Université de Montréal

Programming tools for intelligent systems

par

Breandan Considine

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Informatique

février 2020

Université de Montréal

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

Programming tools for intelligent systems

présenté par

Breandan Considine

a été évalué par un jury composé des personnes suivantes :

Marc Feeley
(président-rapporteur)

Liam Paull
(directeur de recherche)

Michalis Famelis
(codirecteur)

Eugène Syriani
(membre du jury)

Abstract

Programming tools are computer programs which help humans program computers. Tools come in all shapes and forms, from editors and compilers to debuggers and issue trackers. Each of these facilitates a core task in the programming workflow which consumes cognitive resources when performed manually. In this thesis, we share some lessons learned designing four tools that facilitate the development of intelligent software systems, particularly in robotics settings. First, we introduce an integrated development environment (IDE) for programming in the Robot Operating System (ROS), called Hatchery. Second, we describe Kotlin∇, a language and type system for differentiable programming, an emerging programming paradigm in intelligent systems. Third, we present an automated test suite for differentiable programming, based on adversarial and metamorphic testing. Fourth, we explore a container infrastructure based on Docker, which enables the development of reproducible robotics software for the Duckietown platform. Finally, we offer some concluding remarks about the past, present and future of programming tools for intelligent systems.

Acknowledgements

I would like to thank Gimmey, Mom, Uncle Mark, and Dad for their unfailing love and support. Hanneli Tavante for teaching me type theory and the beauty of functional programming. Siyan Wang for his fellowship and adventures. Xiaoyan Liu for planting in me the seed of mathematics. Uncle Andy for watering the seed for many years. Aunt Shannon, Adam Devoe and Jacquie Kirrane for encouraging me to pursue grad school. Arthur Nunes-Harwitt for teaching me algorithmic differentiation a long time ago. Renee Miller for sparking my interest in neural science. Ian Clarke for showing me a clever new language called Kotlin. Hadi Hariri for placing more trust in me than I deserved. Lukas Eder and Eugene Petrenko for showing me the magic of type-safe DSLs and giving me advice about grad school. Rusi Hristov for his patience and mentorship. Dmitry Serdyuk for introducing me to Montréal. Isabela Albuquerque and João Monteiro for showing me what good research looks like. Manfred Diaz and Maxime Chevalier Boisvert for the inspiration, conversations and feedback. Florian Golemo for his excellent engineering and architectural advice. Ryan Turner, Saikrishna Gottipati, Vincent Mai, Krishna Murthy, Bhairav Mehta, Christos Tsirigotis, Konstantin Solomatov and Xujie Si for the interesting conversations. Pascal Lamblin, Olivier Breleux and Bart van Merriënboer and for lighting the path between ML and PL. Conal Elliott for teaching me the importance of simplicity and denotational semantics. Christian Perone for introducing me to PyTorch, Alexander Nozik, Erik Meijer, Kiran Gopinathan, Roman Elizarov and Jacob Miller for their useful comments and feedback related to Kotlin∇. Miltos Allamanis for showing me there is room for SE in ML. Celine Begin at the Université de Montréal for helping a stranger on a cold winter's eve in 2017. Stefan Monnier for thoughtfully and thoroughly replying to my rambling emails. Andrea Censi for his advice and encouragement. Last but not least, I wish to thank my brilliant advisors Liam Paull for taking a chance on an out-of-distribution sample, providing strong gradients and giving me far more credit than I deserved, and Michalis Famelis for teaching me the value of intuitionistic logic, formal methods, and self-discipline. Thank you!

Contents

Abstract v

Acknowledgements vii

List of tables xv

List of figures xvii

Chapter 1. Introduction 1

1.1. Stages in the software development lifecycle 3

1.2. Design: Programming tools for robotics 4

1.3. Implementation: Type-safe differentiable programming 6

1.4. Verification: Testing intelligent systems 8

1.5. Maintenance: Tools for reproducible robotics 9

1.5.1. Iconography 12

Chapter 2. Programming tools for robotics 15

2.1. Introduction to the Robot Operating System 16

2.2. Installation 19

2.3. Plugin development 19

2.3.1. Refactoring 20

2.3.2. Parsing 20

2.3.3. Running and debugging 22

2.3.4. User interface 23

2.4. Ongoing work . . . 25
2.5. Conclusion . . . 28
Chapter 3. Type-safe differentiable programming . . . 29
3.1. Automatic differentiation . . . 30
3.2. Differentiable programming . . . 32
3.3. Static and dynamic languages . . . 35
3.4. Imperative and functional languages . . . 35
3.5. Kotlin . . . 36
3.6. Kotlin∇ . . . 37
3.7. Usage . . . 39
3.8. Type systems . . . 40
3.9. Shape safety . . . 41
3.10. Testing . . . 45
3.11. Operator overloading . . . 47
3.12. First-class functions . . . 47
3.13. Numeric Tower . . . 48
3.14. Algebraic data types . . . 49
3.15. Multiple Dispatch . . . 50
3.16. Extension Functions . . . 51
3.17. Automatic, Symbolic Differentiation . . . 52
3.18. Coroutines . . . 52
3.19. Comparison . . . 53

3.20. Future work . . . 55
3.21. Conclusion . . . 56
Chapter 4. Testing intelligent systems . . . 57
4.1. Unit Testing . . . 58
4.2. Integration Testing . . . 58
4.3. Fuzz Testing . . . 59
4.4. Property-based Testing . . . 59
4.5. Metamorphic testing . . . 61
4.6. Adversarial Testing . . . 62
4.7. Generative Adversarial Testing . . . 65
4.8. Probabilistic Adversarial Testing . . . 67
4.9. Conclusion . . . 72
Chapter 5. Tools for reproducible robotics . . . 75
5.1. Dependency management . . . 76
5.2. Operating systems and virtualization . . . 77
5.3. Containerization . . . 77
5.4. Introduction to Docker . . . 79
5.4.1. Creating an image snapshot . . . 80
5.4.2. Writing an image recipe . . . 83
5.4.3. Layer Caching . . . 84
5.4.4. Volume Sharing . . . 88
5.4.5. Multi-stage builds . . . 89
5.5. ROS and Docker . . . 90

5.6. Duckiebot Development using Docker . . . 91
5.6.1. Flashing a bootable disk . . . 92
5.6.2. Web interface . . . 92
5.6.3. Testing ROS . . . 93
5.6.4. Build and deployment . . . 93
5.6.5. Multi-architecture support . . . 94
5.6.6. Running a simple HTTP file server . . . 94
5.6.7. Camera testing . . . 95
5.6.8. Graphical User Interface tools . . . 95
5.6.9. Remote control . . . 96
5.6.10. Camera calibration . . . 96
5.6.11. Wheel calibration . . . 96
5.6.12. Lane following . . . 97
5.7. Retrospective . . . 97
5.7.1. Remarks on security . . . 99
5.8. Conclusion . . . 100
Chapter 6. Conclusion . . . 101
References . . . 105
Appendix A. Type-safe differentiable programming . . . 131
A.1. Grammar . . . 131
A.2. Linear regression . . . 132
A.2.1. Finite difference method . . . 133
A.2.2. Partial differentiation . . . 134
A.2.3. Matrix solution . . . 135
A.3. Polynomial regression . . . 137
Appendix B. Tools for reproducible robotics . . . 139

B.1. Useful Docker resources 139
B.1.1. Balena 139
B.1.2. ROS Docker Images 140
B.1.3. Hypriot 140
B.1.4. PiWheels 140
B.1.5. Docker Hub 140
B.1.6. Docker Cloud 142

List of tables

3.1 Kotlin∇'s shape system specifies the output shape for tensor expressions. 42

3.2 Comparison of AD libraries. The ✍symbol indicates work in progress. 54

4.1 Some DFGs generated by Equation 4.8.1 with accompanying 2D plots. 70

List of figures

1.1 Royce's original waterfall model describes the software development process. We use it to guide our discussion and frame our contributions inside of this model. . . 3

2.1 Unique downloads of Hatchery between the time of its release and June 2019. https://plugins.jetbrains.com/plugin/10290-hatchery. 16

2.2 A typical ROS application contains a large graph of dependencies. 17

2.3 Railroad diagram for the grammar shown above (reads from left to right). 22

2.4 ROS Run Configuration. Accessible via: Run » Edit Configurations » + » ROS Launch . . 23

2.5 The evolution of code. On the left are languages that force the user to adapt to the machine. To the right are increasingly flexible representations of source code. 24

2.6 Projectional editors such as MPS [Voelter and Solomatov, 2010, Pech et al., 2013] (shown above) are able to render source code in visually creative ways. This might resemble freehand notation or some other visually appealing format. 24

2.7 Hatchery UI supports syntax highlighting, validation and project navigation. 26

2.8 Detection of local ROS packages. Accessible via: File » Settings » ROS config 26

3.1 *Differentiable programming* includes neural networks, but more broadly, arbitrary programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* [Carpenter et al., 2017, Gorinova et al., 2019] is a generalization of probabilistic graphical models which uses Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function. 33

3.2 Two equivalent programs, both implementing the function $f(l_1, l_2) = l_1 \cdot l_2$. 36

3.3 Adapted from van Merriënboer et al. [2018]. Kotlin∇ models are data structures, constructed by an embedded DSL, eagerly optimized, and lazily evaluated. 38

3.4 Implicit DFG constructed by the original expression, shown above. 40

4.1 We compare numerical drift between AD and SD over a swollen expression using fixed precision and arbitrary precision (AP). AD and SD both exhibit relative errors (i.e. with respect to each other) several orders of magnitude lower than their absolute error. These results are consistent with the findings of Laue [2019]. 60

4.2 Average loss curve of 100 trajectories of momentum SGD in parameter space. . . . 69

4.3 By construction, our differential shrinker detects a greater number of errors per input-output pair than a naïve sampler which does not take the gradient into consideration. 71

4.4 Complexity of detecting various types of programming errors. 72

5.1 Full virtualization is a very resource-hungry process. Containerization is cheaper, as it shares a kernel with the host OS. Emulation lets us emulate hardware as software. Any of these methods can be used in conjunction with any other. 78

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. 78

5.3 Container infrastructure. **Left**: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we build ARM artifacts on x86 using QEMU [Bellard, 2005]. **Right**: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models may be also be run on an ARM device using an accelerator. 91

5.4 Browser interface for individual Duckiebots. It is provided by Portainer, a RESTful web dashboard, which wraps the Docker CLI and offers support for container management, configuration, networking and terminal emulation (shown above). http://DUCKIEBOT_NAME:9000/#/container/container_name "Console" 93

5.5 Early prototype of the Docker image hierarchy. Chaining unversioned autobuilds without disciplined unit testing creates a potential domino effect which allows breaking changes to propagate downstream, resulting in a cascade of silent failures. 98

5.6 The AI Driving Olympics, a primary use case for the system described above. . . . 98

Chapter 1

Introduction

> "There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought."
>
> –Leslie Lamport [2002], *A Discussion with Leslie Lamport*

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.[1] 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 of building software systems.

The objective of 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 virtualization technology.

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] This can be approximated by various metrics like cyclomatic or Halstead complexity.

(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

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 [Bergstra et al., 2010], 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 these tools was crucial for democratizing deep learning techniques.

Intelligent systems are widely deployed in virtual settings like data science and cloud services. But even with the tremendous success of machine learning algorithms in fully-observable domains like image recognition and speech processing, intelligent systems have yet to be widely adopted in robotics (at the time of writing this thesis). This dilemma can be partly attributed to various theoretical problems such as domain adaption and transfer learning. Yet with the proven capabilities of modern learning algorithms, exponential increase in processing power, and decades-long effort in building physically-embodied intelligent agents, we should have more progress to show. Why has this goal evaded researchers for so long? One reason, we conjecture, is a lack of programming tools and abstractions for designing, developing, deploying and evaluating intelligent systems. In practice, these activities consume a large amount of cognitive effort without the right set of tools and abstractions.

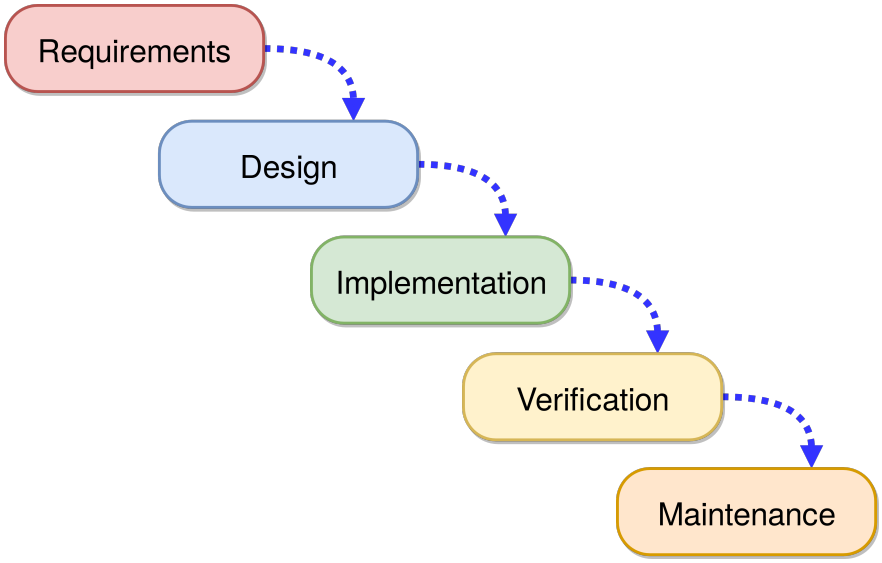


Fig. 1.1. Royce's original waterfall model describes the software development process. We use it to guide our discussion and frame our contributions inside of this model.

