Li, Y., Shi, W.: Energy consumption in java: An early experience. In: 2017 Eighth International Green and Sustainable Computing Conference (IGSC). pp. 1–8. IEEE (2017)
17. Luckow, K.S., Bøgholm, T., Thomsen, B.: Supporting development of energy-optimised java real-time systems using tetasarts (2013)
18. Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J.: An empirical study of practitioners' perspectives on green software engineering. In: Proceedings of the 38th international conference on software engineering. pp. 237–248 (2016)
19. Miller, S., Lerche, C.: Sustainability with rust, https://aws.amazon.com/cn/blogs/opensource/sustainability-with-rust/
20. Mycroft, A.: Programming language design and analysis motivated by hardware evolution: (invited presentation). In: International Static Analysis Symposium. pp. 18–33. Springer (2007)
21. Nielson, F., Nielson, H.R.: From cml to process algebras. In: International Conference on Concurrency Theory. pp. 493–508. Springer (1993)
22. Oliveira, W., Oliveira, R., Castor, F., Pinto, G., Fernandes, J.P.: Improving energy-efficiency by recommending java collections. Empirical Software Engineering **26**, 1–45 (2021)
23. Pearce, D.J.: A lightweight formalism for reference lifetimes and borrowing in rust. ACM Transactions on Programming Languages and Systems (TOPLAS) **43**(1), 1–73 (2021)
24. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Ranking programming languages by energy efficiency. Science of Computer Programming **205**, 102609 (2021)
25. Pereira, R., Simão, P., Cunha, J., Saraiva, J.: jstanley: Placing a green thumb on java collections. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 856–859 (2018)
26. Shao, Y.S., Brooks, D.: Energy characterization and instruction-level energy model of intel's xeon phi processor. In: International Symposium on Low Power Electronics and Design (ISLPED). pp. 389–394. IEEE (2013)
27. Sharvit, Y.: Data-oriented Programming: Reduce Software Complexity. Simon and Schuster (2022)
28. Søndergaard, H., Ravn, A.P., Thomsen, B., Scoeberl, M.: A practical approach to mode change in real-time systems. Tech. rep., Technical Report 08-001, Department of Computer Science, Aalborg University (2008)
29. Thomsen, B.: Polymorphic sorts and types for concurrent functional programs. Techn. Rep. ECRC-93-10 (1993)
30. Tofte, M., Talpin, J.P.: Region-based memory management. Information and computation **132**(2), 109–176 (1997)

Air quality big data analytics using low-cost sensors

Eleftheria Katsiri

Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi 67100, Greece,
ekatsiri@gmail.com

Abstract. Air pollution is the fourth most important risk of death factor in the world with a proven burden of disease that includes billions of deaths and thousands of DALYS [1–4]. Furthermore, 4000 industries and 3000 diagnostic labs, in Greece only, are required by law to monitor air quality. Recently, both the covid-19 pandemic and climate change are changing current legislation worldwide, introducing a need to monitor more closely air quality.

On the other hand, the emergence of the low-cost sensor technology has changed the pollution monitoring paradigm [8–15], by enabling the monitoring of pollutants close to the source, with high temporal and spatial granularity. These features makes it possible to answer new questions about the underlying causes of poor air quality, ensure more accurate modelling and prediction at local scales [6, 16], improve the ability to identify the links between air quality and human health [5, 17, 18] or environmental degradation [19], identify potential air pollution "hot spots", enhance the ability to quantify the impacts of pollutant mitigation techniques and promote savings through on-demand ventilation [7]. However, the exploitation of low-cost sensors requires in-depth knowledge of sensing principles, low-noise electronics, calibration in the lab and in he field, real-time edge processing and device-cloud communication, analytics and AI.

Our team has been engaged in the development of reliable air quality sensing devices using low-cost sensors, custom sensor boards, embedded software and cloud services [40]. The SibaIoT PM device used in this work measures ambient particulate matter concentrations in ug/m^3 of three classes of particles, namely $PM1.0$, $PM2.5$ and $PM10$, humidity and temperature. The device has very good accuracy, response time and sensitivity in indoor pollution levels.

With respect to the methodology, we have conducted a pilot application in a state-of-the art industrial space that is sensitive to infection caused by particulate matter such as dust. Fifteen PM devices were installed in three different production areas with varying air quality sensitivity. Indicative visual analytics are presented in the paper such as descriptive analytics, histograms, density, delay and line plots, as well as outcomes from the application of analytic functions such as pearson correlation, k-means clustering, classification and aggregated exposure to pollution on the sensor data. More specifically, both *Production-Area analytics*,i.e. analysis of multiple time-series generated by multiple devices deployed

at a specific production area and *Intra-Area analytics*, i.e., analysis of multiple time-series generated by multiple devices deployed at logically connected production areas, are discussed. For example, we have calculated annual max, min, average values per day-of-the-week, shift, day of the month, hour-of-the-day, poor, good and fair quality clusters,
Preliminary results show that the above analytics are promising in providing useful insights on the both the level of pollution and the intensity of industrial activity in the dairy. For example we have found that air quality is aggravated indoors during weekdays, that annual average particulate matter concentration follows humidity and temperature with a fixed lag.

Keywords: low-cost sensors, air quality monitoring, internet of things, wireless sensor networks, delay tolerant networking, middleware, analytics, python, pattern detection, outliers, visual analytics

References

1. Krzyzanowski, M., Martin, R. V., Van Dingenen, R., van Donkelaar, A., and Thurston, G. D. (2012) Exposure assessment for estimation of the global burden of disease attributable to outdoor air pollution. Environ. Sci. Technol., 46, 652-660.
2. WHO- Regional office for Europe, Review of evidence on health aspects of air pollution – REVIHAAP project: final technical report, 2013.
3. Samet J.M., Dominici F., Curriero F.C., Coursac I., Zeger, S.L., Fine Particulate Air Pollution and Mortality in 20 U.S. Cities, 1987–1994, *N Engl J Med* , Vol. 343, 2000, pp. 1742-1749
4. WHO- Regional office for Europe, Health effects of particulate matter. Policy implications for countries in eastern Europe, Caucasus and central Asia, 2013.
5. Bano, N., Assessment of indoor environmental impacts on human health (Case study: Glass city, Firozabad (India), *Pollution*, Vol. 3, No.2, 2017, pp. 175-183
6. Setton, E., Marshall J. D., Brauer M., Lundquist K. R., Hystad P., Keller P., and Cloutier-Fisher D., The impact of daily mobility on exposure to traffic-related air pollution and health effect estimates, *Journal of Exposure Science and Environmental Epidemiology*, Vol. 21, No. 1, 2011, pp. 42–48.
7. Kumar, P., Martani, C., Morawska, L., Norford, L., Choudhary, R., Bell, M., Leach, M., (2016) Indoor air quality and energy management through real–time sensing in commercial buildings, Energy and Buildings, (111):145-153, 0378-7788, https://doi.org/10.1016/j.enbuild.2015.11.037
8. Mahajan, S., and Kumar, P. (2020) Evaluation of low-cost sensors for quantitative personal exposure monitoring. Sustainable Cities and Society p. 102076
9. Kumar, P., Morawska, L., Martani, C., Biskos, G., Neophytou, M., Di Sabatino, S., et al. (2015) The rise of low-cost sensing for managing air pollution in cities, Environment International, 75 , pp. 199-205
10. Morawska, L., Thai, P.K., Liu, X., Asumadu-Sakyi, A., Ayoko, G, Bartonova,A., et al.: Applications of low-cost sensing technologies for air quality monitoring and exposure assessment: How far have they gone? Environment International, 116 (2018), pp. 286-299, 10.1016/j.envint.2018.04.018 (2018)

11. Snyder, E.G., Watkins, T.H., Solomon,P.A., Thoma, E.D., Williams, R.W., Hagler, G.S.W., et al. (2013) *Applications of low-cost sensing technologies for air quality monitoring and exposure assessment: How far have they gone?* Environmental Science & Technology, 47 (2013), pp. 11369-11377
12. Anjomshoaa, A., Duarte, F. , Rennings, D., Matarazzo, T.J., deSouza, P., Ratti, C., City scanner: Building and scheduling a mobile sensing platform for smart city services, *IEEE Internet of Things Journal*, Vol.5, 2018, pp. 4567-4579
13. DeSouza, P., Anjomshoaa, A.,Duarte, F., Kahn, R., Kumar, P., Ratti, C., Air quality monitoring using mobile low-cost sensors mounted on trash-trucks: Methods development and lessons learned, *Sustainable Cities and Society*, Vol.60, 2020, pp.102239
14. Elen, B., Peters, J., Poppel, M.V., Bleux, N., Theunis, J., Reggente, M., et al. The aeroflex: A bicycle for mobile air quality measurements. Sensors, Vol.13, 2013, pp. 221-240 (2013)
15. Dutta, P.; Aoki, P.M.; Kumar, N.; Mainwaring, A.; Myers, C.; Willett, W.; Woodruff, A. (2009) Common sense: Participatory urban sensing using a network of handheld air quality monitors. *In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Berkeley, CA, USA, 4–6 November ACM: New York, NY, USA, 2009; pp. 349–350.
16. Hasenfratz, D.; Saukh, O.; Sturzenegger, S.; Thiele, L. (2012) Participatory air pollution monitoring using smartphones. Mobile Sensing 1, 1–5
17. deSouza, P., Kahn, A. R., Limbacher, A. J., Marais, A. E., Duarte, F., and Ratti, C.: Combining low-cost, surface-based aerosol monitors with size-resolved satellite data for air quality applications. Atmospheric Measurement Techniques 13(10):5319-5334 DOI: 10.5194/amt-13-5319-2020 (2020)
18. Koukouli, M.E., Skoulidou, I., Karavias, A., Parcharidis, I., Balis, D, Manders-Groot, A.M.M., Segers, A.J., Eskes, H., van Geffen, J.: Sudden changes in nitrogen dioxide emissions over Greece due to lockdown after the outbreak of Covid-19. Atmospheric Chemistry and Physics, 21 (21), 1759-1774 (2020)
19. Postolache, O.A.; Pereira, J.M.D.; Girao, P.M.B.S.: Smart sensors network for air quality monitoring applications. IEEE Trans. Instrum. Meas., 58, 3253–3262 (2009)
20. Kumar, P., Martani, C., Morawska, L., Norford, L., Choudhary, R., Bell, M., Leach, M. (2016) Indoor air quality and energy management through real–time sensing in commercial buildings, Energy and Buildings, (111):145-153, 0378-7788
21. Hagan, D. H., Gani,S., Bhandari, S., Patel, K., Habib, G., Apte, J., Hildebrandt Ruiz, L., and Kroll, H. J. (2019) Inferring Aerosol Sources from Low-Cost Air Quality Sensor Measurements: A Case Study in Delhi, India, *Environ. Sci. Technol. Lett.*,Vol.6, No.8, pp.467-472
22. Yazdi, M. N., Arhami, N., Delavarrafiee, M., Ketabchy, M. (2019) Developing air exchange rate models by evaluating vehicle in-cabin air pollutant exposures in a highway and tunnel setting: case study of Tehran, Iran, *Environ Sci Pollut Res Int.*, Vol.26, No.1, pp.501-513
23. Bukowiecki, N., Dommen J., , Prévôt, A.S.H., Richter, R., Weingartner, E. (2002) and Baltensperger U., A mobile pollutant measurement laboratory—Measuring gas phase and aerosol ambient concentrations with high spatial and temporal resolution, *Atmospheric Environment*, Vol.36, pp. 5569-5579
24. Apte, J.S., Messier, K.P., Gani, S., Brauer, M., Kirchstetter, T.W., Lunden, M.M. et al. (2017) High-resolution air pollution mapping with google street view cars: Exploiting big data, *Environmental Science & Technology*, Vol.51, pp. 6999-7008

25. Capezzuto, L.; Abbamonte, L.; De Vito, S.; Massera, E.; Formisano, F.; Fattoruso, G.; Di Francia, G.; Buonanno, A. A maker friendly mobile and social sensing approach to urban air quality monitoring. *In Proceedings of the 2014 IEEE on SENSORS*, Valencia, Spain, ; pp. 12–16.
26. Murty, R.N.; Mainland, G.; Rose, I.; Chowdhury, A.R.; Gosain, A.; Bers, J.; Welsh, M. Citysense: An urban-scale wireless sensor network and testbed. *In Proceedings of the 2008 IEEE Conference on Technologies for Homeland Security*, Waltham, MA, USA, 12–13 May 2008; pp. 583–588
27. Kadri, A.; Yaacoub, E.; Mushtaha, M.; Abu-Dayya, A.(2013) Wireless sensor network for real-time air pollution monitoring. *In Proceedings of the 2013 1st International Conference on Communications, Signal Processing, and their applications (ICCSPA)*, Sharjah, United Arab Emirates, 12–14 February; pp. 1–5.
28. Jiang, Y.; Li, K.; Tian, L.; Piedrahita, R.; Yun, X.; Mansata, O.; Lv, Q.; Dick, R.P.; Hannigan, M.; Shang, L. (2011) MAQS: A personalized mobile sensing system for indoor air quality monitoring. *In Proceedings of the 13th International Conference on Ubiquitous Computing*, Beijing, China, 17–21 September 2011; ACM: New York, NY, USA; pp. 271–280.
29. Jelicic, V.; Magno, M.; Brunelli, D.; Paci, G.; Benini, L. (2013) Context-adaptive multimodal wireless sensor network for energy-efficient gas monitoring. *IEEE Sens. J.* 13, 328–338.
30. Mansour, S.; Nasser, N.; Karim, L.; Ali, A. (2014) Wireless sensor network-based air quality monitoring system. *In Proceedings of the 2014 International Conference on Computing, Networking and Communications (ICNC)*, Honolulu, HI, USA, pp. 545–550.
31. Sun, L.; Wong, K.C.; Wei, P.; Ye, S.; Huang, H.; Yang, F.; Westerdahl, D.; Louie, P.K.; Luk, C.W.; Ning, Z.(2016) Development and application of a next generation air sensor network for the Hong Kong marathon 2015 air quality monitoring. *Sensors* 16, 211.
32. Honicky, R.; Brewer, E.A.; Paulos, E.; White, R. (2008) N-smarts: Networked suite of mobile atmospheric real-time sensors. *In Proceedings of the Second ACM SIGCOMM Workshop on Networked Systems for Developing Regions*, Seattle, WA, USA, 18 August 2008; ACM: New York, NY, USA; pp. 25–30
33. Lane, N.D.; Miluzzo, E.; Lu, H.; Peebles, D.; Choudhury, T.; Campbell, A.T. (2010) A survey of mobile phone sensing. *IEEE Commun. Mag.* 48, pp. 140–150.
34. Helbig, C.; Bauer, H.S.; Rink, K.; Wulfmeyer, V.; Frank, M.; Kolditz, O. (2014) Concept and workflow for 3D visualization of atmospheric data in a virtual reality environment for analytical approaches. *Environ. Earth Sci.* 72, pp. 3767–3780.
35. Setti, L., Passarini, F., de Gennaro, G., Di Gilio, A., Palmisani, J., Buono, P., Fornari, G., Perrone, M., G., Piazzalunga, A., Barbieri, P., Rizzo, E. and Miani, A. (2020) Evaluation of the potential relationship between Particulate Matter (PM) pollution and COVID-19 infection spread in Italy:first observational study based on initial epidemic diffusion. BMJ Open ;10
36. Wu, X., Nethery, R. C., Sabath, M. B., Braun, D., Dominici, F. (2020) Air pollution and COVID-19 mortality in the United States: Strengths and limitations of an ecological regression analysis. Science Advances Vol. 6, no. 45
37. Saadat, S., Rawtani, D., Hussain, CM. (2020) Environmental perspective of COVID-19. Sci Total Environ. Aug 1;728
38. Le Quéré, C., Jackson, B.R., Jones, W.M., Smith, P.J.A., Abernethy, S., Andrew, M.R., De-Gol, J.A., Willis, R.D., Shan, Y., Canadell, G.J., Friedlingstein, P., Creutzig, F., Peters, P.G. (2020) Temporary reduction in daily global CO2 emissions during the COVID-19 forced confinement. Nat. Clim. Chang. 10, 647–653

39. Katsiri E.(2020) Sensor Networks with Edge Intelligence for Reliable Air Quality Monitoring in the Covid-19 Era. Proceedings of the ICR'22 International Conference on Innovations in Computing Research, 383-396
40. Katsiri E.(2020) Developing reliable air quality monitoring devices with low cost sensors: Method and lessons learned. International Journal of Environmental Science, 6, 425-444
41. Grafana: The open observability platform. https://grafana.com
42. Fadhel, M., Sekerinski, E., Yao, S. (2019) A Comparison of Time Series Databases for Storing Water Quality Data. Mobile Technologies and Applications for the Internet of Things, IMCL
43. Buelvas, J., Múnera, D., Tobón V., D.P. et al. (2023) Data Quality in IoT-Based Air Quality Monitoring Systems: a Systematic Mapping Study. Water Air Soil Pollut 234, 248 .
44. Sharifi R, Langari R. Nonlinear sensor fault diagnosis using mixture of probabilistic PCA models. Mech Syst Sign Process. 2017;85:638–50.
45. Wang RY, Strong DM. (1996) Beyond accuracy: what data quality means to data consumers. J Manag Inform Syst.;12(4):5–33.
46. Li Y, Parker LE. (2014) Nearest neighbor imputation using spatial-temporal correlations in wireless sensor networks. Inform Fusion. ;15:64–79.
47. Aggarwal CC. (2013) An introduction to outlier analysis. Outlier analysis. Springer: New York; p. 1–40.
48. Ahmad NF, Hoang DB, Phung MH. (2009) Robust preprocessing for health care monitoring framework. In: 2009 11th international conference on e-Health networking, applications and services (Healthcom). pp. 169–74.
49. Anderson, R. L.,(1942) Distribution of the Serial Correlation Coefficient, Annals of Mathematical Statistics, Volume 13, Number 1 1–13.
50. Bosman HH, Iacca G, Tejada A, Wörtche HJ, Liotta A. (2017) Spatial anomaly detection in sensor networks using neighborhood information. Inform Fusion; 33:41–56.
51. Moursi, A.S., El-Fishawy, N., Djahel, S. et al. An IoT enabled system for enhanced air quality monitoring and prediction on the edge. Complex Intell. Syst. 7, 2923–2947
52. InfluxDB line protocol reference. https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_reference/
53. Glantz, Stanton A.; Slinker, Bryan K.; Neilands, Torsten B. (2016), Primer of Applied Regression & Analysis of Variance (Third ed.), McGraw Hill
54. Aho, Ken A. (2014), Foundational and Applied Statistics for Biologists (First ed.), Chapman & Hall / CRC Press
55. Bartlett, M. S. (1946) On the Theoretical Specification and Sampling Properties of Autocorrelated Time-Series. Supplement to the Journal of the Royal Statistical Society, vol. 8, no. 1, pp. 27–41. JSTOR, http://www.jstor.org/stable/2983611.
56. Quenouille, M. H. (1949) The Joint Distribution of Serial Correlation Coefficients, The Annals of Mathematical Statistics, Vol. 20, 4 pp. 561–571

Abstract. The *trie* data structure is a good choice for finite maps whose keys are data structures (trees) rather than atomic values. But what if we want the keys to be *patterns*, each of which matches many lookup keys? Efficient matching of this kind is well studied in the theorem prover community, but much less so in the context of statically typed functional programming. Doing so yields an interesting new viewpoint — and a practically useful design pattern, with good runtime performance.

