





**Université de Montréal**

**Programming tools for intelligent systems**

**with a case study in autonomous robotics**

par

**Breandan Considine**

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Discipline

juin 2019



## **Summary**

---



# Table des matières

---

<b>Summary .....</b>	iii
<b>Liste des tableaux .....</b>	ix
<b>Liste des figures .....</b>	xi
<b>Chapitre 1. Introduction .....</b>	3
1.1. Stages in the software development lifecycle .....	5
1.2. Designing intelligent systems .....	5
1.3. Implementation: Languages and compilers .....	7
1.4. Testing: Verification and validation .....	9
1.5. Software reproducibility and maitenance .....	10
1.5.1. Case Study .....	12
<b>Chapitre 2. Design: Programming tools for robotics .....</b>	15
2.1. Software architecture for robotics application .....	16
2.2. Foundations of a modern IDE .....	17
2.2.1. The parser .....	17
2.2.2. Refactoring .....	17
2.2.3. Running and debugging .....	17
2.3. More ROS Tools .....	17
<b>Chapitre 3. Implementation: languages and compilers .....</b>	19
3.1. Automatic differentiation .....	20

3.2.	Differentiable programming .....	22
3.3.	Static and dynamic languages.....	25
3.4.	Imperative and functional languages .....	25
3.5.	Kotlin .....	26
3.6.	Kotlin $\nabla$ .....	27
3.7.	Usage.....	28
3.8.	Type systems .....	30
3.9.	Shape safety .....	31
3.10.	Testing .....	35
3.11.	Operator overloading .....	36
3.12.	First class functions.....	37
3.13.	Numeric Tower .....	37
3.14.	Algebraic data types.....	38
3.15.	Multiple Dispatch.....	40
3.16.	Extension Functions .....	40
3.17.	Automatic, Symbolic Differentiation .....	41
3.18.	Coroutines.....	41
3.19.	Comparison.....	42
<b>Chapitre 4.</b>	<b>Verification and validation .....</b>	<b>45</b>
4.1.	Background.....	45
<b>Chapitre 5.</b>	<b>Software Maintenance and Reproducibility .....</b>	<b>49</b>

5.1.	Dependency management .....	49
5.2.	Operating systems and virtualization.....	50
5.3.	Containerization .....	51
5.4.	Docker and ROS.....	53
5.5.	Retrospective .....	54
<b>Chapitre 6.</b>	<b>Case study: application for autonomous robotics .....</b>	<b>55</b>
6.1.	Design .....	55
6.2.	Implementation .....	55
6.3.	Verification and validation .....	55
6.4.	Containerization .....	55
<b>Chapitre 7.</b>	<b>Conclusion.....</b>	<b>57</b>
7.1.	Future work .....	57
7.1.1.	Requirements Engineering .....	57
7.1.2.	Continuous Delivery and Continual Learning.....	58
<b>Bibliography</b> .....		<b>59</b>



## Liste des tableaux

---

- 3.1 Two programs, both implementing the function  $f(l_1, l_2) = l_1 \cdot l_2$ ..... 26



## Liste des figures

---

1.1	Royce's original Waterfall model, originally intended to describe the software development process, but the same sequence can be found in most engineering disciplines. We use it to help guide our discussion and frame our work inside of this process model. ....	5
3.1	<i>Differentiable programming</i> includes neural networks, but more broadly, arbitrary differentiable programs which use automatic differentiation and gradient-based optimization to approximate a loss function. <i>Probabilistic programming</i> is a generalization of probabilistic graphical models, and uses various forms of Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function. ....	23
3.2	Adapted from van Merriënboer et al. [2018]. Kotlin $\nabla$ models are data structures constructed by an embedded DSL, which are eagerly optimized and lazily evaluated at runtime. ....	28
3.3	Implicit DFG constructed by the original expression, $z$ in Listing ??....	29
5.1	Virtualization is a very resource expensive proposition. Containerization is cheaper, as it shares a kernel with the host OS. Emulation allows us to emulate hardware as software. Any of these methods can be used in conjunction with any other method. ....	51
5.2	Containers live in user space. By default they are sandboxed from the host OS and sibling containers, but unlike VMs, share a common kernel with each other and the host OS. All system calls are passed through host kernel. ....	52
5.3	Container infrastructure. Left: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and	

emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their architecture, may be run on an ARM device using an accelerator..... 53





# Chapitre 1

---

## Introduction

Intelligent system: *A computer system that uses techniques derived from artificial intelligence, particularly one in which such techniques are central to the operation of the system.*

---

—Wikipedia

Computational complexity is of such concern in computer science that a great deal of the field is dedicated to understanding it through the lens of function analysis and information theory. In software engineering, researchers are primarily interested in the complexity of building software – the digital manifestation of algorithms on physical hardware. One kind of software complexity is the cognitive effort required to understand a program.<sup>1</sup> While today's software is becoming rapidly more intelligent, it shows few signs of becoming more intelligible. Better tools are needed for managing the complexity in building software systems.

*The objective of writing this thesis is to develop methods that reduce the cognitive effort required to build intelligent systems, using developer tools, programming language abstractions, automated testing, and container-based virtualization.*

Broadly speaking, intelligent systems differ from ordinary software systems in that they enable machines to detect patterns, perform tasks, and solve problems which they are not explicitly programmed to solve and which human experts were previously incapable of solving by hard-coding explicit rules. Typically, these systems are able to:

- (1) learn generalizable rules by processing large amounts of data
- (2) tune a large number of free parameters (thousands to billions)
- (3) outperform well-trained humans in domain specific tasks

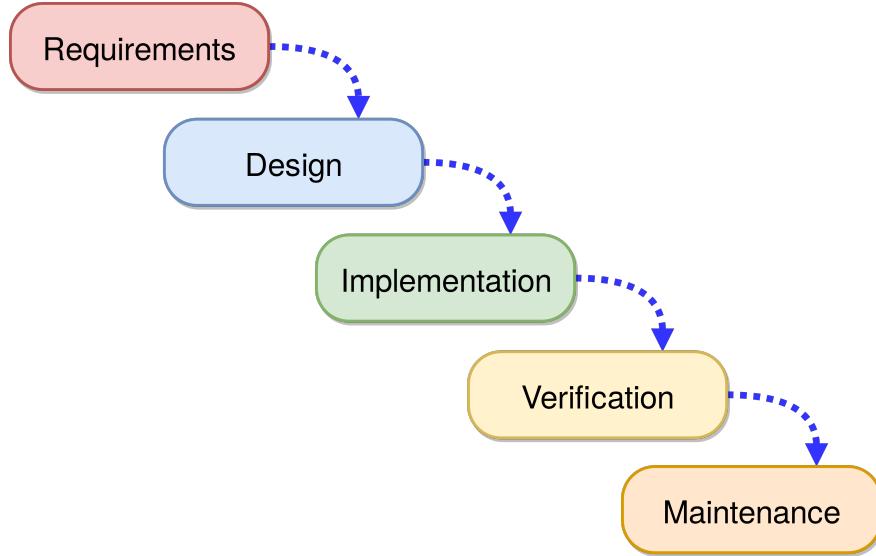
---

<sup>1</sup>This can be approximated by various metrics like cyclomatic or Halstead complexity.

While the idea of intelligent systems has existed for decades, three critical developments made modern intelligent systems ultimately successful. First, computer processing power has become faster, cheaper, and much more readily available. Similarly, the digitalization of new datasets has made vast amounts of information available, and data storage costs have plummeted dramatically. (A \$5 thumb drive today has 200 times more storage capacity than a 2,000 pound, 5 MB, IBM hard drive that leased for \$3,000 per month in 1956.) Most importantly, has been the development of more efficient learning algorithms.

In recent years, computer science and software engineering has made significant strides in building and deploying intelligent systems. Nearly every mobile computer in the world is able to detect objects in images, perform speech-to-text and language translation. These breakthroughs were the direct result of fundamental progress in neural networks and representation learning. Also key to the success of modern intelligent systems was the adoption of collaborative open source practices, pioneered by the software engineering community. Software engineers developed automatic differentiation libraries like Theano [Al-Rfou et al., 2016], Torch [Collobert et al., 2002] and Caffe [Jia et al., 2014], and built many popular simulators for reinforcement learning, the ease of use and availability of which was crucial for popularizing deep learning techniques.

In this thesis, we explore several tools that facilitate the process of programming intelligent systems, and which reduce the cognitive effort required to understand an intelligent program. First, we demonstrate an integrated development environment that assists users writing robotics software (Chapter 2). Next, we show a type-safe domain specific language for differentiable programming, an emerging paradigm in deep learning (Chapter 3). To test this application, we use a set of techniques borrowed from property-based testing [Fink and Bishop, 1997] and adversarial learning [Lowd and Meek, 2005] (Chapter 4). Docker containers [Merkel, 2014] are used to automate the building, testing and deployment of reproducible robotics applications on heterogeneous hardware platforms (Chapter 5). Finally, as a proof of concept for these ideas, we build an intelligent system comprised of a mobile autonomous vehicle and an Android mobile application, using the tools developed (Chapter 6).



**Fig. 1.1.** Royce’s original Waterfall model, originally intended to describe the software development process, but the same sequence can be found in most engineering disciplines. We use it to help guide our discussion and frame our work inside of this process model.

## 1.1. Stages in the software development lifecycle

In traditional software engineering, the Waterfall Model 1.1 is a classical software process model comprised of five stages. While the Waterfall Model was an early model in software engineering, the same activities it describes can be found in most engineering process models. We propose contributions to four such areas: design in Chapter 2, implementation in Chapter 3, verification in Chapter 4 and maintenance in Chapter 5.

## 1.2. Designing intelligent systems

Today’s software systems are deeply complex entities. Gone are the days where a solitary programmer, even a very skilled one, can maintain a large software system alone. To effectively scale software systems, programmers must pool their mental capacity to form a knowledge graph. Software projects which rely on a small set of maintainers tend to perish due to the so-called *bus factor* [Cosentino et al., 2015] – large portions of the knowledge graph are locked inside someone’s head. Successful software projects learn how to distribute this graph and form new connections to the outside world. The knowledge graph which accumulates around a large software project contains facts, but it also contains workflows for programming, debugging, and delivery – all well-traveled paths through the labyrinth of

software development. Components of this graph can be committed to writing in the form of documentation, but documentation is time-consuming and grows stale over time. What is needed is a system that preserves the benefits of documentation without the burdens of maintenance.

The development of software systems has a second component, the social graph. The social graph of a successful software project contains product designers, managers and software engineers who work in concert to build software that is well-designed, cohesive, and highly performant. Sometimes this means revising the specification to accommodate engineering challenges, or rewriting source code to remove technical debt. Software design is a multi-objective optimization process and requires contributors with a broad set of skills and common set of goals. To produce software that approximates criteria of its stakeholders, developers are asked to provide rapid prototypes, and continuously integrate user feedback. Yet today’s software systems are larger and more unwieldy than ever. So finding ways to work together more efficiently is critical to building and maintaining intelligent systems.

First, let us consider the mechanical process of writing software with a keyboard.

Integrated development environments (IDEs) can assist developers building complex software applications by automating certain repetitive programming tasks. For example, IDEs perform static analyses and inspections for catching bugs quickly. They provide completion, refactoring and source code navigation, and they automate the process of building, running and debugging programs. While these tasks may seem trivial, their automation promises increased developer productivity by delivering earlier feedback, detecting clerical errors, and allows developers to focus on fundamental design tasks. Rather than being forced to concentrate on the structure and organization of text, if developers are able to manipulate code at a semantic level, they will be much happier and more productive. Furthermore, automating frequent tasks in software development, these mechanical tools enable them to focus on the fundamentals of writing and understanding programs.

But what are IDEs really doing? They are guiding developers through the knowledge graph of a software project. Consider what a new developer must learn to get up to speed: in addition to learning the language, developers must learn to use libraries and frameworks (arguably languages in their own right). They must become familiar with command line tools for software development, from build tools to version control and continuous integration.

They must become familiar with the software ecosystem, programming styles, conventions and development workflows. And they must learn how to collaborate on a distributed team of developers. By automating common tasks in an interactive programming environment and making the graph connectivity explicit through document markup [Goldfarb, 1981] and projectional editing [Voelter et al., 2014], a well-designed IDE is a tool for graph traversal. It should come as no surprise IDEs are really graph databases.

In some aspects, the development of intelligent systems is no different than classical software engineering. While the former requires more human intervention, the same principles and best-practices which guide software engineering are also applicable to intelligent systems. And the same activities, from analysis, design, implementation, verification and maintenance will continue to play an important role in building intelligent systems. But in other aspects, the generic programming tools we use for developing traditional software will require domain-specific adaptations for machine learning to become a truly first class citizen in the next generation of software systems, particularly in the case of physically embodied agents. Towards that end, we propose Hatchery, an IDE for the Robot Operating System (ROS), a popular robotics middleware.