In this thesis, we explore several tools that facilitate the process of building 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, we offer some concluding remarks and lessons learned for incorporating these programming tools in (Chapter 6).

1.1. Stages in the software development lifecycle

In traditional software engineering, the Waterfall model (Figure 1.1) is a classical model for software development comprising of various stages [Royce, 1987]. Intended to describe the traditional software development process, a similar framework can be found in most engineering disciplines. We propose contributions to four stages: *design* in Chapter 2, *implementation* in Chapter 3, *verification* in Chapter 4 and *maintenance* in Chapter 5.

1.2. Design: Programming tools for robotics

Today's software systems are deeply complex entities. Gone are the days where a solitary programmer, even a very skilled one, could maintain a large software system alone. To effectively scale modern 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 connections to the outside world. The knowledge graph which accumulates around a software project contains facts, but it also contains workflows for programming, debugging, and delivery – common paths through the labyrinth of software development. Components of this graph can be committed to writing, but documentation is time-consuming and grows stale over time. What is needed is a system that offers 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 the 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 collaborate more effectively 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 freeing mental resources to be used elsewhere. 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, by automating mechanical tasks in software development, these tools free one's attention towards the fundamental activity 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 little surprise IDEs are really graph databases.

In some aspects, the development of intelligent systems is no different than classical software engineering. The same principles and best-practices which guide software engineering are also applicable to intelligent systems. And the same activities, from design to maintenance will continue to play an important role in building intelligent systems. But in other respects, the generic programming tools used to develop traditional software will require domain-specific adaptations for learning systems to become truly first-class citizens in the next generation of intelligent software, particularly in the case of robotics development.

Towards that goal, we developed an IDE for the Robot Operating System (ROS) called Hatchery. It supports a number of common workflows for ROS development, such as creating ROS nodes, Gazebo simulator integration, support for remote debugging, static analysis, autocompletion and refactoring. In Chapter 2 we discuss the implementation of these features and some of the challenges of building language support, programming tools and integrating with the ROS middleware. We argue that such tools reduce the cognitive complexity of building ROS applications by adopting explicit coding conventions, annotating unstructured text and automating repetitive development tasks.

1.3. Implementation: Type-safe differentiable programming

In the early days of machine learning, it was widely believed that human-level intelligence would emerge from 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 to capture human knowledge. 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 artificial intelligence research for several decades. While computer scientists underestimated the difficulty of learning, expert systems excelled in areas where current machine learning systems struggle such as classical reasoning and interpretability, and there is growing evidence to suggest many of these ideas were simply ahead of their time. In our work, we take inspiration from some early work in symbolic reasoning, type systems and functional programming.

What was finally shown to scale, is the idea of connectionist learning. By nesting random function approximators, called perceptrons, and updating the free parameters using back-propagation [Werbos et al., 1990, Rumelhart et al., 1988], the resulting system is capable of learning a surprising amount of intelligent behavior. The approach, termed artificial neural networks (ANNs), 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 training 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 circa 2013 when deep learning was competitive with human experts in specific 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

yet 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 were 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 dating back to the 1970s, called *automatic differentiation* (AD) [Linnainmaa, 1976, 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 [Bergstra et al., 2010] 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].

Engineered intelligent systems must think carefully about languages and abstractions. If developers are required to implement backpropagation by hand, they will have scarce 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 relationship between source code and runtime behavior is already difficult to debug, due to the complexity of neural networks and array programming. One component of that complexity can be found in the type system.

Most existing AD frameworks for machine learning are written in dynamically-typed languages like Python, Lua and JavaScript, with some early implementations including projects like Theano [Bergstra et al., 2010], Torch [Collobert et al., 2002] and Caffe [Jia et al., 2014]. Similar ideas have arisen in statically-typed, functional languages, such as Java (JAutoDiff [Nureki, 2012], DL4J Team [2016a]), Scala (Nexus [Chen, 2017]), F# (DiffSharp [Baydin et al., 2015b]), Swift [Lattner and Wei, 2018], Haskell (TensorSafe [Piñeyro et al., 2019]) et al., but few of these are able to check the shape of multidimensional arrays in their type system, and those which do are typically implemented in experimental languages with sophisticated type-level programming features. In our work, we demonstrate the viability of

shape-checked differentiable programming in a widely-used language. This ensures that programs on matrices, if they compile, are the correct shape and can be numerically evaluated at runtime.

Kotlin∇ is an embedded domain-specific language (eDSL) for differentiable programming in a language called Kotlin, a statically-typed programming language with support for asynchronous programming and multi-platform compilation. In Kotlin∇ (Chapter 3), we describe an algebraically-grounded implementation of forward-mode automatic differentiation with shape-safe tensor operations. Our approach differs from most existing AD frameworks in that Kotlin∇ is the first shape-safe AD library fully compatible with the Java type system, requiring no metaprogramming, reflection or compiler intervention to use.

1.4. Verification: Testing intelligent systems

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 with problems which are simple to pose, but seemingly 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 forty orders of magnitude [Shannon, 1988]. At the time, it was believed that such problems would be insurmountable without fundamental breakthroughs in algorithms and 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 discovered in the Cambrian explosion 541 million years ago, were implemented to a competitive margin in modern computers [Pratt, 2015].

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 dispersed. Because the dimensionality of many real-world functions that we would like to approximate is intractably large, it is difficult to verify the behavior of a candidate solution in all regimes, especially in settings where failure is rare but catastrophic. According to some studies, humans drivers average 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 operator 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 speaking, 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 are 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 interest.

More broadly, type checking (as explored in Chapter 3) and testing can be seen as part of a wide spectrum of techniques for software verification and validation. The sooner developers can detect software anomalies, the easier they are to fix and the safer autonomous systems can become. Previous techniques in the software testing literature have called for domain experts to handcraft input-output examples or otherwise constrain the search space, but with advances in adversarial machine learning, automated software validation becomes an increasingly attractive proposition.

Towards that end, in Chapter 4 we present preliminary work in adversarial fuzz testing, building on prior literature in adversarial learning, metamorphic and property-based testing. A similar technique is used for testing the numerical correctness of Kotlin∇'s implementation. We present a simple algorithm for property-based shrinking using projected gradient descent and suggest several future directions for improvement.

1.5. Maintenance: Tools for reproducible robotics

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 follow 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 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,[2] a vanishingly small fraction of the information corresponds to the source code 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 specialized accelerators to train and run these programs rapidly. But unlike most programming, deep learning is a much

[2] https://help.github.com/en/articles/what-is-my-disk-quota

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, selling competing chips with incompatible standards (e.g. MLIR [Lattner and Pienaar, 2019], ONNX [Bai et al., 2019], nGraph [Cyphers et al., 2018], Glow [Rotem et al., 2018], TVM [Chen et al., 2018] et al.) While some have tried to leverage existing compilers such as GHC [Elliott, 2018] or DLVM/LLVM [Wei et al., 2017], there are few signs of broader interoperability at the time of writing this thesis.

At the end of the day, researchers need to reproduce the work of other researchers, but the mental effort of re-implementing basic abstractions can impede scientific progress. Tools which facilitate software reproducibility and incremental development are essential. Fortunately, this is the same problem which has concerned the software industry for many years and produced a variety of version control systems (VCS). But VCS alone is insufficient, since these tools are primarily intended to store text. Text-based representations are temporarily stable, but when dependencies are updated and rebuilt, important details about the original development environment can be misplaced. To reproduce a program in its entirety, a snapshot of all digital information available during execution, and ideally, the physical computer itself is needed. Short of a full snapshot, the minimal set of dependencies and a near physical replica is highly desirable. Any variability in the physical or digital dependency graph can be a source of discrepancies which requires time and energy to later isolate.

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-related set of tools for build automation and developer operations which we shall refer to as *container infrastructure*. Docker allows developers to freeze a software application and its host environment, allowing developers (e.g. using a different environment) to quickly reproduce these applications. Docker itself is a technical solution, but also encompasses a set of best-practices and guidelines which are more methodological in nature. While Docker

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, which incorporates the notion of time: simulators. Simulators are used in nearly every engineering discipline to imitate a physical process which may be expensive, dangerous or impractical to bring into reality. For example, simulators are often used to model the dynamics of another instruction set architecture [Bellard, 2005], the dynamics of electromagnetic transients [Tavante et al., 2018], the dynamics of orbital motion [Bellman et al., 1965], the dynamics of human transportation systems [Ruch et al., 2018], or the dynamics of driving [Chevalier-Boisvert et al., 2018]. Today's computers are capable of running increasingly high fidelity simulations, but most practitioners agree that simulation alone will never be enough to capture the full distribution of real-world data. In this view, simulation can be a useful tool for detecting errors, but it cannot fully reproduce all the subtleties of the real-world, and should not be a surrogate for testing on real-world data. Others have suggested a middle road [Bousmalis et al., 2018], where judicious use of simulator training, alongside domain adaptation is a sufficiently rigorous environment for evaluating intelligent systems. Regardless of which view prevails, our goal is to provide rapid feedback to developers, and to make the entire process from testing to deployment as reproducible as possible.

1.5.1. Iconography

Throughout this thesis, the following iconography is used to denote:

 Shell commands intended for a personal computer, or output derived thereof.

 GrammarKit's `.bnf` parsing expression grammar (PEG) [3]

 Either `Dockerfile` [4] or Docker Compose [5] syntax.

[3] GrammarKit usage notes: https://github.com/JetBrains/Grammar-Kit/blob/master/HOWTO.md

[4] Dockerfile reference: https://docs.docker.com/engine/reference/builder/

[5] Compose file reference: https://docs.docker.com/compose/compose-file/

 Shell commands which should be run on a Raspberry Pi.[6]

 Duckietown Shell (`dts`) commands.[7]

 roslaunch `.launch` files.[8]

 Python source code.[9]

 Kotlin source code.[10]

[6]Raspberry Pi: https://www.raspberrypi.org/

[7]Duckietown Shell: https://github.com/duckietown/duckietown-shell-commands

[8]ROS Launch XML: https://wiki.ros.org/roslaunch/XML

[9]Python documentation: https://www.python.org/doc/

[10]Kotlin documentation: https://kotlinlang.org/docs/reference/

Chapter 2

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."
>
> –Joseph Licklider [1992], *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 driven 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 and robotics will be tremendously fruitful when fully realized.

Hatchery is a tool designed to assist programmers writing robotics applications using the ROS middleware. At the time of its release, Hatchery was the first ROS plugin for the IntelliJ Platform [1], and today, is the most widely used with over 10,000 unique downloads. While the idea is simple, its prior absence and subsequent adoption suggest there is unmet demand for such tools in the development of intelligent software systems, particularly in domain-specific applications like robotics.

[1] An IDE platform for C/C++, Python and Android development, among other languages.

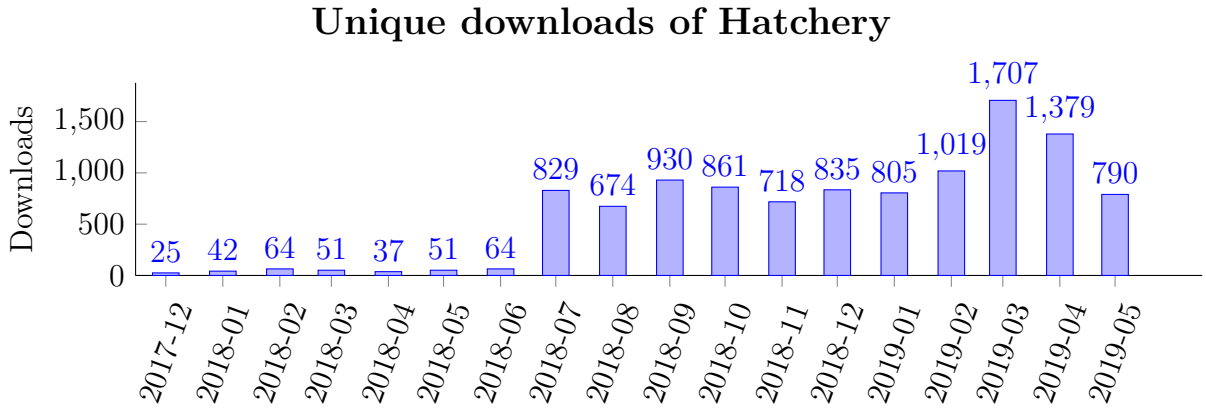


Fig. 2.1. Unique downloads of Hatchery between the time of its release and June 2019. https://plugins.jetbrains.com/plugin/10290-hatchery.

2.1. Introduction to the Robot Operating System

The Robot Operating System (ROS) [Quigley et al., 2009] 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 support similar functionality such as shared memory and inter-process communication. Unlike pure message-oriented systems such as DDS [Pardo-Castellote, 2003] and ZMQ [Hintjens, 2013], 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 deserializing geometric data, coordinate frames, maps, sensor messages, and imagery.