Triemaps that match
Technical Report

Simon Peyton Jones[1] and Sebastian Graf[2]

[1] Epic Games
[2] Karlsruhe Institute of Technology

1 Introduction

Many functional languages provide *finite maps* either as a built-in data type, or as a mature, well-optimised library. Generally the keys of such a map will be small: an integer, a string, or perhaps a pair of integers. But in some applications the key is large: an entire tree structure. For example, consider the Haskell expression

let $x = a + b$ **in** ... (**let** $y = a + b$ **in** $x + y$) ....

We might hope that the compiler will recognise the repeated sub-expression $(a + b)$ and transform to

let $x = a + b$ **in** ... $(x + x)$ ....

An easy way to do so is to build a finite map that maps the expression $(a + b)$ to x. Then, when encountering the inner **let**, we can look up the right hand side in the map, and replace y by x. All we need is a finite map keyed by syntax trees.

Traditional finite-map implementations tend to do badly in such applications, because they are often based on balanced trees, and make the assumption that comparing two keys is a fast, constant-time operation. That assumption is false for large, tree-structured keys.

Another time that a compiler may want to look up a tree-structured key is when rewriting expressions: it wants to see if any rewrite rule matches the sub-expression in hand, and if so rewrite with the instantiated right-hand side of the rule. To do this we need a fast way to see if a target expression matches one of the patterns in a set of ($\mathit{pattern}$, rhs) pairs. If there is a large number of such ($\mathit{pattern}$, rhs) entries to check, we would like to do so faster than checking them one by one. Several parts of GHC, a Haskell compiler, need matching lookup, and currently use an inefficient linear algorithm to do so.

In principle it is well known how to build a finite map for a deeply-structured key: use a *trie*. The matching task is also well studied but, surprisingly, only in the automated reasoning community (Section 7.1): they use so-called *discrimination trees*. In this paper we apply these ideas in the context of a statically-typed functional programming language, Haskell. This shift of context is surprisingly fruitful, and we make the following contributions:

- Following [Hinze(2000a)], we develop a standard pattern for a *statically-typed triemap* for an arbitrary algebraic data type (Section 3.2). In contrast, most of the literature describes untyped tries for a fixed, generic tree type. In particular:
  - Supported by type classes, we can make good use of polymorphism to build triemaps for polymorphic data types, such as lists (Section 3.6).
  - We cover the full range of operations expected for finite maps: not only *insert*ion and *lookup*, but *alter*, *union*, *fold*, *map* and *filter* (Section 3.2).
  - We develop a generic optimisation for singleton maps that compresses leaf paths. Intriguingly, the resulting triemap *transformer* can be easily mixed into arbitrary triemap definitions (Section 3.7).
- We show how to make our triemaps insensitive to α*-renamings* in keys that include binding forms (Section 4). Accounting for α-equivalence is not hard, but it is crucial for the applications in compilers.
- We extend our triemaps to support *matching* lookups (Section 5). This is an important step, because the only readily-available alternative is linear lookup. Our main contribution is to extend the established idea of tries keyed by arbitrary data types, so that it can handle matching too.
- We present measurements that compare the performance of our triemaps (ignoring their matching capability) with traditional finite-map implementations in Haskell (Section 6).

We discuss related work in Section 7. Our contribution is not so much a clever new idea as an exposition of some old ideas in a new context. Nevertheless, we found it surprisingly tricky to develop the "right" abstractions, such as the *TrieMap* and *Matchable* classes, the singleton-and-empty map data type, and the combinators we use in our instances. These abstractions have been through *many* iterations, and we hope that by laying them out here, as a functional pearl, we may shorten the path for others.

2 The problem we address

Our general task is as follows: *implement an efficient finite mapping from keys to values, in which the key is a tree.* Semantically, such a finite map is just a set of *(key,value)* pairs; we query the map by looking up a *target*. For example, the key might be a data type of syntax trees, defined like this:

```
type Var = String
data Expr = App Expr Expr | Lam Var Expr | Var Var
```

Here *Var* is the type of variables; these can be compared for equality and used as the key of a finite map. Its definition is not important for this paper, but for the sake of concreteness, you may wish to imagine it is simply a string: The data type *Expr* is capable of representing expressions like $(add\,x\,y)$ and $(\lambda x.\,add\,x\,y)$. We will use this data type throughout the paper, because it has all the features that occur in real expression data types: free variables like *add*, represented by a *Var*

```
type TF v = Maybe v → Maybe v
data Map k v = ...    -- Keys k, values v
Map.empty      :: Map k v
Map.insert     :: Ord k ⇒ k → v → Map k v → Map k v
Map.lookup     :: Ord k ⇒ k → Map k v → Maybe v
Map.alter      :: Ord k ⇒ TF v → k
                  → Map k v → Map k v
Map.foldr      :: (v → r → r) → r → Map k v → r
Map.map        :: (v → w) → Map k v → Map k w
Map.unionWith  :: Ord k ⇒ (v → v → v)
                  → Map k v → Map k v → Map k v
Map.size       :: Map k v → Int
Map.compose    :: Ord b ⇒ Map b c → Map a b → Map a c

infixr 1 >=>              -- Kleisli composition
(>=>) :: Monad m ⇒ (a → m b) → (b → m c)
                  → a → m c

infixr 1 >>>              -- Forward composition
(>>>) :: (a → b) → (b → c) → a → c

infixr 0 ▷                -- Reverse function application
(▷) :: a → (a → b) → b
```

Fig. 1: API for library functions

node; lambdas which can bind variables (*Lam*), and occurrences of those bound variables (*Var*); and nodes with multiple children (*App*). A real-world expression type would have many more constructors, including literals, let-expressions and suchlike.

2.1 Alpha-renaming

In the context of a compiler, where the keys are expressions or types, the keys may contain internal *binders*, such as the binder x in $(\lambda x.x)$. If so, we would expect insertion and lookup to be insensitive to α-renaming, so we could, for example, insert with key $(\lambda x.x)$ and look up with key $(\lambda y.y)$, to find the inserted value.

2.2 Lookup modulo matching

Beyond just the basic finite maps we have described, our practical setting in GHC demands more: we want to do a lookup that does *matching*. GHC supports so-called *rewrite rules* [Peyton Jones et al.(2001)], which the user can specify in their source program, like this:

```
{−# RULES "map/map" ∀f g xs. map f (map g xs)
                                  = map (f ∘ g) xs #−}
```

This rule asks the compiler to rewrite any target expression that matches the shape of the left-hand side (LHS) of the rule into the right-hand side (RHS). We use the term *pattern* to describe the LHS, and *target* to describe the expression we are looking up in the map. The pattern is explicitly quantified over the *pattern variables* (here f, g, and xs) that can be bound during the matching process. In other words, we seek a substitution for the pattern variables that makes the pattern equal to the target expression. For example, if the program we are compiling contains the expression $map\ double\ (map\ square\ nums)$, we would like to produce a substitution $[f \mapsto double, g \mapsto square, xs \mapsto nums]$ so that the substituted RHS becomes $map\ (double \circ square)\ nums$; we would replace the former expression with the latter in the code under consideration.

Of course, the pattern might itself have bound variables, and we would like to be insensitive to α-conversion for those. For example:

$\{-\#\ \textbf{RULES}\ \texttt{"map/id"}\ map\ (\lambda x \to x) = \lambda y \to y\ \#-\}$

We want to find a successful match if we see a call $map\ (\lambda y \to y)$, even though the bound variable has a different name.

Now imagine that we have thousands of such rules. Given a target expression, we want to consult the rule database to see if any rule matches. One approach would be to look at the rules one at a time, checking for a match, but that would be slow if there are many rules. Similarly, GHC's lookup for type-class instances and for type-family instances can have thousands of candidates. We would like to find a matching candidate more efficiently than by linear search.

2.3 Non-solutions

At first sight, our task can be done easily: define a total order on *Expr* and use a standard finite map library. Indeed that works, but it is terribly slow. A finite map is implemented as a binary search tree; at every node of this tree, we compare the key (an *Expr*, remember) with the key stored at the node; if it is smaller, go left; if larger, go right. Each lookup thus must perform a (logarithmic) number of potentially-full-depth comparisons of two expressions.

Another possibility might be to hash the *Expr* and use the hash-code as the lookup key. That would make lookup much faster, but it requires at least two full traversals of the key for every lookup: one to compute its hash code for every lookup, and a full equality comparison on a "hit" because hash-codes can collide.

But the killer is this: *neither binary search trees nor hashing is compatible with matching lookup.* For our purposes they are non-starters.

What other standard solutions to matching lookup are there, apart from linear search? The theorem proving and automated reasoning community has been working with huge sets of rewrite rules, just as we describe, for many years. They have developed term indexing techniques for the job [Sekar et al.(2001), Chapter 26], which attack the same problem from a rather different angle, as we discuss in Section 7.1.

3 Tries

A standard approach to a finite map in which the key has internal structure is to use a *trie*[3]. Generalising tries to handle an arbitrary algebraic data type as the key is a well established, albeit under-used, idea [Connelly and Morris(1995),Hinze(2000a)]. We review these ideas in this section. Let us consider a simplified form of expression:

```
data Expr = Var Var | App Expr Expr
```

We omit lambdas for now, so that all *Var* nodes represent free variables, which are treated as constants. We will return to lambdas in Section 4.

3.1 The interface of a finite map

Building on the design of widely used functions in Haskell (see Fig. 1), we seek these basic operations:

```
emptyEM :: ExprMap v
lkEM    :: Expr → ExprMap v → Maybe v
atEM    :: Expr → TF v → ExprMap v → ExprMap v
```

The lookup function *lkEM*[4] has a type that is familiar from every finite map. The update function *atEM*, typically called *alter* in Haskell libraries, changes the value stored at a particular key. The caller provides a *value transformation function* *TF v*, an abbreviation for *Maybe v* → *Maybe v* (see Fig. 1). This function transforms the existing value associated with the key, if any (hence the input *Maybe*), to a new value, if any (hence the output *Maybe*). We can easily define *insertEM* and *deleteEM* from *atEM*:

```
insertEM :: Expr → v → ExprMap v → ExprMap v
insertEM e v = atEM e (\_ → Just v)

deleteEM :: Expr → ExprMap v → ExprMap v
deleteEM e = atEM e (\_ → Nothing)
```

You might wonder whether, for the purposes of this paper, we could just define *insert*, leaving *atEM* for the Appendix[5], but as we will see in Section 3.3, our approach using tries requires the generality of *atEM*.

These fundamental operations on a finite map must obey the following properties:

$$\begin{aligned} &\mathit{lookup}\ e\ \mathit{empty} && \equiv \mathit{Nothing} \\ &\mathit{lookup}\ e\ (\mathit{alter}\ e\ \mathit{xt}\ m) && \equiv \mathit{xt}\ (\mathit{lookup}\ e\ m) \\ e_1 \neq e_2 \Rightarrow\ &\mathit{lookup}\ e_1\ (\mathit{alter}\ e_2\ \mathit{xt}\ m) && \equiv \mathit{lookup}\ e_1\ m \end{aligned}$$

We also support other standard operations on finite maps, with types analogous to those in Fig. 1, including *unionEM*, *mapEM*, and *foldrEM*.

[3] `https://en.wikipedia.org/wiki/Trie`

[4] We use short names *lkEM* and *atEM* consistently in this paper to reflect the single-column format.

[5] In the supplemental file `TrieMap.hs`

3.2 Tries: the basic idea

Here is a trie-based implementation for *Expr*:

```
data ExprMap v
  = EM {em_var :: Map Var v, em_app :: ExprMap (ExprMap v)}
```

Here *Map Var v* is any standard finite map (e.g. in *containers*[6]) keyed by *Var*, with values *v*. One way to understand this slightly odd data type is to study its lookup function:

```
lkEM :: Expr → ExprMap v → Maybe v
lkEM e (EM {em_var = var, em_app = app}) = case e of
  Var x       → Map.lookup x var
  App e1 e2 → case lkEM e1 app of
    Nothing → Nothing
    Just m1 → lkEM e2 m1
```

This function pattern-matches on the target *e*. The *Var* alternative says that to look up a variable occurrence, just look that variable up in the *em_var* field. But if the expression is an *App* e_1 e_2 node, we first look up e_1 in the *em_app* field, *which returns an ExprMap*. We then look up e_2 in that map. Each distinct e_1 yields a different *ExprMap* in which to look up e_2.

We can substantially abbreviate this code, at the expense of making it more cryptic, thus:

```
lkEM (Var x)       = em_var  >>> Map.lookup x
lkEM (App e1 e2) = em_app >>> lkEM e1 >=> lkEM e2
```

The function *em_var* :: *ExprMap v* → *Map Var v* is the auto-generated selector that picks the *em_var* field from an *EM* record, and similarly *em_app*. The functions (>>>) and (>=>) are right-associative forward composition operators, respectively monadic and non-monadic, that chain the individual operations together (see Fig. 1). Finally, we have η-reduced the definition, by omitting the *m* parameter. These abbreviations become quite worthwhile when we add more constructors, each with more fields, to the key data type.

Notice that in contrast to the approach of Section 2.3, *we never compare two expressions for equality or ordering*. We simply walk down the *ExprMap* structure, guided at each step by the next node in the target.

This definition is extremely short and natural. But it embodies a hidden complexity: *it requires polymorphic recursion*. The recursive call to *lkEM* e_1 instantiates *v* to a different type than the parent function definition. Haskell supports polymorphic recursion readily, provided you give type signature to *lkEM*, but not all languages do.

[6] `https://hackage.haskell.org/package/containers`

3.3 Modifying tries

It is not enough to look up in a trie – we need to *build* them too. First, we need an empty trie. Here is one way to define it:

```
emptyEM :: ExprMap v
emptyEM = EM {em_var = Map.empty, em_app = emptyEM}
```

It is interesting to note that *emptyEM* is an infinite, recursive structure: the *em_app* field refers back to *emptyEM*. We will change this definition in Section 3.5, but it works perfectly well for now. Next, we need to *alter* a triemap:

```
atEM :: Expr → TF v → ExprMap v → ExprMap v
atEM e tf m@(EM {em_var = var, em_app = app}) = case e of
  Var x     → m {em_var = Map.alter tf x var}
  App e1 e2 → m {em_app = atEM e1 (liftTF (atEM e2 tf)) app}
liftTF :: (ExprMap v → ExprMap v) → TF (ExprMap v)
liftTF f Nothing  = Just (f emptyEM)
liftTF f (Just m) = Just (f m)
```

In the *Var* case, we must just update the map stored in the *em_var* field, using the *Map.alter* function from Fig. 1. In the *App* case we look up e_1 in *app*; we should find a *ExprMap* there, which we want to alter with *tf*. We can do that with a recursive call to *atEM*, using *liftTF* for impedance-matching.

The *App* case shows why we need the generality of *alter*. Suppose we attempted to define an apparently-simpler *insert* operation. Its equation for (*App* e_1 e_2) would look up e_1 — and would then need to *alter* that entry (an *ExprMap*, remember) with the result of inserting (e_2, v). So we are forced to define *alter* anyway.

We can abbreviate the code for *atEM* using combinators, as we did in the case of lookup, and doing so pays dividends when the key is a data type with many constructors, each with many fields. However, the details are fiddly and not illuminating, so we omit them here. Indeed, for the same reason, in the rest of this paper we will typically omit the code for *alter*, though the full code is available in the Appendix.