### 1.3. Implementation: Languages and compilers

In the early days of machine learning, it was widely believed the development of human-level intelligence simply required a sufficiently descriptive first-order logic. By accumulating a database of facts and their relations, researchers believed they could use symbolic reasoning to bypass learning altogether. This rule-based approach dominated a large portion of early research in artificial intelligence and considerable effort was poured into the creation of domain-specific ontologies. Despite the best efforts of roboticists, signal processing engineers and natural language researchers, *expert systems* were unable to scale to real-world applications, causing a great disillusionment in A.I. research for a number of decades. While they underestimated the difficulty of learning and were largely unsuccessful because of it, expert systems excelled in areas where current machine learning systems struggle such as reasoning and interpretability, and there is reason to believe these ideas were just ahead of their time.

What did finally work, is the idea of connectionist learning. By nesting random function approximators, called *artificial neural networks* (ANNs), and training the network using

backpropagation [Werbos et al., 1990, Rumelhart et al., 1988], the resulting system is capable of learning a surprising amount of intelligent behavior. The approach can be traced back to the mid-20th century [Ivakhnenko and Lapa, 1965, Rosenblatt, 1958], but was not fully-realized in silico until after the widespread availability of cheap computing and large datasets [LeCun et al., 2015]. In theory, a single layer of nesting is able to approximate any continuous differentiable function [Hornik et al., 1989], but in practice learning requires composing many such approximators in a deeply nested fashion, hence the term, *deep neural networks* (DNNs). The importance of depth was suspected for many years, but the original backpropagation algorithm had difficulty on DNNs due to the vanishing gradient problem [Bengio et al., 1994]. Solving this problem required a number of adaptations and many years to fully debug. It was not until 2013 when deep learning was competitive with humans in a number of domains.

While it took fundamental research in deep learning to realize the connectionist blueprint, the success of modern deep learning can be partly attributed to software tools for calculating mathematical derivatives, a key step in the backpropagation algorithm. Although it has not been established if or how derivatives might be calculated in biological circuits, derivatives are essential for ANN training. For many years, the symbolic form of these derivatives had to be analytically derived when prototyping a new neural network architecture, a tedious and error-prone process. There is a well-known algorithm in the scientific computing community, called *automatic differentiation* (A.D.) [Linnainmaa, 1970, Griewank et al., 1989], which is able to calculate derivatives for arbitrary differentiable functions. But surprisingly, it was not until much later, after the development of Theano [Al-Rfou et al., 2016] when AD became widely adopted in the machine learning community. This library alone greatly accelerated the pace of deep learning research and spurred the development of others like TensorFlow [Abadi et al., 2016] and PyTorch [Paszke et al., 2017]

Intelligent systems engineers must think carefully about languages and abstractions. If developers are required to implement backpropogation by hand, they will have little time to think about the high-level characteristics of these systems. Similarly, if programming abstractions are too specific, small variations will require costly reimplementation. This is no different from traditional software engineering – as engineers, we need to choose the right abstractions for the task at hand. Too low-level and the design is lost in the details, too

abstract and the details are lost completely. With deep learning, the necessity of choosing good abstractions is even more important, as the connection of between source code and runtime behavior is already difficult to grasp, due to the nature of neural networks.

#### 1.4. Testing: Verification and validation

Most naturally arising phenomena, particularly those related to vision, planning and locomotion are high dimensional creatures. Richard Bellman famously coined this problem as the “curse of dimensionality”. Our physical universe is populated by problems which are simple to pose, but impossible to solve inside of it. Claude Shannon, a contemporary of Bellman, calculated the number of unique chess games to exceed  $10^{120}$ , more than the number of atoms in the universe by approximately 40 orders of magnitude [Shannon, 1950]. At the time, it was believed that such problems would be insurmountable without a fundamental change in algorithms or computing machinery. Indeed, while Bellman or Shannon did not live to see the day, it took only half a century of progress in computer science before solutions to problems with the same order of complexity, first solved in the Cambrian explosion 541 million years ago, were approximated to a competitive margin in modern computers.

While computer science has made enormous strides in solving the common cases, Bellman’s curse of dimensionality still haunts the long tail of machine learning, particularly for distributions that are highly disperse. Because the dimensionality of many real world problems we would like to solve is intractably large, it is difficult to verify the behavior of a candidate solution in all regimes. According to some studies, a human driver averages 1.09 fatalities per hundred million miles [Kalra and Paddock, 2016]. A new software build for an autonomous vehicle would need to accumulate 8.8 billion miles of driving in order to approximate the fatality rate of a human driver to within 20% with a 95% confidence interval. Deploying such a scheme in the real world would be logistically, not to mention ethically problematic.

Realistically, intelligent systems need better ways to practice their skills and probe the effectiveness of a candidate solution within a limited computational budget, without harming humans in the process. The goal of this testing is to highlight errors, but ultimately to provide feedback to the system. In software engineering, the real system under test is the ecosystem of humans and machines which provide each other’s means of subsistence. The success of this

arrangement depends on an external testing mechanism to enforce a minimum bar of rigor, typically some form of hardware- or human-in-the-loop testing. If the testing mechanism is not somehow opposed to the system under test, an intelligent system can deceive itself, which is neither in the system's nor its users' best interests.

## 1.5. Software reproducibility and maintenance

One of the challenges of building intelligent systems and programming in general, is the problem of reproducibility. Software reproducibility has several challenging aspects, including hardware compatibility, operating systems, file systems, build systems, and runtime determinism. While writing programs and feeding them directly into a computer may have once been common practice, today's source code is far too removed from its mechanical realization to be meaningfully executed in isolation. Today's handwritten programs are like schematics for a traffic light – built inside a factory, and which require a city's-worth of infrastructure, cars, and traffic laws to serve their intended purpose. Like traffic lights, source code does not exist in a vacuum – built by compilers, interpreted by virtual machines, executed inside an operating system, and which following a specific communication protocol – programs are essentially meaningless abstractions outside this context.

As necessary in any good schematic, much of the information required to build a program is divided into layers of abstraction. Most low-level instructions carried out by a computer during the execution of a program were not written nor intended to be read by the programmer and have long since been automated and forgotten. In a modern programming language like Java, C# or Python, the total information required to run a simple program often numbers in the trillions of bits. A portion of that data pertains to the software for building and running programs, including the build system, software dependencies, and development tools. Part of the data pertains to the operating system, firmware, drivers, and embedded software. For most programs, such as those found in a typical GitHub repository, a vanishingly small fraction correspond to the handwritten program itself.

Applied machine learning shares many of the same practical challenges as traditional software development, with source code, release and dependency management. The current process of training a deep learning model can be seen as particularly long compilation step, but it differs significantly in that the source code is a high-level language which does not

directly describe the computation being performed, but is a kind of meta-meta-program. The first meta-program describes the connectivity of a large directed graph (i.e. a computation graph or probabilistic graphical model), parameterized by weights and biases. The tuning of those parameters is another meta-program, describing the sequence of operations required to approximate a program which we do not have access, save for some input-output examples. Emerging techniques in meta-learning and hyper-parameter optimization (e.g. differentiable architecture search [Liu et al., 2018]) add even further meta-programming layers to this stack, by searching over the space of directed graphs themselves.

Hardware manufacturers have developed a variety of custom silicon to train and run these programs rapidly. But unlike most programming, deep learning is a much simpler model of computation – so long as a computer can add and multiply, it has the ability to run a deep neural network. Yet due to the variety of hardware platforms which exist and the software churn associated with them, reproducing deep learning models can be painstakingly difficult on new hardware, even with the same source code and dependencies. Many graph formats, or *intermediate representations* (IRs) in compiler parlance, promise hardware portability but if developers are not careful, their models may not converge during training, or may produce different results on different hardware. Complicating the problem, IRs are produced by competing vendors, with competing chips and incompatible standards (e.g. MLIR, ONNX, NNEF, OpenVINO, et al.). While some have tried to leverage existing compilers such as GHC [Elliott, 2018] or LLVM [Wei et al., 2017], there are few signs of convergence.

At the end of the day, researchers need to reproduce the work of other researchers, but the mental effort of re-implementing their abstractions can be tedious and detrimental towards scientific progress. Since it is necessary to reuse programs written by others, it would be convenient to have tools for reproducibility and incremental development. Fortunately, this is the same problem software developers have been attempting to solve for many years, through open source software. But source control management (SCM) alone is insufficient, since SCM tools are primarily intended for text. While text-based representations may be temporarily stable, as dependencies are periodically updated and rebuilt, important details about the original development environment can be misplaced. To reproduce a program in its entirety, we need a snapshot of all digital information available to the computer at the

time of its execution, and ideally, the physical computer itself. Short of that, the minimal set of dependencies and a close replica is essential.

In order to mitigate the effects of software variability and assist the development of intelligent systems on heterogeneous platforms, we use a developer tool called Docker, part of a loosely-affiliated set of tools for build automation and developer operations which we shall refer to as *container infrastructure*. Docker allows developers to freeze a software system and its host environment, allowing developers (e.g. using a different environment) to quickly reproduce software on another computer. Docker itself is a technical solution, but also encompasses a set of best-practices which are more procedural in nature. While Docker itself does not address the incompatibility of vendor standards and hardware drivers, it makes these variables explicit, and reduces the difficulty of reproducing software artifacts.

There is a second component to software reproducibility of intelligent systems, at the boundary of software and hardware: simulators. Today’s simulators have become increasingly realistic, but most roboticists agree that simulation alone will never be enough to capture the full distribution of real world data. In this view, while simulation can be a useful tool for detecting errors, it cannot fully reproduce all the intricacies of the real world, and must never be used as a surrogate for training on real-world data. Others have suggested a middle road [Bousmalis et al., 2018], where judicious use of simulator training, alongside domain adaption is a sufficiently rigorous environment for training for intelligent systems. Regardless of which view prevails, our goal is to provide rapid feedback to developers, and to make this entire process as reproducible as possible.

### 1.5.1. Case Study

All great software has a secret recipe: software gets better when engineers use the product. In the best case, software engineers are the core users, ideally by choice, if not by necessity. When software engineers are using their own software on a regular basis – bumping into sharp corners and encountering edge cases firsthand – the product gets better. When there is an obviously missing feature, it gets implemented. When there is a bug, it gets fixed. It may not be easy to find engineers who are so invested, or to build software which is so useful, but there must be some overlap in order for good software to become

great. Termed “dogfooding” [Harrison, 2006], this practice has been employed for public relations, but is a very effective mechanism for building self-improving cybernetic systems. It is also an important principle for open source software and safety-critical systems. Putting this principle into practice, we, as authors and primary users of these tools, validate their effectiveness by developing a robotics application within an IDE (Chapter 2), containing Kotlin $\nabla$  code (Chapter 3), tested using adversarial fuzz testing (Chapter 4), and which is built and maintained using the Docker stack (Chapter 5).



# Chapitre 2

---

## Design: Programming tools for robotics

“The hope is that, in not too many years, human brains and computing machines will be coupled together very tightly, and that the resulting partnership will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today.”

---

—Licklider [1960]

, *Man-Computer Symbiosis*

In this chapter we will discuss the design and implementation of an integrated development environment (IDE) for building intelligent robotic software. Modern robots are increasingly characterized by systems which learn and improve over time. Most researchers would agree that modern robotic systems have not yet achieved biologically competitive sensorimotor capabilities and most intelligent systems are not physically embodied. However, it is our view that any closed-loop control system that is not explicitly programmed to perform a specific task, but which learns it from experience is an *intelligent system*. Furthermore, any closed-loop system with physical motors is a *robotic system*. While research has demonstrated successful applications in both areas separately, it is widely believed the integration of intelligent systems in robotics will be tremendously fruitful when fully realized.

Hatchery is a tool designed to assist programmers writing robotics applications with ROS middleware. At the time of its release, Hatchery was the first ROS plugin for the IntelliJ Platform<sup>1</sup>, and one year later, is also the most widely used with close to 10,000 downloads. While the idea is simple, its prior absence and subsequent adoption suggest there is unmet

---

<sup>1</sup>A popular IDE for C/C++, Python and Android development

demand for such tools in the development of intelligent software systems, particularly in robotics where the system is highly specialized.

## 2.1. Software architecture for robotics application

The Robot Operating System (ROS) is a popular middleware for robotics applications. At its core, ROS provides software infrastructure for distributed messaging, but also includes a set of community-developed libraries and graphical tools for building robotics applications. ROS is not an operating system (OS) in the traditional sense, but it does offer OS-like functionality such as shared memory and inter-process communication. Unlike pure message-oriented systems like DDS and ZMQ, in addition to the communication infrastructure, ROS provides specific APIs for building decentralized robotic systems, particularly those which are capable of mobility. This includes standard libraries for serializing and consuming geometric data, coordinate frames, maps, sensor messages, and imagery.

A ROS application consists of several components, including the source code, configuration files, build infrastructure, compiled artifacts and the deployment environment. To build a ROS application, several steps are necessary. First, one must install the ROS system, which is only officially supported on Debian-based Linux distributions. Then, a workspace must be created. ROS nodes, implemented as event based “talkers” and “listeners”.