The ROS middleware provides several language front-ends for polyglot programming. According to one community census taken in 2018, 55% of all ROS applications on GitHub are written in C/C++, followed by Python with a 25% [Guenther, 2018] developer share. Source code for a typical ROS application contains a mixture of C/C++ and Python code, corresponding to the respective language preferences in the robotics and machine learning communities. Hatchery is compatible with most common ROS client libraries, including rosjava for Java, rospy for Python, roscpp for C/C++, and other language front ends.

A typical ROS project has several components, including the source code, configuration files, build infrastructure, compiled artifacts and the deployment environment. To build a

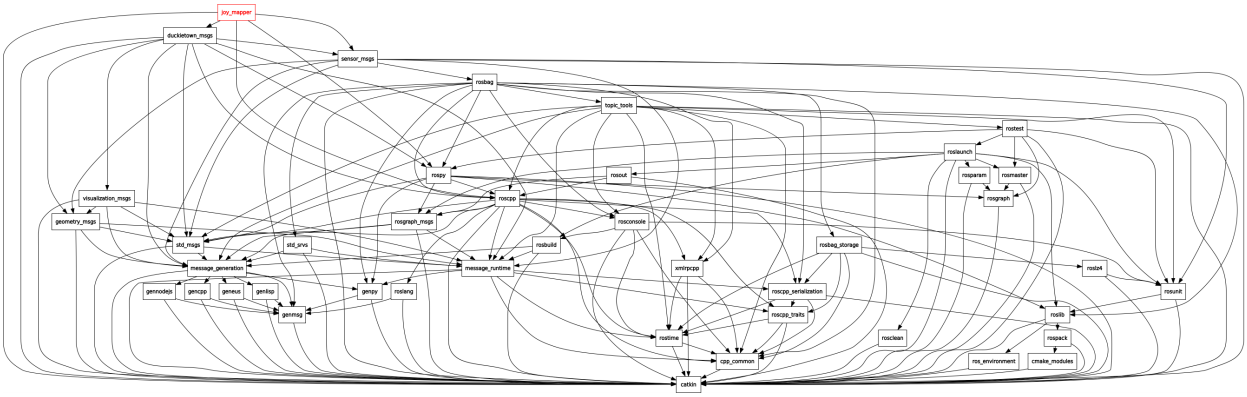


Fig. 2.2. A typical ROS application contains a large graph of dependencies.

simple ROS application, several steps are necessary. First, one must install the ROS system, which is only officially supported on Debian-based Linux distributions.[2] Assuming ROS has been installed to the default location, it can be sourced like so:

```
~$ source /opt/ros/<ROS DISTRO>/setup.[ba]sh
```

A minimal ROS application contains at least one *publisher* and *subscriber*, which pass messages over a shared communication channel. The publisher might be defined as follows:

./catkin_ws/src/pubsub/publisher.py

```
import rospy
from std_msgs.msg import String

pub = rospy.Publisher("channel", String, queue_size=10)
rospy.init_node("publisher", anonymous=True)
rate = rospy.Rate(10)
while not rospy.is_shutdown():
pub.publish("Some message")
rate.sleep()
```

As the publisher writes messages to `channel`, another node which is subscribed to the same channel will receive a callback when new messages arrive and can read them off the channel:

[2]Detailed installation instructions may be found here: https://wiki.ros.org/ROS/Installation

./catkin_ws/src/pubsub/subscriber.py

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "received data %s", data.data)

    rospy.init_node("subscriber", anonymous=True)
    rospy.Subscriber("channel", String, callback)
    rospy.spin()
```

All ROS packages have launch file, which contain a manifest of available nodes:

./catkin_ws/src/pubsub/pubsub.launch

```
<launch>
<node name="publisher" pkg="pubsub" type="publisher.py" output="screen"/>
<node name="subscriber" pkg="pubsub" type="subscriber.py" output="screen"/>
</launch>
```

To build and run the application, the following series of commands are required:

```
~$ cd catkin_ws && catkin_make
```

```
~$ roslaunch pubsub pubsub.launch
```

Rather than interacting with the command line, it would be convenient to have a graphical tool to perform all of these tasks automatically. Additionally, it would be helpful to detect if there were a typographical error or navigable reference in the launch file:

./catkin_ws/src/pubsub/pubsub.launch

```
<launch>
<node name="publisher" pkg="pubsub" type="pubsher.py" output="screen"/>
<node name="subscriber" pkg="pubsub" type="subscriber.py" output="screen"/>
</launch>
```

Notice how the typographical error is printed in red and the valid file reference is underlined in blue, indicating it can be selected to open the file shown above. Broadly, these are the kinds of features IDEs provide and are examples of specific functionality in Hatchery.

2.2. Installation

To simply run the tool, users should have the following software dependencies:

(1) MacOS or Debian-based Linux distribution

(2) Robot Operating System (Electric Emys or later)

(3) Java SE (JRE 8+) or CLion/PyCharm 2019.1+

ROS users can use the following command to open an existing ROS project:

```
~$ git clone https://github.com/duckietown/hatchery && cd hatchery && \
   ./gradlew runIde [-Project="<ABSOLUTE_PATH_TO_ROS_PROJECT>"]
```

Duckietown users can simply use `dts`, the Duckietown Shell:

```
dt> hatchery
```

Hatchery can also be installed directly from inside the CLion or PyCharm IDEs, via the following menu options: File » Settings » Plugins » Marketplace » Q"Hatchery"

2.3. Plugin development

To build an IDE, some tools are helpful. First, is an IDE, and its source code. Assume that IDE_0 exists. In order to build a new IDE, IDE_1, we can load the source code from IDE_0 into IDE_0 and use IDE_0, to modify, compile and re-run the code, which becomes IDE_1, in which the process is repeated. However, this approach has some disadvantages. First, most IDEs are already quite cumbersome to compile and run. As most auxiliary features are small by comparison, modern IDEs have adopted a modular design, which allows them to load specific packages (i.e. *plugins*) as needed. So most developers can skip the first step, and load their plugin inside IDE_0 directly. It is still convenient to have the platform source code for reference purposes, but in most cases this code is read-only.

Hatchery uses the IntelliJ Platform, an IDE platform which supports most common programming languages. By targeting a platform with support for polyglot programming, we are able to focus on language-agnostic features in the ROS ecosystem, such as parsing and editing ROS-specific configuration files, build and run configuration and other common development tasks.

2.3.1. Refactoring

Refactoring is an essential feature in any IDE, and the essence of refactoring is renaming. Consider what must occur when a user wishes to rename a token in her program, such as the parameter named `data` on line #1 below:

```
def callback(data):                                                    1
    rospy.loginfo(rospy.get_caller_id() + "received data: %s", data.data)   2
```

If she were using the `vim` text editor, one solution would be to replace all textual occurrences of the string `data` within the file using `:%s/data/msg/g`, producing the following result:

```
def callback(msg):                                                     1
    rospy.loginfo(rospy.get_caller_id() + "received msg: %s", msg.msg)  2
```

There were four occurrences of the string `data`, only two of which were correctly renamed. Instead, only those strings which refer to the function parameter should be renamed:

```
def callback(data):                                                    1
    rospy.loginfo(rospy.get_caller_id() + "received data: %s", data.data)   2
```

Generally, we would like the ability to rename identifiers across files and languages. To do so, we need a richer understanding of code that transcends text – we need a parser.

2.3.2. Parsing

One of the most important and unappreciated components of an IDE is the parser. Unlike compilers, most IDEs do not use recursive descent or shift-reduce parsing as treated in most compiler textbooks [Appel and Palsberg, 2003], as these algorithms are not well-suited for real-time editing of source code. Edits are typically short, localized changes inside a large file, and are frequently invalid or incomplete between keystrokes. As most IDEs are expected to recover from local errors and provide responsive feedback while editing source code, re-parsing the entire program between minor edits would be expensive and unnecessary. In order to analyze source code undergoing simultaneous modification and provide interactive feedback, special consideration must be taken to ensure robust and responsive parsing.

Various techniques have been developed to improve the responsiveness of modern parsers. Incremental parsing techniques like those first proposed in Ghezzi and Mandrioli [1979] and further developed by Wagner [1998], Wagner and Graham [1997] seek to incorporate caching and differential parsing to accelerate the analysis of programs under simultaneous modification. Fuzzy parsing techniques like those described in Koppler [1997] aim to increase the flexibility and robustness of parsing in the presence of local errors. Both of these techniques have played a role in the development of language-aware programming tools, which must be able to provide rapid and specific feedback whilst the user is typing.

The procedural instructions for modern parsers are seldom written by hand unless the language being parsed is very simple or raw performance is desired. Even parsers designed for IDEs, where incremental parsing and error-tolerance is so important, metacompilation toolkits such as ANTLR [Parr and Quong, 1995], or Xtext [Eysholdt and Behrens, 2010] cover a surprising number of common use-cases. Hatchery uses Grammar-Kit, a toolkit designed to assist users developing custom language plugins for the IntelliJ Platform. It uses a DFA-based lexer generator, JFlex [Klein et al., 2001], and a custom parser-generator loosely based on the parsing expression grammar (PEG) [Ford, 2004], a descendant of the Backus-Naur Form (BNF) grammar specification. This specification is consumed by the GrammarKit parser generator and translated to Java source code, producing a parser which reads source code written in the specified language and constructs a program structure interface (PSI), the IntelliJ Platform's internal data structure for representing abstract syntax trees (ASTs). Here is an excerpt of a PEG BNF grammar for parsing ROS `.msg` files:

BNF

```
rosInterfaceFile ::= ( property | COMMENT )*
property ::= ( TYPE FIELD SEPARATOR CONSTANT ) | ( TYPE FIELD ) {
    pin=3 // Identifies an unambiguous delimiter or fallback point
    recoverWhile="recover_property" // Error recovery predicate
    mixin="edu.umontreal.hatchery.psi.impl.RosMsgNamedElementImpl"
    implements="edu.umontreal.hatchery.psi.RosMsgNamedElement"
    methods=[getType getKey getValue getName setName getNameIdentifier]
}
private recover_property ::= ! ( TYPE | FIELD | SEPARATOR | COMMENT )
```

The lexical rules for the tokens, `TYPE`, `FIELD`, `CONSTANT` et al. are defined in a separate `.flex`

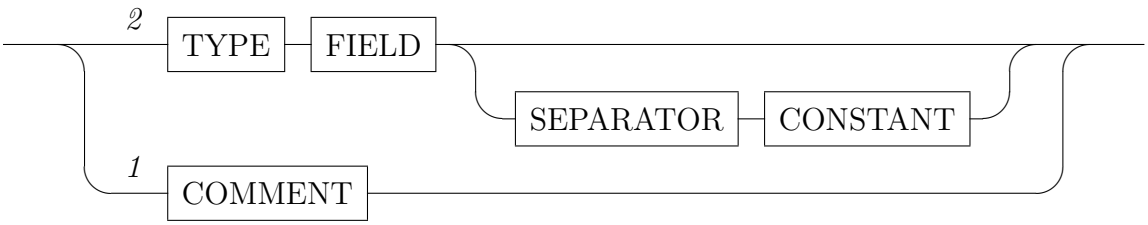


Fig. 2.3. Railroad diagram for the grammar shown above (reads from left to right).

file, the JFlex grammar. Below is an excerpt from the accompanying `.flex` lexer:

```
TYPE_CHARACTER=[^:=#\ \r\n\t\f\\]
FIELD_CHARACTER=[^:=#\ \r\n\t\f\\]
SEPARATOR_CHARACTER=[:=]
CONSTANT_CHARACTER=[^\r\n\f#]
COMMENT_CHARACTER=#[^\r\n\f]*
```

Grammar-Kit consumes these files and generates Java source code for parsing ROS `.msg` files. Generated sources can be manually refined to provide support for more advanced functionality such as more flexible error-recovery. For regular languages like the interface description languages (IDL) found in ROS `.msg` and `.srv` files, the default generated parser and lexer are usually sufficient. Hatchery is also capable of parsing URDF, package manifest and roslaunch XML.

2.3.3. Running and debugging

The process of compiling and running ROS applications often requires several steps, ex.:

```
~$ . /opt/ros/<DISTRO>/setup.[ba]sh &&
   cd <PROJECT>/catkin_ws &&
   catkin_make &&
   . devel/setup.sh &&
   [export ROS_MASTER_URI=<URI> &&]
   roslaunch [OPTIONS] src/.../<LAUNCH FILE> [ARGUMENTS]"
```

Hatchery provides assistance for configuring, building and running ROS applications inside a custom graphical user interface (GUI). This GUI effectively serves as a wrapper for the ROS command line interface (CLI). Visual elements like configuration options and command line

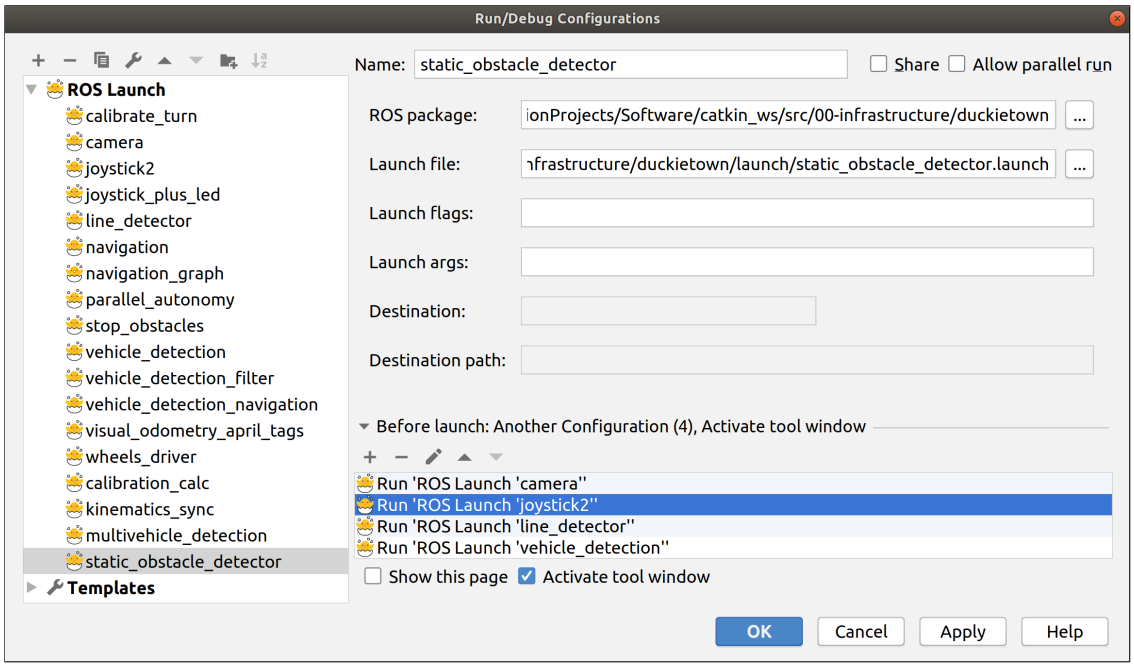


Fig. 2.4. ROS Run Configuration. Accessible via: Run > Edit Configurations > + > ROS Launch

flags are written to an internal model called the "Run Configuration" (Figure 2.4). When a run configuration is manually triggered, Hatchery's internal model is serialized to a `String`, representing the command to be executed. This `String` is then sent to a terminal emulator, which invokes the command and displays the corresponding output.

2.3.4. User interface

An often overlooked, but important aspect of development tools is the graphical user interface, as the primary interface for editing source code. In the early days of modern computing, the only way of getting information in or out of a computer involved punching holes in paper. Later, computers were equipped with technology to emit the same binary pattern as pixels, which could be used to display a small alphabet called ASCII. With higher density and frequency displays, computers could render more sophisticated shapes and animations. These improvements are the direct result of graphical innovation, but can also be seen as progress in program representation, where the symbolic medium was itself just a notational convention which developers and machines used to communicate.

ASCII is still the dominant medium for modern programming, although machines still use various forms of low-level assembly code for execution. A great deal of software infrastructure

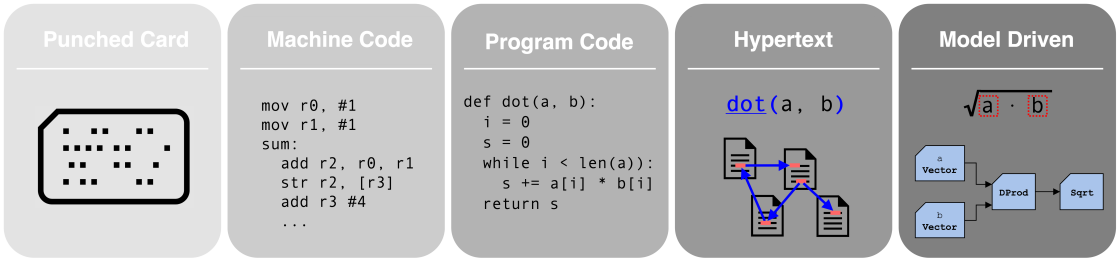


Fig. 2.5. The evolution of code. On the left are languages that force the user to adapt to the machine. To the right are increasingly flexible representations of source code.

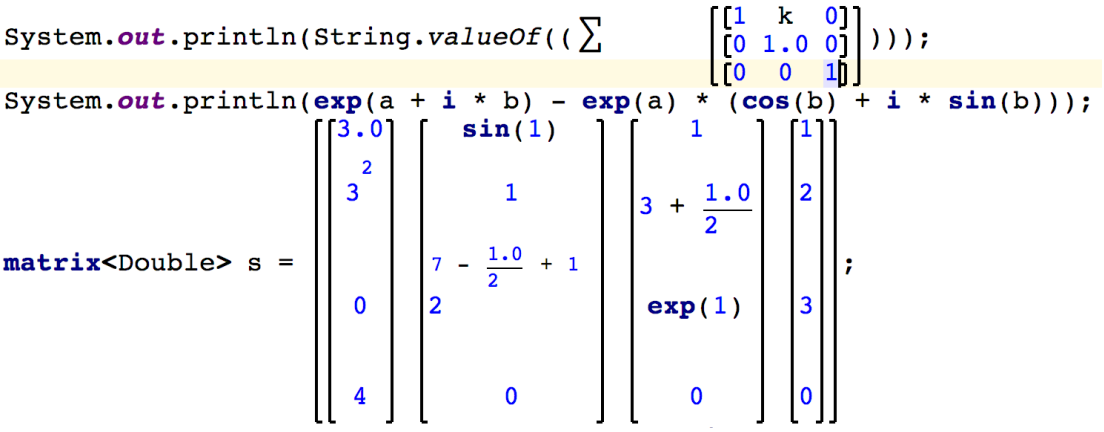


Fig. 2.6. Projectional editors such as MPS [Voelter and Solomatov, 2010, Pech et al., 2013] (shown above) are able to render source code in visually creative ways. This might resemble freehand notation or some other visually appealing format.

is dedicated to translating between such representations via programming languages and compilers. While many software frameworks provide a minimal command line interface (CLI) and some even provide sophisticated programming environments, these tools are fairly restrictive. In the same way that early computer scientists probably did not invent new algorithms by imagining patterns of holes in paper, ASCII is also an indirect medium for expressing ideas, albeit one slightly less contrived. As hardware and software technology progressed, programming languages moved "up the stack", allowing their users to express ideas in a notation which was more familiar and easy to reason about its execution.

With the development of modern languages came programming tools capable of representing code as a mixture of hypertext and graphical user interfaces. Such tools provide a richer representation for code than plaintext and help to capture programs' graph-based

structure, but still use ASCII with sparse visual cues to render code. Some tools support larger character sets and font-based typographic ligatures, although the visual representation of source code remains mostly linear and textual.

More experimental UIs, as proposed in the language oriented programming [Dmitriev, 2004] and model-driven engineering [Famelis et al., 2015] literature, suggest the possibility of more visually flexible layouts. This uncoupling between the composition and representation of text raises many intriguing questions. With the proliferation of new abstractions and programming shorthands, what is the appropriate level of notation required for a given programming task? And who is the intended audience? These are important questions to consider when designing a new programming tool.

The Hatchery plugin provides a lightweight GUI overlaying the program's source code. This interface (Figure 2.7) primarily consists of simple visual cues such as text highlighting, navigation assistance and other menus and configuration panels for performing various programming tasks. The host IDE offers a visual language consisting of iconography and repetitive visual motifs, which serve as cognitive landmarks to guide the developer's procedural memory. The IntelliJ Platform offers a palette of common design elements, which users who are familiar with the IDE can recognize at a glance. Plugins can use these same patterns to access procedural memories implanted in the userbase, facilitating transfer learning. To do this properly requires familiarity with the design language and careful integration with the target framework (i.e. ROS).

Hatchery provides a settings menu for configuring and managing ROS installations which can automatically detect local ROS distributions and also allows users to manually configure the ROS environment, as shown in Figure 2.8.

2.4. Ongoing work

While it supports many common use cases such as rudimentary code navigation, static analysis and run assistance, Hatchery is currently a work in progress. We are working to expand Hatchery's support for ROS programming in some of the following areas:

- **Syntax support** – Highlighting, navigation, autocompletion
- **Program analysis** – Code inspections, intentions, and linting
- **Testing support** – Unit and integration testing, code coverage

- **Project creation** – Project setup and boilerplate code generation
- **Dependency management** – Track installed and missing packages

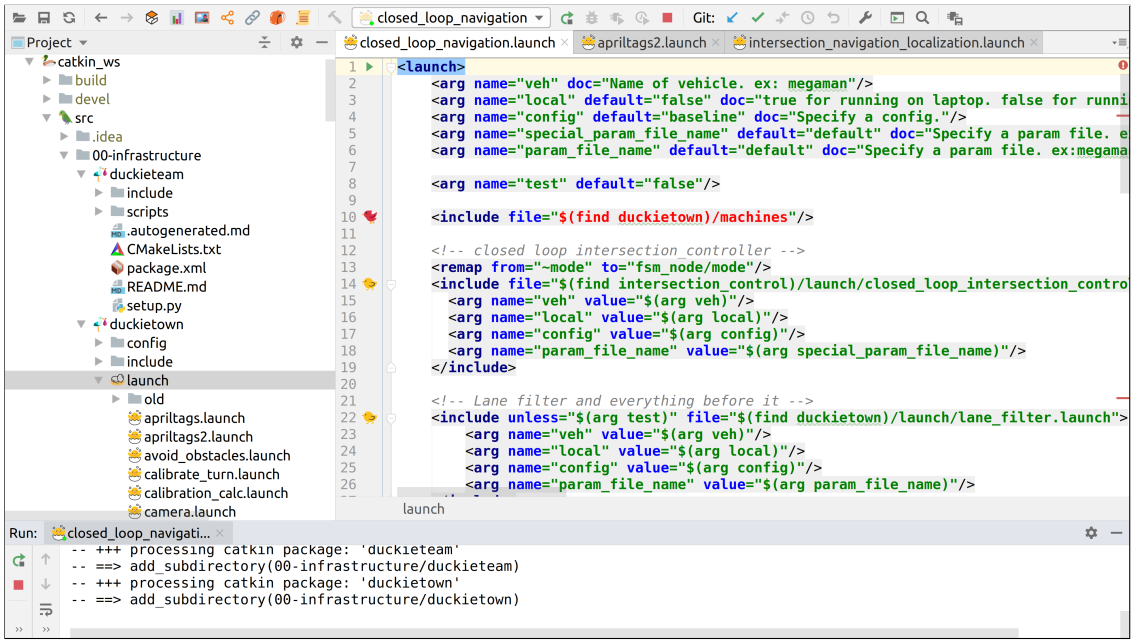


Fig. 2.7. Hatchery UI supports syntax highlighting, validation and project navigation.

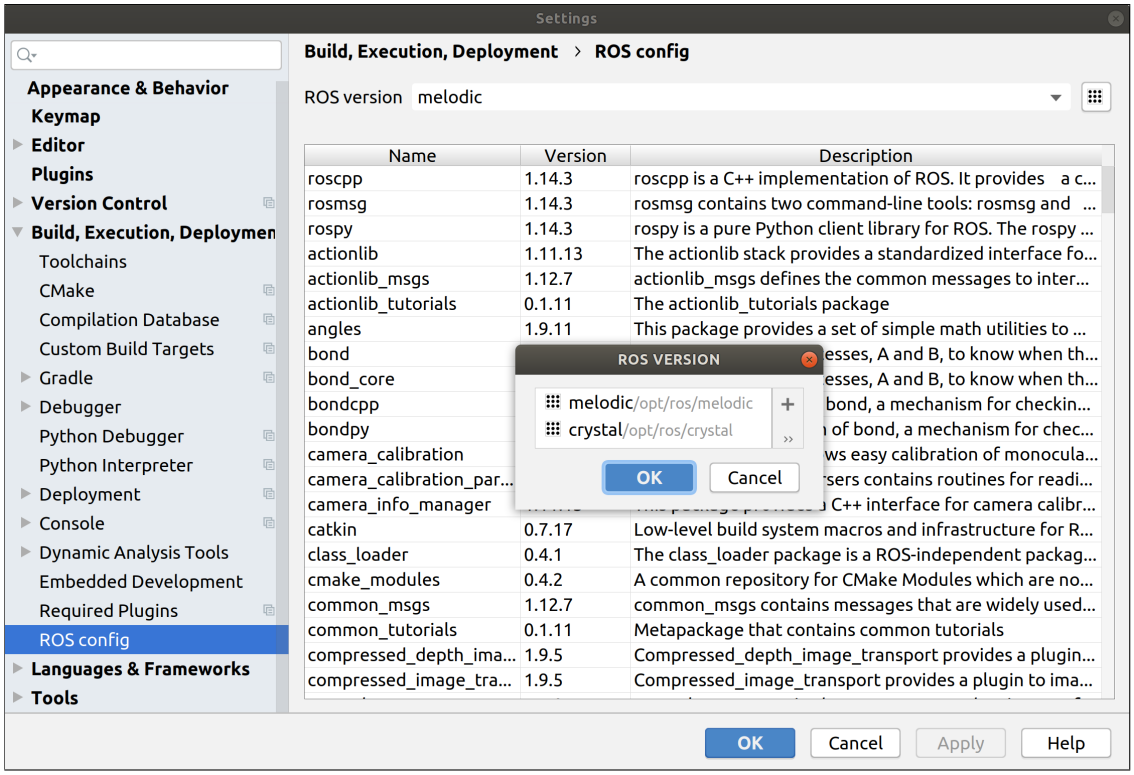


Fig. 2.8. Detection of local ROS packages. Accessible via: File › Settings › ROS config

- **Monitoring utils** – Logging, diagnostics, profiling and visualization
- **Crash analytics** – Enhanced stack traces with source navigation
- **Build automation** – Delta rebuilds, cmake magic, code hotswap
- **ROS integration** – Nodes, topics, services, parameters, graphs
- **Duckumentation** – Usage instructions and supported features

A more comprehensive list of currently supported and upcoming features are detailed below:

- ☑ ROS Launch (`*.launch`, `*.test`)
  - ☑ Syntax highlighting
  - ☑ Resource references (`$(find <directory>)...`)
- ☑ Package manifest (`package.xml`)
  - ☑ Syntax highlighting
  - ☑ Package dependencies (`<build_depend>`, `<test_depend>`, `<run_depend>`)
- ☑ ROS URDF (`*.urdf.xacro`)
  - ☑ Syntax highlighting
  - ☑ Resource references (`$(find <directory>)...`)
- ☑ ROS Bag (`*.bag`)
  - ☑ Syntax highlighting
- ☑ ROS Message (`*.msg`)
- ☑ ROS Service (`*.srv`)
- ☑ Implement preliminary project structure and XML support
- ☑ Write an MVP/POC app that supports file renaming and refactoring
- ☑ Add support for project templates and skeleton project creation
- ☑ Add support for deploying a project from the local machine to the remote
- ☐ Add support for monitoring and tracking running code, viewing logs
  - ☐ Live logfile tracking
  - ☐ Save to local disk
  - ☐ Searching the log
- ☐ Collect crash dumps and link to the corresponding code points
  - ☐ Link stack traces to source code
  - ☐ Copy environment info and crash dump to clipboard
- ☐ Integration with the Robot Operating System (ROS)
  - ☑ ROS 1 support (Kinetic Kame recommended)
  - ☐ ROS 2 support
  - ☑ Managing ROS installations.
- ☑ Gazebo simulator integration
- ☐ CMake build integration
- ☐ Remote debugging support
- ☐ Docker integration
  - ☑ Basic Docker support
  - ☐ Remote host and script support
  - ☐ Docker Hub namespace awareness
  - ☐ Support for platformio tooling

- ☐ X11 forwarding and rqt support
- ☐ Static analysis for Python API misuse
  - ☑ Invalid dependency detection
  - ☐ Validate `.msg`/`.srv` compatibility
  - ☐ ROS nodes and graph analysis via `rosdep`/`rqt_dep`
- ☐ rqt plugin support
  - ☑ `rqt_img_view` - View images
  - ☑ `rqt_plot` - Plot data visually
  - ☑ `rqt_graph` - Graph messages
  - ☑ `rqt_dep` - Visualize dependencies
  - ☑ `rqt_bag` - Replay and edit bag files
  - ☐ rqt_common - Other common plugins

2.5. Conclusion

In this chapter we demonstrate the value of IDEs for general purpose software development and present a domain-specific IDE plugin for robotics development, originally developed as a final project in the Duckietown class [Paull et al., 2017]. By using Hatchery, developers can receive assistance when writing, compiling and running ROS applications. The author wishes to express his gratitude to Paolo Achdjian for contributing several features.

Chapter 3

Type-safe differentiable programming

> "Although mathematical notation undoubtedly possesses parsing rules, they are rather loose, sometimes contradictory, and seldom clearly stated... Because of their application to a broad range of topics, their strict grammar, and their strict interpretation, programming languages can provide new insights into mathematical notation."
>
> –Kenneth Iverson [1999], *Math for the Layman*

In this chapter, we will discuss the theory and implementation of a type-safe domain-specific language for automatic differentiation (AD), an algorithm with a variety of applications in numerical optimization and machine learning. The key idea behind AD is fairly simple. A small set of primitive mathematical 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, we are often given a computable function in the form of a program which does not work properly. We would like an algorithm for determining how to change the input slightly, 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, recognizing 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 [1976] where the idea of calculating derivatives over computation graphs was first recorded. Linnaimaa's algorithm was particularly important for neural networks, and is today known as reverse-mode AD. But it was not until 2010 when standard software tools [Bergstra et al., 2010] 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 $P_k : \mathbb{R} \to \mathbb{R}$, composed of a series of nested functions, each with the same type:

$$P_k(x) = \begin{cases} p_1 \circ x = x & \text{if } k = 1 \\ p_k \circ P_{k-1} \circ x & \text{if } k > 1 \end{cases} \tag{3.1.1}$$

From the chain rule, we recall the derivative of a composition is a product of the derivatives:

$$\frac{dP}{dp_1} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \cdots \frac{dp_2}{dp_1} = \prod_{i=1}^{k-1} \frac{dp_{i+1}}{dp_i} \tag{3.1.2}$$

Given $Q(q_1, \ldots, q_m) : \mathbb{R}^m \to \mathbb{R}$, the *gradient* is a function $\nabla Q : \mathbb{R}^m \to \mathbb{R} \to \mathbb{R}^m$ defined as:

$$\nabla Q = \left[\frac{\partial Q}{\partial q_1}, \ldots, \frac{\partial Q}{\partial q_m} \right] \tag{3.1.3}$$

The *Hessian* is a function $\mathbf{H} : \mathbb{R}^m \to \mathbb{R} \to \mathbb{R}^{m \times m}$ returning a matrix of second-order partials:

$$\mathbf{H}(Q) = \begin{bmatrix} \dfrac{\partial^2 Q}{\partial x_1^2} & \dfrac{\partial^2 Q}{\partial x_1 \, \partial x_2} & \cdots & \dfrac{\partial^2 Q}{\partial x_1 \, \partial x_m} \\ \dfrac{\partial^2 Q}{\partial x_2 \, \partial x_1} & \dfrac{\partial^2 Q}{\partial x_2^2} & \cdots & \dfrac{\partial^2 Q}{\partial x_2 \, \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 Q}{\partial x_m \, \partial x_1} & \dfrac{\partial^2 Q}{\partial x_m \, \partial x_2} & \cdots & \dfrac{\partial^2 Q}{\partial x_m^2} \end{bmatrix} \tag{3.1.4}$$

For vector functions $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^n$, the *Jacobian*, $\mathcal{J}_\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^n \to \mathbb{R}^{n \times m}$ is defined as:

$$\mathcal{J}_\mathbf{f} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \cdots & \dfrac{\partial f_n}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix} \tag{3.1.5}$$

For scalar functions, the transpose of the Hessian is equivalent to the Jacobian of the gradient:

$$\mathbf{H}(Q)^\intercal = \mathcal{J}_\mathbf{q}(\nabla Q) \tag{3.1.6}$$

For a vector function $\mathbf{P}_k(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^n$, the chain rule from Equation 3.1.2 still applies:

$$\mathcal{J}_{\mathbf{P_k}} = \prod_{i=1}^{k} \mathcal{J}_{p_i} = \underbrace{\Big(\big((\mathcal{J}_{p_k} \mathcal{J}_{p_{k-1}}) \ldots \mathcal{J}_{p_2} \big) \mathcal{J}_{p_1} \Big)}_{\textit{“Reverse accumulation”}} = \underbrace{\Big(\mathcal{J}_{p_k} \big(\mathcal{J}_{p_{k-1}} \ldots (\mathcal{J}_{p_2} \mathcal{J}_{p_1}) \big) \Big)}_{\textit{“Forward accumulation”}} \tag{3.1.7}$$

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

$$\mathbf{H}(\mathbf{f}) = [\mathbf{H}(f_1), \mathbf{H}(f_2), \ldots, \mathbf{H}(f_n)] \tag{3.1.8}$$

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 how a is changed, the output remains the same. We say that such functions have zero gradient for that input.

$$(\nabla F)(a) \approx \mathbf{0} \tag{3.1.9}$$

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.

$$||\nabla F|| \approx \pm\infty \tag{3.1.10}$$