3.4 Unions of maps

A common operation on finite maps is to take their union:

```
unionEM :: ExprMap v → ExprMap v → ExprMap v
```

In tree-based implementations of finite maps, such union operations can be tricky. The two trees, which have been built independently, might not have the same left-subtree/right-subtree structure, so some careful rebalancing may be required. But for tries there are no such worries – their structure is identical, and we can simply zip them together. There is one wrinkle: just as we had to generalise *insert* to *alter*, to accommodate the nested map in *em_app*, so we need to generalise *union* to *unionWith*:

```
unionWithEM :: (v → v → v)
            → ExprMap v → ExprMap v → ExprMap v
```

When a key appears on both maps, the combining function is used to combine the two corresponding values. With that generalisation, the code is as follows:

```
unionWithEM f (EM {em_var = var1, em_app = app1})
              (EM {em_var = var2, em_app = app2})
  = EM {em_var = Map.unionWith f var1 var2
       , em_app = unionWithEM (unionWithEM f) app1 app2}
```

It could hardly be simpler.

3.5 Folds and the empty map

The strange, infinite definition of *emptyEM* given in Section 3.3 works fine (in a lazy language at least) for lookup, alteration, and union, but it fails fundamentally when we want to *iterate* over the elements of the trie. For example, suppose we wanted to count the number of elements in the finite map; in *containers* this is the function *Map.size* (Fig. 1). We might attempt:

```
sizeEM :: ExprMap v → Int
sizeEM (EM {em_var = var, em_app = app})
  = Map.size var+???
```

We seem stuck because the size of the *app* map is not what we want: rather, we want to add up the sizes of its *elements*, and we don't have a way to do that yet. The right thing to do is to generalise to a fold:

```
foldrEM :: ∀v. (v → r → r) → r → ExprMap v → r
foldrEM k z (EM {em_var = var, em_app = app})
  = Map.foldr k z1 var
  where
    z1 = foldrEM kapp z (app :: ExprMap (ExprMap v))
    kapp m1 r = foldrEM k r m1
```

In the binding for z_1 we fold over *app*, using *kapp* to combine the map we find with the accumulator, by again folding over the map with *foldrEM*.

But alas, *foldrEM* will never terminate! It always invokes itself immediately (in z_1) on *app*; but that invocation will again recursively invoke *foldrEM*; and so on forever. The solution is simple: we just need an explicit representation of the empty map. Here is one way to do it (we will see another in Section 3.7):

```
data ExprMap v = EmptyEM | EM {em_var :: ..., em_app :: ...}
emptyEM :: ExprMap v
emptyEM = EmptyEM
foldrEM :: (v → r → r) → r → ExprMap v → r
foldrEM k z EmptyEM = z
foldrEM k z (EM {em_var = var, em_app = app})
```

```
  = Map.foldr k z1 var
  where
    z1 = foldrEM kapp z app
    kapp m1 r = foldrEM k r m1
```

Equipped with a fold, we can easily define the size function, and another that returns the range of the map:

```
sizeEM :: ExprMap v → Int
sizeEM = foldrEM (λ_ n → n + 1) 0

elemsEM :: ExprMap v → [v]
elemsEM = foldrEM (:) []
```

3.6 A type class for triemaps

Since all our triemaps share a common interface, it is useful to define a type class for them:

```
class Eq (Key tm) ⇒ TrieMap tm where
  type Key tm :: Type
  emptyTM       :: tm a
  lkTM          :: Key tm → tm a → Maybe a
  atTM          :: Key tm → TF a → tm a → tm a
  foldrTM       :: (a → b → b) → tm a → b → b
  unionWithTM :: (a → a → a) → tm a → tm a → tm a
  ...
```

The class constraint *TrieMap tm* says that the type *tm* is a triemap, with operations *emptyTM*, *lkTM* etc. The class has an *associated type* [Chakravarty et al.(2005)], *Key tm*, a type-level function that transforms the type of the triemap into the type of *keys* of that triemap.

Now we can witness the fact that *ExprMap* is a *TrieMap*, like this:

```
instance TrieMap ExprMap where
  type Key ExprMap = Expr
  emptyTM = emptyEM
  lkTM    = lkEM
  atTM    = atEM
  ...
```

Having a class allow us to write helper functions that work for any triemap, such as

```
insertTM :: TrieMap tm ⇒ Key tm → v → tm v → tm v
insertTM k v = atTM k (\_ → Just v)

deleteTM :: TrieMap tm ⇒ Key tm → tm v → tm v
deleteTM k = atTM k (\_ → Nothing)
```

But that is not all. Suppose our expressions had multi-argument apply nodes, *AppV*, thus

```
data Expr = ... | AppV Expr [Expr]
```

Then we would need to build a trie keyed by a *list* of *Expr*. A list is just another algebraic data type, built with nil and cons, so we *could* use exactly the same approach, thus

```
lkLEM :: [Expr] → ListExprMap v → Maybe v
```

But rather than to define a *ListExprMap* for keys of type [*Expr*], and a *ListDeclMap* for keys of type [*Decl*], etc, we would obviously prefer to build a trie for lists of *any type*, like this [Hinze(2000a)]:

```
instance TrieMap tm ⇒ TrieMap (ListMap tm) where
  type Key (ListMap tm) = [Key tm]
  emptyTM = emptyLM
  lkTM    = lkLM
  ...
data ListMap tm v = LM { lm_nil :: Maybe v
                       , lm_cons :: tm (ListMap tm v) }
emptyLM :: TrieMap tm ⇒ ListMap tm
emptyLM = LM { lm_nil = Nothing, lm_cons = emptyTM }
lkLM :: TrieMap tm ⇒ [Key tm] → ListMap tm v → Maybe v
lkLM []     = lm_nil
lkLM (k : ks) = lm_cons ⋙ lkTM k ≻⇒ lkLM ks
```

The code for *atLM* and *foldrLM* is routine. Notice that all of these functions are polymorphic in *tm*, the triemap for the list elements.

3.7 Singleton maps, and empty maps revisited

Suppose we start with an empty map, and insert a value with a key (an *Expr*) such as

```
App (App (Var "f") (Var "x")) (Var "y")
```

Looking at the code for *atEM* in Section 3.3, you can see that because there is an *App* at the root, we will build an *EM* record with an empty *em_var*, and an *em_app* field that is... another *EM* record. Again the *em_var* field will contain an empty map, while the *em_app* field is a further *EM* record.

In effect, the key is linearised into a chain of *EM* records. This is great when there are a lot of keys with shared structure, but once we are in a sub-tree that represents a *single* key-value pair it is a rather inefficient way to represent the key. So a simple idea is this: when a *ExprMap* represents a single key-value pair, represent it directly as a key-value pair, like this:

```
data ExprMap v = EmptyEM
               | SingleEM Expr v   -- A single key/value pair
               | EM { em_var :: ..., em_app :: ... }
```

But in the triemap for for each new data type X, we will have to tiresomely repeat these extra data constructors, *EmptyX* and *SingleX*. For example we would have to add *EmptyList* and *SingleList* to the *ListMap* data type of Section 3.6. It is better instead to abstract over the enclosed triemap, as follows[7]:

```
data SEMap tm v = EmptySEM
                | SingleSEM (Key tm) v
                | MultiSEM (tm v)
instance TrieMap tm ⇒ TrieMap (SEMap tm) where
  type Key (SEMap tm) = Key tm
  emptyTM = EmptySEM
  lkTM    = lkSEM
  atTM    = atSEM
  ...
```

The code for lookup practically writes itself. We abstract over *Maybe* with some *MonadPlus* combinators to enjoy forwards compatibility to Section 5:

```
lkSEM :: TrieMap tm ⇒ Key tm → SEMap tm v → Maybe v
lkSEM k m = case m of
  EmptySEM       → mzero
  SingleSEM pk v → guard (k == pk) ≫ pure v
  MultiSEM m     → lkTM k m
```

Where *mzero* means *Nothing* and *pure* means *Just*. The *guard* expression in the *SingleSEM* will return *Nothing* when the key expression k doesn't equate to the pattern expression *pk*. To test for said equality we require an *Eq* (*Key tm*) instance, hence it is a superclass of *TrieMap tm* in the **class** declaration in Section 3.6.

The code for alter is more interesting, because it governs the shift from *EmptySEM* to *SingleSEM* and thence to *MultiSEM*:

```
atSEM :: TrieMap tm
      ⇒ Key tm → TF v → SEMap tm v → SEMap tm v
atSEM k tf EmptySEM = case tf Nothing of Nothing → EmptySEM
                                         Just v  → SingleSEM k v
atSEM k1 tf (SingleSEM k2 v2) = if k1 == k2
  then case tf (Just v2) of
    Nothing → EmptySEM
    Just v' → SingleSEM k2 v'
  else case tf Nothing of
    Nothing → SingleSEM k2 v2
    Just v1 → MultiSEM (insertTM k1 v1 (insertTM k2 v2 emptyTM))
atSEM k tf (MultiSEM tm) = MultiSEM (atTM k tf tm)
```

Adding a new item to a triemap can turn *EmptySEM* into *SingleSEM* and *SingleSEM* into *MultiSEM*; and deleting an item from a *SingleSEM* turns it back into *EmptySEM*. You might wonder whether we can shrink a *MultiSEM*

[7] *SEMap* stands for "singleton or empty map".

back to a *SingleSEM* when it has only one remaining element? Yes we can, but it takes quite a bit of code, and it is far from clear that it is worth doing so.

Finally, we need to re-define *ExprMap* and *ListMap* using *SEMap*:

```
type ExprMap    = SEMap ExprMap'
data ExprMap' v = EM {em_var :: ..., em_app :: ExprMap (ExprMap v)}
type ListMap    = SEMap ListMap'
data ListMap' tm v = LM {lm_nil :: ..., lm_cons :: tm (ListMap tm v)}
```

The auxiliary data types *ExprMap'* and *ListMap'* have only a single constructor, because the empty and singleton cases are dealt with by *SEMap*. We reserve the original, un-primed, names for the user-visible *ExprMap* and *ListMap* constructors.

3.8 Generic programming

We have not described a triemap *library*; rather we have described a *design pattern*. More precisely, given a new algebraic data type X, we have described a systematic way of defining a triemap, *XMap*, keyed by values of type X. Such a triemap is represented by a record:

- Each *constructor* K of X becomes a *field* *x_k* in *XMap*.
- Each *field* of a constructor K becomes a *nested triemap* in the type of the field *x_k*.
- If X is polymorphic then *XMap* becomes a triemap transformer, like *ListMap* above.

Actually writing out all this boilerplate code is tiresome, and it can of course be automated. One way to do so would be to use generic or polytypic programming, and Hinze describes precisely this [Hinze(2000a)]. Another approach would be to use Template Haskell.

We do not develop either of these approaches here, because our focus is only the functionality and expressiveness of the triemaps. However, everything we do is compatible with an automated approach to generating boilerplate code.

4 Keys with binders

If our keys are expressions (in a compiler, say) they may contain binders, and we want insert and lookup to be insensitive to α-renaming. That is the challenge we address next. Here is our data type *Expr* from Section 2.1, which brings back binding semantics through the *Lam* constructor:

```
data Expr = App Expr Expr | Lam Var Expr | Var Var
```

The key idea is simple: we perform De Bruijn numbering on the fly, renaming each binder to a natural number, from outside in. So, when inserting or looking up a key $(\lambda x.\, foo\ (\lambda y.\, x + y))$ we behave as if the key was $(\lambda.\, foo\ (\lambda.\#_1 + \#_2))$, where each $\#_i$ stands for an occurrence of the variable bound by the i'th lambda,

```
type DBNum = Int
data DBEnv = DBE { dbe_next :: DBNum, dbe_env :: Map Var DBNum }

emptyDBE :: DBEnv
emptyDBE = DBE { dbe_next = 1, dbe_env = Map.empty }

extendDBE :: Var -> DBEnv -> DBEnv
extendDBE v (DBE { dbe_next = n, dbe_env = dbe })
  = DBE { dbe_next = n + 1, dbe_env = Map.insert v n dbe }

lookupDBE :: Var -> DBEnv -> Maybe DBNum
lookupDBE v (DBE { dbe_env = dbe }) = Map.lookup v dbe
```

Fig. 2: De Bruijn leveling

counting from the root of the expression. In effect, then, we behave as if the data type was like this:

```
data Expr' = App Expr Expr | Lam Expr | FVar Var | BVar BoundKey
```

Notice (a) the *Lam* node no longer has a binder and (b) there are two sorts of *Var* nodes, one for free variables and one for bound variables, carrying a *BoundKey* (see below). We will not actually build a value of type *Expr'* and look that up in a trie keyed by *Expr'*; rather, we are going to *behave as if we did*. Here is the code (which uses Fig. 2):

```
data ModAlpha a = A DBEnv a
type AlphaExpr = ModAlpha Expr
instance Eq AlphaExpr where ...

type BoundKey = DBNum
type ExprMap = SEMap ExprMap'
data ExprMap' v
  = EM { em_fvar :: Map Var v            -- Free vars
       , em_bvar :: Map BoundKey v       -- Lambda-bound vars
       , em_app  :: ExprMap (ExprMap v)
       , em_lam  :: ExprMap v }

instance TrieMap ExprMap' where
  type Key ExprMap' = AlphaExpr
  lkTM = lkEM
  ...

lkEM :: AlphaExpr -> ExprMap' v -> Maybe v
lkEM (A bve e) = case e of
  Var v -> case lookupDBE v bve of
    Nothing -> em_fvar >>> Map.lookup v
    Just bv -> em_bvar >>> Map.lookup bv
  App e1 e2 -> em_app >>> lkTM (A bve e1) >=> lkTM (A bve e2)
  Lam v e   -> em_lam >>> lkTM (A (extendDBE v bve) e)

lookupClosedExpr :: Expr -> ExprMap v -> Maybe v
lookupClosedExpr e = lkEM (A emptyDBE e)
```

We maintain a *DBEnv* (cf. Fig. 2) that maps each lambda-bound variable to its De Bruijn level[8] [de Bruijn(1972)], its *BoundKey*. The expression we look up — the first argument of *lkEM* — becomes an *AlphaExpr*, which is a pair of a *DBEnv* and an *Expr*. At a *Lam* node we extend the *DBEnv*. At a *Var* node we look up the variable in the *DBEnv* to decide whether it is lambda-bound or free, and behave appropriately[9].

The construction of Section 3.7, to handle empty and singleton maps, applies without difficulty to this generalised map. To use it we must define an instance *Eq AlphaExpr* to satisfy the *Eq* super class constraint on the trie key, so that we can instantiate *TrieMap ExprMap'*. That *Eq AlphaExpr* instance simply equates two α-equivalent expressions in the standard way. The code for *alter* and *foldr* holds no new surprises either.

And that is really all there is to it: it is remarkably easy to extend the basic trie idea to be insensitive to α-conversion and even mix in trie transformers such as *SEMap* at no cost other than writing two instance declarations.

5 Tries that match

A key advantage of tries over hash-maps and balanced trees is that we can support *matching* (Section 2.2).

5.1 What "matching" means

Semantically, a matching trie can be thought of as a set of *entries*, each of which is a (pattern, value) pair. What is a pattern? It is a pair (vs, p) where

- vs is a set of *pattern variables*, such as $[a, b, c]$.
- p is a *pattern expression*, such as $f\, a\, (g\, b\, c)$.

A pattern may of course contain free variables (not bound by the pattern), such as f and g in the above example, which are regarded as constants by the algorithm. A pattern (vs, p) *matches* a target expression e iff there is a unique substitution S whose domain is vs, such that $S(p) = e$.

We allow the same variable to occur more than once in the pattern. For example, the pattern $([x], f\ x\ x)$ should match targets like $(f\ 1\ 1)$ or $(f\ (g\ v)\ (g\ v))$, but not $(f\ 1\ (g\ v))$. This ability is important if we are to use matching tries to implement class or type-family look in GHC.

[8] The De Bruijn *index* of the occurrence of a variable v counts the number of lambdas between the occurrence of v and its binding site. The De Bruijn *level* of v counts the number of lambdas between the root of the expression and v's binding site. It is convenient for us to use *levels*.

[9] The implementation from the Appendix uses more efficient *IntMap*s for mapping *BoundKey*. *IntMap* is a itself trie data structure, so it could have made a nice "Tries all the way down" argument. But we found it distracting to present here, hence regular ordered *Map*.

In implementation terms, we can characterise matching by the following type class:

```
class (Eq (Pat k), MonadPlus (Match k)) ⇒ Matchable k where
  type Pat   k :: Type
  type Match k :: Type → Type
  match :: Pat k → k → Match k ()
```

For any key type k, the *match* function takes a pattern of type *Pat* k, and a key of type k, and returns a monadic match of type *Match* k (), where *Pat* and *Match* are associated types of k. Matching can fail or can return many results, so *Match* k is a *MonadPlus*:

```
mzero :: MonadPlus m ⇒ m a
mplus :: MonadPlus m ⇒ m a → m a → m a
```

To make this more concrete, here is a possible *Matchable* instance for *AlphaExpr*:

```
instance Matchable AlphaExpr where
  type Pat   AlphaExpr = PatExpr
  type Match AlphaExpr = MatchExpr
  match = matchE
```

Let's look at the pieces, one at a time.