The ROS middleware provides several language front-ends for polyglot programming. According to one community census in 2018, 55% of ROS applications on GitHub are written in C/C++, followed by Python at around 25% [Guenther]. Source code for a typical ROS application will include a mixture of C/C++ and Python code, roughly corresponding to the language preferences in the robotics and machine learning communities. Hatchery supports most common language front-ends for ROS, including Java, Python, C/C++, and several others, through the IntelliJ Platform. By targeting an IDE platform with support for polyglot programming, we are able to focus on language-agnostic features for the ROS ecosystem, such as parsing and editing ROS-specific configuration files, build and run configuration and other common development tasks.

## 2.2. Foundations of a modern IDE

To build an IDE, a few prerequisites are necessary. First, is an IDE, which we shall refer to as  $\text{IDE}_0$ , and its source code. Assume that  $\text{IDE}_0$  exists. In order to build a new IDE,  $\text{IDE}_1$ , first load the source code from  $\text{IDE}_0$  into  $\text{IDE}_0$ . Then, using  $\text{IDE}_0$ , modify and compile the source code. Finally, run the compiled artifact, which becomes  $\text{IDE}_1$ . Iterate as necessary.

While valid, this approach has some disadvantages. First, most IDEs are already quite large and cumbersome to download, install, compile, and run. Since most plugins are small by comparison, modern IDEs have adopted a modular design, which allows them to load certain packages (i.e. *plugins*), as needed. So most developers can skip the first step, and load such a plugin, using  $\text{IDE}_0$  directly. It is still convenient to have the platform source code for reference purposes, but in most cases this source code is read-only.

### 2.2.1. The parser

An IDE consists of a few important components. First is a parser. This is typically not an ordinary parser, because most of the time users are modifying a program, their code is invalid. Even when the source code is invalid, the parser needs guide the user towards a functioning program. So this parser needs to be able to recover from syntactical errors.

We can parse URDF, package and launch XML, and srv files.

### 2.2.2. Refactoring

Refactoring support is implemented.

### 2.2.3. Running and debugging

Assistance for running ROS applications.

## 2.3. More ROS Tools

Detecting and managing ROS installations.



# Chapitre 3

---

## Implementation: languages and compilers

“The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both [the domain and the range] consist of functions.”

---

—Church [1985], *The Calculi of Lambda Conversion*

In this chapter, we will discuss the theory and implementation of a type safe domain specific language for automatic differentiation (AD), which has a variety of applications in numerical optimization and machine learning. The key idea behind AD is fairly simple. A small set of primitive operations form the basis for all modern computers, and by composing these operations over the real numbers in an orderly fashion, one can compute any computable function. In machine learning, it is often the case we are given a computable function, in the form of a program<sup>1</sup>, that does not work properly. We would like an algorithm for determining how to change the input slightly, so as to produce a more suitable output.

In 1964, such an algorithm was first conceived in Wengert [1964], whose method is known today as forward-mode AD. Not long after, a certain Richard Bellman reproduced Wengert’s algorithm to numerically estimate the orbital dynamics of a two body system, and recognized its potential for, “the treatment of large systems of differential equations which might not otherwise be undertaken” [Bellman et al., 1965]. Around the same time, key details of the backpropagation algorithm first emerged [Dreyfus, 1990]. It was in Linnainmaa [1970] where the idea of calculating derivatives over computation graphs was first recorded. Linnainmaa’s algorithm was particularly important for the development of neural networks, and is today known as reverse-mode AD. But it was not until 2010 when standard software tools [Bergstra

---

<sup>1</sup>Not all programs are computable functions, but all computable functions are programs.

et al., 2010, Collobert et al., 2011] for AD became widely available in machine learning. It is here where our journey begins.

### 3.1. Automatic differentiation

Given some input to a function, AD tells us how to change the input by a minimal amount, in order to maximally change the outputs. Suppose we are handed a function  $P_k : \mathbb{R} \rightarrow \mathbb{R}$ , composed of a series of nested functions, each with the same type:

$$P_k(x) = \begin{cases} p_0(x) = x & \text{if } k = 0 \\ (p_k \circ P_{k-1})(x) & \text{if } k > 0 \end{cases} \quad (3.1.1)$$

From the chain rule of calculus, we recall that:

$$\frac{dP}{dp_0} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \cdots \frac{dp_1}{dp_0} = \prod_{i=1}^k \frac{dp_i}{dp_{i-1}} \quad (3.1.2)$$

Likewise, for a scalar function  $Q(q_0, q_1, \dots, q_n) : \mathbb{R}^n \rightarrow \mathbb{R}$ , the gradient  $\nabla Q$  tells us:

$$\nabla Q = \left( \frac{\partial Q}{\partial q_1}, \dots, \frac{\partial Q}{\partial q_n} \right) \quad (3.1.3)$$

Occasionally, we may wish to compute the second-order partials for  $Q$ , i.e. the Hessian,  $\mathbf{H}$ :

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 Q}{\partial x_1^2} & \frac{\partial^2 Q}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_1 \partial x_n} \\ \frac{\partial^2 Q}{\partial x_2 \partial x_1} & \frac{\partial^2 Q}{\partial x_2^2} & \cdots & \frac{\partial^2 Q}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 Q}{\partial x_n \partial x_1} & \frac{\partial^2 Q}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_n^2} \end{bmatrix} \quad (3.1.4)$$

More generally, for a vector function  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , the Jacobian  $\mathbf{J}$  is defined as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_n \end{bmatrix} \quad (3.1.5)$$

For a vector function  $\mathbf{P}_k(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , the chain rule still applies:

$$\mathbf{J}_{\mathbf{P}_k} = \prod_{i=1}^k \mathbf{J}_{p_i} = \underbrace{\left( \left( (\mathbf{J}_{p_k} \mathbf{J}_{p_{k-1}}) \dots \mathbf{J}_{p_2} \right) \mathbf{J}_{p_1} \right)}_{\text{"Reverse accumulation"}} = \underbrace{\left( \mathbf{J}_{p_k} \left( \mathbf{J}_{p_{k-1}} \dots (\mathbf{J}_{p_2} \mathbf{J}_{p_1}) \right) \right)}_{\text{"Forward accumulation"}}$$

For completeness, but rarely used in practice, is the second-order partials for vector functions:

$$\mathbf{H}(\mathbf{f}) = [\mathbf{H}(f_1), \mathbf{H}(f_2), \dots, \mathbf{H}(f_m)] \quad (3.1.6)$$

We can use these tools to compute the direction to adjust the inputs of a computable function, in order to maximally change that function's output, i.e. the direction of steepest descent. Sometimes a function has the property that given an input  $a$ , no matter  $a$  is changed, the output stays the same. We say that such functions have zero gradient for that input.

$$(\nabla F)(a) \approx \mathbf{0} \quad (3.1.7)$$

The cost of calculating the Hessian,  $\mathbf{H}$  is approximately quadratic [Griewank, 1993] with respect to the number of independent variables under differentiation. If  $\mathbf{H}(a)$  is tractable to compute and invertible, we could use the second-partial derivative test to determine that:

- (1) If all eigenvalues of  $\mathbf{H}(a)$  are positive,  $a$  is a local minimum
- (2) If all eigenvalues of  $\mathbf{H}(a)$  are negative,  $a$  is a local maximum
- (3) If  $\mathbf{H}$  contains a mixture of positive and negative eigenvalues,  $a$  is a *saddle point*

For some classes of computable functions, small changes to the input will produce a sudden large change in the output. We say that such functions are non-differentiable.

$$\|\nabla F\| \approx \pm\infty \quad (3.1.8)$$

It is an open question whether non-differentiable functions exist in the real world [Buny et al., 2005]. At the current physical (10nm) and temporal (10ns) scale of modern computing, there exist no such functions, but most modern computers are incapable of reporting the true value of their binary-valued functions. For all intents and purposes, programs implemented by most physical computers are discrete relations. Nevertheless, discrete programs are capable of approximating bounded functions of  $R^n$  to arbitrary precision given enough time and space. For most applications, a low precision (32-64 bit) approximation is sufficient.

There exists at the heart of machine learning a theorem that states a simple family of functions, which compute a weighted sum of a non-linear function  $\varphi : R \rightarrow R$  composed with a linear function  $\theta^\top x + b$ , can approximate any bounded function on  $\mathbb{R}^m$  to arbitrary precision. More precisely, the universal approximation theorem [Hornik et al., 1989] states that for all real-valued continuous functions  $f : C(\mathbb{I}_m)$ , where  $\mathbb{I}_m = [0, 1]^m$ , there exists a function  $\hat{f}$ , parameterized by constants  $n \in \mathbb{N}, \beta \in \mathbb{R}^n, b \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$  and  $\theta \in \mathbb{R}^{m \times n}$ :

$$\begin{aligned}\hat{f}(x) &= \beta^\top \varphi(\theta^\top x + b) \\ \forall x \in \mathbb{I}_m, \quad &|\hat{f}(x) - f(x)| < \epsilon\end{aligned}\tag{3.1.9}$$

This theorem does not provide any method for how to find  $\theta$  nor place an upper bound on the constant  $n$ , somewhat limiting its practical applicability. But for reasons not yet fully understood, empirical results suggest it is possible to approximate many naturally-arising functions in a relatively short time by composing several *layers* of  $\theta_j^\top x + b$  and  $\varphi$  in an alternating fashion. The resulting model can be expressed as,

$$\hat{\mathbf{P}}_k(\mathbf{x}; \Theta) = \begin{cases} \mathbf{p}_0(\boldsymbol{\theta}_0)(\mathbf{x}) & \text{if } k = 0 \\ (\mathbf{p}_k(\boldsymbol{\theta}_k) \circ \hat{\mathbf{P}}_{k-1}(\Theta_{<k}))(\mathbf{x}) & \text{if } k > 0 \end{cases}\tag{3.1.10}$$

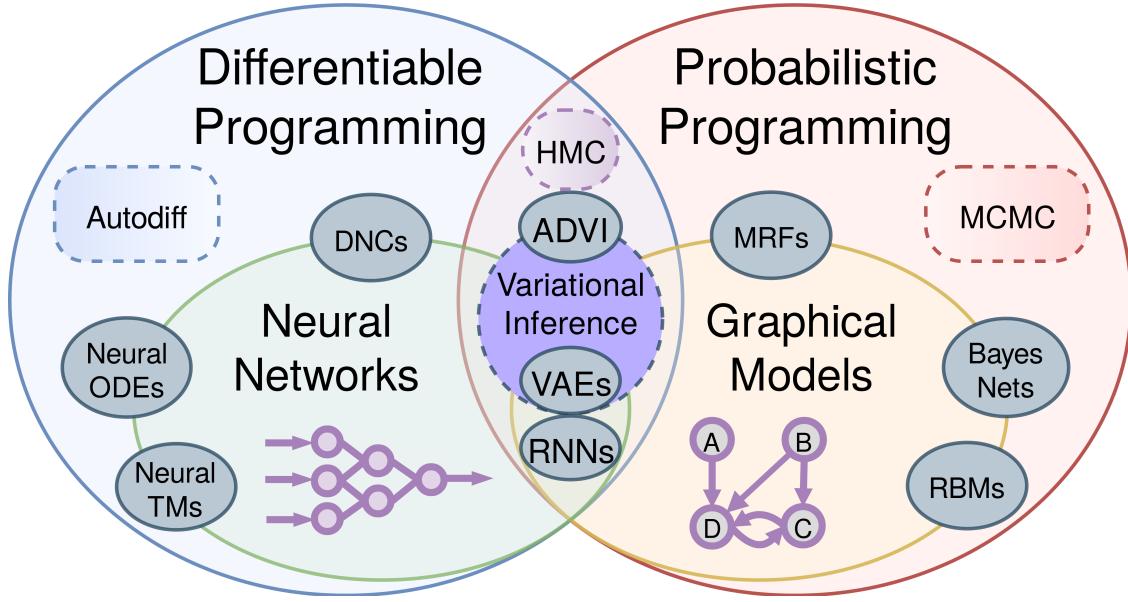
where  $\Theta = \{\boldsymbol{\theta}_0, \dots, \boldsymbol{\theta}_k\}$  are free parameters and  $\mathbf{x} \in \mathbb{R}^n$  is a single input. To approximate  $\mathbf{P}(\mathbf{x})$ , one must obtain  $\mathbf{X} = \{\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(z)}\}, \mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \dots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$  in as great and varied a quantity as possible and repeat the following procedure until  $\Theta$  converges:

$$\Theta \leftarrow \Theta - \frac{1}{z} \nabla_\Theta \sum_{i=0}^z \mathcal{L}(\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)})\tag{3.1.11}$$

For most common  $\mathcal{L}$ , the complexity of this procedure is linear with  $z$ . As  $z$  can be quite large in practice, a single iteration can take prohibitively long. Since obtaining the exact gradient is not very important, we use a stochastic variant by randomly resampling a *minibatch*  $\mathbf{X}' \sim \mathbf{X}$  on each update, which is slightly noisier, but runs considerably more quickly.

