1 Research Statement: Functional Foundations for Scalable Heterogeneous Systems

In the last decade, there has been a widespread adoption of both multicore and cloud computing. Multicore processors have become the norm in mobile, desktop and enterprise computing. Cloud computing has paved the way for companies to rent farms of such multicore processors on a pay-per-use basis, with the ability to dynamically scale on demand. Simultaneously, the decreasing hardware cost, miniaturisation and latency concerns have pushed computation closer to the edge. Emerging applications such as Internet-of-things (IoT), Augmented Reality (AR) and self-driving vehicles have necessitated moving computation closer to the data for real-time decision making, but also depend on cloud computing for machine learning models used for making those decisions.

Increasingly, the compute platforms are also heterogeneous incorporating many-core general-purpose processors with Graphics Processing Units (GPU) for massively data parallel computations and Field Programmable Gate Arrays (FPGA) for custom hardware acceleration. For example, Amazon Elastic Compute Cloud (EC2) offers multicore compute instances with up to 64 virtual CPUs, FPGAs and up to 8 GPUs each pairing 5100 NVIDIA CUDA cores and 640 Tensor cores in the same instance. On the other end of the spectrum, Apple's new A11 processor incorporates 2 high-performance cores, 4 energy-efficient cores and a GPU in the same chip. These trends indicate that the future compute platforms will be highly scalable and heterogeneous.

As a result, there is a growing proportion of developers who must tackle the complexity of programming for a cloud of heterogeneous multicore processors. In particular, they must ensure correct application behaviour in the face of asynchrony, heterogeneity, concurrency, and partial failures, all the while providing good scalability as well as minimising the user's perception of latency. Traditionally, developers have relied on the infrastructure providing strong consistency guarantees such as sequential consistency, linearizability and distributed transactions in order to build correct applications for scalable platforms. Strong consistency provides a semblance of a single consistent view of memory, where the operations appear to be performed sequentially in some linear order. Unfortunately, providing such strong guarantees in unviable in the face of modern heterogeneous multicore processors and geo-distributed massively scalable services.

Modern multicore processors and concurrent programming languages expose subtle relaxed memory behaviours that arise from hardware and compiler optimisations to the developer. The state of the art distributed stores also resort to weak consistency guarantees, where the different geo-distributed replicas eventually converge to a uniform state at some arbitrary point in the future. In particular, no assumptions can be made about the global state of the system at any particular time. Such weakly consistent systems make a well-known trade-off: developers avoid the cost associated with latency, contention and availability to achieve strong consistency, in exchange for weaker consistency guarantees about the data; the semblance of a uniform consistent view of memory is lost. Thus, as we head into heterogeneous many-core cloud of loosely connected computers, the existing ad hoc programming models and systems are simply not equipped for writing correct and scalable concurrent programs.

Research Goal

My research vision is to enable the construction of reliable, global-scale, low-latency heterogeneous distributed systems by developing mathematically rigorous functional programming platform. In particular, the goal is to build programming language abstractions to shield the developer from unintuitive behaviours and bugs that might result from concurrent program execution on modern multicore processors and heterogeneous geo-distributed compute clouds.

Progress to date

Programming next-generation multicore hardware. During my PhD studies, I lead the MultiMLton project [J5,C4,C5,C6,C7,C9,W8,W10,W11], a parallel extension of the MLton Standard ML compiler, targeted at future many core processors. In MultiMLton, concurrent programs were organised as a large number of cooperative lightweight thread, that communicated by passing messages between each other. I developed a novel communication abstraction

for composable asynchronous communications [C7] that allowed MultiMLton to scale to the 864-core Azul Vega 3 machine. MultiMLton's multicore garbage collector was designed to minimise inter-core communication. The key innovation was to trade some of the ample concurrency in the source language to offset some of the GC costs [C6].

MultiMLton was designed not only for traditional cache-coherent multicore machines, but also to take advantage of exotic architectures that provided fine-grained control over caches. I developed a port of MultiMLton to the non-cache-coherent Intel Single-chip Cloud Computer (SCC), which preserved the familiar shared memory parallel programming model [C5]. This work took advantage of MultiMLton's ability to statically distinguish mutable and immutable data to manage them in separate cache coherence domains. This work won the Best Paper Award at the Intel Many-core Architecture Community (MARC) Symposium at RWTH, Aachen, Germany. My work on MultiMLton was recognised by Purdue University with the Halstead award for outstanding research in software engineering.