Patterns A pattern *PatExpr* over *AlphaExpr* can be defined like this:

```
data PatExpr = P PatKeys AlphaExpr
type PatKeys = Map PatVar PatKey
type PatVar  = Var
type PatKey  = DBNum
```

A pattern *PatExpr* is a pair of an *AlphaExpr* and a *PatKeys* that maps each of the quantified pattern variables to a canonical De Bruijn *PatKey*. Just as in Section 4, *PatKeys* make the pattern insensitive to the particular names, and order of quantification, of the pattern variables. We canonicalise the quantified pattern variables before starting a lookup, numbering pattern variables in the order they appear in a left-to-right scan. For example

Original pattern	Canonical *PatExpr*
$([a, b], f\ a\ b\ a)$	$P\ [a \mapsto 1, b \mapsto 2]\ (f\ a\ b\ a)$
$([x, g], f\ (g\ x)$	$P\ [x \mapsto 2, g \mapsto 1]\ (f\ (g\ x))$

The matching monad There are many possible implementation of the *MatchExpr* monad, but here is one:

```
type MatchExpr v = MR (StateT Subst [] v)
type Subst = Map PatKey Expr
```

The *MatchExpr* type is isomorphic to $Subst \to [(v, Subst)]$; matching takes a substitution for pattern variables (more precisely, their canonical *PatKey*s), and yields a possibly-empty list of values paired with an extended substitution. Notice that the substitution binds pattern keys to *Expr*, not *AlphaExpr*, because the pattern variables cannot mention lambda-bound variables within the target expression.

The formulation in terms of *StateT* endows us with just the right *Monad* and *MonadPlus* instances, as well as favorable performance because of early failure on contradicting *match*es and the ability to share work done while matching a shared prefix of multiple patterns.

The monad comes with some auxiliary functions that we will need later:

```
runMatchExpr :: MatchExpr v → [(Subst, v)]
liftMaybe    :: Maybe v → MatchExpr v
refineMatch :: (Subst → Maybe Subst) → MatchExpr ()
```

Their semantics should be apparent from their types. For example, *runMatchExpr* runs a *MatchExpr* computation, starting with an empty *Subst*, and returning a list of all the successful $(Subst, v)$ matches.

Matching summary The implementation of *matchE* is entirely straightforward, using simultaneous recursive descent over the pattern and target. The code is given in the Appendix.

The key point is this: nothing in this section is concerned with tries. Here we are simply concerned with the mechanics of matching, and its underlying monad. There is ample room for flexibility. For example, if the key terms had two kinds of variables (say type variables and term variables) we could easily define *Match* to carry two substitutions; or *Match* could return just the first result rather than a list of all of them; and so on.

5.2 The matching trie class

The core of our matching trie is the class *MTrieMap*, which generalises the *TrieMap* class of Section 3.6:

```
class Matchable (MKey tm) ⇒ MTrieMap tm where
  type MKey tm :: Type
  emptyMTM :: tm v
  lkMTM    :: MKey tm → tm v → Match (MKey tm) v
  atMTM    :: Pat (MKey tm) → TF v → tm v → tm v
```

The lookup function takes a key of type *MKey tm* as before, but it returns something in the *Match* monad, rather than the *Maybe* monad. The *atMTM* takes a *pattern* (rather than just a key), of type *Pat* (*MKey tm*), and alters the trie's value at that pattern[10].

[10] Remember, a matching trie represents a set of (pattern,value) pairs.

We can generalise *SEMap* (Section 3.7) in a similar way:

```
data MSEMap tm v = EmptyMSEM
                 | SingleMSEM (Pat (MKey tm)) v
                 | MultiMSEM (tm v)
instance MTrieMap tm ⇒ MTrieMap (MSEMap tm) where
  type MKey (MSEMap tm) = MKey tm
  emptyMTM = EmptyMSEM
  lkMTM    = lkMSEM
  atMTM    = atMSEM
```

Notice that *SingleMSEM* contains a *pattern*, not merely a *key*, unlike *SingleSEM* in Section 3.7. The code for *lkMSEM* and *atMSEM* is straightforward; we give the former here, leaving the latter for the Appendix

```
lkMSEM :: MTrieMap tm ⇒ MKey tm → MSEMap tm a
                      → Match (MKey tm) a
lkMSEM k EmptyMSEM          = mzero
lkMSEM k (MultiMSEM m)      = lkMTM k m
lkMSEM k (SingleMSEM pat v) = match pat k ≫ pure v
```

Notice the call to *mzero* to make the lookup fail if the map is empty; and, in the *SingleMSEM* case, the call *match* to match the pattern against the key.

5.3 Matching tries for *Expr*

Next, we show how to implement a matching triemap for our running example, *AlphaExpr*. The data type follows closely the pattern we developed for *ExprMap* (Section 4):

```
type MExprMap = MSEMap MExprMap'
data MExprMap' v
  = MM { mm_fvar :: Map Var v          -- Free var
       , mm_bvar :: Map BoundKey v     -- Bound var
       , mm_pvar :: Map PatKey v       -- Pattern var
       , mm_app  :: MExprMap (MExprMap v)
       , mm_lam  :: MExprMap v }
instance MTrieMap MExprMap' where
  type MKey MExprMap' = AlphaExpr
  emptyMTM = ...  -- boring
  lkMTM    = lookupPatMM
  atMTM    = alterPatMM
```

The main difference is that we add an extra field *mm_pvar* to *MExprMap'*, for occurrences of a pattern variable. You can see how this field is used in the lookup code:

```
lookupPatMM :: ∀v. AlphaExpr → MExprMap' v → MatchExpr v
lookupPatMM ae@(A bve e) (MM {..})
```

```
  = rigid `mplus` flexi
  where
    rigid = case e of
      Var x       → case lookupDBE x bve of
        Just bv   → mm_bvar  ▷  liftMaybe ∘ Map.lookup bv
        Nothing   → mm_fvar  ▷  liftMaybe ∘ Map.lookup x
      App e1 e2   → mm_app   ▷  lkMTM (A bve e1)
                             >=> lkMTM (A bve e2)
      Lam x e     → mm_lam   ▷  lkMTM (A (extendDBE x bve) e)
    flexi = mm_pvar ▷ IntMap.toList ▷ map match_one ▷ msum
    match_one :: (PatVar, v) → MatchExpr v
    match_one (pv, v) = matchPatVarE pv ae >> pure v
```

Matching lookup on a trie matches the target expression against *all patterns the trie represents*. The *rigid* case is no different from exact lookup; compare the code for *lkEM* in Section 4. The only difference is that we need *liftMaybe* (from Section 5.1) to take the *Maybe* returned by *Map.lookup* and lift it into the *MatchExpr* monad.

The *flexi* case handles the triemap entries whose pattern is simply one of the quantified pattern variables; these entries are stored in the new *mm_pvar* field. We enumerate these entries with *toList*, to get a list of $(PatVar, v)$ pairs, match each such pair against the target with *match_one*, and finally accumulate all the results with *msum*. In turn *match_one* uses *matchParVarE* to match the pattern variable with the target and, if successful, returns corresponding value *v*.

The *matchPatVarE* function does the heavy lifting, using some simple auxiliary functions whose types are given below:

```
matchPatVarE :: PatKey → AlphaExpr → MatchExpr ()
matchPatVarE pv (A bve e) = refineMatch $ λms →
  case Map.lookup pv ms of
    Nothing   -- pv is not bound
      | noCaptured bve e → Just (Map.insert pv e ms)
      | otherwise        → Nothing
    Just sol  -- pv is already bound
      | noCaptured bve e
      , eqExpr e sol     → Just ms
      | otherwise        → Nothing
eqExpr     :: Expr → Expr → Bool
noCaptured :: DBEnv → Expr → Bool
```

To match a pattern variable *pv* against an expression $(A\ bve\ e)$, we first look up *pv* in the current substitution (obtained from the *MatchExpr* state monad. If *pv* is not bound we simply extend the substitution.

But wait! Consider matching the pattern $([p], \lambda x \to p)$ against the target $(\lambda y \to 3)$. That's fine: we should succeed, binding *a* to 3. But suppose we match that same pattern against target $(\lambda y \to y)$. It would be nonsense to "succeed" binding *a* to *y*, because *y* is locally bound within the target. Hence the

noCaptured test, which returns *True* iff the expression does not mention any of the locally-bound variables.

If *pv* is already bound in the substitution, we have a repeated pattern variable (see Section 5.1), and we must check that the target expression is equal (using *eqExpr*) to the the one already bound to *pv*. Once again, however, we must check that the target does not contain any locally-bound variables, hence the *noCaptured* check.

lookupPatMM is the trickiest case. The code for *alterPatMM*, and the other operations of the class, is very straightforward, and is given in the Appendix.

5.4 The external API

The matching tries we have described so far use canonical pattern keys, a matching monad, and other machinery that should be hidden from the client. We seek an external API more like this:

```
type PatMap :: Type → Type
alterPM    :: ([Var], Expr) → TF v → PatMap v → PatMap v
lookupPM :: Expr → PatMap v → [(PatSubst, v)]
type PatSubst = [(Var, Expr)]
```

When altering a *PatMap* we supply a client-side pattern, which is just a pair $([Var], Expr)$ of the quantified pattern variables and the pattern. When looking up in a *PatMap* we supply a target expression, and get back a list of matches, each of which is a pair of the value and the substitution for those original pattern variables that made the pattern equal to the target.

So *alterPM* must canonicalise the client-side pattern variables before altering the trie; that is easy enough. But how can *lookupPM* recover the client-side *PatSubst*? Somehow we must remember the canonicalisation used when *inserting* so that we can invert it when *matching*. For example, suppose we insert the two (pattern, value pairs)

$$(([p], f\ p\ True), v_1) \quad \text{and} \quad (([q], f\ q\ False), v_2)$$

Both patterns will canonicalise their (sole) pattern variable to the De Bruin index 1. So if we look up the target $(f\ e\ True)$ the *MatchExpr* monad will produce a final *Subst* that maps $[1 \mapsto e]$, paired with the value v_1. But we want to return $([(\texttt{"p"}, e)], v_1)$ to the client, a *PatSubst* that uses the client variable `"p"`, not the internal index 1.

The solution is simple enough: *we store the mapping in the triemap's domain*, along with the values, thus:

```
type PatMap v = MExprMap (PatKeys, v)
```

Now the code writes itself. Here is *alterPM*:

```
alterPM :: ∀v. ([Var], Expr) → TF v → PatMap v → PatMap v
alterPM (pvars, e) tf pm = atMTM pat ptf pm
```

```
  where
    pks :: PatKeys = canonPatKeys pvars e
    pat :: PatExpr = P pks (A emptyDBE e)
    ptf :: TF (PatKeys, v)
    ptf Nothing       = fmap (λv → (pks, v)) (tf Nothing)
    ptf (Just (_, v)) = fmap (λv → (pks, v)) (tf (Just v))
canonPatKeys :: [Var] → Expr → PatKeys
```

The auxiliary function *canonPatKeys* takes the client-side pattern $(pvars, e)$, and returns a *PatKeys* (Section 5.1) that maps each pattern variable to its canonical De Bruijn index. *canonPatKeys* is entirely straightforward: it simply walks the expression, numbering off the pattern variables in left-to-right order.

Then we can simply call the internal *atMTM* function, passing it: a canonical $pat :: PatExpr$; and a transformer $ptf :: TF\ (PatKeys, v)$ that will pair the *PatKeys* with the value supplied by the user via $tf :: TF\ v$. Lookup is equally easy:

```
lookupPM :: Expr → PatMap v → [(PatSubst, v)]
lookupPM e pm
  = [(Map.toList (subst `Map.compose` pks), x)
    | (subst, (pks, x)) ← runMatchExpr $
                          lkMTM (A emptyDBE e) pm]
```

We use *runMatchExpr* to get a list of successful matches, and then pre-compose (see Fig. 1) the internal *Subst* with the *PatKeys* mapping that is part of the match result. We turn that into a list to get the client-side *PatSubst*. The only tricky point is what to do with pattern variables that are not substituted. For example, suppose we insert the pattern $([p, q], f\ p)$. No lookup will bind q, because q simply does not appear in the pattern. One could reject this on insertion, but here we simply return a *PatSubst* with no binding for q.

5.5 Most specific match, and unification

It is tempting to ask: can we build a lookup that returns only the *most specific* matches? And, can we build a lookup that returns all values whose patterns *unify* with the target. Both would have useful applications, in GHC at least.

However, both seem difficult to achieve. All our attempts became mired in complexity, and we leave this for further work, and as a challenge for the reader. We outline some of the difficulties of unifying lookup in Appendix B.

6 Evaluation

So far, we have seen that trie maps offer a significant advantage over other kinds of maps like ordered maps or hash maps: the ability to do a matching lookup (in Section 5). In this section, we will see that query performance is another advantage. Our implementation of trie maps in Haskell can generally compete in performance with other map data structures, while significantly outperforming

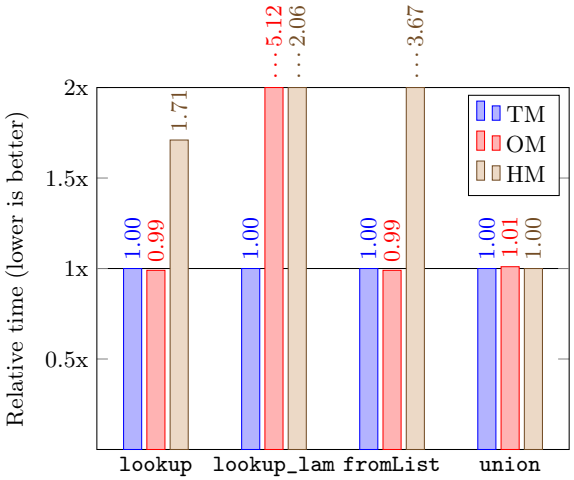


Fig. 3: Benchmarks comparing our trie map (TM) to ordered maps (OM) and hash maps (HM)

traditional map implementations on some operations. Not bad for a data structure that we can also extend to support matching lookup!

We took runtime measurements of the (non-matching) *ExprMap* data structure on a selection of workloads, conducted using the *criterion*[11] benchmarking library[12]. Fig. 3 presents a quick overview of the results.

Appendix A is an extended version of this section, featuring a more in-depth analysis, finer runtime as well as space measurements and indicators for statistical significance.

Setup All benchmarks except `fromList` are handed a pre-built map containing 10000 expressions, each consisting of roughly 100 *Expr* data constructors drawn from a pseudo-random source with a fixed (and thus deterministic) seed.

We compare three different non-matching map implementations, simply because we were not aware of other map data structures with matching lookup modulo α-equivalence and we wanted to compare apples to apples. The *ExprMap* forms the baseline. Asymptotics are given with respect to map size n and key expression size k:

[11] `https://hackage.haskell.org/package/criterion`

[12] The benchmark machine runs Ubuntu 18.04 on an Intel Core i5-8500 with 16GB RAM. All programs were compiled on GHC 9.0.2 with `-O2 -fproc-alignment=64` to eliminate code layout flukes and run with `+RTS -A128M -RTS` for 128MB space in generation 0 in order to prevent major GCs from skewing the results.

- *ExprMap* (designated "TM" in Fig. 3) is the trie map implementation from this paper. Insertion and lookup perform at most one full traversal of the key, so performance should scale with $\mathcal{O}(k)$.
- *Map Expr* (designated "OM") is the ordered map implementation from the mature, well-optimised *containers*[13] library. It uses size balanced trees under the hood [Adams(1993)]. Thus, lookup and insert operations incur an additional log factor in the map size n, for a total of $\mathcal{O}(k \log n)$ factor compared to both other maps.
- *HashMap Expr* (designated "HM") is an implementation of hash array mapped tries [Bagwell(2001)] from the *unordered-containers*[14] library. Like *ExprMap*, map access incurs a full traversal of the key to compute a hash and then a $\mathcal{O}(\log_{32} n)$ lookup in the array mapped trie, as well as an expected constant number of key comparisons to resolve collisions. The log factor can be treated like a constant for all intents and purposes, so lookup and insert is effectively in $\mathcal{O}(k)$.

Some clarification as to what our benchmarks measure:

- The `lookup` benchmark looks up every expression that is part of the map. So for a map of size 10000, we perform 10000 lookups of expressions each of which have approximately size 100.
- `lookup_lam` is like `lookup`, but wraps a shared prefix of 100 layers of (*Lam* `"$"`) around each expression.
- `fromList` benchmarks a naïve *fromList* implementation on *ExprMap* against the tuned *fromList* implementations of the other maps, measuring map creation performance from batches.

Querying The results suggest that *ExprMap* is about as fast as *Map Expr* for completely random expressions in `lookup`. But in a more realistic scenario, at least some expressions share a common prefix, which is what `lookup_lam` embodies. There we can see that *ExprMap* wins against *Map Expr* by a huge margin: *ExprMap* looks at the shared prefix exactly once on lookup, while *Map* has to traverse the shared prefix of length $\mathcal{O}(k)$ on each of its $\mathcal{O}(\log n)$ comparisons.

Although *HashMap* loses on most benchmarks compared to *ExprMap* and *Map*, most measurements were consistently at most a factor of two slower than *ExprMap*. We believe that is due to the fact that it is enough to traverse the *Expr* twice during lookup barring any collisions (hash and then equate with the match), thus it is expected to scale similarly as *ExprMap*. Thus, both *ExprMap* and *HashMap* perform much more consistently than *Map*.

Modification While *ExprMap* consistently wins in query performance, the edge is melting into insignificance for `fromList` and `union`. One reason is the uniform distribution of expressions in these benchmarks, which favors *Map*. Still, it is a

[13] `https://hackage.haskell.org/package/containers`
[14] `https://hackage.haskell.org/package/unordered-containers`

surprise that the naïve *fromList* implementations of *ExprMap* and *Map* as list folds beat the one of *HashMap*, although the latter has a tricky, performance-optimised implementation using transient mutability.