## 3.2. Differentiable programming

The renaissance of modern deep learning is widely attributed to progress in three research areas: algorithms, data and hardware. Among algorithms, most research has focused on deep learning architectures and representation learning. Equally important, arguably, is the role that automatic differentiation (AD) has played in facilitating the implementation of these



**Fig. 3.1.** *Differentiable programming* includes neural networks, but more broadly, arbitrary differentiable programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* is a generalization of probabilistic graphical models, and uses various forms of Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function.

ideas. Prior to the adoption of general-purpose AD libraries such as Theano, PyTorch and TensorFlow, gradients had to be derived manually. The widespread adoption of AD software simplified and accelerated the pace of gradient-based machine learning, allowing researchers to build deeper network architectures and new learning representations. Some of these ideas in turn, formed the basis for new methods in AD, which continues to be an active area of research in the programming language community.

A key aspect of the connectionist paradigm is gradient descent on a statistical loss function with respect to the free parameters of a neural network. For gradient descent to work, the representation must be differentiable almost everywhere. However many representations are non-differentiable in their natural domain. For example, the structure of language in its written form is not easily differentiable, as small changes to a word's symbolic representation can cause sudden changes to its semantic meaning. A key insight from representation learning is that many discrete data types can be mapped to a smoother latent space. For example, if we represent words as a vector of real numbers, then it is possible to learn a

mapping from the textual domain to the vector representation so that the semantic relations between words (as measured by their statistical co-occurrence in large language corpora) are geometrically preserved in vector space [Pennington et al., 2014] – words with similar meanings map to similar vectors. It so happens that many classes of discrete problems can be relaxed to continuous surrogates by learning such representations, or *embeddings* in an unsupervised manner.

Around the same time, the deep learning community realized that perhaps strict differentiability was not so important all along. It was shown in practice, that computers using low-precision arithmetic such as 8-bit floating point [Wang et al., 2018d] and integer [Jacob et al., 2018] quantization are able to train neural networks without sacrificing performance. Strong assumptions like Lipschitz-continuity and  $\beta$ -smoothness once thought to be indispensable for gradient-based learning could be relaxed, as long as the noise introduced by quantization was negligible compared to stochastic gradient methods. In hindsight, this should not have come as a big surprise, since all digital computers use discrete representations anyway and were capable of training neural networks for nearly half a century. This suggests strict differentiability was not as important as having a good metric. As long as the loss surface permits metric learning, network training is surprisingly resilient to quantization.

As deep learning solved problems across various domains, researchers observed that neural networks were part of a broader class of differentiable architectures that could be designed, implemented and analyzed in a manner not unlike computer programs. Hence the term *differentiable programming* (DP) was born. Today, DP has found a wide range of applications, from protein folding [AlQuraishi, 2018], to physics engines [de Avila Belbute-Peres et al., 2018, Degrave et al., 2016] and graphics rendering [Loper and Black, 2014] to meta-learning [Liu et al., 2018]. These domains have well-studied dynamics models, with parameters that can be tuned via gradient descent. Traditionally, handcrafted optimization algorithms were required to learn these parameters, but given a smooth metric, DP promises to do this for a broad class of models, more or less automatically. For discrete optimization however, DP is not sufficient. To automatically learn discrete relations without ad hoc embedding, additional tools, such as probabilistic programming, are likely needed. As seen in Fig. 3.1, these two fields have developed many productive collaborations in recent years.

### 3.3. Static and dynamic languages

Most programs in machine learning and scientific computing are written in dynamic languages, such as Python. In contrast, most of the industry uses statically typed languages [Ray et al., 2017]. Dynamically typed languages are popular for experimentation and prototyping, but are they scalable to building complex software systems? According to some studies, type related errors account for over 15% of software bugs [Gao et al., 2017]. While the causality between statically typing and fewer defects has not been conclusively established, the majority of robotics [Guenther] and embedded systems today rely on statically typed languages.

Statically typed languages eliminate a broad class of runtime errors and allow us to reason about the behavior of programs without needing to run them. A well-designed library in a statically typed language will eliminate errors related to API misuse, which would otherwise require documentation and code samples. Finally, type systems make it easier to implement automated reasoning tools, which can provide more relevant autocompletion, source code navigation, and earlier detection of runtime errors.

### 3.4. Imperative and functional languages

Most programs written today are written in the imperative style, due the prevalence of the Turing Machine and von Neumann architecture [Backus, 2007].  $\lambda$ -calculus provides an equivalent<sup>2</sup> language for computing, which we argue, is a more appropriate notation for expressing mathematical functions and computing their derivatives. In imperative programming the sole purpose of using a function is to pass it values, and there is no way to refer to a function directly. More troubling in the case of AD, is imperative programs have mutable state, which requires taking extra precautions when computing their derivatives.

The mathematical notion of function composition is a first-class citizen in functional programming. Just like in calculus, to take the derivative of a programming function composed with another function, simply apply the chain rule (cf.3.1). Since there is no mutable state, no exotic data structures or compiler tricks are required to keep track of the program state.

For example, consider the vector function  $f(l_1, l_2) = l_1 \cdot l_2$ , seen in 3.1. Imperative programs, by allowing mutation, are destroying intermediate information. In order to recover

---

<sup>2</sup>In the sense that the Turing Machine and  $\lambda$ -calculus are both Turing Complete.

Imperative	Functional
<pre> 1 <b>fun</b> dot(l1, l2) { 2     <b>if</b> (len(l1) != len(l2)) 3         <b>return</b> error 4     <b>var</b> sum = 0 5     <b>for</b>(i <b>in</b> 0 <b>to</b> len(l1)) 6         sum += l1[i] * l2[i] 7     <b>return</b> sum 8 }</pre>	<pre> <b>fun</b> dot(l1, l2) {     <b>return if</b> (len(l1) != len(l2))         error     <b>else if</b> (len(l1) == 0) 0     <b>else</b>         head(l1) * head(l2) +         dot(tail(l1), tail(l2)) }</pre>

**Tab. 3.1.** Two programs, both implementing the function  $f(l_1, l_2) = l_1 \cdot l_2$ .

the computation graph for reverse mode AD, we either need to override the assignment operator, or use a tape to store the intermediate values, which is quite tedious. In pure functional programming, mutable variables do not exist, which makes our lives much easier.

Kotlin $\nabla$  treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed to form a data-flow graph (DFG). An expression is simply a **Function**, which is only evaluated when invoked with numerical values, e.g. `z(θ, θ)`. In this way, Kotlin $\nabla$  is similar to other compiled graph-based approaches like TensorFlow and Theano.

### 3.5. Kotlin

When programming in a statically typed language, one common question is, “Given a value, `x`, can `x` be assigned to a variable of type `Y`? ” (e.g. `x instanceof Y`) In Java, this question is both unsound [Amin and Tate, 2016] and undecidable [Grigore, 2017] in the general case. It is possible to construct a Java program where the answer is “yes” regardless of `Y`, or for which the answer cannot be determined in a finite amount of time.

Kotlin is a strong, statically typed language, that is well-suited for building cross-platform applications, with implementations in native, JVM, and JavaScript. Kotlin’s type system [Tate, 2013] is strictly less expressive, but fully compatible with Java.

In this work, we make use of various Kotlin specific language features, such as extension functions (3.16), operator overloading (3.11),

### 3.6. Kotlin $\nabla$

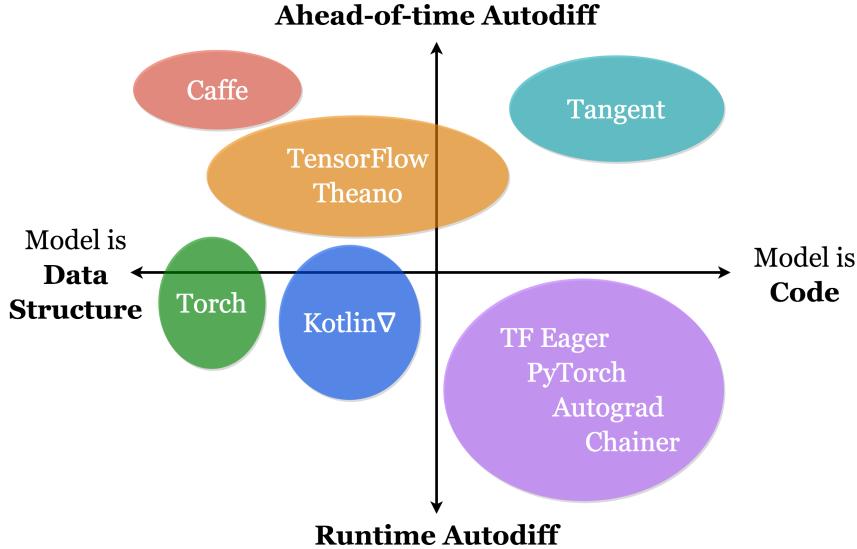
Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [Gil and Levy, 2016] in Java. This result was strengthened to prove Java’s type system is Turing complete [Grigore, 2017]. As a practical consequence, we can use the same technique to perform shape-safe automatic differentiation (AD) in Java, using type-level programming. A similar technique is feasible in any language with generic types.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano [Al-Rfou et al., 2016], Torch [Collobert et al., 2002], and TensorFlow [Abadi et al., 2016]. Similar ideas have been implemented in statically typed, functional languages, such as Haskell’s Stalin $\nabla$  [Pearlmutter and Siskind, 2008b], DiffSharp in F# [Baydin et al., 2015b] and recently Swift [Lattner and Wei, 2018]. However, the majority of existing automatic differentiation (AD) libraries use a loosely-typed DSL, and few offer shape-safe tensor operations in a widely-used programming language.

Existing AD implementations for the JVM include Lantern [Wang et al., 2018b], Nexus [Chen, 2017] and DeepLearning.scala [Bo, 2018], however these are Scala-based and do not interoperate with other JVM languages. Kotlin $\nabla$  is fully interoperable with vanilla Java, enabling broader adoption in neighboring languages. To our knowledge, Kotlin has no prior AD implementation. However, the language contains a number of desirable features for implementing a native AD framework. Kotlin $\nabla$  primarily relies on the following language features:

- **Operator overloading and infix functions** allow a concise notation for defining arithmetic operations on tensor-algebraic structures, i.e. groups, rings and fields.
- **$\lambda$ -functions and coroutines** support functional differentiable programming with lambdas and shift-reset continuations, following Pearlmutter and Siskind [2008a] and Wang et al. [2018b].
- **Extension functions** support extending classes with new fields and methods and can be exposed to external callers without requiring sub-classing or inheritance.

Kotlin $\nabla$  models are embedded domain specific languages (eDSLs), which are essentially data structures masquerading as code. These structures may look and act like code, but are really just functions building an abstract syntax tree (AST), which can be evaluated eagerly



**Fig. 3.2.** Adapted from van Merriënboer et al. [2018]. Kotlin $\nabla$  models are data structures, constructed by an embedded DSL, eagerly optimized, and lazily evaluated at runtime.

or lazily depending on the developer’s needs. Typically these ASTs represent simple state machines, but they can also be used to implement a full-fledged programming language. Popular examples include SQL/LINQ [Meijer et al., 2006], OptiML [Sujeeth et al., 2011] and other fluent interfaces [Fowler, 2005]. In a sufficiently expressive host language, one can implement any language as a library, without needing to write a lexer, parser, compiler or interpreter. And with a carefully designed type system, users will automatically receive code completion and static analysis from their favorite developer tools. Many have observed that functional programming languages are suitable host languages [Elliott et al., 2003, Rompf and Odersky, 2010], perhaps owing to the notion of code as data.<sup>3</sup>

### 3.7. Usage

Kotlin $\nabla$  allows users to implement differentiable programs by composing operations on numerical fields to form algebraic expressions. In Listing ??, we define a function with two variables and take a series of partial derivatives with respect to each variable. The function is numerically evaluated on the interval  $(-1, 1)$  in each dimension and rendered in 3-space.