Safe lightweight concurrency substrate. During my PhD studies, I undertook a research internship at Microsoft Research, Cambridge, where I developed a new concurrency substrate for the Glasgow Haskell Compiler (GHC) [J3]. The aim was to describe the thread scheduler, one of the core runtime system components, in Haskell rather than in C as it currently is implemented. This design allows concurrency libraries to be implemented as Haskell libraries, taking advantage of Haskell's type system for describing the libraries. Concurrency primitives and their interactions with the RTS are particularly tricky to specify and reason about. I formalised not only the concurrency substrate primitives, but also their interaction with the RTS components using small step operational semantics. Thus, I bring forth to a research program, a unique mix of formal programming language expertise and strong system implementation skills.

Declarative programming for distributed systems. Unlike parallel programs, geo-distributed web applications often favour high availability (being responsive to user-requests) over strong consistency (always producing the correct result). In response, modern-day replicated databases provide much weaker consistency guarantees than programs running on multicore hardware. Writing correct programs under this setting is further complicated by poorly defined semantics of modern distributed databases. To alleviate the burden of developing correct programs, I developed Quelea [C3], a programming model that associates user program with declarative contracts for their consistency expectation. Quelea utilised recent developments in automated theorem proving tools to automatically and correctly insert the necessary coordination to ensure that application's consistency expectations are preserved.

Current Work: Multicore OCaml

Following my PhD studies, I moved to the University of Cambridge to lead the development of Multicore OCaml project. The goal of the project is to add native support for concurrency and parallelism in OCaml. This research has been supported by Research Fellowships from the Royal Commission for the Exhibition of 1851 and Darwin College, Cambridge.

Concurrency For concurrent programming, I introduced *algebraic effects and their handlers* to Multicore OCaml. Algebraic effect handlers are an abstraction for concurrent programming that generalises idioms such as exceptions, generators, async/await, that are primitively provided by other languages. As a result, we can now unify the myriad of primitives that introduce non-local control-flow in the program under the same roof in Multicore OCaml, obviating the need for any special support. Experimental results show that feature rich, highly scalable web servers written with effect handlers match the performance of industrial-strength concurrent programming libraries in OCaml as well as Go programming language [J1]. Importantly, I also show that efficient effect handlers can be incorporated into an existing programming language without adversely affecting sequential performance.

I have also developed (and proved correct) an alternative compilation mechanism for algebraic effect handlers using continuation passing style [C1]. Such a translation is essential for compiling effect handlers to uncooperative environments such as JavaScript (to run in the browser) and the Java Virtual Machine.

This work has garnered good attention not only from the academic community, but also from the industry. Algebraic effect handlers in OCaml has inspired the new architecture for Facebook's React UI library called the React Fiber. I have also undertaken efforts to popularise algebraic effect handlers for structured concurrent programming. In order to make the ideas more approachable, I ran a tutorial on *Concurrent Programming with Effect Handlers* at Commercial Users of Functional Programming (CUFP) workshop, 2017. The tutorial is freely available and actively

maintained on Github. I am also organising a Dagstuhl Seminar titled *Algebraic Effect go Mainstream* in April 2018 to bring together academics who work on effect handlers and leading programming language implementers.

Parallelism For parallel programming in Multicore OCaml, I have developed expressive composable concurrency libraries. In particular, parallel programs written with low-level operating system mechanisms such as mutexes and condition variables are not composable; individually correct program fragments when put together will fail to work as expected due to deadlocks. Software transactional memory is an alternate abstraction that ensures the composition of two correct parallel program fragments is a correct program, but at the cost of performance even in the case of moderate contention for shared resources between the parallel program threads. To solve this problem, I have developed Reagents [W4] abstraction for Multicore OCaml, that allows program composition in the vein of software transactional memory, but does not compromise performance by taking advantage of hardware support for transactional memory available on modern multicore hardware.

In order to have scalable shared memory parallelism in Multicore OCaml, I have put enormous amount of work in developing a mostly-concurrent collector with private minor heaps. This is design minimises stop-the-world phase where all the cores are stopped running OCaml code and the garbage collector runs. This is garbage collector is particularly suited for latency sensitive programs that OCaml is often used for such as trading systems, user interfaces, and network facing micro-services.

Programs running on multicore processors exhibit non-trivial behaviours due to reordering by modern multicore hardware and compiler optimisations. Languages like C++ and Java have adopted complicated *memory models* which specify which of these *relaxed* behaviours programs may observe. However, these models prescribe global correctness properties, which makes them difficult to program against directly. I have developed new memory model, which is not only simple and allows local reasoning about program fragments, but can also be efficiently realised on modern multicore processors. This memory model not only fits Multicore OCaml but is also a suitable candidate for other safe languages like Java and JavaScript.