What would a non-naïve version of *fromList* for *ExprMap* look like? Perhaps the process could be sped up considerably by partitioning the input list according to the different fields of *ExprMap* and then calling the *fromList* implementations of the individual fields in turn. The process would be very similar to discrimination sort [Henglein(2008)], which is a generalisation of radix sort to tree-like data and very close to tries. Indeed, the *discrimination*[15] library provides such an optimised $\mathcal{O}(n)$ *toMap* implementation for *Map*.

7 Related work

7.1 Matching triemaps in automated reasoning

Matching triemaps, a kind of *term index*, have been used in the automated reasoning community for decades. An automated reasoning system has hundreds or thousands of axioms, each of which is quantified over some variables (just like the RULEs described in Section 2.2). Each of these axioms might apply at any sub-tree of the term under consideration, so efficient matching of many axioms is absolutely central to the performance of these systems.

This led to a great deal of work on so-called *discrimination trees*, starting in the late 1980's, which is beautifully surveyed in the Handbook of Automated Reasoning [Sekar et al.(2001), Chapter 26]. All of this work typically assumes a single, fixed, data type of "first order terms" like this[16]

```
data MKey = Node Fun [MKey]
```

where *Fun* is a function symbol, and each such function symbol has a fixed arity. Discrimination trees are described by imagining a pre-order traversal that (uniquely, since function symbols have fixed arity) converts the *MKey* to a list of type [*Fun*], and treating that as the key. The map is implemented like this:

```
data DTree v = DVal v | DNode (Map Fun DTree)
lookupDT :: [Fun] → DTree v → Maybe v
lookupDT []       (DVal v)    = Just v
lookupDT (f : fs) (DNode m) = case Map.lookup f m of
                                Just dt → lookupDT fs dt
                                Nothing → Nothing
lookupDT _        _           = Nothing
```

Each layer of the tree branches on the first *Fun*, and looks up the rest of the [*Fun*] in the appropriate child. Extending this basic setup with matching is done by some kind of backtracking.

[15] `https://hackage.haskell.org/package/discrimination`

[16] Binders in terms do not seem to be important in these works, although they could be handled fairly easily by a De Bruijn pre-pass.

Discrimination trees are heavily used by theorem provers, such as Coq, Isabelle, and Lean. Moreover, discrimination trees have been further developed in a number of ways. Vampire uses *code trees* which are a compiled form of discrimination tree that stores abstract machine instructions, rather than a data structure at each node of the tree [Voronkov(1995)]. Spass [Weidenbach et al.(2009)] uses *substitution trees* [Graf and Meyer(1996)], a refinement of discrimination trees that can share common *sub-trees* not just common *prefixes*. (It is not clear whether the extra complexity of substitution trees pays its way.) Z3 uses *E-matching code trees*, which solve for matching modulo an ever-growing equality relation, useful in saturation-based theorem provers. All of these techniques except E-matching are surveyed in [Sekar et al.(2001)].

If we applied our ideas to *MKey* we would get a single-field triemap which (just like *lookupDT*) would initially branch on *Fun*, and then go though a chain of *ListMap* constructors (which correspond to the *DNode* above). You have to squint pretty hard — for example, we do the pre-order traversal on the fly — but the net result is very similar, although it is arrived at by entirely different thought process.

Many of the insights of the term indexing world re-appear, in different guise, in our triemaps. For example, when a variable is repeated in a pattern we can eagerly check for equality during the match, or instead gather an equality constraint and check those constraints at the end [Sekar et al.(2001), Section 26.14].

A related application of matching tries appear in [Perera et al.(2022), Section 2.2], where *eliminators* express both parameter-binding and pattern-matching in a single Core language construct, with a semantics not unlike GHC's own `-XLambdaCase` extension. They realise that their big-step interpreter implements eliminators via special generalised tries that can express variable matching – which corresponds to our triemaps applied to linear patterns.

7.2 Haskell triemaps

Trie data structures have found their way into numerous Haskell packages over time. There are trie data structures that are specific to *String*, like the *StringMap*[17] package, or polymorphically, requiring just a type class for trie key extraction, like the *TrieMap*[18] package. None of these libraries describe how to index on expression data structures modulo α-equivalence or how to perform matching lookup.

Memoisation has been a prominent application of tries in Haskell [Hinze(2000b),Elliott(2008b),Elliott(2008a)] Given a function f, the idea is to build an *infinite*, lazily-evaluated trie, that maps every possible argument x to (a thunk for) $(f\ x)$. Now, a function call becomes a lookup in the trie. The ideas are implemented in the *MemoTrie*[19] library. For memo tries, operations like alter, insert, union, and fold are all irrelevant: the infinite trie is built once, and then used only for lookup.

[17] `https://hackage.haskell.org/package/StringMap`
[18] `https://hackage.haskell.org/package/TrieMap`
[19] `https://hackage.haskell.org/package/MemoTrie`

A second strand of work concerns data type generic, or polytypic, approaches to generating tries, which nicely complements the design-pattern approach of this paper (Section 3.8). [Hinze(2000a)] describes the polytypic approach, for possibly parameterised and nested data types in some detail, including the realisation that we need *alter* and *unionWith* in order to define *insert* and *union*. A generalisation of those ideas then led to *functor-combo*[20]. The *representable-tries*[21] library observes that trie maps are representable functors and then vice versa tries to characterise the sub-class of representable functors for which there exists a trie map implementation.

The *twee-lib*[22] library defines a simple term index data structure based on discrimination trees for the *twee* equation theorem prover. We would arrive at a similar data structure in this paper had we started from an expression data type

data *Expr* = *App Con* [*Expr*] | *Var Var*

In contrast to our *ExprMap*, *twee*'s *Index* does path compression not only for paths ending in leaves (as we do) but also for internal paths, as is common for radix trees.

It is however unclear how to extend *twee*'s *Index* to support α-equivalence, hence we did not consider it for our benchmarks in Section 6.

8 Conclusion

We presented trie maps as an efficient data structure for representing a set of expressions modulo α-equivalence, re-discovering polytypic deriving mechanisms described by [Hinze(2000a)]. Subsequently, we showed how to extend this data structure to make it aware of pattern variables in order to interpret stored expressions as patterns. The key innovation is that the resulting trie map allows efficient matching lookup of a target expression against stored patterns. This pattern store is quite close to discrimination trees [Sekar et al.(2001)], drawing a nice connection to term indexing problems in the automated theorem proving community.

Acknowledgments

We warmly thank Leonardo de Moura and Edward Yang for their very helpful feedback.

References

[Adams(1993)] Stephen Adams. 1993. Functional Pearls Efficient sets—a balancing act. *Journal of Functional Programming* 3, 4 (1993), 553–561. https://doi.org/10.1017/S0956796800000885

[20] https://hackage.haskell.org/package/functor-combo
[21] https://hackage.haskell.org/package/representable-tries
[22] https://hackage.haskell.org/package/twee-lib

[Bagwell(2001)] Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report. Infoscience Department, École Polytechnique Fédérale de Lausanne.

[Chakravarty et al.(2005)] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 241–253. https://doi.org/10.1145/1086365.1086397

[Connelly and Morris(1995)] Richard Connelly and F Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical structure in computer science* 5 (1995), 381–418. Issue 3.

[de Bruijn(1972)] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[Elliott(2008a)] Conal Elliott. 2008a. Composing memo tries. http://conal.net/blog/posts/composing-memo-tries.

[Elliott(2008b)] Conal Elliott. 2008b. Elegant memoization with functional memo tries. http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries.

[Graf and Meyer(1996)] P Graf and C Meyer. 1996. Advanced indexing operations on substitution trees. In *Proc International Conference on Automated Deduction (CADE'96), LNCS 1104*, MA McRobbie and Slaney JK (Eds.). Springer.

[Henglein(2008)] Fritz Henglein. 2008. Generic Discrimination: Sorting and Paritioning Unshared Data in Linear Time. *SIGPLAN Not.* 43, 9 (Sept. 2008), 91–102. https://doi.org/10.1145/1411203.1411220

[Hinze(2000a)] Ralf Hinze. 2000a. Generalizing Generalized Tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. http://journals.cambridge.org/action/displayAbstract?aid=59745

[Hinze(2000b)] Ralf Hinze. 2000b. Memo Functions, Polytypically!. In *Proceedings of the 2nd Workshop on Generic Programming, Lima, Portugal*. 17–32.

[Perera et al.(2022)] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proc. ACM Program. Lang.* 6, POPL, Article 7 (jan 2022), 29 pages. https://doi.org/10.1145/3498668

[Peyton Jones et al.(2001)] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN, ACM. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

[Sekar et al.(2001)] R Sekar, IV Ramakrishnan, and A Voronkov. 2001. *Handbook of Automated Reasoning*. Vol. 2. Elsevier.

[Voronkov(1995)] A Voronkov. 1995. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning* 15 (1995), 237–265. Issue 2.

[Weidenbach et al.(2009)] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS Version 3.5. In *Proc International Conference on Automated Deduction (CADE)*. Springer, 140–145.

The Contributions of Alan Mycroft to Abstract Interpretation

Patrick Cousot

CIMS, CS, New York University

1 Introduction

Abstract interpretation started in the late 70's in Grenoble, France. At that time Rod Burstall was visiting the computer science department (IMAG), and Radhia and I have had numerous conversations (and steak-frites-salade lunches at the GUC (Grenoble University Club) restaurant) with Rod. I suspect that Alan Mycroft got news about abstract interpretation far north in Edinburgh, Scotland, thanks to Rod Burstall supervising his thesis together with Robin Milner. This led to his first publication [15] in 1980 and his thesis "Abstract Interpretation and Optimising Transformations for Applicative Programs [16] in 1982 (his second most cited paper on Google Scholar, not so common for a thesis). His work originated a lot of research on strictness analysis and more generally the static analysis of functional programs, a crucial contribution to the theory, diffusion, and application of abstract interpretation.

Alan Mycroft has had a very productive career with numerous applied and theoretical achievements in various domains of computer science and so my overview of his contributions will be restricted to those explicitly connected to abstract interpretation.

2 Strictness Analysis

The first problem Alan Mycroft solved was to determine statically when a lazy call by need (or call by name) in a side-effect free, higher-order, functional language (where the formal parameter is reevaluated in the context of the call whenever (or the first time) it is used in the function body) by a more efficient call by value (where the formal parameter is evaluated at call time and stored in the context/environment of evaluation of the function body). Because of the absence of side effects, the only difference is when the evaluation of the parameter does not terminate and it is never used in the function body/definition so that call be need/name will terminate while call by value will not, as in `let f x = f x and g y = if true then 0 else y in g (f 0)`. The two call methods are therefore equivalent when the function call evaluated with call by need/name does not terminates when the evaluation of the parameter does not terminate (assuming the "observations" of a functional program execution is its final value or nontermination). Explained in terms of denotational semantics, this is $f(\bot) = \bot$, where $\bot$ denotes nontermination, which is the infimum in Scott domain, meaning that f is strict, hence the term "strictness analysis".

2.1 Alan Mycroft's starting point

The state of the theory of abstract interpretation that Alan Mycroft relied on, was for reachability/invariant verification/analysis of transition systems $\langle \Sigma, \tau \rangle$ [1].

The objective is to infer an invariant $I \in \wp(\Sigma)$ over approximating the reachable states $\{\sigma' \mid \exists \sigma \in P \,.\, \langle \sigma, \sigma' \rangle \in \tau^*\} \subseteq I$ from initial states $P \in \wp(\Sigma)$ where $\tau^* \triangleq \bigcup_{n \in \mathbb{N}} \tau^n = \mathsf{lfp}^{\subseteq} T$ with $T \triangleq \boldsymbol{\lambda} X \bullet \tau^0 \cup \tau \circ X$ is the reflexive transitive closure of $\tau \in \wp(\Sigma \times \Sigma)$, τ^0 is the identity relation on the set of states Σ, and $\tau^{n+1} \triangleq \tau^n \circ \tau$ is the relation power. Define $\mathsf{post}(r)P \triangleq \{\sigma' \mid \exists \sigma \in P \,.\, \langle \sigma, \sigma' \rangle \in r\}$ to be the right-image of the states in P by the relation r on states. We look for an invariant I such that $\mathsf{post}(\mathsf{lfp}^{\subseteq} T) \subseteq I$. The method is explained in proposition 2 thereafter, proofs are relegated to the appendix.

Proposition 1. *Given $P \in \wp(\Sigma)$, we have the Galois connection $\langle \wp(\Sigma \times \Sigma), \subseteq \rangle \underset{\boldsymbol{\lambda} r \bullet \mathit{post}(r) P}{\overset{\gamma_P}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq \rangle$[2].*

Proposition 2. *For all $P \in \wp(\Sigma)$, we have the following commutation property $\boldsymbol{\lambda} r \bullet \mathit{post}(r)P \circ T = F_P \circ \boldsymbol{\lambda} r \bullet \mathit{post}(r)P$ with $T(X) \triangleq \tau^0 \cup \tau \circ X$ and $F_P(X) \triangleq P \cup \mathit{post}(\tau)X$.*

By the fixpoint exact abstraction under this commutation condition, if follows that $\mathsf{post}(\mathsf{lfp}^{\subseteq} T) = \mathsf{lfp}^{\subseteq} F_P$, as shown by the following

Proposition 3. *if $\langle C, \preceq \rangle$ and $\langle A, \preccurlyeq \rangle$ are CPO's (every increasing chain has a lub, including the empty chain, so has an infimum), $f \in C \overset{c}{\longrightarrow} C$ and $\bar{f} \in A \overset{c}{\longrightarrow} A$ are continuous, $\langle C, \preceq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \preccurlyeq \rangle$ is a Galois connection, then the commutation condition $\alpha \circ f = \bar{f} \circ \alpha$ (respectively semi-commutation $\alpha \circ f \mathrel{\dot{\preccurlyeq}} \bar{f} \circ \alpha$, pointwise) implies that $\alpha(\mathsf{lfp}^{\preceq} f) = \mathsf{lfp}^{\preccurlyeq} \bar{f}$ (resp. $\alpha(\mathsf{lfp}^{\preceq} f) \preccurlyeq \mathsf{lfp}^{\preccurlyeq} \bar{f}$).*

The problem is thus to find an invariant I such that $\mathsf{lfp}^{\subseteq} F_P \subseteq I$. It is essential to remark that the computation ordering used for the fixpoint $\mathsf{lfp}^{\subseteq} F_P$ and the logical ordering in $\mathsf{lfp}^{\subseteq} F_P \subseteq I$ to over approximate the reachable states are the same so that the above fixpoint approximate abstraction under the semi-commutation condition is directly applicable

Therefore, by proposition 3, using a Galois connection $\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \leqslant \rangle$, the problem can be reduced to the computation of an abstract invariant $\mathsf{lfp}^{\leqslant} \alpha \circ F_P \circ \gamma$ such that $\mathsf{lfp}^{\subseteq} F_P \subseteq \gamma(\mathsf{lfp}^{\leqslant} \alpha \circ F_P \circ \gamma)$, as desired.

Using a semi-commuting over approximation $\bar{F}_P$ such that $\alpha \circ \bar{F}_P \mathrel{\dot{\leqslant}} F_P \circ \alpha$ is also feasible since then $\mathsf{lfp}^{\subseteq} F_P \subseteq \gamma(\mathsf{lfp}^{\leqslant} \bar{F}_P)$, by proposition 3.

---

[1] Alan and his followers refer to "flowchart abstract interpretation" whereas in my thesis and POPL79, I had moved from flowcharts to transition systems for conciseness.

[2] $\langle C, \preceq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \preccurlyeq \rangle$ denotes the fact that $\langle C, \preceq \rangle$ and $\langle A, \preccurlyeq \rangle$ are posets, $\alpha \in C \longrightarrow A$, $\gamma \in A \longrightarrow C$, and $\forall x \in C, y \in A \,.\, \alpha(x) \preccurlyeq y \Leftrightarrow x \preceq \gamma(y)$.

2.2 Alan Mycroft's pioneer strictness analysis

Alan Mycroft formulates strictness analysis by first defining the denotational semantics of the (recursive) function as a fixpoint $\mathsf{lfp}^{\sqsubseteq} F \in \mathcal{D}_\perp \xrightarrow{c} \mathcal{D}_\perp$ where the domain $\mathcal{D}_\perp$ is a CPO $\langle \mathcal{D}_\perp, \sqsubseteq, \perp, \sqcup \rangle$ with ordering $\sqsubseteq$ and the functional $F \in (\mathcal{D}_\perp \xrightarrow{c} \mathcal{D}_\perp) \xrightarrow{c} (\mathcal{D}_\perp \xrightarrow{c} \mathcal{D}_\perp)$ is continuous. (F is defined on pages 38/39 of his thesis by structural induction on the functional programs he considers. As stated in [16, page 54], $\sqsubseteq$ is taken to be Scott ordering $\perp \sqsubseteq \perp \sqsubset x \sqsubseteq x, x \in D$ on the flat domain $\mathcal{D}_\perp = D \cup \{\perp\}$, $\perp \notin D$).