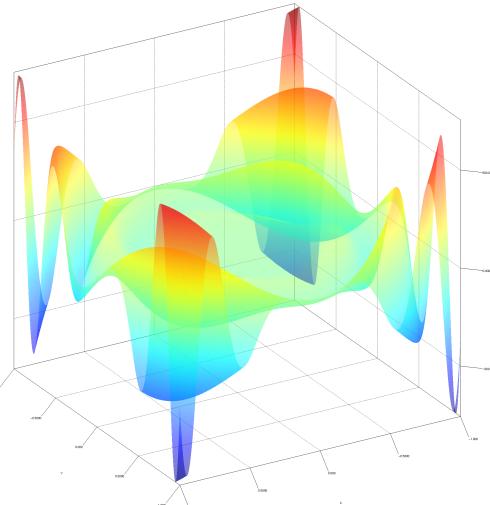
---

<sup>3</sup>i.e. homoiconicity, notably introduced by Lisp, one of the first functional programming languages

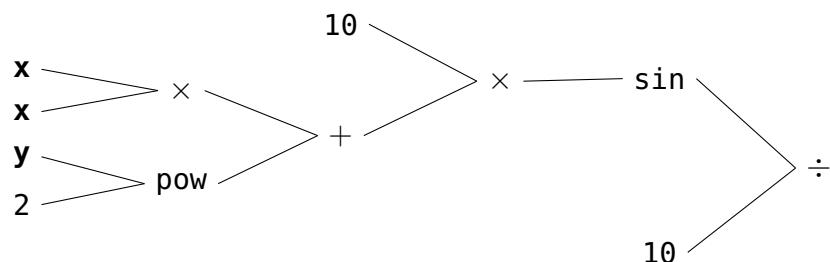
```

with(DoublePrecision) { // Uses double precision numerics for evaluation
  val x = variable("x") // Declare immutable variables (these variables
  val y = variable("y") // are symbolic constructs for differentiation)
  val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazily evaluated
  val dz_dx = d(z) / d(x) // Supports Leibniz's notation
  -----
  val d2z_dxdy = d(dz_dx) / d(y) // Mixing higher order partials
  val d3z_d2xdy = grad(d2z_dxdy)[x] // Equivalent to d(f)/d(x)
  plot3D(d3z_d2xdy, -1.0, 1.0) // Plot in 3-space (-1 < x, y, z < 1)
}

```



```
val z = sin(10 * (x * x + pow(y, 2))) / 10 // Does not perform any calculation
```



**Fig. 3.3.** Implicit DFG constructed by the original expression,  $z$  in Listing ??.

### 3.8. Type systems

Early work in type safe dimension analysis can be found in Kennedy [1994, 1996] showing how static types in array programming can be used to encode dimensionality and prevent common bugs due to dimension mismatch. Kennedy is particularly interested in units of measurement, and was followed by Jay and Sekanina, Rittri, and Zenger [1997]. Later, Kiselyov [2005], Kiselyov et al. [2010] and Griffioen [2015], show how to encode numbers to infer array sizes in more complex ways. More recently, with the resurgence of interest in tensor algebra and array programming, Chen [2017] and Rink [2018] explore how shape safe tensor operations can be encoded in a sufficiently expressive type system.

The problem we would like to solve can be summarized as follows. Given two variables  $x$  and  $y$ , and operator  $\diamond$ , how do we determine whether the expression  $z = x \diamond y$  is valid, and if so, what is the result type of  $z$ ? For matrix multiplication, when  $x \in \mathbb{R}^{m \times n}$  and  $y \in \mathbb{R}^{n \times p}$ , the expression is well-typed and we can infer  $z \in \mathbb{R}^{m \times p}$ . More generally, we would like to infer the type of  $z$  for some operator  $\otimes : (\mathbb{R}^{\mathbf{a}}, \mathbb{R}^{\mathbf{b}}) \rightarrow \mathbb{R}^{\mathbf{c}}$  where  $\mathbf{a} \in \mathbb{Z}^q, \mathbf{b} \in \mathbb{Z}^r, \mathbf{c} \in \mathbb{Z}^s$  and  $q, r, s \in \mathbb{Z}^{\geq}$ . For most linear algebra operations such as matrix multiplication,  $\mathcal{T}$  is computable in  $\mathcal{O}(1)$  – we can simply check the inner dimensions for equivalence and use the outer dimensions for inference.

Shape checking matrix operations is not always decidable, however. For arbitrary type functions  $\mathcal{T}(\mathbf{a}, \mathbf{b})$ , checking  $\mathcal{T}(\mathbf{a}, \mathbf{b}) \stackrel{?}{=} \mathbf{c}$  requires a Turing machine, at which point it would be easier to implement at runtime rather than at the type level. If  $\mathcal{T}$  is allowed to use the multiplication operator, as in the case of convolutional arithmetic [Dumoulin and Visin, 2016], shape inference becomes equivalent to Peano arithmetic, which is known to be undecidable [Gödel, 1931].

First class dependent types are useful for ensuring arbitrary shape safety (e.g. when concatenating and reshaping matrices [Xi and Pfenning, 1998]), but are unnecessary for simple equality checking, such as when multiplying two matrices.<sup>4</sup>. When the shape of a tensor is known at compile time, it is possible to type check arithmetical operations in a

---

<sup>4</sup>Less expressive type systems are still capable of performing arbitrary computation in the type checker. As specified, Java’s type system is known to be Turing Complete [Grigore, 2017] It may be possible to emulate a limited form of dependent types in Java by exploiting this property, although this may not be computationally tractable due to the practical limitations noted by Grigore

Math†	Infix	Prefix	Postfix	Type
$A(B)$	$a(b)$			$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^\lambda \rightarrow \mathbb{R}^\pi)$
$A + B$	$a + b$ $a.plus(b)$	$\text{plus}(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^\pi)$
$A - B$	$a - b$ $a.minus(b)$	$\text{minus}(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^\pi)$
$AB$	$a * b$ $a.times(b)$	$\text{times}(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times n}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{n \times p}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times p})$
$\frac{A}{B}$ $AB^{-1}$	$a / b$ $a.div(b)$	$\text{div}(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times n}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{p \times n}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times p})$
$-A$		$-a$	$a.\text{unaryMinus}()$	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^\pi)$
$+A$		$+a$	$a.\text{unaryPlus}()$	
$A+1$	$a + 1$	$\text{++}a$	$a.\text{++}, a.\text{inc}()$	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m})$
$A-1$	$a - 1$	$-a$	$a.-, a.\text{dec}()$	
$\sin(a)$		$\sin(a)$	$a.\sin()$	
$\cos(a)$		$\cos(a)$	$a.\cos()$	$(a : \mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$
$\tan(a)$		$\tan(a)$	$a.\tan()$	
$\ln(A)$		$\ln(a)$ $\log(a)$	$a.\ln()$ $a.\log()$	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m})$
$\log_b A$	$a.\log(b)$	$\log(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R})$
$A^b$	$a.\text{pow}(b)$	$\text{pow}(a, b)$		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times m})$
$\sqrt{a}$	$a.\text{pow}(1.0/2)$	$a.\text{pow}(1.0/2)$	$a.\text{sqrt}()$	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R}^{m \times m})$
$\sqrt[3]{a}$	$a.\text{root}(3)$	$a.\text{root}(3)$	$a.\text{cbrt}()$	
$\frac{da}{db}$ $a'(b)$	$a.\text{diff}(b)$	$\text{grad}(a)[b]$	$d(a) / d(b)$	$(a : C(\mathbb{R}^m)^*, b : \mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^m \rightarrow \mathbb{R})$
$\nabla a$		$\text{grad}(a)$	$a.\text{grad}()$	$(a : C(\mathbb{R}^m)) \rightarrow (\mathbb{R}^m \rightarrow \mathbb{R}^m)$

decidable type system with support for subtyping and parametric polymorphism. In practice, we can implement a shape-safe tensor algebra in Java, Kotlin, C++, C# or Typescript.

### 3.9. Shape safety

There are three broad strategies for handling shape errors in array programming:

- (1) Hide the error somehow by implicitly reshaping or broadcasting arrays
- (2) Announce the error at runtime, with a relevant message, e.g. `InvalidArgumentException`
- (3) Do not allow programs which can result in a shape error to compile

In Kotlin $\nabla$ , we use the third strategy. Consider the following program:

```

val a = Vec(1.0, 2.0) // Inferred type: Vec<Int, '2'>
val b = Vec(1.0, 2.0, 3.0) // Inferred type: Vec<Int, '3'>
val c = b + b
val d = a + b // Does not compile, shape mismatch

```

Attempting to sum two vectors whose shapes do not match will fail to compile.

```

val a = Mat('1', '4', 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, '1', '4'>
val b = Mat('4', '1', 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, '4', '1'>
val c = a * b
val d = a * a // Does not compile, inner dimension mismatch

```

Similarly, multiplying two tensors whose inner dimensions do not match will not compile.

```

val a = Mat('2', '4',
             1.0, 2.0, 3.0, 4.0,
             5.0, 6.0, 7.0, 8.0)
val b = Mat('4', '2',
             1.0, 2.0,
             3.0, 4.0,
             5.0, 6.0,
             7.0, 8.0)
val c: Mat<Double, '2', '2'> = a * b // Types are optional, but encouraged
val d = Mat('2', '1', 1.0, 2.0)
val e = c * d
val f = Mat('3', '1', 1.0, 2.0, 3.0)
val g = e * f // Does not compile, inner dimension mismatch

```

It is required to specify the input types in a method signature. Explicit return types are optional but encouraged for readability. If omitted, the type system can usually infer them:

```

fun someMatFun(m: Mat<Double, '3', '1'>): Mat<Double, '3', '3'> = ...
fun someMatFun(m: Mat<Double, '2', '2'>) = ...

```

. Shape safety is currently supported up to rank-2 tensors, i.e. matrices. To do so, we enumerate a list of integer type literals as a chain of subtypes, so that  $C <: C-1 <: C-2 <: \dots <: 1 <: 0$ , where  $C$  is the largest fixed-length dimension we wish to represent. Using this encoding, we are guaranteed linear growth in space and time for subtype checking.  $C$  can be specified by the user, who will need to regenerate the code.

```
interface Nat<T: '0'> { val value: Int }

// Integer literals have reified Int values should we need to compare them at runtime

sealed class '0'(open val value: Int = 0) { companion object: '0'(), Nat<'0'> }

open class '1'(override val value: Int = 1): '0'(i) { companion object: '1'(), Nat<'1'> }

open class '2'(override val value: Int = 2): '1'(i) { companion object: '2'(), Nat<'2'> }

open class '3'(override val value: Int = 3): '2'(i) { companion object: '3'(), Nat<'3'> }

//...This is generated code

open class '100'(open val value: Int = 100) { companion object: '100'(), Nat<'100'> }
```

Kotlin $\nabla$  supports shape-safe tensor operations by encoding tensor rank as a parameter of the operand type. Since integer literals are a chain of subtypes, we need only define tensor operations once using the highest literal, and can rely on Liskov substitution to preserve shape safety for all subtypes. Let us consider the rank-1 tensor (i.e. vector) case:

```
// <C: '1'> will accept 0 <= C via Liskov substitution
infix operator fun <C: '1', V: Vec<Float, C>> V.plus(v: V): Vec<Float, C> =
    Vec(length, contents.zip(v.contents).map { it.first + it.second })
```

The operator ‘+’ can now be used like so. Incompatible operands will cause a type error:

```
// Type-checked vector addition with shape inference
val Y = Vec('2', listOf(1, 2)) + Vec('2', listOf(3, 4)) // Y: Vec<Float, '2'>
val X = Vec('1', listOf(1, 2)) + Vec('3') // Undefined!
```

This technique can be easily extended to additional infix operators. We can also define a shape-safe vector initializer by overloading the invoke operator on a companion object:

```
open class Vec<E, Len: '1'> constructor(val len: Nat<Len>, val contents: List<E>) {
    companion object {
```



```

operator fun <T> invoke(t: T): Vec<T, '1'> = Vec('1', listOf(t))
operator fun <T> invoke(t0: T, t1: T): Vec<T, '2'> = Vec('2', listOf(t0, t1))
operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, '3'> = Vec('3', listOf(t0, t1, t2))
//...
}
}

```

Dynamic length construction is possible, although it may fail at runtime. For example:

```

val one = Vec('3', 1, 2, 3) + Vec('3', 1, 2, 3)    // Always runs safely
val add = Vec('3', 1, 2, 3) + Vec('3', listOf(t)) // May fail at runtime
val vec = Vec('2', 1, 2, 3)                         // Does not compile
val sum = Vec('2', 1, 2) + add                      // Does not compile

```

A similar syntax is used for matrices and tensors. For example, Kotlin $\nabla$  can infer the shape of matrix multiplication, and will not compile if their inner dimensions disagree:

```

val l = Mat('4', '4', // Inferred type: Mat<Int, '4', '4'>
            1, 2, 3, 4,
            5, 6, 7, 8,
            9, 0, 0, 0,
            9, 0, 0, 0)
val m = Mat('4', '3', // Inferred type: Mat<Int, '4', '3'>
            1, 1, 1,
            2, 2, 2,
            3, 3, 3,
            4, 4, 4)
val lm = l * m // Inferred type: Mat<Int, '4', '3'>
val mm = m * m // Does not compile

```

This technique originates in Haskell, which supports more powerful forms of type-level computation, *type arithmetic* [Kiselyov, 2005]. Type arithmetic makes it easy to express convolutional arithmetic [Dumoulin and Visin, 2016] and other arithmetical operations on shape variables, which is currently not possible in Kotlin $\nabla$ , or would require enumerating every possible combination of type literals.

### 3.10. Testing

Kotlin $\nabla$  claims to eliminate certain runtime errors, but how do we know the proposed implementation is not incorrect? One method, called property-based testing (PBT), is borrowed from the Haskell community and closely related to metamorphic testing [Chen et al., 1998]. Notable implementations include QuickCheck [Claessen and Hughes, 2011], Hypothesis [MacIver, 2018] and KotlinTest [Samuel and Lopes, 2018]. PBT uses algebraic properties to verify the result of an operation by constructing semantically equivalent but syntactically distinct expressions, which should produce the same answer. Kotlin $\nabla$  uses two such equivalences to validate its AD implementation:

- (1) **Analytical differentiation:** manually differentiate common functions and numerically evaluate a subset of the domain with AD.
- (2) **Finite difference approximation:** sample space of symbolic (differentiable) functions, comparing results of AD to FD.