It is an open question whether non-differentiable functions exist in the real-world [Buniy 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 on $\mathbb{R}^m$ 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 $\varphi : \mathbb{R} \to \mathbb{R}$ composed with a linear function $\theta^\intercal x + b$, 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 $\mathbf{f} : C(\mathbb{I}_m)$, where $\mathbb{I}_m = [0,1]^m \to [0,1]$, there exists a function $\widehat{\mathbf{f}} : \mathbb{R}^m \times \mathbb{R}^{n \times m}$, parameterized by $\Theta \in \mathbb{R}^{n \times m}$, taking an input $\mathbf{x} \in [0,1]^m$ and constants $n \in \mathbb{N}, \beta \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$ such that following statement holds:

$$\begin{aligned} \widehat{\mathbf{f}}(\mathbf{x}; \Theta) &= \beta^\intercal \varphi_\odot \left(\Theta^\intercal \mathbf{x} + \mathbf{b} \right) \\ \forall \mathbf{x} \in \mathbb{I}_m, \ & \left| \widehat{\mathbf{f}}(\mathbf{x}) - \mathbf{f}(\mathbf{x}) \right| < \epsilon \end{aligned} \tag{3.1.11}$$

Where $\varphi_\odot$ indicates the nonlinear function φ applied elementwise to a vector. This theorem only tells us that Θ exists, but does not tell us how to find it nor does it 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 number of steps by composing several *layers* of $\Theta^\intercal \mathbf{x} + \mathbf{b}$ and φ in an alternating fashion, and updating each Θ using a procedure based on gradient descent. The resulting model might be expressed as follows[1],

$$\widehat{\mathbf{P}}_k(\mathbf{x}; \boldsymbol{\Theta}) = \begin{cases} \widehat{\mathbf{p}}_1(\Theta_1) \circ \mathbf{x} & \text{if } k = 1 \\ \widehat{\mathbf{p}}_k(\Theta_k) \circ \widehat{\mathbf{P}}_{k-1}(\boldsymbol{\Theta}_{[1,k-1]}) \circ \mathbf{x} & \text{if } k > 1 \end{cases} \tag{3.1.12}$$

where $\boldsymbol{\Theta} = \{\Theta_1, \ldots, \Theta_k\}$ are free parameters and $\mathbf{x} \in \mathbb{R}^m$ is a single input. To approximate $\mathbf{P}(\mathbf{x})$, one must obtain $\mathbf{X} = \{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(z)}\}, \mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ in as great and varied a quantity as possible and repeat the following procedure until $\boldsymbol{\Theta}$ converges:

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{1}{z} \nabla_{\boldsymbol{\Theta}} \sum_{i=1}^{z} \mathcal{L}\left(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)} \right) \tag{3.1.13}$$

In the general case, we can solve for the gradient using Equation 3.1.7. For most common $\mathcal{L}$, the complexity of this procedure is linear with z. As z can be quite large in practice, and since obtaining the exact gradient is not important, we use a stochastic variant by resampling a *minibatch* $\mathbf{X}', \mathbf{Y}'$ consisting of pairs $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ for $i \sim \{0, \ldots, z\}$ without replacement on each update step. This 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

[1] The notation below assumes some familiarity with currying and partial function application, in which $\widehat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}^n \equiv \underbrace{\mathbb{R} \to \ldots \to \mathbb{R}}_{m} \to \mathbb{R}^n$. For further details, see Schönfinkel [1924], Curry and Feys [1958] et al.

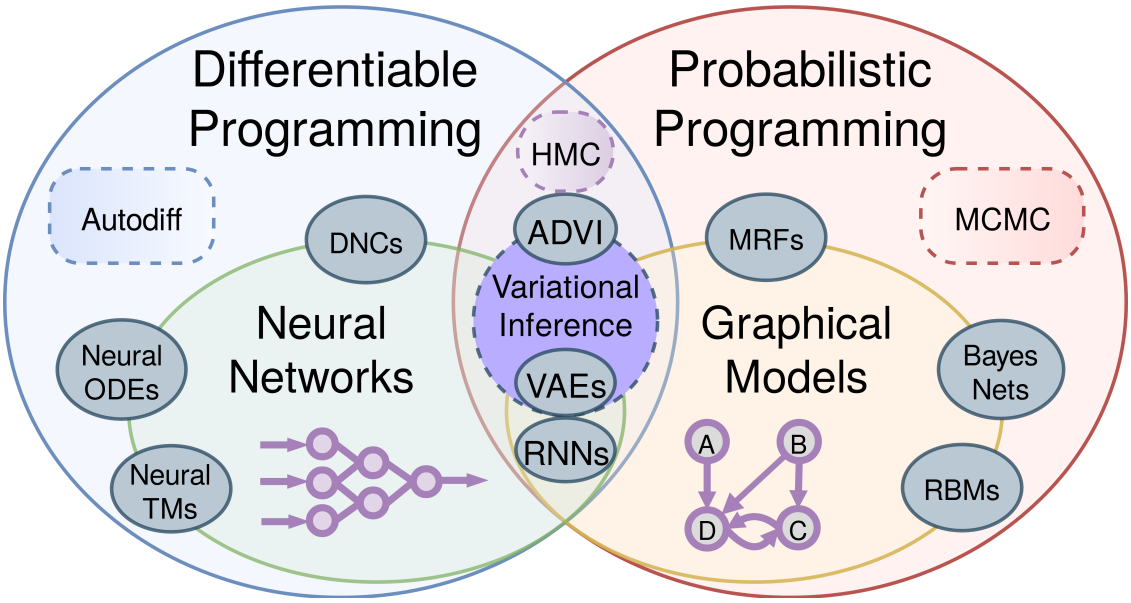


Fig. 3.1. *Differentiable programming* includes neural networks, but more broadly, arbitrary programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* [Carpenter et al., 2017, Gorinova et al., 2019] is a generalization of probabilistic graphical models which uses Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function.

learning architectures and representation learning. Equally important, arguably, is the role that automatic differentiation (AD) has played in facilitating the implementation of these ideas. Prior to the advent 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 and scientific computing communities.

A key aspect of the connectionist paradigm is gradient descent of a statistical loss function defined on a neural network with respect to its free parameters. 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 written language is not easily differentiable, as small changes to a word's symbolic form can cause sudden changes to its semantics [van Merriënboer, 2018]. A key insight from representation

learning is that many discrete data types can be mapped into a smoother latent space. For example, if we represent words as a vector of real numbers, $\mathbb{R}^N$, then it is possible to learn a mapping from words to $\mathbb{R}^N$ so that semantic relations between words (as defined by their statistical co-occurrence in large corpora) are geometrically preserved in vector space [Pennington et al., 2014] – words with similar meanings map to similar vectors. Many classes of discrete problems can be relaxed to continuous surrogates by learning such representations, or *embeddings* in an unsupervised, or semi-supervised 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 8-bit floating point [Wang et al., 2018d] and integer [Wu et al., 2018, Jacob et al., 2017] arithmetic are able to train neural networks without sacrificing performance. Strong assumptions like Lipschitz-continuity and β-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 have been less surprising, 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, gradient descent 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* [Olah, 2015] (DP) was born. Today, DP has many applications, from protein folding [AlQuraishi, 2018], to physics engines [Hu et al., 2019, 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 all have parameters that are tuned via gradient descent. Traditionally, handcrafted optimization algorithms were required, but given a smooth metric, DP promises to learn these parameters for a broad class of models, more or less automatically. Discrete optimization, however, remains an open question. To "learn" discrete relations without ad hoc embedding, additional techniques (§ 3.20), such as probabilistic programming, are likely needed. As seen in Figure 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]. According to some studies, type-related errors account for over 15% of software bugs [Gao et al., 2017]. While the causality between defectiveness and static typing has not been conclusively established, dynamically-typed languages are seldom used for building safety-critical systems, and the majority of robotics applications [Guenther, 2018] are written in statically-typed languages.

Static typing eliminates a broad class of runtime errors, allowing developers and tools to reason more carefully about the behavior of programs without needing to execute them. In addition to stronger syntax validation for general-purpose programming, a well-designed library in a strongly-typed language can eliminate domain-specific errors related to API misuse that would otherwise require documentation and code samples to avert, improving usability and reducing maintenance. Furthermore, strong type systems allow us to build more intelligent static analysis tools, which can provide relevant autocompletion, source code navigation, and earlier detection of runtime errors.

One frequent objection to using strongly-typed languages is attributed to the additional burden of manual type annotation. While early type-safe languages like C++ and Java required programmers to exhaustively annotate function and variable declarations, with judicious use of type inference in modern languages like Kotlin, Scala, Rust et al., most type signatures may be safely omitted and easily recovered from the surrounding context. Type inference enables modern languages to offer the brevity of dynamic-typed languages with the safety of static type checking.

3.4. Imperative and functional languages

Most programs today are written in the imperative style, due the prevalence of the Turing Machine and von Neumann architecture [Backus, 1978]. λ-calculus provides an equivalent[2] 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

[2]In the sense that the Turing Machine and λ-calculus are both Turing complete.

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

Fig. 3.2. Two equivalent programs, both implementing the function $f(l_1, l_2) = l_1 \cdot l_2$.

without doing so. More troubling in the case of AD, is that imperative programs have mutable state, which requires taking extra precautions when computing their derivatives.

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

For example, consider the vector function $f(l_1, l_2) = l_1 \cdot l_2$, seen in Figure 3.2. Imperative programs, by allowing mutation, are effectively destroying intermediate information. In order to recover the computation graph for reverse-mode AD, we either need to override the assignment operator, or use a tape to store the intermediate values. In pure functional programming, mutable variables do not exist, which makes our lives much easier.

Functional programming lets Kotlin∇ use the same abstraction for representing mathematical functions and programming functions. All functions in Kotlin∇ are pure functions, composed of expressions forming a data-flow graph (DFG). An expression is simply a `Function`, which is only evaluated when invoked with numerical values, e.g. `z(0, 0)`. In this way, Kotlin∇ is similar to other graph-based frameworks like TensorFlow and Theano.

3.5. Kotlin

When programming in a statically-typed language, a common question one might ask the compiler is, "Given a value, `x`, can `x` be assigned to a variable of type `Y`?" (e.g. type checking `x instanceof Y`) In Java, this question turns out to be ill-posed [Amin and Tate,

2016] and undecidable [Grigore, 2017] in the general case. It is possible to construct a Java program in which the answer is "yes" regardless of `Y`, or for which an answer cannot always be determined in finite time. Undecidability is not necessarily a showstopper, but Java's unsoundness is more critical and unclear how to fix, even though it rarely occurs in practice.

Kotlin is a statically-typed language that is well-suited for building cross-platform applications, with compiler support for JVM, JavaScript and native targets. Unlike most programming languages, Kotlin was designed with IDE support from the outset, and has gained some traction in the JVM ecosystem due to its ergonomics. Kotlin's type system [Tate, 2013] is strictly less expressive, but fully interoperable with Java's. It is unknown whether the same issues which affect Java's type system are present in Kotlin's, but interoperability with Java has broadened its adoption and remains a key usability feature of the language.

In this work, we make use of several language features unique to Kotlin, such as first-class functions (§ 3.12), extension functions (§ 3.16), operator overloading (§ 3.11), and algebraic data types (§ 3.14). Furthermore, we make heavy use of Kotlin's DSL support to implement shape-safe array programming. Together, these language features provide a concise, flexible and type-safe platform for mathematical programming.

3.6. Kotlin∇

Prior work has demonstrated the possibility of encoding a deterministic context free (DCF) language in the Java type system as a *fluent interface* [Gil and Levy, 2016, Nakamaru et al., 2017]. This result was strengthened to prove Java's type system is Turing complete (TC) [Grigore, 2017], which enables us to perform shape checking and inference on array programs written in Java. Kotlin is a Java descendant which is at least DCF at the type level. Kotlin∇, an embedded DSL in the Kotlin language is TC at the value level and DCF at the type level. A similar approach is feasible in most languages with generic types.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano [Bergstra et al., 2010], Torch [Collobert et al., 2002], and TensorFlow [Abadi et al., 2016]. Similar ideas have been implemented in functional languages such as Scheme (Stalin∇ [Pearlmutter and Siskind, 2008b]), and statically-typed languages like F# (DiffSharp [Baydin et al., 2015b])

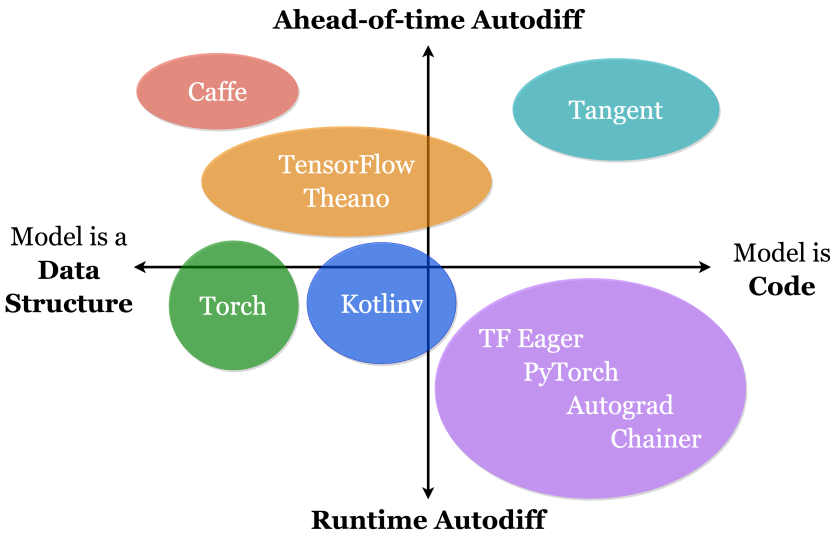


Fig. 3.3. Adapted from van Merriënboer et al. [2018]. Kotlin∇ models are data structures, constructed by an embedded DSL, eagerly optimized, and lazily evaluated.

and 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∇ is fully interoperable with vanilla Java, enabling broader adoption in neighboring languages. To our knowledge, Kotlin has no prior AD implementation. However, the language has several useful features for implementing a native AD framework. Kotlin∇ 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.
- **λ-functions** support functional programming, following Pearlmutter and Siskind [2008a,b], Siskind and Pearlmutter [2008], Elliott [2009, 2018], et al.
- **Extension functions** support extending classes with new fields and methods which can be exposed to external callers without requiring sub-classing or inheritance.

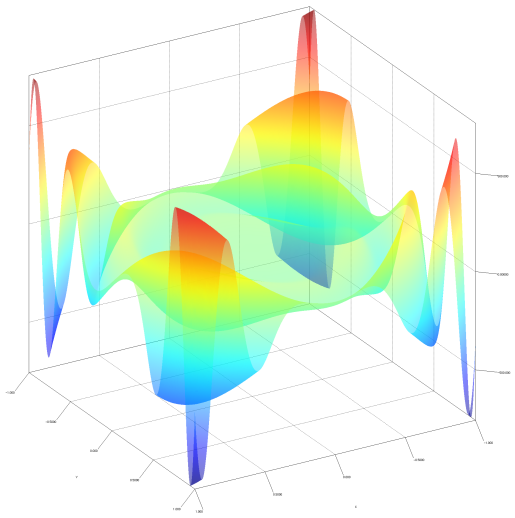
Kotlin∇ models are embedded domain-specific languages (eDSLs). These languages may look and act different from the host language, but are really just carefully disguised functions for building an abstract syntax tree (AST). Often these ASTs represent simple

state machines, but are also used to embed a 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 the need to write a lexer, parser, compiler or interpreter. And with proper typing, users will receive code completion and static analysis from their favorite developer tools. Functional languages are often suitable host languages [Elliott et al., 2003, Rompf and Odersky, 2010], perhaps owing to the notion of code as data.

3.7. Usage

Kotlin∇ allows users to implement differentiable programs by composing expressions. Consider the following Kotlin∇ program with two inputs and one output:

```
with(DoublePrecision) { // Uses double precision numerics for evaluation
  val x = Var("x") // Declare immutable variables (these variables are
  val y = Var("y") // just symbolic constructs used for differentiation)
  val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazily evaluated
  val dz_dx = d(z) / d(x) // Supports Leibniz notation [Christianson, 2012]
  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 t = (1 + x * 2 + z / y).d(y).d(x) + z / y * 3 - 4 * (y pow y).d(y)
```

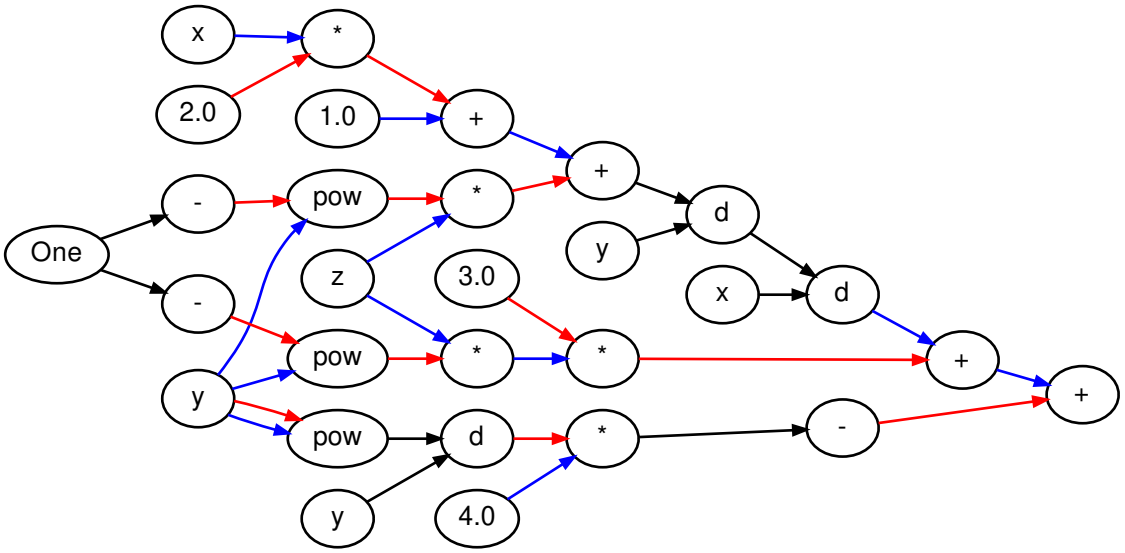


Fig. 3.4. Implicit DFG constructed by the original expression, shown above.

Above, we define a function with two variables and take a series of partial derivatives with respect to each variable. Expressions are lazily evaluated inside a numerical context, which may be imported on a per-file basis or lexically scoped for finer-grained control over the runtime behavior. The function is numerically evaluated on the interval $(-1, 1)$ in each dimension and rendered in 3-space.

3.8. Type systems

Early work in type-safe dimension analysis can be found in Kennedy [1994, 1996] which uses types to encode dimensionality and prevent common bugs related to dimension mismatch from arising, and was later realized in the F# language [Kennedy, 2010]. Jay and Sekanina [1997], Rittri [1995], and Zenger [1997] explore the application of dimension types for linear algebra. More recently, Kiselyov [2005], Kiselyov et al. [2009] and Griffioen [2015], show how to manipulate arrays in more complex ways. With the resurgence of interest in tensor algebra and array programming, Chen [2017] and Rink [2018] demonstrate how to encode shape-safety for tensor algebra in various type systems.

The problem we attempt to solve can be summarized as follows. Given two values `x` and `y`, and operator `$`, how do we determine whether the expression `z = x $ 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 `@` : $(\mathbb{R}^{\mathbf{a}}, \mathbb{R}^{\mathbf{b}}) \to \mathbb{R}^{\mathbf{c}}$ where $\mathbf{a} \in \mathbb{N}^q, \mathbf{b} \in \mathbb{N}^r, \mathbf{c} \in \mathbb{N}^s$ and $q, r, s \in \mathbb{N}$. For many linear algebra operations such as matrix multiplication, $\mathcal{T}(\mathbf{a}, \mathbf{b}) \stackrel{?}{=} \mathbf{c}$ is computable in $\mathcal{O}(1)$ – we can simply check the inner dimensions for equivalence ($\mathbf{a}_2 \stackrel{?}{=} \mathbf{b}_1$).

Shape checking operations on multidimensional arrays is not always decidable. For arbitrary type functions $\mathcal{T}(\mathbf{a}, \mathbf{b})$, checking $\mathcal{T}(\mathbf{a}, \mathbf{b}) \stackrel{?}{=} \mathbf{c}$ requires a Turing machine. 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 undecidable [Gödel, 1931]. Addition, subtraction, indexing and comparison of integers are all decidable operations in Presburger arithmetic [Suzuki and Jefferson, 1980, Bradley et al., 2006, Charlier et al., 2011]. Equality checking is trivially decidable, and can be implemented in most static type systems.

Evaluating an arbitrary $\mathcal{T}$ which uses multiplication or division (e.g. convolutional arithmetic) requires a dependently typed language [Xi and Pfenning, 1998, Piñeyro et al., 2019], but checking shape equality (e.g. shape checking ordinary arithmetic operations) is feasible in Java and its cousins.[3] Furthermore, we believe that shape checking ordinary matrix arithmetic is decidable in any type system loosely based on System $F_{<:}$ [Cardelli et al., 1994]. We propose a type system for enforcing shape-safety which can be implemented in any language with subtyping and generics, such as Java [Naftalin and Wadler, 2007], Kotlin [Tate, 2013], TypeScript [Bierman et al., 2014] or Rust [Crozet et al., 2019].

3.9. Shape safety

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

(1) Conceal the error by implicitly reshaping or broadcasting arrays.

(2) Announce the error at runtime with a relevant message, e.g. `InvalidArgumentError`.

[3] Java's type system is known to be Turing complete [Grigore, 2017]. Thus, emulation of dependent types in Java is theoretically possible, but likely intractable due to the practical limitations noted by Grigore.

Math	Infix	Prefix	Postfix	Operator Type Signature
$A(B)$	`a(b)`			$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\tau}) \to (\mathbb{R}^{\lambda} \to \mathbb{R}^{\pi})$
$A \pm B$	`a + b` `a - b`	`plus(a, b)` `minus(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\pi}) \to (\mathbb{R}^{?} \to \mathbb{R}^{\pi})$
AB	`a * b` `a.times(b)`	`times(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times n}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{n \times p}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times p})$
$\frac{A}{B}$ AB^{-1}	`a / b` `a.div(b)`	`div(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times n}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{p \times n}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times p})$
$\pm A$		`-a` `+a`	`a.unaryMinus()` `a.unaryPlus()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}) \to (\mathbb{R}^{\tau} \to \mathbb{R}^{\pi})$
$\ln(A)$		`ln(a)` `log(a)`	`a.ln()` `a.log()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^{\tau} \to \mathbb{R}^{m \times m})$
$\log_b A$	`a.log(b)`	`log(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^{?} \to \mathbb{R})$
A^b	`a.pow(b)`	`pow(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times m})$
$\sqrt{a}$ $\sqrt[3]{a}$	`a.pow(1.0/2)` `a.root(3)`	`a.pow(1.0/2)` `a.root(3)`	`a.sqrt()` `a.cbrt()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}) \to (\mathbb{R} \to \mathbb{R}^{m \times m})$
$\frac{da}{db}$ $a'(b)$	`a.d(b)`	`grad(a)[b]`	`d(a) / d(b)`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\omega}) \to (\mathbb{R}^{?} \to \mathbb{R}^{\pi \times \omega})$

Table 3.1. Kotlin∇'s shape system specifies the output shape for tensor expressions.

(3) Do not allow programs which can result in a shape error to compile.

Most array programming libraries such as NumPy [Van Der Walt et al., 2011] or TensorFlow [Abadi et al., 2016] use the first or second strategy. In Kotlin∇, we adopt the third, which allows an incremental type checker, such as those typically found in modern IDEs, to instantaneously detect when a matrix operation is invalid. Consider the following example:

```
val vecA = Vec(1.0, 2.0)      // Inferred type: Vec<Int, D2>
val vecB = Vec(1.0, 2.0, 3.0) // Inferred type: Vec<Int, D3>
val vecC = vecB + vecB
val vecD = vecA + vecB // Compile error: Expected Vec<2>, found Vec<3>
```

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

```
val matA = Mat1x4(1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, D1, D4>
val matB = Mat4x1(1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, D4, D1>
val matC = matA * matB
val matD = matA * matC // Compile error: Expected Mat<4, *>, found Mat<1, 1>
```

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

```
val matA = Mat2x4(1.0, 2.0, 3.0, 4.0,
                  5.0, 6.0, 7.0, 8.0)
val matB = Mat4x2(1.0, 2.0,
                  3.0, 4.0,
                  5.0, 6.0,
                  7.0, 8.0)
val matC: Mat<Double, D2, D2> = a * b // Types are optional, but encouraged
val matD = Mat2x1(1.0, 2.0)
val matE = matC * matD
val matF = Mat3x1(1.0, 2.0, 3.0)
val matG = matE * matF // Compile error: Expected Mat<1, *>, found Mat<3, 1>
```

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

```
fun someMatFun(m: Mat<Double, D3, D1>): Mat<Double, D3, D3> = ...
fun someMatFun(m: Mat<Double, D2, D2>) = ...
```

Shape safety is currently supported up to rank-2 tensors, i.e. matrices. To perform dimension checking in our type system, first we enumerate a list of integer type literals as a chain of subtypes, $C <: C-1 <: C-2 <: \cdots <: 1 <: 0$, where C is the largest fixed-length dimension we wish to represent, which can be specified by the user prior to compilation. This guarantees linear space and time complexity for subtype checking, with a constant upper bound.

```
interface Nat<T: D0> { val i: Int }
// Integer literals have reified Int values should we need to compare them at runtime
sealed class D0(open val i: Int = 0) { companion object: D0(), Nat<D0> }
sealed class D1(override val i: Int = 1): D0(i) { companion object: D1(), Nat<D1> }
sealed class D2(override val i: Int = 2): D1(i) { companion object: D2(), Nat<D2> }
sealed class D3(override val i: Int = 3): D2(i) { companion object: D3(), Nat<D3> }
//...Code for integer literals should be generated
sealed class D99(override val i: Int = 99): D98(i) { companion object: D99(), Nat<D99> }
```

Next, we overload the call operator to emulate instantiating a collection literal, using arity to infer its dimensionality. Consider the rank-1 case for length inference on vector literals:

```
open class Vec<E, Len: D1> constructor(val contents: List<E>) {
    companion object {
        operator fun <T> invoke(t: T): Vec<T, D1> = Vec(listOf(t))
        operator fun <T> invoke(t0: T, t1: T): Vec<T, D2> = Vec(listOf(t0, t1))
        operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, D3> = Vec(listOf(t0, t1, t2))
    }
}
```

Finally, we overload arithmetical operators using generic shape constraints. Since our type-level integers are a chain of subtypes, we only need to define one operator and can rely on Liskov substitution [Liskov, 1987] to preserve shape safety for all subtypes.

```
// <C: D1> will accept 1 <= C <= 99 via Liskov substitution
operator fun <E, C: D1, V: Vec<X, C>> V.plus(v: V): V = TODO()
```

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(0, 0) + Vec(0, 0) // Y: Vec<Float, D2>
val X = Vec(0, 0) + Vec(0, 0, 0) // Compile error: Expected Vec<Int, D2>, found Vec<Int, D3>
```

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

```
val one = Vec(0, 0, 0) + Vec(0, 0, 0) // Always runs safely
val add = Vec(0, 0, 0) + Vec<Int, D3>(listOf(...)) // Compiles, but may fail at runtime
val vec = Vec(0, 0, 0) // Inferred type: Vec<3>
val sum = Vec(0, 0) + add // Compile error: Expected Vec<Int, D2>, found Vec<Int, D3>
```

Matrices and tensors have a similar syntax. For example, Kotlin∇ can infer the shape of matrix multiplication, and will not compile if the arguments' inner dimensions disagree:

```
open class Mat<X, R: D1, C: D1>(vararg val rows: Vec<X, C>)
fun <X> Mat1x2(d0: X, d1: X): Mat<X, D1, D2> = Mat(Vec(d0, d1))
fun <X> Mat2x1(d0: X, d1: X): Mat<X, D2, D1> = Mat(Vec(d0), Vec(d1))

operator fun <X, Q: D1, R: D1, S: D1> Mat<X, Q, R>.times(m: Mat<X, R, S>): Mat<X, Q, S> =
    Mt( *(rows.indices).map { i -> /* ... */ }.toTypedArray() )

val matM = Mat1x2(0, 0)
```

```
val mat0 = matM * matM // Compile error: Expected Mat<2, *>, found Mat<1, 2>
```

A similar technique can be found in nalgebra [Crozet et al., 2019], a shape-checked linear algebra library for the Rust language which also uses synthetic type-level integers. This technique originates in Haskell, a language which supports more powerful forms of type-level computation, such as *type arithmetic* [Kiselyov, 2005]. Type arithmetic simplifies array concatenation, convolutional arithmetic [Dumoulin and Visin, 2016] and other operations which are currently difficult to express in Kotlin∇, where arbitrary type-level functions $\mathcal{T}(\mathbf{a}, \mathbf{b})$ (ref. § 3.8) can require enumerating up to C^{q+r} Kotlin functions to compute.

3.10. Testing

Kotlin∇ claims to eliminate certain runtime errors, but how do we know the implementation is not incorrect? One method is known as property-based testing (PBT) [Fink and Bishop, 1997] (§ 4.4), closely related to the notion of metamorphic testing [Chen et al., 1998] (§ 4.5). Notable implementations include QuickCheck [Claessen and Hughes, 2000], Hypothesis [MacIver, 2018] and KotlinTest [Samuel and Lopes, 2018], on which our test suite is based. PBT uses algebraic properties to verify the result of a calculation by constructing semantically equivalent but syntactically distinct expressions. When evaluated on the same inputs, these should produce the same answer, to within numerical precision. Two such equivalences are used to test Kotlin∇:

(1) **Analytical differentiation**: manually differentiate selected functions and compare the numerical result of evaluating random chosen inputs from their domain with the numerical result obtained by evaluating AD on the same inputs.
(2) **Finite difference approximation**: sample the space of symbolic differentiable functions, comparing the numerical results suggested by the finite difference method and the equivalent AD result, up to a fixed-precision approximation.

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

```
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 `x0` and `y0`, which violate the specification, then "shrink" them to discover pass-fail boundary values. We can construct a similar test using the finite difference method, e.g. $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$:

```
val dx = 1E-8
val f = sin(x)
val df_dx = d(f) / d(x)
val fd_dx = (sin(x + dx) - sin(x)) / dx

"d(sin x)/dx should be equal to (sin(x + dx) - sin(x)) / dx" {
    NumericalGenerator.assertAll { x0 ->
    val autoEval = df_dx(x0)
        val fdEval = fd_dx(x0)
        autoEval shouldBeApproximately fdEval // Fails iff eps < |adEval - fdEval|
    }
}
```

There are many other ways to independently verify the numerical gradient, such as dual numbers or the complex step derivative. Another method to validate Kotlin∇'s implementation would be to compare the numerical output against the output of a well-known AD framework, such as TensorFlow. In future work, we intend to conduct a more thorough comparison of numerical accuracy and performance.

3.11. Operator overloading

Operator overloading [Corliss and Griewank, 1993] is one of the simplest ways to implement automatic differentiation. We use Kotlin's operator overloading functionality on a numeric tower (ref. § 3.13) to provide a concise notation for abstract algebraic operations. For example, suppose we have an interface `Group`, which overloads the operators `+` and `*`:

```
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 value of the type variable `T`, rather than something more abstract 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 Kotlin∇, 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

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

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

Currently, it is possible to represent functions where all inputs and outputs share a single data type. It may be possible to extend support for building functions with varying input/output types and enforcing constraints on both, by using covariant and contravariant type bounds.