The collecting semantics of a function with denotational semantics $f \in \mathcal{D}_\perp \xrightarrow{c} \mathcal{D}_\perp$ is $\mathsf{post}(f)P \triangleq \{f(x) \mid x \in P\}$ where $P \in \wp(\mathcal{D}_\perp)$ is a set of possible values of the parameter (including $\perp$ in case of non termination) and $\mathsf{post}(f)P$ provides the possible results of the function for these parameters. Since the considered functions are total and determinitic, the image of a singleton is a singleton. Moreover, post preserves arbitrary joins so that the collecting semantics $\mathsf{post}(f)$ belongs to $\wp(\mathcal{D}_\perp) \xrightarrow{1\cup} \wp(\mathcal{D}_\perp)$ defined as

$$\wp(\mathcal{D}_\perp) \xrightarrow{1\cup} \wp(\mathcal{D}_\perp) \triangleq \{\phi \in \wp(\mathcal{D}_\perp) \longrightarrow \wp(\mathcal{D}_\perp) \mid \forall x \in \mathcal{D}_\perp \,.\, |\phi(\{x\})| = 1 \wedge \forall X \in \wp(\wp(\mathcal{D}_\perp)) \,.\, \mathsf{post}(f) \textstyle\bigcup_{P\in X} P = \bigcup_{P\in X} \mathsf{post}(f)P\}$$

as well as to $\mathsf{post}(f) \in \wp(\mathcal{D}_\perp) \setminus \{\emptyset\} \xrightarrow{1\cup} \wp(\mathcal{D}_\perp) \setminus \{\emptyset\}$ since $\mathsf{post}(f)X = \emptyset$ if and only if $X = \emptyset$.

Proposition 4. *We have the Galois connection* $\langle \mathcal{D}_\perp \longrightarrow \mathcal{D}_\perp, \dot{\sqsubseteq} \rangle \underset{\mathit{post}}{\overset{\hat{\gamma}}{\longleftrightarrow}} \langle \wp(\mathcal{D}_\perp) \setminus \{\emptyset\} \xrightarrow{1\cup} \wp(\mathcal{D}_\perp) \setminus \{\emptyset\}, \dot{\hat{\sqsubseteq}} \rangle$ *where* $\hat{\gamma}(\phi) \triangleq \boldsymbol{\lambda} x \bullet \mathit{let}\ \{y\} = \phi(\{x\})\ \mathit{in}\ y$ *and* $\hat{\sqsubseteq}$ *is Egli-Milner ordering* $X \hat{\sqsubseteq} Y \triangleq (\forall x \in X \,.\, \exists y \in Y \,.\, x \sqsubseteq y \wedge \forall y \in Y \,.\, \exists x \in X \,.\, x \sqsubseteq y)$ *and* $\dot{\hat{\sqsubseteq}}$ *is its pointwise extension.*

In order to characterize $\mathsf{post}(\mathsf{lfp}^{\sqsubseteq} F)$ as a fixpoint, we consider the commutation condition

Proposition 5. $\mathit{post} \circ F = \hat{F} \circ \mathit{post}$ *with* $\hat{F}(\phi)P \triangleq \mathit{post}(F(\hat{\gamma}(\phi)))P$.

By propositions 4, 5, and 3, we have

$$\mathsf{post}(\mathsf{lfp}^{\dot{\sqsubseteq}} F) = \mathsf{lfp}^{\dot{\hat{\sqsubseteq}}} \hat{F} \tag{1}$$

(where $\hat{F}$ is defined by the equations on top of page 42 of Alan's thesis).

On page 53, Mycroft defines $\alpha^\sharp(S) \triangleq (\!| S = \{\perp\} \,;\, 0 : 1 |\!)$ and $\alpha^\flat(S) \triangleq (\!| \perp \in S \,;\, 0 : 1 |\!)$ so that

Proposition 6. *Let* $\mathbb{B} \triangleq \{0, 1\}$ *and* $\leq$ *be logical implication. Then* $\langle \wp(D_\perp) \setminus \{\emptyset\}, \subseteq \rangle \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\longleftrightarrow}} \langle \mathbb{B}, \leq \rangle$ *and* $\langle \wp(D_\perp), \subseteq \rangle \underset{\alpha^\flat}{\overset{\gamma^\flat}{\longleftrightarrow}} \langle \mathbb{B}, \leq \rangle$ *with* $\gamma^\sharp(b) = (\!| b = 0 \,;\, \{\perp\} : D_\perp |\!)$, $\gamma^\flat(b) = (\!| b = 1 \,;\, D : D_\perp |\!)$, *and* $0 < 1$.

Observe that $\langle \wp(D_\bot) \setminus \{\hat{\sqsubseteq}\}, \subseteq\rangle \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \langle \mathbb{B}, \leq\rangle$ does not hold since e.g. for Scott ordering, we have $\alpha^\sharp(S) \leq 1$ but not $S \hat{\subseteq} \gamma^\sharp(1) = D_\bot$ since for $S \subset D$ and $y \in D \setminus S \subseteq D_\bot$ we don't have $\exists x \in S \,.\, x \sqsubseteq y$ since for Scott ordering the only possibility is $x = y \notin S$. So proposition 3 is not applicable to express $\alpha^\sharp(\mathsf{post}(\mathsf{lfp}^{\dot{\sqsubseteq}} F) = \alpha^\sharp(\mathsf{lfp}^{\dot{\hat{\sqsubseteq}}} \hat{F}))$ as a fixpoint. This is the main difficulty Alan Mycroft had to solve with the theory of abstract interpretation, as available as the time. The required generalization of proposition 3 is the following

Proposition 7. *Let $\langle C, \bot, \sqsubseteq, \sqcup\rangle$ be a concrete CPO for the computational ordering $\sqsubseteq$ and $f \in C \xrightarrow{c} C$ be continuous. Let $\langle C, \leq\rangle$ be a poset for the approximation ordering $\leq$.*

Let $\langle A, \bot^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp\rangle$ be an abstract CPO and $f^\sharp \in A \xrightarrow{c} A$ be continuous.

Let $\langle C, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq^\sharp\rangle$ be an abstraction such that

$$\bot \leq \gamma(\bot^\sharp) \tag{2}$$

$$\forall x \in C, y \in A \,.\, (x \leq \gamma(y)) \Rightarrow (f(x) \leq \gamma(f^\sharp(y)) \tag{3}$$

for all increasing chains $\langle x_i,\, i \in \mathbb{N}\rangle$ for $\sqsubseteq$ and $\langle y_i,\, i \in \mathbb{N}\rangle$ for $\sqsubseteq^\sharp$.

$$(\forall i \in \mathbb{N} \,.\, x_i \leq \gamma(y_i)) \Rightarrow \bigsqcup_{i\in\mathbb{N}} x_i \leq \gamma(\bigsqcup_{j\in\mathbb{N}}^\sharp y_j) \tag{4}$$

Then $\mathsf{lfp}^{\sqsubseteq} f \leq \gamma(\mathsf{lfp}^{\sqsubseteq^\sharp} f^\sharp)$.

Notice that in proposition 7, the computational ordering $\sqsubseteq$ and the approximation ordering $\leq$ may differ, whereas in proposition 3 they must be the same. This solves Alan problem for strictness analysis. Define $\overrightarrow{\alpha}^\sharp(f) \triangleq \alpha^\sharp \circ f \circ \gamma^\sharp$ and $\overrightarrow{\gamma}^\sharp(f) \triangleq \gamma^\sharp \circ f \circ \alpha^\sharp$ so that

$$\langle \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot),\, \dot{\subseteq}\rangle \underset{\overrightarrow{\alpha}^\sharp}{\overset{\overrightarrow{\gamma}^\sharp}{\leftrightarrows}} \langle \mathbb{B} \xrightarrow{i} \mathbb{B},\, \dot{\leq}\rangle \tag{5}$$

where $\mathbb{B} \xrightarrow{i} \mathbb{B}$ is the set of $\leq$-increasing Boolean functions. Define $\hat{F}^\sharp \triangleq \overrightarrow{\alpha}^\sharp \circ \hat{F} \circ \overrightarrow{\gamma}^\sharp$ so that the hypotheses (2), (3), and (4) of proposition 7 are satisfied with $C = \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot)$, $\sqsubseteq = \dot{\subseteq}$, $A = \mathbb{B} \xrightarrow{i} \mathbb{B}$, $\sqsubseteq^\sharp = \dot{\leq}$, $\leq = \dot{\subseteq}$, $\alpha = \overrightarrow{\alpha}^\sharp$, and $\gamma = \overrightarrow{\gamma}^\sharp$. Proposition 7 applies to $\hat{F}^\sharp$.

Proposition 8. $\mathsf{lfp}^{\dot{\hat{\sqsubseteq}}} \hat{F} \,\dot{\subseteq}\, \overrightarrow{\gamma}^\sharp(\mathsf{lfp}^{\dot{\leq}} \hat{F}^\sharp)$.

Therefore by (1) and proposition 8, we have $\mathsf{post}(\mathsf{lfp}^{\dot{\sqsubseteq}} F) = \mathsf{lfp}^{\dot{\hat{\sqsubseteq}}} \hat{F} \,\dot{\subseteq}\, \overrightarrow{\gamma}^\sharp(\mathsf{lfp}^{\dot{\leq}} \hat{F}^\sharp)$. Since $\hat{F}^\sharp$ operates on a finite domain, $f^\sharp = \mathsf{lfp}^{\dot{\leq}} \hat{F}^\sharp$ is computable for any functional program. Assume that $f^\sharp(0) = 0$. Then $\mathsf{post}(\mathsf{lfp}^{\dot{\sqsubseteq}} F)\{\bot\} \subseteq \gamma^\sharp(0) = \{\bot\}$, proving strictness $F(\bot) = \bot$. Mycroft's strictness analysis method is sound (and also incomplete by Rice's theorem).

Alan applies the same approach to the lower abstraction $\alpha^\flat$ but this is of limited applicability since function nontermination be can proved with this abstraction only when it does not depend upon the values of the parameters. However,

it is anticipating the present-day interest in under approximation verification and static analysis!

For functions with multiple parameters, Mycroft uses a Cartesian abstraction in the collecting semantics [16, top of page 42] which is more efficient but less precise that a relational analysis.

The thesis goes on in chapter 4 to show that call-by-need can be replaced by call-by-value for strict functions, see also [15].

2.3 The large body of research on strictness analysis in the 1980/90s

Alan Mycroft strictness analysis originated an enormous amount of work on the subject in the 80's and early 90's, see e.g. [1]. Strictness analysis has routinely found its way in modern compilers for lazy purely functional languages such as Haskell[3].

Although purely Boolean, strictness analysis suffers combinatorial explosions at higher-orders, although widenings, which have not been much investigated in this context, might certainly have helped [11].

2.4 Connection between call-by-value and call-by-name

In his thesis, Alan proved that call-by-need strict functions without side effects can be replaced by call-by-value functions. Nearly 40 years later, he came back to the subject, establishing a Galois connection between call-by-name and call-by-value for functions with limited side effects [14,13,12]. The Galois connection is between pre-ordered programs, where programs can be understood as encoding their set-based semantics, which established the link with abstract interpretation.

3 Sharing Analysis

In chapter 5 of his thesis [16], Alan Mycroft considers Lisp-like lists and two versions of LISP, a pure applicative LISP-A (declarative without side effect) and destructive LISP-D (with data structure alteration rplaca, rplacd, nconc, expressed using an explicit deallocation by free). The objective is to analyse declarative programs and optimize them into destructive ones.

To go into more details of LISP-A (we don't consider LISP-D and the transformation of LISP-A into LISP-D described in [16, section 5.8] and proved correct in [16, section 5.9]). Let $\mathbb{A}$ be a set of atoms, $\mathbb{L}$ be a disjoint set of locations and $\mathbb{V} = \mathbb{A} \cup \mathbb{L}$, with $\mathbb{A} \cap \mathbb{L} = \emptyset$ be the set of values. Assume that the memory heap/store is represented by binary directed acyclic graphs (DAGs) $H \in \mathbb{H}$ which are sets of nodes/cells $\langle v,\ v_h,\ v_t,\ f\rangle \in H$ such that $v \in \mathbb{L}$ is the location of the node, $v_h, v_t \in \mathbb{V}$ are the respective left/car/head and right/cdr/tail

[3] e.g. in the open source compiler and interactive environment GHC wiki.haskell.org/Performance/Strictness or the strcitness analysis of The Helium Compiler for a subset of Haskell.

values of the node, and $f \in \mathbb{B}$ records whether the node has been explicitly freed. The locations uniquely identify nodes in that if $H \ni \langle v,\ v_h,\ v_t,\ f\rangle \neq \langle v',\ v'_h,\ v'_t,\ f'\rangle \in H$ then $v \neq v'$. If $v \in \mathbb{L}$, we define the head $\mathsf{h}(v) = v_h$ and the tail $\mathsf{t}(v) = v_t$, this is an error if $v_h, v_t \in \mathbb{A}$ or $f = \mathtt{true}$. The locations allocated in the heap H are $\mathsf{L}(H) \triangleq \{v \mid \langle v,\ v_h,\ v_t,\ f\rangle \in H\}$. The roots of the $\mathsf{R}(H)$ graph H have no predecessors $\mathsf{R}(H) \triangleq \{v \in \mathsf{L}(H) \mid \forall v', v'', f\ .\ \langle v',\ v,\ v'',\ f\rangle \notin H \wedge \langle v',\ v'',\ v,\ f\rangle \notin H\}$. The heap/graph has no cycles, meaning that if $v_1 \ldots v_n \in \mathsf{L}(H)^n,\ n \geqslant 0$ is any sequence of heap locations such that $v_{i+1} \in \{\mathsf{h}(v_i), \mathsf{t}(v_i)\},\ i \in [1, n]$, then $\forall 0 \leqslant i < j \leqslant n\ .\ v_i \neq v_j$. The construction operation c (cons) is $\mathsf{c}(v_h, v_t)H \triangleq \mathsf{let}\ v \notin \mathsf{L}(H)\ \mathsf{in}\ H \cup \{\langle v,\ v_h,\ v_t,\ \mathsf{false}\rangle\}$ where $v_h, v_t \in \mathbb{A} \cup \mathsf{L}(H)$ are atoms or locations of nodes allocated in H. It follows that $\mathsf{h}(\mathsf{c}(v_h, v_t)) = v_h$ and $\mathsf{t}(\mathsf{c}(v_h, v_t)) = v_t$. The considered language [16, page 154] is a functional language with a denotation for atoms, primitives including c, h, t, free, atom (checking for atomicity). The fixpoint denotational semantics of a function denotation [16, Section 5.12] for LISP-D is a function f taking the value $v \in \mathsf{A} \cup \mathsf{L}(H)$ of the actual parameter and a memory heap H as a parameter and returning the possibly modified heap H' and a returned value $v' \in \mathsf{A} \cup \mathsf{L}(H')$ or $\perp$ in case of non termination (so $f \in \mathbb{L} \times \mathbb{H} \to \mathbb{L} \cup \{\perp\} \times \mathbb{H}$).

Alan looks for "an approximation to the set of paths which will actually exist at run time, but as usual in abstract interpretation (see chapter 2) the paths we infer will be a superset of those which can occur at run time". He claims [16, Section 5.7, page 134] that his abstraction was inspired and generalizes the isolation classes of Jacob T. Schwarz[4], to provide information on "how shared an object might be".

Given a heap $H \in \mathbb{H}$, $v, v' \in \mathsf{L}(H)$, the paths $\mathit{\Pi}(H)\langle v,\ v'\rangle$ from v to v' are

$$\mathit{\Pi}(H)\langle v,\ v'\rangle \triangleq \{x_0 \ldots x_n \in \mathsf{L}(H)^n \mid x_0 = v \wedge \forall i \in [0, n[\ .\ x_{i+1} \in \{\mathsf{h}(x_i), \mathsf{t}(x_i)\} \wedge x_n = v'\}$$

$\mathit{\Pi}(H)\langle v,\ v'\rangle$ is empty when v or v' is an atom (including erroneous h and t). Let $O(H)$ be the set of locations on the heap H reachable from the roots of H through heads and tails by one path only.

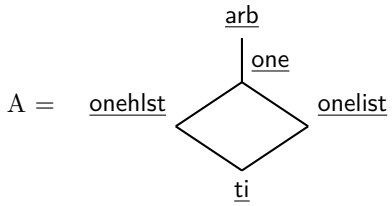
$$O(H) \triangleq \{v \in \mathbb{A} \cup \mathsf{L}(H) \cup \{\perp\} \mid (v \in \mathsf{L}(H)) \Rightarrow (\forall v' \in \mathsf{R}(H)\ .\ |\mathit{\Pi}(H)\langle v',\ v\rangle| = 1)\}$$

(where $|S|$ is the cardinality of a set S). $\mathit{\Delta}_{\mathsf{h}}(H)v$ (respectively $\mathit{\Delta}_{\mathsf{t}}(H)v$, $\mathit{\Delta}(H)v$) is the set of descendants of location $v \in \mathsf{L}(H)$ in the heap H going exclusively through heads (resp. through tails only, through heads or tails).

$$\mathit{\Delta}_{\mathsf{h}}(H)v \triangleq \{v' \mid \exists x_0 \ldots x_n \in \mathsf{L}(H)^n\ .\ x_0 = v \wedge \forall i \in [0, n[\ .\ x_{i+1} = \mathsf{h}(x_i) \wedge x_n = v'\}$$
$$\mathit{\Delta}_{\mathsf{t}}(H)v \triangleq \{v' \mid \exists x_0 \ldots x_n \in \mathsf{L}(H)^n\ .\ x_0 = v \wedge \forall i \in [0, n[\ .\ x_{i+1} = \mathsf{t}(x_i) \wedge x_n = v'\}$$
$$\mathit{\Delta}(H)v \triangleq \{v' \mid \mathit{\Pi}(H)\langle v,\ v'\rangle \neq \emptyset\}$$

[4] citing Schwarz, J. *Verifying the safe use of destructive operations in applicative programs.* Program Transformations - Proc. of the 3rd Int'l Symp. on Programming, Dunod Informatique, 1978, pp. 395–411, also DAI research report 55, Dept. of Artificial Intelligence, Edinburgh University, published while Jacob was visiting Edinburgh.