For example, the following test checks whether the analytical derivative and the automatic derivative, when evaluated at a given point, are equal to within numerical precision:

```
val x = Var("x")
val y = Var("y")
val z = y * (sin(x * y) - x)           // Function under test
val dz_dx = d(z) / d(x)                 // Automatic derivative
val manualDx = y * (cos(x * y) * y - 1) // Manual derivative

"dz/dx should be y * (cos(x * y) * y - 1)" {
    NumericalGenerator.assertAll { x0, y0 ->
        // Evaluate the results at a given seed
        val autoEval = dz_dx(x to x0, y to y0)
        val manualEval = manualDx(x to x0, y to y0)
        autoEval shouldBeApproximately manualEval // Fails iff eps < |adEval - manualEval|
    }
}
```

PBT will search the input space for two numerical values  $x_0$  and  $y_0$ , which violate the specification, then “shrink” them to discover pass-fail boundary values. We can construct a similar test using finite differences:

```
"d(sin x)/dx should be equal to (sin(x + dx) - sin(x)) / dx" {
    NumericalGenerator.assertAll { x0 ->
        val f = sin(x)
        val df_dx = d(f) / d(x)
        val adEval = df_dx(x0)
        val dx = 1E-8
        val fdEval = (sin(x0 + dx) - sin(x0)) / dx
        adEval shouldBeApproximately fdEval
    }
}
```

There are many other ways to independently verify the numerical gradient, such as dual numbers or the complex step derivative. Another method is to compare the numerical output against a well-known implementation, such as TensorFlow. We plan to conduct a more thorough comparison of numerical accuracy and performance.

### 3.11. Operator overloading

Operator overloading enables concise notation for arithmetic on abstract types, where the types encode algebraic structures, e.g. **Group**, **Ring**, and **Field**. These abstractions are extensible to other mathematical structures, such as complex numbers and quaternions.

For example, we have an interface **Group** which overloads the operators  $+$  and  $\times$ :

```
interface Group<T: Group<T>> {
    operator fun plus(addend: T): T
    operator fun times(multiplicand: T): T
}
```

Here, we specify a recursive type bound using a method known as F-bounded polymorphism [Canning et al., 1989] to ensure that operations return the concrete type variable  $T$ ,

rather than something more generic like **Group** (effectively,  $T$  is a **self** type). Imagine a class **Fun** which has implemented **Group**. It can be used as follows:

```
fun <T: Group<T>> cubed(t: T): T = t * t * t
fun <X: Fun<X>> twiceExprCubed(e: X): X = cubed(e) + cubed(e)
```

Like Python, Kotlin supports overloading a limited set of operators, which are evaluated using a fixed precedence. In the current version of  $\text{Kotlin}\nabla$ , operators do not perform any computation, they simply construct a directed acyclic graph representing the symbolic expression. Expressions are only evaluated when invoked as a function.

### 3.12. First class functions

With higher-order functions and lambdas, Kotlin treats functions as first class citizens, following a number of recent papers in functional AD [Pearlmutter and Siskind, 2008a, Wang et al., 2018a]. This allows us to represent mathematical functions and programming functions with the same underlying abstractions (i.e. typed FP). In  $\text{Kotlin}\nabla$ , all expressions are treated as functions. For example:

```
fun <T: Group<T>> makePoly(x: Var<T>, y: Var<T>) = x * y + y * y + x * x
val x: Var<Double> = Var(1.0)
val f = makePoly(x, y)
val z = f(1.0, 2.0) // Returns a value
println(z) // Prints: 7
```

### 3.13. Numeric Tower

$\text{Kotlin}\nabla$  uses a numeric tower [St-Amour et al., 2012]. First pioneered in the functional programming language, Scheme[Sperber et al., 2009], this strategy is also suited to object oriented programming [Niculescu, 2003, 2011] and widely used in other JVM libraries such as KMath [Nozik, 2019] and Apache Commons Math [Developers, 2012]. By doing so, we are able to define common functionality on the supertype.

```

interface Group<X: Group<X>> {
    operator fun unaryMinus(): X
    operator fun plus(addend: X): X
    operator fun minus(subtrahend: X): X = this + -subtrahend
    operator fun times(multiplicand: X): X
}

interface Field<X: Field<X>>: Group<X> {
    val e: X
    val one: X
    val zero: X
    operator fun div(dividend: X): X = this * dividend.pow(-one)
    infix fun pow(exp: X): X
    fun ln(): X
}

```

The numeric tower allows us to define common behavior over abstract structures, which are easily extended to concrete number systems. For example, to later define a field over the complex numbers or other algebraic structures<sup>5</sup>, one must simply extend the numeric tower and override the default implementation. Most common mathematical operations can be defined by composing a small set of primitive operations, which can later be symbolically differentiated.

### 3.14. Algebraic data types

Algebraic data types (ADTs) in the form of sealed classes (a.k.a. sum types) facilitates a limited form of pattern matching over a closed set of subclasses. When matching against subclasses of a sealed class, the compiler enforces the author to provide an exhaustive control flow over all concrete subtypes of an abstract class. Consider the following classes:

```

class Const<T: Fun<T>>(val number: Number) : Fun<T>()
class Sum<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()
class Prod<T:Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()

```




---

<sup>5</sup>ex. In order to calculate derivatives in a quaternion neural network.[Isokawa et al., 2003]

```

class Var<T: Fun<T>>: Fun<T>() { override val variables: Set<Var<X>> = setOf(this) }

class Zero<T: Fun<T>>: Const<T>(0.0)

class One<T: Fun<T>>: Const<T>(1.0)

```

When branching on the type of a sealed class, consumers must explicitly handle every subclass, since incomplete control flow will not compile, rather than fail silently at runtime.

```

sealed class Fun<X: Group<X>>(open val variables: Set<Var<X>> = emptySet()): Group<Fun<X>> {

    constructor(vararg fns: Fun<X>): this(fns.flatMap { it.variables }.toSet())

    // Since the subclasses of Fun are a closed set, no 'else -> ...' is required.

    operator fun invoke(map: Map<Var<X>, X>): Fun<X> = when (this) {
        is Const -> this
        is Var -> map.getOrDefault(this) { this }
        is Prod -> left(map) * right(map) // Smart casting implicitly casts after checking
        is Sum -> left(map) + right(map)
    }

    fun diff(): Fun<X> = when(this) {
        is Const -> Zero
        is Var -> One
        // Product rule: d(u*v)/dx = du/dx * v + u * dv/dx
        is Prod -> left.diff() * right + left * right.diff()
        is Sum -> left.diff() + right.diff()
    }

    operator fun plus(addend: Fun<T>) = Sum(this, addend)

    operator fun times(multiplicand: Fun<T>) = Prod(this, multiplicand)
}

```

Kotlin's smart-casting implicitly downcasts the abstract type `Fun` as a subtype, e.g. `Sum` after performing an `is Sum` check. Otherwise, we would need to write `(this as Sum).left` in order to access its member, `left`. If the type cast was mistaken, a `ClassCastException` would be thrown, which smart casting also prevents.

### 3.15. Multiple Dispatch

In conjunction with ADTs, Kotlin $\nabla$  also uses multiple dispatch to instantiate the most specific result type of applying an operator based on the type of its operands. While multiple dispatch is not an explicit language feature, it can be emulated using inheritance.

Building on the previous example, suppose we would like to perform algebraic simplification. This can be useful for reducing expression swell and improving numerical stability. We can use multiple dispatch to branch on the type of a subexpression at runtime. *Smart casting* lets us access members of a class after checking its type, as if it were casted:

```
override fun times(multiplicand: Fun<X>): Fun<X> =  
    when {  
        this == zero -> this  
        this == one -> multiplicand  
        multiplicand == one -> this  
        multiplicand == zero -> multiplicand  
        this == multiplicand -> pow(two)  
        // w/o smart cast: Const((this as Const).number * (multiplicand as Const).number)  
        this is Const && multiplicand is Const -> Const(number * multiplicand.number)  
        // Further simplification is possible using rules of replacement  
        else -> Prod(this, multiplicand)  
    }  
  
val result = Const(2.0) * Sum(Var(2.0), Const(3.0))  
//           = Sum(Prod(Const(2.0), Var(2.0)), Const(6.0))
```

### 3.16. Extension Functions

Extension functions augment external classes with new fields and methods. By using a technique known as context oriented programming [Hirschfeld et al., 2008], we can expose custom extensions (e.g. through **DoubleContext**) to consumers without requiring subclassing or inheritance.

```
data class Const<T: Fun<T>>(val number: Double) : Fun<T>()  
data class Sum<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>(left, right)
```

```

data class Prod<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>(left, right)

object DoubleContext {
    operator fun Number.times(expr: Fun<Double>) = Const(toDouble()) * expr
}

```

Now, we can use the context to define another extension, `Fun.multiplyByTwo`, which computes the product inside a `DoubleContext`, using the operator overload defined above:

```

fun Fun<Double>.multiplyByTwo() = with(DoubleContext) { 2 * this }

```

Extensions can also be defined in another file or context and imported on demand. This approach was borrowed from KMath [Nozik, 2019], another mathematical library for Kotlin.

### 3.17. Automatic, Symbolic Differentiation

It has long been claimed by the AD literature that automatic differentiation is not symbolic differentiation [Baydin et al., 2015a]. Many, including the author of this thesis, has suspected this claim to be misleading. Recently, the claim has been questioned [Wang et al., 2018c] and refuted [Laue, 2019]. While it may be true that certain implementations of automatic differentiation interleave numerical and symbolic differentiation during a program's execution, interleaving differentiation and numerical execution is certainly not a prerequisite for a differentiation library to be considered *automatic*, particularly when popular AD implementations, such as Theano [Al-Rfou et al., 2016], have chosen the symbolic route. It is our view that symbolic differentiation is one type of automatic differentiation, in particular, one which affords more flexibility to perform global optimizations and rewriting procedures. These optimizations would otherwise be impossible if attempted at runtime, with only partial information about the graph.

### 3.18. Coroutines

Coroutines are a generalization of subroutines for non-preemptive multitasking, typically implemented using continuations. Continuations are a mechanism that allow functions to access and modify subsequent computation. In continuation passing style (CPS), every function, in addition to its usual arguments, takes a second function representing the subsequent

computation to be performed. Rather than returning to its caller, the function invokes its continuation immediately prior to completion, and the process is restarted. If the continuation is empty, the program halts.

One form of continuation, known as shift-reset a.k.a. delimited continuations, are sufficient for implementing reverse mode AD with operator overloading alone (without any additional data structures) as described by Wang et al. in *Shift/Reset the Penultimate Backpropagator* [Wang et al., 2018c] and later in *Backpropagation with Continuation Callbacks* [Wang et al., 2018a]. Delimited continuations can be implemented with Kotlin’s Coroutines support.

### 3.19. Comparison

Inspired by Stalin $\nabla$ , Autograd, DiffSharp, Myia, Nexus, Lantern, Tangent et al., Kotlin $\nabla$  attempts to port recent developments in automatic differentiation (AD) to the Kotlin language. It introduces a number of experimental ideas, including compile-time shape-safety, algebraic simplification and numerical stability checking through property-based testing. Unlike most existing automatic differentiation libraries, Kotlin $\nabla$  is a purely symbolic, graph-based solution that does not require any compiler augmentation or runtime reflection. As we have seen, it achieves this primarily through operator overloading, polymorphism, and pattern matching. The practical advantage of this approach is that it can be implemented as a simple library or *embedded domain specific language*, thereby employing the host language’s type system. It is particularly well suited to functional programming, and makes use of a number of functional programming concepts, including lambda expressions, higher order functions, partial application, algebraic data types and more.

Framework	Language	Symbolic Differentiation	Automatic Differentiation	Functional Programming	Type Safe	Shape Safe	Differentiable Programming	Multiplatform
Kotlin $\nabla$	Kotlin	✓	✓	✓	✓	✓	✗	✗
DiffSharp	F#	✗	✓	✓	✓	✗	✓	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗
Myia	Python	✓	✓	✓	✓	✓	✓	✗
Deeplearning.scala	Scala	✗	✓	✓	✓	✗	✓	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗
Lantern	Scala	✗	✓	✓	✓	✗	✓	✗
Grenade	Haskell	✗	✓	✓	✓	✓	✗	✗
Eclipse DL4J	Java	✓	✓	✗	✓	✗	✗	✗
Halide	C++	✗	✓	✗	✓	✗	✓	✗
Stalin $\nabla$	Scheme	✗	✓	✓	✗	✗	✗	✗



# Chapitre 4

---

## Verification and validation

“If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere... then we had better be quite sure the purpose put into the machine is the purpose which we really desire.”