Future Work

Building upon my previous and current research focus, my long term vision is to develop foundational functional programming technology to solve problems of concurrency, parallelism and distribution in emerging heterogeneous platforms.

The software artefacts resulting from my current and past research has been made available as free, liberally licensed open-source software. I am also strongly committed to the quality of my software artefacts and reproducibility of research. I will continue this by making all research outputs also open source under similar liberal licensing, well documented and published academically for longer term archival.

Handlers and Memory model for WebAssembly. WebAssembly is a low-level bytecode format for compiling high-level programs such as C++, Java and OCaml to the browser. It is designed to be more efficient than JavaScript in terms of program size and load time, but also provide high-level features such as concurrency, parallelism and garage collection. Every major browser vendor now supports WebAssembly. It is poised to be the safe and managed target platform for applications of the future.

WebAssembly does not yet have a concurrency model. I plan to propose algebraic effect handlers as the concurrency model for WebAssembly. There has been some expression of interest in applying effect handlers to WebAssembly. WebAssembly project provides a unique opportunity for incorporating state of the art concurrent programming abstraction in on of most widely used platform: the web browser. This augurs well with my research goal of making it easy to develop correct concurrent programs.

WebAssembly does not yet support shared memory parallelism. But there is ongoing efforts to add this feature. The new memory model that I have developed would be a good fit for WebAssembly. I plan to evaluate the suitability of my memory model for WebAssembly. If this effort succeeds, it will have an enormous effect on the future of the web platform.

Retrofitting Typed Algebraic Effects Algebraic effects in Multicore OCaml are said to be *unchecked*; the effects of a function are not tracked in the type of the function. As a result, programs performing effects (I/O, nondeterminism, etc.) can fail with a runtime error if the effects are not *handled*. Several research languages like Koka, Links and Frank have incorporated an *effect system* to statically guarantee that effectful programs do not go wrong.

However, retrofitting typed algebraic effects in a full-featured language like OCaml involves overcoming several challenges, most of which are due to OCaml's powerful features for abstraction. I plan to pursue develop typed algebraic effects for Multicore OCaml. Such a system would not only make effectful programs safer and easier to reason about, but also open up opportunities for compiler optimisations.

Metaprogramming for Heterogeneous Systems Metaprogramming is the act of writing programs, which we call metaprograms, that manipulate other programs. Examples of metaprograms are compilers and theorem provers. I will develop the theory and practice of *statically staged metaprogramming* techniques to build programs that do not have a dependency on certain portions of the OCaml runtime, most notably the garbage collector, whose execution can compromise the latency profile of highly-optimised systems.

The overall goal is not to simply run more efficiently on conventional CPUs, but instead to map functional code to foreign hardware architectures that do not resemble existing CPUs and offer domain-specific performance benefits. My target platforms are diverse: FPGAs hosted in the cloud; GPUs for high-performance vector processing; and modern web browsers moving beyond JavaScript towards emerging runtimes such as WebAssembly. I aim to test the hypothesis that functional programming can be used to make systems more future proof in the face of the next few decades-worth of hardware; and to use our discoveries to guide emerging industry standards in this space.

Mergeable Types for Distributed Programming Application logic for distributed databases is usually expressed using high-level, often domain-specific, language abstractions, while data management issues are typically relegated to an opaque monolithic data querying and storage service. While this architecture encourages separation of concerns, it provides little opportunity for synergies between the application and database/storage boundary. To overcome these drawbacks, requires radical rethink of how applications and databases interact.

I am developing the idea of *mergeable types*, a language extension to reconcile the impedance mismatch between the application and the database logic. Mergeable types draw inspiration from distributed version control systems. Distributed state is now viewed in terms of a tree of immutable object versions with application defined merge semantics. This model frees the programmer from having to think of low-level operational notions related to distribution and replication, and therefore does not entail wholesale restructuring of application logic to enable distributed programming.

In general, the problem of applying cutting-edge programming language technology to the challenges of distributed databases has been under explored. I am organising a Shonan meeting in August 2018 to bring together programming language experts and practitioners of cutting-edge data-intensive applications to discuss how we can unify data representation issues across different layers of the application stack to exploit the benefits of program verification and optimisation to realise correctness and performance in the data processing layer.

Summary

My career to date has spanned programming languages, compilers, multicore runtime systems and distributed databases. I have established successful ongoing collaborations with Purdue University, University of Cambridge, University of Edinburgh, Tohoku University, ENS, Jane Street Group and Microsoft Research. I am currently looking for a long-term academic position to drive my research vision forward and build a world-class research group of my own.