The abstract domain is the (complete) lattice



The meaning of the abstract values (called "isolation classes") is as follows.

- The supremum $\underline{\mathsf{arb}}$ can denote any atom (including error), element on the heap, or non termination. $\gamma(\underline{\mathsf{arb}}) \triangleq \{\langle v,\ H\rangle \mid H \in \mathbb{H} \wedge v \in \mathbb{A} \cup \mathsf{L}(H) \cup \{\bot\}\}$.
- The abstract value $\underline{\mathsf{one}}$ can denote any atom, non termination, or element on the heap accessible from the roots of the heap by one path only, so, "the object described [by $\underline{\mathsf{one}}$] cannot be a shared CONS node" [16, page 138]. $\gamma(\underline{\mathsf{one}}) \triangleq \{\langle v,\ H\rangle \mid H \in \mathbb{H} \wedge v \in O(H)\}$.
- The abstract value $\underline{\mathsf{onehlst}}$ can denote any atom, non termination, or element on the heap accessible from the roots of the heap by one path only, and such that all its descendants by the head h are not accessible from the roots in any other way. $\gamma(\underline{\mathsf{onehlst}}) \triangleq \{\langle v,\ H\rangle \mid \forall v' \in \Delta_{\mathsf{h}}(H)v\ .\ v' \in O(H)\}$ (this includes $v' = v$).
- The abstract value $\underline{\mathsf{onelist}}$ is similar, but for tails only. So these nodes are uniquely accessible from the roots of H and so are all their descendants through tails only. $\gamma(\underline{\mathsf{onelist}}) \triangleq \{\langle v,\ H\rangle \mid \forall v' \in \Delta_{\mathsf{t}}(H)v\ .\ v' \in O(H)\}$
- The infimum $\underline{\mathsf{ti}}$ denotes any atom, nontermination, or location on the heap such that all its descendants are accessible form the roots by one path only ("objects totaly unshared from other objects" [16, page 135]). $\gamma(\underline{\mathsf{ti}}) \triangleq \{\langle v,\ H\rangle \mid \forall v' \in \Delta(H)v\ .\ v' \in O(H)\}$

Observe that $\gamma(\underline{\mathsf{onehlst}}) \sqcap \gamma(\underline{\mathsf{onelist}}) = \gamma(\underline{\mathsf{ti}})$ so we have a Galois connection with the abstraction of $P \in \wp(\mathbb{V} \times \mathbb{H})$ such that $\alpha(P) \triangleq \bigsqcap\{\underline{a}) \in A \mid P \subseteq \gamma(\underline{a})\}$. The abstraction is extended to functions of the collecting semantics $f \in \wp(\mathbb{V} \times \mathtt{H}) \xrightarrow{i} \wp(\mathbb{V} \times \mathtt{H})$ by $\alpha(f) \triangleq \alpha \circ f \circ \gamma$. This provides a fixpoint definition of the isolation class of a function in terms of the isolation classes of its parameters and its textual definition" [16, page 140], provided variables are handled correctly, as in [16, section 5.7.5], as roots of the DAG. Since the abstract domain is finite, the abstraction is computable for each subexpression appearing in a program.

3.1 Static analysis of shared data structures

Alan is one of the early users of abstract interpretation[5] for analyzing programs manipulating shared recursive data structures, a complex problem which, with parallelism, is still a hot research subject nowadays.

[5] following Cousot and Cousot, IFIP FDPC, 1978.

4 Main contributions of Alan Mycroft to Static Program Analysis

We have already underlined the pioneer work of Alan Mycroft on strictness analysis in Sect. 2.3 and linked data structure analysis in Sect. 3.1. In this section, we outline other important contributions to static program analysis (excluding dynamic analysis).

4.1 Denotational semantics based abstract interpretation

The work of Alan Mycroft [15,16] originated the used of denotational semantics as a standard semantics, as opposed to operational semantics, for abstract interpretation [20]. "The motivation is that abstract interpretation of denotational language definitions allows approximation of a wide class of programming language properties" [9, section 1.5]. The interest in denotational semantics, which is an abstract interpretation of operational semantics, later declined since it is not expressive enough (e.g. for trace or hyper properties) although basic principles like structural induction on the program syntax have perdured.

4.2 Static Analysis

Alan interest in static program analysis has persisted all along his scientific carrier, including for strictness analysis [3], analysis of procedures and functions [8,19], microcode [17], hardware [27,26], although his concerns in correctness proofs somewhat faded while concentrating on data flow analysis (in which programs are represented by graphs[6] and analysis algorithms are traditionally postulated rather than proved sound with respect to a semantics[7]) [7,6,4]. This informal approach is nevertheless with some exceptions, mainly without reference to a formal semantics [24,5,10].

4.3 Completeness in abstract interpretation

Alan Mycroft was one of the first [18] to develop completeness[8] in abstract interpretation, a subject that has since proliferated. His example of multiplication complete for sign analysis but not addition, is provided as introductory example in (almost) all papers on the subject! Moreover, [18] introduces predicate abstraction, somewhat before the standard reference [9]. Unfortunately fixpoints are postponed to later research. The fixpoint abstraction completeness problem is not yet solved satisfactorily.

[6] ironically close to decried "flowchart abstract interpretation".

[7] for example the classical liveness analysis in [6] is wrong since use is a syntactic over approximation of the semantic notion of use of a variable value, whereas, being negated, it should be an underapproximation, see Patrick Cousot: *Syntactic and Semantic Soundness of Structural Dataflow Analysis*. SAS 2019: 96-117

[8] Cousot and Cousot, POPL, 1979

[9] Susanne Graf, Hassen Saïdi: *Construction of Abstract State Graphs with PVS*. CAV 1997: 72-83

4.4 Types and Effects

Alan Mycroft most cited work is on types and type inference [21] and he has also maintained interest and important contributions on the subject [2,23], including effect systems [22,25]. Alan certainly understood the close link between types and abstract interpretation. An example is his thesis "There is an analogy between the system described here and the "most general type" inference system used in a language such as ML [17]"[10] [16, page 67]. Another is "Control-flow operators also provide a link to abstract interpretation [6]. Primitive effectful operations in the concrete semantics are abstracted to effects in an effect algebra." [22, page 18]. Finally in [18], he writes "Mycroft and Jones [10] manage to exhibit the Hindley-Milner type system as an abstract interpretation of the untyped -calculus but the result is much more unwieldly to use than the inference rule formulation." (where [10] in this citation is our [19]).

He has certainly been too overwhelmed by new ideas to take time to explore this connection in more depth.

5 Conclusion

Starting from his PhD thesis [16], we have taken a rapid tour of Alan Mycroft's publications related to abstract interpretation, where he played an outstanding pioneer role and went on to regularly introduce new inspiring ideas in the field. Personnaly, I would have hoped this he spend more time on the subject since he one of the few who, beyond theory, have a strong interest in applicability and practical applications as his other works in other domains do show.

References

1. Abramsky, S., Hankin, C. (eds.): Abstract Interpretation of Declarative Languages. Ellis Horwood (1987)
2. Dolan, S., Mycroft, A.: Polymorphism, subtyping, and type inference in mlsub. In: POPL. pp. 60–72. ACM (2017)
3. Ernoult, C., Mycroft, A.: Untyped strictness analysis. J. Funct. Program. **5**(1), 37–49 (1995)
4. Feigin, B., Mycroft, A.: Formally efficient program instrumentation. In: RV. Lecture Notes in Computer Science, vol. 6418, pp. 245–252. Springer (2010)
5. Gharat, P.M., Khedker, U.P., Mycroft, A.: Generalized points-to graphs: A precise and scalable abstraction for points-to analysis. ACM Trans. Program. Lang. Syst. **42**(2), 8:1–8:78 (2020)
6. Ivaskovic, A., Mycroft, A., Orchard, D.: Data-flow analyses as effects and graded monads. In: FSCD. LIPIcs, vol. 167, pp. 15:1–15:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)

[10] where [17] refers to Michael J. C. Gordon, Robin Milner, F. Lockwood Morris, Malcolm C. Newey, Christopher P. Wadsworth: *A Metalanguage for Interactive Proof in LCF*. POPL 1978: 119-130.

7. Jaiswal, S., Khedker, U.P., Mycroft, A.: A unified model for context-sensitive program analyses: : The blind men and the elephant. ACM Comput. Surv. **54**(6), 114:1–114:37 (2022)
8. Jones, N.D., Mycroft, A.: Data flow analysis of applicative programs using minimal function graphs. In: POPL. pp. 296–306. ACM Press (1986)
9. Jones, N.D., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of logic in computer science. Volume 4. Semantic modelling, pp. 527–636. Clarendon Press (1995)
10. Khedker, U.P., Dhamdhere, D.M., Mycroft, A.: Bidirectional data flow analysis for type inferencing. Comput. Lang. Syst. Struct. **29**(1-2), 15–44 (2003)
11. Mauborgne, L.: Abstract interpretation using typed decision graphs. Sci. Comput. Program. **31**(1), 91–112 (1998)
12. McDermott, D., Mycroft, A.: Call-by-need effects via coeffects. Open Comput. Sci. **8**(1), 93–108 (2018)
13. McDermott, D., Mycroft, A.: Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In: ESOP. Lecture Notes in Computer Science, vol. 11423, pp. 235–262. Springer (2019)
14. McDermott, D., Mycroft, A.: Galois connecting call-by-value and call-by-name. In: FSCD. LIPIcs, vol. 228, pp. 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
15. Mycroft, A.: The theory and practice of transforming call-by-need into call-by-value. In: Symposium on Programming. Lecture Notes in Computer Science, vol. 83, pp. 269–281. Springer (1980)
16. Mycroft, A.: Abstract interpretation and optimising transformations for applicative programs. Ph.D. thesis, University of Edinburgh, UK (1982)
17. Mycroft, A.: A study on abstract interpretation and 'validating microcode algebraically'. In: Abramsky and Hankin [1], pp. 199–218
18. Mycroft, A.: Completeness and predicate-based abstract interpretation. In: PEPM. pp. 179–185. ACM (1993)
19. Mycroft, A., Jones, N.D.: A relational framework for abstract interpretation. In: Programs as Data Objects. Lecture Notes in Computer Science, vol. 217, pp. 156–171. Springer (1985)
20. Mycroft, A., Nielson, F.: Strong abstract interpretation using power domains (extended abstract). In: ICALP. Lecture Notes in Computer Science, vol. 154, pp. 536–547. Springer (1983)
21. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for prolog. In: Logic Programming Workshop. pp. 107–122. Núcleo de Intelligência Artificial, Universidade Nova De Lisboa, Portugal (1983)
22. Mycroft, A., Orchard, D.A., Petricek, T.: Effect systems revisited - control-flow algebra and semantics. In: Semantics, Logics, and Calculi. Lecture Notes in Computer Science, vol. 9560, pp. 1–32. Springer (2016)
23. Orchard, D.A., Mycroft, A.: Efficient and correct stencil computation via pattern matching and static typing. In: DSL. EPTCS, vol. 66, pp. 68–92 (2011)
24. Rodriguez-Prieto, O., Mycroft, A., Ortin, F.: An efficient and scalable platform for java source code analysis using overlaid graph representations. IEEE Access **8**, 72239–72260 (2020)
25. Schrijvers, T., Mycroft, A.: Strictness meets data flow. In: SAS. Lecture Notes in Computer Science, vol. 6337, pp. 439–454. Springer (2010)

26. Thompson, S., Mycroft, A.: Abstract interpretation of combinational asynchronous circuits. In: SAS. Lecture Notes in Computer Science, vol. 3148, pp. 181–196. Springer (2004)
27. Thompson, S., Mycroft, A.: Abstract interpretation of combinational asynchronous circuits. Sci. Comput. Program. **64**(1), 166–183 (2007)

A Proofs for Section 2 (Strictness Analysis)

Proof (proposition 1).

$$
\begin{aligned}
& \mathsf{post}(r)P \subseteq Q \\
\Leftrightarrow\ & \{\sigma' \mid \exists \sigma \,.\, \sigma \in P \wedge \langle \sigma,\, \sigma' \rangle \in r\} \subseteq Q && \text{(def. post)} \\
\Leftrightarrow\ & \forall \sigma' \,.\, (\exists \sigma \,.\, \sigma \in P \wedge \langle \sigma,\, \sigma' \rangle \in r) \Rightarrow \sigma' \in Q && \text{(def. } \subseteq) \\
\Leftrightarrow\ & \forall \sigma' \,.\, \forall \sigma \,.\, (\sigma \in P \wedge \langle \sigma,\, \sigma' \rangle \in r) \Rightarrow \sigma' \in Q) && \text{(def. } \Rightarrow) \\
\Leftrightarrow\ & \forall \sigma \,.\, \forall \sigma' \,.\, (\langle \sigma,\, \sigma' \rangle \in r) \Rightarrow (\sigma \in P \Rightarrow \sigma' \in Q) && \text{(def. } \forall \text{ and } \Rightarrow) \\
\Leftrightarrow\ & r \subseteq \{\langle \sigma,\, \sigma' \rangle \mid \sigma \in P \Rightarrow \sigma' \in Q\} && \text{(def. } \subseteq) \\
\Leftrightarrow\ & r \subseteq \gamma_P(Q) && \text{(by defining } \gamma_P(Q) = (P \times Q) \cup (\Sigma \setminus P) \times \Sigma) \qquad \square
\end{aligned}
$$

Proof (proposition 2).

$$
\begin{aligned}
& \boldsymbol{\lambda} r \bullet \mathsf{post}(r)P \circ T(X) \\
=\ & \mathsf{post}(T(X))P && \text{(def. function composition } \circ) \\
=\ & \mathsf{post}(\tau^0 \cup \tau \circ X)P && \text{(def. } T) \\
=\ & \mathsf{post}(\tau^0)P \cup \mathsf{post}(\tau \circ X)P && (\mathsf{post} \text{ preserves arbitrary unions}) \\
=\ & P \cup \mathsf{post}(\tau)(\mathsf{post}(X)P) && \text{(def. post)} \\
=\ & P \cup \mathsf{post}(\tau)(\boldsymbol{\lambda} r \bullet \mathsf{post}(r)(X)) && \text{(def. function application)} \\
=\ & F_P(\boldsymbol{\lambda} r \bullet \mathsf{post}(r)(X)) && (\text{ with } F_P(X) \triangleq P \cup \mathsf{post}(\tau)X\ ) \\
=\ & F_P \circ \boldsymbol{\lambda} r \bullet \mathsf{post}(r)(X) && \text{(def. function composition } \circ) \qquad \square
\end{aligned}
$$

Proof (proposition 3). Let f^n and $\bar{f}^n$, $\in \mathbb{N}$ be the iterates of f and $\bar{f}$ from the infima $\bot$ and $\bar{\bot}$. In a Galois connection, α preserves existing arbitrary joins in particular the infimum so $\alpha(f^0) = \alpha(\bot) = \bar{\bot} = \bar{f}^0$. Assume $\alpha(f^n) = \bar{f}^n$ by induction hypothesis. By the commutation condition, $\alpha(f^{n+1}) = \alpha(f(f^n)) = \bar{f}(\alpha(f^n)) = \bar{f}(\bar{f}^n) = \bar{f}^{n+1}$. By recurrence, $\forall n \in \mathbb{N} \,.\, \alpha(f^n) = \bar{f}^n$. Since f and $\bar{f}$ are continuous, they are increasing (isotone/monotone), so their iterates are increasing, and their limits do exist in the CPO. By continuity $\alpha(\mathsf{lfp}^{\preccurlyeq} f) = \alpha(\bigcup_{n\in\mathbb{N}} f^n) = \bigvee_{n\in\mathbb{N}} \alpha(f^n) = \bigvee_{n\in\mathbb{N}} \bar{f}^n = \mathsf{lfp}^{\preccurlyeq} \bar{f}$. The proof is similar in the inequality case. $\square$

Proof (proposition 4).

$$
\begin{aligned}
& \mathsf{post}(f) \mathbin{\dot{\hat{\sqsubseteq}}} \phi \\
\Leftrightarrow\ & \forall P \in \wp(D_\bot) \,.\, \mathsf{post}(f)P \mathbin{\hat{\sqsubseteq}} \phi(P) && \text{(pointwise def. of } \dot{\hat{\sqsubseteq}}) \\
\Leftrightarrow\ & \forall P \in \wp(D_\bot) \,.\, \{f(x) \mid x \in P\} \mathbin{\hat{\sqsubseteq}} \phi(P) && \text{(def. post)} \\
\Leftrightarrow\ & \forall P \in \wp(D_\bot) \,.\, (\forall x \in \{f(x) \mid x \in P\} \,.\, \exists y \in \phi(P) \,.\, x \sqsubseteq y \wedge \forall y \in \phi(P) \,.\, \exists x \in \{f(x) \mid x \in P\} \,.\, x \sqsubseteq y) && \text{(def. } \hat{\sqsubseteq}) \\
\Leftrightarrow\ & \forall P \in \wp(D_\bot) \,.\, (\forall x \in P \,.\, \exists y \in \phi(P) \,.\, f(x) \sqsubseteq y \wedge \forall y \in \phi(P) \,.\, \exists x \in P \,.\, f(x) \sqsubseteq y) && \text{(def. } \in) \\
\Leftrightarrow\ & \forall x \in D_\bot \,.\, f(x) \sqsubseteq \mathsf{let}\ \{y\} = \phi(\{x\})\ \mathsf{in}\ y
\end{aligned}
$$

⦇($\Rightarrow$) Take $P = \{x\}$, $x \in D_\bot$, then $\exists y \in \phi(\{x\})\ .\ f(x) \sqsubseteq y$. By $\phi \in \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot)$, we have $\{y\} = \phi(\{x\})$ so $f(x) \sqsubseteq \mathsf{let}\ \{y\} = \phi(\{x\})\ \mathsf{in}\ y$.
($\Leftarrow$) Assume that $P \in \wp(D_\bot)$ and $x \in P$. Then $\{y\} = \phi(\{x\}) \subseteq \bigcup_{z\in P} \phi(\{z\}) = \phi(\bigcup_{z\in P}\{z\}) = \phi(P)$ proving $\exists y \in \phi(P)\ .\ f(x) \sqsubseteq y$.