---

—Wiener [1960], *Some moral and technical consequences of automation*

Roughly speaking, in backpropagation we perform gradient descent on the parameters of a program for a given input. In adversarial testing, we do gradient ascent on the input, at a given parameter setting.

Neural networks and differentiable programming have provided a powerful set of optimization tools for training learning algorithms. However, these methods are often brittle to small variations in the input space, and have difficulty with generalization. In contrast, these same techniques used for probing the failure modes of neural networks can be applied to adversarial test case generation for traditional programs.

### 4.1. Background

Suppose we have a program  $P : \mathbb{R} \rightarrow \mathbb{R}$  where:

$$P(p_0) = p_n \circ p_{n-1} \circ p_{n-2} \circ \dots \circ p_1 \circ p_0 \circ p_0 \quad (4.1.1)$$

From the chain rule of calculus, we know that:

$$\nabla p_i(p_0) = \frac{dp_{i+1}}{dp_i} \cdot \frac{dp_i}{dp_{i-1}}, \forall i \in (0, n) \quad (4.1.2)$$

More broadly, if  $P : \mathbb{R}^n \rightarrow \mathbb{R}^k$ , and  $p_i : \mathbb{R}^{\dim_{\mathbb{R}}(p_{i-1})} \rightarrow \mathbb{R}^{\dim_{\mathbb{R}}(p_{i+1})} \forall i \in (0, n)$ :

$$\nabla p_i(p_0) = \nabla p_i(p_{i-1}(p_0)) \cdot \nabla p_{i-1}(p_0) \quad (4.1.3)$$

Imagine a single test  $T : \mathbb{R} \rightarrow \mathbb{B}$ . Consider the following example:

$$T(P) = \forall x \in (0, 1), P(x) < C \quad (4.1.4)$$

How can we find a set of inputs that break the test under a fixed computational budget (i.e. constant number of program evaluations)? In other words:

$$D_T : \{x^i \sim \mathbb{R}(0, 1) \mid P(x^i) \implies \neg T\}, \text{maximize}|D_T| \quad (4.1.5)$$

If we have no information about the program implementation or its input distribution,  $D_P$ , we can do no better than random search [Wolpert et al., 1997]. However, if we know something about the input distribution, we could re-parameterize the distribution to incorporate our knowledge. Assuming the program has been tested on common inputs, we might consider sampling  $x \sim \frac{1}{D_P}$  for inputs that are infrequent. If we knew how  $P$  were implemented, we could prioritize our search in input regions leading towards internal discontinuities (e.g. edge cases in software testing). However for functions that are continuous and differentiable, these heuristics are almost certainly insufficient.

Another strategy, independent of how candidate inputs are selected, is to use some form of gradient based optimization in the search procedure. The gradient of  $P$ 's loss with respect to  $x$  (assuming  $\theta$  is fixed<sup>1</sup>) is  $\nabla_x \mathcal{L}(P(x; \theta))$ . Instead of minimizing the loss, we want to maximize it. So the vanilla gradient update step ?? becomes:

$$x_{n+1}^i = x_n^i + \alpha \nabla_{x_n} \mathcal{L}(P(x_n; \theta)) \quad (4.1.6)$$

We hypothesize that if the implementation of  $P$  were flawed and a counterexample to (3) existed, as sample size increased, a subset of gradient descent trajectories would fail to converge, a portion would converge to local minima, and a subset of trajectories would discover inputs violating the program specification. How would such a search procedure look in practice?

Consider Algorithm 1.

---

<sup>1</sup>In contrast with backpropagation, where the parameters are updated.

**Input** : Program  $P$ , specification  $T$ , evaluation budget  $Budget$

**Output:**  $D_T$ , the set of inputs which cause  $P$  to fail on  $T$

$D_T = [];$

$evalCount = 0;$

**while**  $evalCount \leq Budget$  **do**

*sample candidate input  $x^i$  according to selection strategy  $S$ ;*

**if**  $P(x^i) \implies \neg T$  **then**

*append  $x^i$  to  $D_T$*

**else**

$n = 0;$

$x_n^i = x^i;$

**while**  $n \leq C \wedge evalCount \leq Budget \wedge \neg converged$  **do**

$n++;$

$x_n^i = x_{n-1}^i - \alpha \frac{dP}{dx};$

**if**  $P(x_n^i) \implies T$  **then**

*append  $x_n^i$  to  $D_T$*

*break;*

**end**

$evalCount++;$

**end**

**end**

$evalCount++; i++;$

**end**

**Algorithm 1:** Algorithm for finding test failures. First select a candidate input  $x^i$  according to sampling strategy  $S$  (e.g. uniform random, or a neural network which takes  $P$  and  $T$  as input). If  $P(x^i)$  violates  $T$ , we can append  $x^i$  to  $D_T$  and repeat. Otherwise, we follow the gradient of  $\mathcal{L}(P, x)$  with respect to  $x$  and repeat until test failure, gradient descent convergence, or a fixed number of steps  $C$  are reached before resampling  $x^{i+1}$  from the initial sampling strategy  $S$  to ensure each gradient descent trajectory will terminate before exhausting our budget.



# Chapitre 5

---

## Software Maintenance and Reproducibility

In this chapter, we will discuss the challenges of software reproducibility and how best practices in software engineering such as continuous integration and containers can help researchers mitigate the variability associated with building and running software. Our work focuses on computational determinism, and does not consider the variability of distributional shift or related statistical notions of variation.

In order to address the issue of software reproducibility, we constructed a set of tools and development workflows that draw on best practices in software engineering. These tools are primarily built around containerization, a widely adopted virtualization technology in the software industry. In order to lower the barrier of entry for developers and minimize variability across hardware platforms, we provide a state-of-the-art container infrastructure based on Docker [Merkel, 2014], one popular container engine. Docker allows us to construct versioned deployment artifacts that represent the entire filesystem, and manages resource constraints via a sandboxed runtime environment.

### 5.1. Dependency management

One common source of variability in software development are software dependencies. For many years, developers struggled with dependency management before it was discovered the dependency resolution problem was NP-complete [Abate et al., 2012]. If we assume no two versions of the same dependency can be installed simultaneously, then for a set of software packages which need to be installed, and dependencies required to install them, determining the latest version of the dependencies which satisfy all requirements is as hard

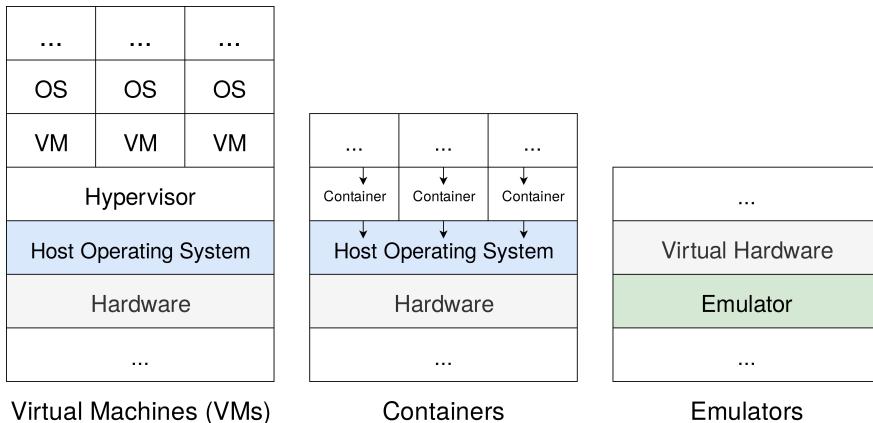
as the hardest problems in NP. Informally, this problem is known as *dependency hell* and grows increasingly troublesome as software projects grow and introduce new dependencies.

Dependency hell does not just occur inside individual software projects, but across projects and development environments. Hundreds of package managers have been developed for various operating systems, programming languages, and development frameworks. Ubuntu has the Advanced Package Tool (**apt**), macOS has Homebrew (**brew**), Windows has Chocolatey (**choco**). Most programming language ecosystems have their own bespoke package managers; Conan for C/C++, Maven for Java, and Cabal for Haskell. Python has developed several overlapping solutions for package management, including pip, Anaconda, PyEnv, Virtualenv, and others. Some of these install system-wide packages, and others provide command line environments. Over the lifetime of a computer system, as packages are installed and removed it becomes difficult to keep track of changes and their effects.

The problem basically stems from the requirement that no two versions of the same dependency can be installed simultaneously. In addition, software installers tend to spray files across the file system, which can become corrupted and are difficult to completely remove. To address these issues, some notion of “checkpointing” is required, so that when new software is installed, any future changes can be traced and reverted. Backups would do the job, but are cumbersome to manage and are unsuitable for development purposes. Rather, it would be convenient if there were a tool which allowed applications to setup a private file system, install their dependencies, and avoid contaminating the host OS.

## 5.2. Operating systems and virtualization

With the growth of developer operations (devops) a number of solutions emerged for building and running generic software artifacts. Most universal are emulators, which effectively simulate the physical circuitry of a foreign processor architecture, and thereby any software which runs on it. Another solution was virtual machines (VMs), a form of isolated runtime environment which use a *hypervisor* to mediate access to hardware, but otherwise run software on physical hardware. The downside of both methods is their efficiency. Virtual machines contain full-fledged operating systems and are cumbersome to run and debug, especially to build or run a small application on a foreign OS, as is often their use case.



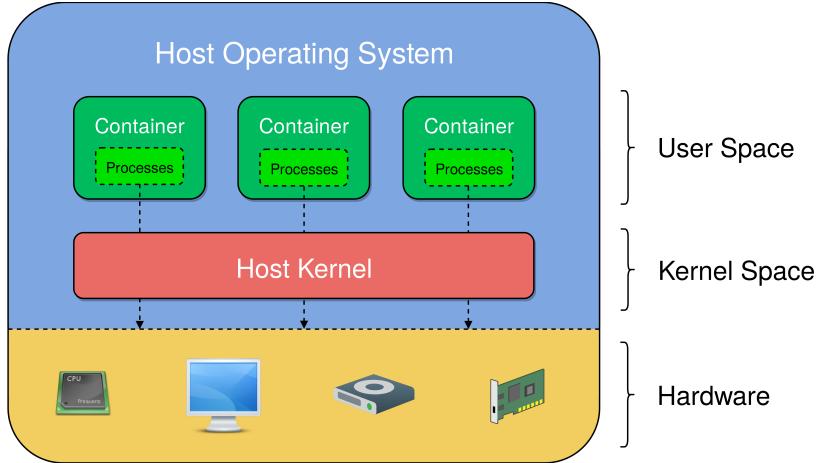
**Fig. 5.1.** Virtualization is a very resource expensive proposition. Containerization is cheaper, as it shares a kernel with the host OS. Emulation allows us to emulate hardware as software. Any of these methods can be used in conjunction with any other method.

Emulators are running a foreign processor architecture, which can run significantly more slowly depending on the host CPU.

In 2006, Linux introduced a variety of new kernel features for controlling groups of processes, under the aegis of **cgroups** [Menage, 2007]. Collectively, these features provide a form of lightweight virtualization, offering many benefits of virtual machines (VMs) such as resource control and namespace isolation, without the computational overhead. These features also paved the way for a set of tools that are today known as containers. Unlike VMs, containers share a common kernel, but remain isolated from their host OS and sibling containers. Where VMs often require server-class hardware to run smoothly, containers are suitable for a much broader class of mobile and embedded platforms.

### 5.3. Containerization

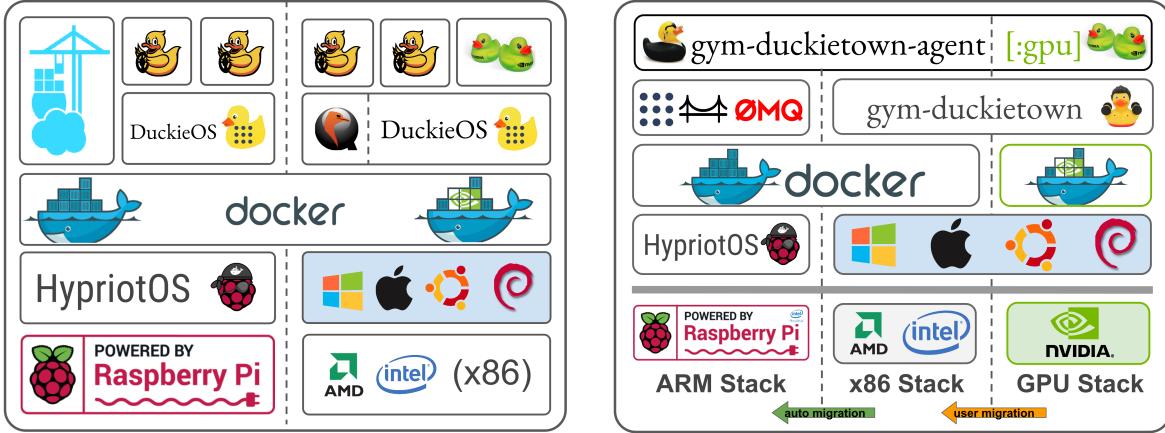
One of the challenges of distributed software development across heterogeneous platforms is the problem of variability. With the increasing pace of software development comes the added burden of software maintenance. As hardware and software stacks evolve, so too must source code be updated to build and run correctly. Maintaining a stable and well documented codebase can be a considerable challenge, especially in an academic setting where contributors are frequently joining and leaving the project. Together, these challenges present significant obstacles to experimental reproducibility and scientific collaboration.



**Fig. 5.2.** Containers live in user space. By default they are sandboxed from the host OS and sibling containers, but unlike VMs, share a common kernel with each other and the host OS. All system calls are passed through host kernel.

Docker containers are sandboxed runtime environments that are portable, reproducible and version controlled. Each environment contains all the software dependencies necessary to run the packaged application(s), but remains isolated from the host OS and file system. Docker provides a mechanism to control the resources each container is permitted to access, and a separate Linux namespace for each container, isolating the network, users, and file system mounts. Unlike virtual machines, container-based virtualization like Docker only requires a lightweight kernel, and can support running many simultaneous containers with close to zero overhead. A single Raspberry Pi is capable of supporting hundreds of running containers.

While containerization simplifies the process of building and deploying applications considerably, it also introduces some additional complexity to the software development lifecycle. Docker, like most container platforms, uses a layered filesystem. This enables Docker to take an existing “image” and change it by installing new dependencies or modifying its functionality. Images may be based on a number of lower layers, which must periodically be updated. Care must be taken when designing the development pipeline to ensure that such updates do not silently break a subsequent layer as described earlier.



**Fig. 5.3.** Container infrastructure. Left: The ROS Stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their architecture, may be run on an ARM device using an accelerator.

## 5.4. Docker and ROS

Prior work has explored the Dockerization of ROS containers [White and Christensen, 2017]. This work forms the basis for our own, which is extended specifically for the Duckietown platform [Paull et al., 2017].

The Duckietown platform supports two primary instruction set architectures: x86 and ARM. To ensure the runtime compatibility of Duckietown packages, we cross-build using hardware virtualization to ensure build artifacts can be run on all target architectures. Runtime emulation of foreign artifacts is also possible, using a similar technique.<sup>1</sup> For performance and simplicity, we only build ARM artifacts and use emulation where necessary (e.g., on x86 devices). On ARM-native, the base operating system is HypriotOS, a light-weight Debian distribution with built-in support for Docker. For both x86 and ARM-native, Docker is the underlying container platform upon which all user applications are run, inside a container.

<sup>1</sup>For more information, this technique is described in further depth at the following URL: <https://www.balena.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/>.

## 5.5. Retrospective

One issue encountered is whether to package source code directly inside the container, or to store it separately. If source code is stored separately, a developer can use a shared volume on the host to build the artifacts. In this case, while build artifacts are standalone reproducible, they are not easily modified or inspected. The alternative is to ship code directly inside the container, where any changes to the source code will trigger a subsequent rebuild, effectively tying the sources and the build artifacts together. Including source code alongside build artifacts has the benefit of improved readability and diagnostics. If a user requires technical assistance, troubleshooting becomes much easier when the source code is directly accessible. But doing so adds some friction during development, which has caused users to struggle with environment setup. Our solution is to store all sources on the local development environment and rebuild the image periodically, by copying sources into the image.

# Chapitre 6

---

## Case study: application for autonomous robotics

As a case study, we have implemented a mobile application using ROS, Docker, and Android, using the proposed toolchain.

### 6.1. Design

Designed with Hatchery.

### 6.2. Implementation

Implementation includes Kotlin

### 6.3. Verification and validation

Verified using property-based testing.

### 6.4. Containerization

Deployed and CI-tested using Docker.



# Chapitre 7

---

## Conclusion

### 7.1. Future work

#### 7.1.1. Requirements Engineering

Often it is not possible, or desirable to summarize the performance of a complex system using a single variable. In multi-objective optimization, we have the notion of pareto-efficiency...

Traditional software engineering has followed a rigorous process model and testing methodology. This model has guided the development of traditional software engineering, intelligent systems will require a re-imagining of these ideas to build systems that adapt to their environment during operation. Intelligent systems are designed with objective functions, which are typically one- or low-dimensional metrics for evaluating the performance of the system. Most often, these take the form of a single criteria, such as an *error* or *loss* which can represent descriptive phenomena such as latency, safety, energy efficiency or any number of objective measures.

For example, in the design of a web based advertisement recommendation system, we can optimize for various objectives such as click rate, engagement, sales conversion. So long as we can measure these parameters, with today's powerful function approximators, we can optimize for any single criterion or combination thereof. Much of the work involved in machine learning is to find representations which are amenable to learning, and preventing unintended consequences. For example, by optimizing for click rate, we create an artificial market for click bots. Similarly, in self driving cars, we often want to optimize for passenger

safety. However by doing so naively, we create a vehicle that never moves, or always yields to nearby vehicles.

When building an intelligent system developers must first ask, “What are the requirements of the system?” This question is often the most troublesome part, because the requirements must not be fuzzy specifications like traditional software engineering, but precise, programmable directives. “The system must be fast,” is not sufficiently precise. These kinds of requirements must be translated into statistical loss functions, so intelligent systems engineers must be very precise when specifying requirements. If we simply say, “The system must produce a valid response as quickly as possible, in less than 100ms,” is better, but leaves open the possibility of returning an empty response.

In traditional software engineering, it is reasonable to assume the people who are implementing a system have some implicit knowledge and are generally well-intentioned human beings working towards the same goal. When building an intelligent system, a more reasonable assumption is that the entity implementing our requirements is a naive but powerful genie, and possibly an adversarial one. When given an optimization metric, it will take every available shortcut to meet that metric. If we are not careful about requirements engineering, this entity can produce a system that does not work, or has unintended consequences.

In the strictest sense, designing a good set of requirements is indistinguishable from implementing the system. With the right language abstractions (e.g. declarative programming), requirements and implementation can be the same thing. These ideas have been explored in recent decades with languages like SQL and Prolog. While these are toy systems, neural networks can express much larger classes of functions than traditional software engineering.

### 7.1.2. Continuous Delivery and Continual Learning

An ongoing trend in modern software and systems engineering is the transition away from long development cycles towards continuous integration and deployment. Development teams across the industry are encouraged to iterate in a series of short sprints between feature development and deployment. In some cases, software is shipped to users on a nightly basis, with automated testing and deployment. Similarly, intelligent systems have a need to continuously adapt to their environment, and will change their code on an even shorter basis.

## Bibliography

---

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülcöhre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus

Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Speckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguy, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016. URL <http://arxiv.org/abs/1605.02688>.

Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. *Available at SSRN 3239970*, 2018.

Nada Amin and Ross Tate. Java and scala’s type systems are unsound: the existential crisis of null pointers. *Acm Sigplan Notices*, 51(10):838–848, 2016.

John Backus. *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. ACM, 2007.

Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015a. URL <http://arxiv.org/abs/1502.05767>.

Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: Automatic differentiation library. *CoRR*, abs/1511.07727, 2015b. URL <http://arxiv.org/abs/1511.07727>.

Richard Ernest Bellman, Ho Kagiwada, and Robert E Kalaba. Wengert’s numerical method for partial derivatives, orbit determination and quasilinearization. *Communications of the ACM*, 8(4):231–232, 1965.

Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

Yang Bo. Deep Learning.scala: A simple library for creating complex neural networks. 2018. URL <https://github.com/ThoughtWorksInc/DeepLearning.scala>.

Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4243–4250. IEEE, 2018.

Roman V Bunyi, Stephen DH Hsu, and Anthony Zee. Is hilbert space discrete? *Physics Letters B*, 630(1-2):68–72, 2005.

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, volume 89, pages 273–280, 1989.

Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017. doi: 10.1145/3136000.3136001. URL <http://doi.acm.org/10.1145/3136000.3136001>.

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong . . . , 1998.

Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1985.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of git repositories. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503. IEEE, 2015.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pages 7178–7189, 2018.

Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. *CoRR*, abs/1611.01652, 2016. URL <http://arxiv.org/abs/1611.01652>.

[org/abs/1611.01652](https://arxiv.org/abs/1611.01652).

Commons Math Developers. Apache commons math. *Forest Hill, MD, USA: The Apache Software Foundation*, 2012.

Stuart E Dreyfus. Artificial neural networks, back propagation, and the kelley-bryson gradient procedure. *Journal of guidance, control, and dynamics*, 13(5):926–928, 1990.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70, 2018.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of functional programming*, 13(3):455–481, 2003.

George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

M. Fowler. Fluent interface, 2005. URL <http://martinfowler.com/bliki/FluentInterface.html>.

Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, pages 758–769. IEEE Press, 2017.

Yossi Gil and Tomer Levy. Formal language recognition with the java type checker. 56, 2016.

Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

Charles F Goldfarb. A generalized approach to document markup. In *ACM Sigplan Notices*, volume 16, pages 68–73. Citeseer, 1981.

Andreas Griewank. Some bounds on the complexity of gradients, jacobians, and hessians. In *Complexity in numerical optimization*, pages 128–162. World Scientific, 1993.

Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

PR Griffioen. Type inference for array programming with dimensioned vector spaces. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, page 4. ACM, 2015.

- Radu Grigore. Java generics are Turing Complete. pages 73–85, 2017. doi: 10.1145/3009837.3009871. URL <http://doi.acm.org/10.1145/3009837.3009871>.
- Martin Guenther. Are serious things done with ros in python? - discourse.ros.org. <https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6>. (Accessed on 04/12/2019).
- Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006.
- Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. *Journal of Object technology*, 7(3):125–151, 2008.
- Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- Teijiro Isokawa, Tomoaki Kusakabe, Nobuyuki Matsui, and Ferdinand Peper. Quaternion neural network and its application. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 318–324. Springer, 2003.
- Aleksei Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. *Cybernetic predicting devices*. CCM Information Corporation, 1965.
- Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.
- C Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, Citeseer.
- Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.
- Nidhi Kalra and Susan M Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.
- Andrew Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

- Andrew John Kennedy. Programming languages and dimensions. Technical report, University of Cambridge, Computer Laboratory, 1996.
- Oleg Kiselyov. Number-parameterized types. *The Monad. Reader*, 5:73–118, 2005.
- Oleg Kiselyov, S Peyton Jones, and Chung-chieh Shan. Fun with type functions (version 3). *Tony Hoare’s 75th birthday celebration*, 2010.
- Chris Lattner and Richard Wei. Swift for tensorflow. 2018. URL <https://github.com/tensorflow/swift>.
- Soeren Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. *arXiv preprint arXiv:1904.02990*, 2019.
- Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- Joseph Carl Robnett Licklider. Man-computer symbiosis. *IRE transactions on human factors in electronics*, (1):4–11, 1960.
- Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master’s Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.
- Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.
- Matthew M Loper and Michael J Black. OpenDr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.
- Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.
- David R. MacIver. Hypothesis. <https://github.com/HypothesisWorks/hypothesis>, 2018.
- Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.
- Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium*, volume 2, pages 45–57. Citeseer, 2007.

- Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
- VIRGINIA Niculescu. A design proposal for an object oriented algebraic library. *Studia Universitatis "Babes-Bolyai", Informatica*, 48(1):89–100, 2003.
- Virginia Niculescu. On using generics for implementing algebraic structures. *Studia Universitatis Babes-Bolyai, Informatica*, 56(4), 2011.
- Alexander Nozik. Kotlin - new language for scientific programming. In *Proceedings of 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Mar 2019. URL <https://indico.cern.ch/event/708041/contributions/3276141/>.
- Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.
- Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008a.
- Barak A Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make automatic differentiation sound and efficient. pages 79–90, 2008b.
- Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.
- Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, September 2017. ISSN 0001-0782. doi: 10.1145/3126905. URL <http://doi.acm.org/10.1145/3126905>.

Norman A Rink. Modeling of languages for tensor manipulation. *arXiv preprint arXiv:1801.08771*, 2018.

Mikael Rittri. Dimension inference under polymorphic recursion. Citeseer.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Stephen Samuel and Leonardo Colman Lopes. KotlinTest.  
<https://github.com/kotlintest/kotlintest>, 2018.

Claude E Shannon. Xxi. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised 6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *International Symposium on Practical Aspects of Declarative Languages*, pages 289–303. Springer, 2012.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

Ross Tate. Mixed-site variance. FOOL, 2013.

Bart van Merriënboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, pages 6256–6265, 2018.

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

- Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *Advances in Neural Information Processing Systems*, pages 10180–10191, 2018a.
- Fei Wang, Xilun Wu, Gr é gory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018b. URL <http://arxiv.org/abs/1803.10228>.
- Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228*, 2018c.
- Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7675–7684, 2018d.
- Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.
- Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.
- Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- Ruffin White and Henrik Christensen. Ros and docker. In *Robot Operating System (ROS)*, pages 285–307. Springer, 2017.
- Norbert Wiener. Some moral and technical consequences of automation. *Science*, 131(3410):1355–1358, 1960.
- David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.
- Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Notices*, volume 33, pages 249–257. ACM, 1998.
- Christoph Zenger. Indexed types. *Theoretical computer science*, 187(1-2):147–165, 1997.