Moreover, take any $y \in \phi(P) = \phi(\bigcup_{x\in P}\{x\}) = \bigcup_{x\in P}\phi(\{x\})$, there exists $x \in P$ such that $y \in \phi(\{x\})$, that is $\{y\} = \phi(\{x\})$ since $\phi(\{x\})$ is a singleton, proving $f(x) \sqsubseteq y$ by hypothesis.⦈

$\Leftrightarrow f \mathrel{\dot\sqsubseteq} \boldsymbol{\lambda} x \bullet \mathsf{let}\ \{y\} = \phi(\{x\})\ \mathsf{in}\ y$ ⦇pointwise def. $\dot\sqsubseteq$⦈

$\Leftrightarrow f \mathrel{\dot\sqsubseteq} \hat\gamma(\phi)$ ⦇def. $\hat\gamma$⦈

It follows that $\langle \mathcal{D}_\bot \longrightarrow \mathcal{D}_\bot, \dot\sqsubseteq\rangle \underset{\mathsf{post}}{\overset{\hat\gamma}{\leftrightarrows}} \langle \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot), \dot{\hat\sqsubseteq}\rangle$ so that by restriction to the continuous functions, we also have $\langle \mathcal{D}_\bot \xrightarrow{c} \mathcal{D}_\bot, \dot\sqsubseteq\rangle \underset{\mathsf{post}}{\overset{\hat\gamma}{\leftrightarrows}} \langle \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot), \dot{\hat\sqsubseteq}\rangle$. Moreover $\mathsf{post}(f)X = \emptyset$ if and only if $X = \emptyset$ so that $\langle \mathcal{D}_\bot \xrightarrow{c} \mathcal{D}_\bot, \dot\sqsubseteq\rangle \underset{\mathsf{post}}{\overset{\hat\gamma}{\leftrightarrows}} \langle \wp(\mathcal{D}_\bot \setminus \{\emptyset\}) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot \setminus \{\emptyset\}), \dot{\hat\sqsubseteq}\rangle$. □

Proof (proposition 5). Given $F \in (\mathcal{D}_\bot \longrightarrow \mathcal{D}_\bot) \longrightarrow (\mathcal{D}_\bot \longrightarrow \mathcal{D}_\bot)$ and $f \in \mathcal{D}_\bot \longrightarrow \mathcal{D}_\bot$, we have

$\mathsf{post}(F(f))P$
$= \mathsf{post}(F(\boldsymbol{\lambda} x \bullet f(x)))P$ ⦇def. $\boldsymbol{\lambda}$ notation⦈
$= \mathsf{post}(F(\boldsymbol{\lambda} x \bullet \mathsf{let}\ \{y\} = \{f(x)\}\ \mathsf{in}\ y))P$ ⦇def. singleton equality⦈
$= \mathsf{post}(F(\boldsymbol{\lambda} x \bullet \mathsf{let}\ \{y\} = \{f(z) \mid z \in \{x\}\}\ \mathsf{in}\ y))P$ ⦇def. $\in$⦈
$= \mathsf{post}(F(\boldsymbol{\lambda} x \bullet \mathsf{let}\ \{y\} = \boldsymbol{\lambda} X \bullet \{f(x) \mid x \in X\}(\{x\})\ \mathsf{in}\ y))P$ ⦇def. application⦈
$= \mathsf{post}(F(\hat\gamma(\boldsymbol{\lambda} X \bullet \{f(x) \mid x \in X\})))P$ ⦇def. $\hat\gamma$⦈
$= \mathsf{post}(F(\hat\gamma(\mathsf{post}(f))))P$ ⦇def. post⦈
$= \hat F(\mathsf{post}(f))P$ ⦇def. $\hat F(\phi)P \triangleq \mathsf{post}(F(\hat\gamma(\phi)))P$⦈ □

Proof (proposition 6). Let $S \in \wp(D_\bot) \setminus \{\emptyset\}$ and $b \in \mathbb{B}$.

– $\alpha^\sharp(S) \le b$
$\Leftrightarrow (\!|\, S = \{\bot\}\ ¿\ 0 : 1 \,|\!) \le b$ ⦇def. $\alpha^\sharp$⦈
$\Leftrightarrow (\!|\, S \subseteq \{\bot\}\ ¿\ 0 : 1 \,|\!) \le b$ ⦇since $S \in \wp(D_\bot) \setminus \{\emptyset\}$⦈
$\Leftrightarrow (b = 0) \Rightarrow (S \subseteq \{\bot\})$ ⦇by cases 0 or 1 for b with $0 \le 0$ and $1 \not\le 0$⦈
$\Leftrightarrow S \subseteq (\!|\, b = 0\ ¿\ \{\bot\} : D_\bot \,|\!)$ ⦇$S \subseteq D_\bot$⦈
$\Leftrightarrow S \subseteq \gamma^\sharp(b)$ ⦇def. $\gamma^\sharp$⦈

– $\alpha^\flat(S) \le b$
$\Leftrightarrow (\!|\, \bot \in S\ ¿\ 0 : 1 \,|\!) \le b$ ⦇def. $\alpha^\flat$⦈
$\Leftrightarrow (b = 0) \Rightarrow (\bot \in S)$ ⦇by cases 0 or 1 for b with $0 \le 0$ and $1 \not\le 0$⦈

$\Leftrightarrow (b = 1) \Rightarrow (\bot \notin S)$ ⦇by cases $b = 0$ or $b = 1$⦈

$\Leftrightarrow S \subseteq (\!| b = 1 \,¿\, D : D_\bot |\!)$ ⦇since $S \subseteq D_\bot$⦈

$\Leftrightarrow S \subseteq \gamma^\flat(b)$ ⦇def. $\gamma^\flat$⦈ □

Proof (proposition 7). Let $\langle f^n,\ n \in \mathbb{N}\rangle$ be the increasing iterates of f from $\bot$ such that $\mathsf{lfp}^{\sqsubseteq} f = \bigsqcup_{n\in\mathbb{N}} f^n$. Let $\langle {f^\sharp}^n,\ n \in \mathbb{N}\rangle$ be the increasing iterates of $f^\sharp$ from $\bot^\sharp$ such that $\mathsf{lfp}^{\sqsubseteq^\sharp} f^\sharp = \bigsqcup^\sharp_{l\in\mathbb{N}} {f^\sharp}^n$. By (2), we have $f^0 = \bot \leq \gamma(\bot^\sharp) = \gamma({f^\sharp}^0)$. Assume by induction hypothesis that $f^n \leq \gamma({f^\sharp}^n)$. Then by (3), we have $f^{n+1} = f(f^n) \leq \gamma(f^\sharp(f^\sharp{}^n)) = \gamma(f^\sharp{}^{n+1})$, proving that $\forall n \in \mathbb{N}\,.\, f^n \leq \gamma({f^\sharp}^n)$. By (4), we have $\mathsf{lfp}^{\sqsubseteq} f = \bigsqcup_{n\in\mathbb{N}} f^n \leq \gamma(\bigsqcup^\sharp_{n\in\mathbb{N}} {f^\sharp}^n) = \gamma(\mathsf{lfp}^{\sqsubseteq^\sharp} f^\sharp)$.[11] □

Proof (of (5)).

$\overrightarrow{\alpha}^\sharp(f) \mathbin{\dot\leq} f^\sharp$

$\Leftrightarrow \forall b \in \mathbb{B}\,.\, \alpha^\sharp \circ f \circ \gamma^\sharp(b) \leq f^\sharp(b)$ ⦇pointwise def. $\dot\subseteq$ and def. $\overrightarrow{\alpha}^\sharp$⦈

$\Leftrightarrow \forall b \in \mathbb{B}\,.\, f \circ \gamma^\sharp(b) \subseteq \gamma^\sharp \circ f^\sharp(b)$ ⦇Galois connection in prop. (6)⦈

$\Leftrightarrow \forall X \in \wp(\mathcal{D}_\bot)\,.\, f(X) \subseteq \gamma^\sharp \circ f^\sharp \circ \alpha^\sharp(X)$

⦇($\Leftarrow$) Take $X = \gamma^\sharp(b)$ and $\gamma^\sharp \circ f^\sharp(b) \subseteq \gamma^\sharp \circ f^\sharp \circ \alpha^\sharp \circ \gamma^\sharp(b)$ since $\alpha^\sharp \circ \gamma^\sharp$ is reductive in $\langle \wp(D_\bot) \setminus \{\emptyset\},\ \subseteq\rangle \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \langle \mathbb{B},\ \leq\rangle$ of prop. 6, and $\gamma^\sharp$ and $f^\sharp(b)$ hence their composition is reductive.

($\Rightarrow$) $\gamma^\sharp \circ \alpha^\sharp$ is expansive in $\langle \wp(D_\bot) \setminus \{\emptyset\},\ \subseteq\rangle \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \langle \mathbb{B},\ \leq\rangle$ of prop. 6, f preserves joins so is increasing, so that taking $b = \alpha^\sharp(X)$ we have $f(X) \subseteq f \circ \gamma^\sharp \circ \alpha^\sharp(X) \subseteq \gamma^\sharp \circ f^\sharp(\alpha^\sharp(X)b)$⦈

$\Leftrightarrow \forall X \in \wp(\mathcal{D}_\bot)\,.\, f(X) \subseteq \overrightarrow{\gamma}^\sharp(f^\sharp)X$ ⦇def. $\overrightarrow{\gamma}^\sharp$⦈

$\Leftrightarrow f \mathbin{\dot\subseteq} \overrightarrow{\gamma}^\sharp(f^\sharp)$ ⦇pointwise def. $\dot\subseteq$⦈ □

Proof (of 8).

– $\bot \leq \overrightarrow{\gamma}^\sharp(\bot^\sharp)$

$\Leftrightarrow \boldsymbol{\lambda} X \bullet \{\bot\} \mathbin{\dot\subseteq} \overrightarrow{\gamma}^\sharp(\boldsymbol{\lambda} x \bullet 0)$ ⦇def. $\bot$ and $\bot^\sharp$⦈

$\Leftrightarrow \forall X \in \wp(\mathcal{D}_\bot)\,.\, \{\bot\} \subseteq \gamma^\sharp \circ \boldsymbol{\lambda} x \bullet 0 \circ \alpha^\sharp(X)$ ⦇pointwise def. of $\dot\subseteq$ and def. $\overrightarrow{\gamma}^\sharp$⦈

$\Leftrightarrow \{\bot\} \subseteq \gamma^\sharp(0)$ ⦇def. function composition $\circ$⦈

$\Leftrightarrow \{\bot\} \subseteq \{\bot\}$ ⦇def. $\gamma^\sharp$, which is true, proving (2)⦈

– Let $f \in \wp(\mathcal{D}_\bot) \xrightarrow{1\cup} \wp(\mathcal{D}_\bot)$, $f^\sharp \in \mathbb{B} \xrightarrow{i} \mathbb{B}$ be such that $f \leq \overrightarrow{\gamma}^\sharp(f^\sharp)$. Then

[11] We have not used the Galois connection hypothesis which is useful to rephrase the hypotheses using α. It is also possible to require equality, see Patrick Cousot, Radhia Cousot: *Galois Connection Based Abstract Interpretations for Strictness Analysis (Invited Paper)*. Formal Methods in Programming and Their Applications 1993: 98-127.

$$
\begin{array}{ll}
& \hat{F}(f) \\
\dot{\subseteq} & \hat{F}(\overrightarrow{\gamma}^\sharp(f^\sharp)) \\
& \quad \wr f \leq \overrightarrow{\gamma}^\sharp(f^\sharp) \text{ and the composition } \hat{F}^\sharp \text{ of increasing functions is increasing} \int \\
\dot{\subseteq} & \overrightarrow{\gamma}^\sharp(\overrightarrow{\alpha}^\sharp \circ \hat{F} \circ \overrightarrow{\gamma}^\sharp(f^\sharp)) \\
& \qquad \wr \text{by the Galois connection } \langle \overrightarrow{\alpha}^\sharp,\ \overrightarrow{\gamma}^\sharp \rangle,\ \overrightarrow{\gamma}^\sharp \circ \overrightarrow{\alpha}^\sharp \text{ is extensive} \int \\
= & \overrightarrow{\gamma}^\sharp(\hat{F}^\sharp(f^\sharp)) \ \wr \text{def. } \hat{F}^\sharp \triangleq \overrightarrow{\alpha}^\sharp \circ \hat{F} \circ \overrightarrow{\gamma}^\sharp \text{, proving hypothesis (3) of proposition 7} \int
\end{array}
$$

– Let $\langle f_i,\ i \in \mathbb{N} \rangle$ be an increasing chain for $\dot{\hat{\subseteq}}$ and $\langle f_i^\sharp,\ i \in \mathbb{N} \rangle$ be an increasing chain for $\dot{\leq}$ such that $\forall i \in \mathbb{N}\ .\ f_i \dot{\subseteq} \overrightarrow{\gamma}^\sharp(f_i^\sharp)$. We have

$$
\begin{array}{ll}
& \dot{\bigsqcup\limits_{i \in \mathbb{N}}} f_i \dot{\subseteq} \overrightarrow{\gamma}^\sharp(\dot{\bigvee\limits_{j \in \mathbb{N}}} f_j^\sharp) \\
\Leftrightarrow & \forall X \in \wp(\mathcal{D}_\perp)\ .\ \hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \subseteq \overrightarrow{\gamma}^\sharp(\dot{\bigvee\limits_{j \in \mathbb{N}}} f_j^\sharp)(X) \qquad \wr \text{pointwise def. } \dot{\hat{\subseteq}} \text{ and } \dot{\leq} \int \\
& \qquad \wr \text{proving (4)} \int \\
\Leftrightarrow & \forall X \in \wp(\mathcal{D}_\perp)\ .\ \hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \subseteq \gamma^\sharp \circ (\dot{\bigvee\limits_{j \in \mathbb{N}}} f_j^\sharp) \circ \alpha^\sharp(X) \quad \wr \text{def. } \overrightarrow{\gamma}^\sharp(f) \triangleq \gamma^\sharp \circ f \circ \alpha^\sharp \int \\
\Leftrightarrow & \forall X \in \wp(\mathcal{D}_\perp)\ .\ \hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \subseteq \gamma^\sharp(\bigvee\limits_{j \in \mathbb{N}} f_j^\sharp(\alpha^\sharp(X))) \quad \wr \text{def. composition } \circ \int \qquad (6)
\end{array}
$$

Since $\langle f_j^\sharp,\ j \in \mathbb{N} \rangle$ is $\dot{\leq}$-increasing, $\langle f_j^\sharp(\alpha^\sharp(X)),\ j \in \mathbb{N} \rangle$ is $\leq$-increasing so is either a sequence of 0s or a sequence of 0s followed by 1s. There are two cases.

- If $\langle f_j^\sharp(\alpha^\sharp(X)),\ j \in \mathbb{N} \rangle$ is a sequence of 0s then $\bigvee_{j \in \mathbb{N}} f_j^\sharp(\alpha^\sharp(X)) = 0$. Moreover the hypothesis $\forall i \in \mathbb{N}\ .\ f_i \dot{\subseteq} \overrightarrow{\gamma}^\sharp(f_i^\sharp) = \gamma^\sharp \circ f_i^\sharp \circ \alpha^\sharp$ implies that $\forall i \in \mathbb{N}\ .\ f_i(X) = \gamma^\sharp(f_i^\sharp(\alpha^\sharp(X))) = \gamma^\sharp(0) = \{\perp\}$. Therefore

$$
\begin{array}{ll}
& \qquad \wr (6) \int \\
\Leftrightarrow & \hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \subseteq \gamma^\sharp(0) \qquad \wr \text{case } \langle f_j^\sharp(\alpha^\sharp(X)),\ j \in \mathbb{N} \rangle = \langle 0,\ j \in \mathbb{N} \rangle \int \\
\Leftrightarrow & \hat{\bigsqcup\limits_{i \in \mathbb{N}}} \{\perp\} \subseteq \{\perp\} \qquad \wr \text{def. } \gamma^\sharp(b) = (\!|\, b = 0\ ¿\ \{\perp\} : D_\perp \,|\!) \int \\
\Leftrightarrow & \{\perp\} \subseteq \{\perp\} \\
& \qquad \wr \text{by def. of the least upper bound } \hat{\bigsqcup} \text{ for } \hat{\subseteq}, \text{ proving (4) in this first case} \int
\end{array}
$$

- Otherwise $\langle f_j^\sharp(\alpha^\sharp(X)), j \in \mathbb{N} \rangle$ is a sequence of 0s followed by 1s so $\bigvee_{j \in \mathbb{N}} f_j^\sharp(\alpha^\sharp(X)) = 1$ in (6) and therefore $\hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \subseteq \gamma^\sharp(1) = D_\perp$ since $\hat{\bigsqcup\limits_{i \in \mathbb{N}}} f_i(X) \in \wp(D_\perp)$ by def. of the lub in the poset $\langle \wp(D_\perp), \hat{\subseteq} \rangle$. Again (4) holds in this second case. □