3.13. Numeric Tower

Kotlin∇ uses a numeric tower [St-Amour et al., 2012]. An early example of this pattern can be found in Scheme [Sperber et al., 2009]. This strategy is also suited to object oriented languages [Niculescu, 2003, 2011, Kennedy and Russo, 2005] and applied in libraries such as KMath [Nozik, 2019] and Apache Commons Math [Developers, 2012].

```
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(divisor: X): X = this * divisor.pow(-one)
    infix fun pow(exp: X): X
    fun ln(): X
}
```

The numeric tower allows us to define common behavior such as subtraction and division on abstract algebraic structures, e.g. `Group`, `Ring`, and `Field`. These abstractions are extensible to concrete number systems, such as complex numbers and quaternions. For example, to later define a field over complex numbers or quaternions,[4] one must simply extend the numeric tower and override the default implementation. Most mathematical operations can be defined using a small set of primitive operators, which can be differentiated in a generic fashion, rather than on an ad hoc basis.

[4] ex. In order to calculate derivatives in a quaternion neural network. [Isokawa et al., 2003]

3.14. Algebraic data types

Algebraic data types (ADTs) in the form of sealed classes (a.k.a. sum types) facilitate a limited form of pattern matching over a closed set of subclasses. When matching against subclasses of a sealed class, the compiler forces 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>()
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 case, since incomplete control flow will not compile rather than fail silently at runtime. Let us now consider a simplified definition of `Fun`, a sealed class which defines the behavior of function invocation and differentiation, using a restricted form of pattern matching. It can be constructed with a set of `Var`s, and can be invoked with a numerical value:

```
sealed class Fun<X: Fun<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.getOrElse(this) { this } // Partial application is permitted
        is Prod -> left(map) * right(map) // Smart casting implicitly casts after checking
        is Sum -> left(map) + right(map)
    }

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

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

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

Kotlin's smart casting is an example of flow-sensitive type analysis [Pearce and Noble, 2011] where the abstract type `Fun` can be treated as `Sum` after performing an `is Sum` check. Without smart casting, we would need to write `(this as Sum).left` to access the member, `left`, creating a potential `ClassCastException` if the cast were mistaken.

3.15. Multiple Dispatch

In conjunction with ADTs, Kotlin∇ uses multiple dispatch to instantiate the most specific result type of an arithmetic operation based on the type of its operands. Although Kotlin does not directly support multiple dispatch, it can be emulated using single dispatch as described by Leavens and Millstein [1998]. Building on § 3.14, suppose we wish to rewrite some algebraic expression, e.g. to reduce expression swell or improve numerical stability. We can use `when` to branch on the type of a subexpression at runtime:

```
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))
```

Multiple dispatch allows us to put all related control flow on a single abstract class which is inherited by subclasses, simplifying readability, debugging and refactoring.

3.16. Extension Functions

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

```
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, an approach borrowed from KMath [Nozik, 2019], another mathematical library for Kotlin. This approach is also suitable for defining convenience methods for variable assignment and type adapters for numerical primitives in a context sensitive manner. For example:

```
object DoubleContext: Proto<DConst, Double>() {
    override val Const<DConst, Number>.value: Double
    get() = c.toDouble()
    override fun wrap(default: Number): DConst = DConst(default.toDouble())
    override val X: X<DConst> = object: X<DConst>(DConst(0.0)) {
        override fun invoke(X: XBnd<DConst>): DConst = X.const
        override fun toString() = "X"
    }
    override val Y: Y<DConst> = object: Y<DConst>(DConst(0.0)) {
        override fun invoke(Y: YBnd<DConst>): DConst = Y.const
        override fun toString() = "Y"
    }
    override infix fun X<DConst>.to(c: Double) = XBnd(DConst(c))
    override infix fun Y<DConst>.to(c: Double) = YBnd(DConst(c))
}
```

This DSL, which is used to support variable capture and currying, can be used as follows:

```
with(DoubleContext) {
    val t = X + Y + 0.0
    val l = t(X to 1.0, Y to 2.0)
    val r = t(X to 1.0)(Y to 3.0) // Currying
    val o = X + Z + 0.0
    val p = o(X to 1.0) // Partial application
    val k = o(Y to 4.0) // Does not compile
}
```

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, have suspected this claim to be misleading. Recently, the claim has been questioned [Wang et al., 2018b] and refuted [Laue, 2019]. While it may be true that certain implementations of automatic differentiation interleave numerical evaluation and symbolic differentiation at runtime, this interleaving is certainly not a prerequisite for a differentiation library to be considered *automatic*. Nor, as suggested by prior literature [Baydin and Pearlmutter, 2014], is the problem of expression swell unique to symbolic differentiation [Laue, 2019], whose findings we can numerically reproduce as shown in Figure 4.3.

The distinction between AD and SD becomes increasingly blurry when we consider more flexible execution models [Wang et al., 2018b] and hybrid ADs [Abadi et al., 2016] which are capable of both eager [Agrawal et al., 2019] and lazy evaluation. Instead, we take the view that symbolic differentiation is a type of automatic differentiation which the AD literature has been too quick to dismiss. SD in particular, affords the compiler far more flexibility to perform global optimizations such as algebraic simplification [Bergstra et al., 2010], loop vectorization [Agarwal, 2019] and tensor comprehension [Vasilache et al., 2018]. These optimizations would otherwise be impossible if their symbolic differentiation and numerical evaluation were performed in lockstep, when the dataflow graph is only partially available.

3.18. Coroutines

Coroutines are a generalization of subroutines for non-preemptive multitasking, typically implemented using continuations [Haynes et al., 1984]. Continuations are a mechanism

that allow functions to access and modify subsequent computation. In continuation-passing style [Sussman and Steele, 1975] (CPS), every function, in addition to its usual arguments, takes another function representing the subsequent routine. Rather than returning to its caller after completion, the function invokes its continuation, and the process is restarted.

One form of continuation, known as delimited continuations, are sufficient for implementing reverse-mode AD with operator overloading alone (without any additional data structures) as described by Wang et al. [2018b] and later in Wang et al. [2018a]. Delimited continuations can be implemented with Kotlin Coroutines, and merits further investigation.

3.19. Comparison

Inspired by Stalin∇ [Pearlmutter and Siskind, 2008b], Autograd [Maclaurin et al., 2015, Maclaurin, 2016], Theano [Bergstra et al., 2010], Myia [Breuleux and van Merriënboer, 2017, van Merriënboernboer et al., 2018], JAutoDiff [Nureki, 2012], Nexus [Chen, 2017], Lantern [Wang et al., 2018b], Tangent [van Merriënboer et al., 2018], Elliott [2018], Halide [Li et al., 2018] et al., Kotlin∇ attempts to port recent developments in automatic differentiation (AD) to the Kotlin language. In the process, it introduces a number of experimental ideas, including compile-time shape-safety, algebraic simplification and numerical stability checking through property-based testing. Prior work, including PyTorch [Paszke et al., 2017], TensorFlow [Abadi et al., 2016], Chainer [Tokui et al., 2015], DL4J Team [2016a] and others have developed general-purpose AD libraries in less safe languages.

Unlike most existing AD implementations, Kotlin∇ is a purely symbolic, graph-based AD that does not require any template metaprogramming, compiler augmentation or runtime reflection to ensure type safety. As we have seen, this approach is primarily achieved through operator overloading, parametric 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 (eDSL), thereby leveraging the host language's type system to receive code completion and type inference for free. Our approach is particularly well-suited to functional programming, and employs several functional programming concepts, including lambda expressions, higher order functions, partial application, currying and algebraic data types. For a more detailed comparison of Kotlin∇ with existing AD libraries, see Table 3.2.

Framework	Language	Symbolic Differentiation	Automatic Differentiation	Differentiable Programming	Functional Programming	Type-Safe	Shape-Safe	Dependently-Typed	Multiplatform
Kotlin∇	Kotlin	✓	✓	✍	✓	✓	✓	✗	✍
DiffSharp	F#	✗	✓	✓	✓	✓	✗	✗	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗	✗
Lantern	Scala	✗	✓	✓	✓	✓	✗	✗	✗
Tensor Safe	Haskell	✗	✓	✗	✓	✓	✓	✓	✗
Hasktorch	Haskell	✗	✓	✓	✓	✓	✓	✗	✗
Eclipse DL4J	Java	✗	✓	✗	✗	✓	✗	✗	✗
JAutoDiff	Java	✓	✓	✗	✗	✓	✗	✗	✗
Stalin∇	Scheme	✗	✓	✗	✗	✗	✗	✗	✗
Myia	Python	✓	✓	✓	✓	✗	✗	✗	✍
JAX	Python	✗	✓	✓	✓	✗	✗	✗	✍
Tangent	Python	✗	✓	✗	✗	✗	✗	✗	✗

Table 3.2. Comparison of AD libraries. The ✍symbol indicates work in progress.

Kotlin∇ advocates for the use of shape safe, functional array programming, but unlike similar frameworks, does not impose its preferences on consumers. If the user omits shape, it falls back to runtime shape checking. In keeping with the design principles of the host language, users can employ their preferred programming style and structure, gradually introducing shape constraints to enjoy the benefits of stronger type checking and avail themselves of richer functional programming support.

3.20. Future work

> "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."
>
> –Alonzo Church [1941], *The Calculi of Lambda Conversion*

The derivative, as it is most commonly used, is usually associated with the calculus of infinitesimals. But the same rules for symbolic differentiation introduced by Leibniz and Newton over three centuries ago have reappeared in strange and marvelous places throughout computer science. In Brzozowski [1964], we encounter an example of symbolic differentiation in a discrete setting, i.e. regular expressions. Brzozowski's work has important and far-reaching applications in automata theory [Berry and Sethi, 1986, Caron et al., 2011, Champarnaud et al., 1999] and incremental parsing [Might et al., 2011, Moss, 2017]. Later in Thayse [1981] the boolean differential calculus was first introduced,[5] a branch of boolean algebra which has important applications in switching theory [Thayse and Davio, 1973] and synthesis of digital circuits [Steinbach and Posthoff, 2017]. Symbolic differentiation has useful applications in other mathematical settings, including λ-calculus [Ehrhard and Regnier, 2003, Cai et al., 2014, Kelly et al., 2016, Brunel et al., 2020], incremental computation [Alvarez-Picallo et al., 2018, Alvarez-Picallo and Ong, 2019], type theory [McBride, 2001, 2008, Chen et al., 2012], category theory [Blute et al., 2006, 2009], domain theory [Edalat and Lieutier, 2002], probability theory [Kac, 1951] and linear logic [Ehrhard, 2016, Clift and Murfet, 2018].

Many further examples of symbolic differentiation can be found in unrelated bodies of literature. This pattern seems unlikely to be mere coincidence, and suggests an unrealized connection between differential and algebraic geometry, perhaps holding important insights for differentiable programming and the study of change propagation in computation graphs.

The work described in this chapter establishes a framework for exploring symbolic differentiation using algebraic structures like `Group`, `Ring`, and `Field` (§ 3.13). In future work, we hope to explore the relationship between differentiable programming and symbolic differentiation in other topologies. Perhaps there exists an analogous mechanism to gradient descent which can be exploited to accelerate optimization in such spaces, e.g. for learning boolean variables and other data structures like graphs and trees.

[5] Although early work on the subject can be traced back to Talantsev [1959] and Sellers et al. [1968]

As shown in prior literature [Bergstra et al., 2010, Baydin et al., 2015a, Laue, 2019], intermediate expression swell is a pernicious issue in computer algebra and automatic differentiation. The ad-hoc algebraic simplification procedure described in § 3.15 is almost certainly inadequate for general use cases. One interesting direction would be training a model to minimize numerical drift, by applying general-purpose rewriting rules. There exists a long list of prior work in rewriting algorithms for numerical stability, dating back to Kahan [1965], Dekker [1971], Ogita et al. [2005] and more recently explored by Zaremba et al. [2014], Zaremba [2016] and Wang et al. [2019] from a machine learning perspective.

Providing a type for matrix structure (e.g. `Singular`, `Symmetric`, `Orthogonal`) would allow specializations of the matrix derivative (§2.8 of Petersen et al. [2012] for a detailed review of specific techniques for differentiating structured matrices). In terms of enhancing the type system, Makwana and Krishnaswami [2019] have developed a linearly-typed encoding of linear algebra which would also be interesting to explore.

From a performance standpoint, migrating to a dedicated linear algebra backend such as ND4J [Team, 2016b], Apache Commons Math [Developers, 2012], EJML [Abeles, 2010] or JBlas [Braun et al., 2011] would likely yield some speedup. Ultimately, we plan to compile to a dedicated intermediate representation such as RelayIR [Roesch et al., 2018] in order to receive hardware acceleration on other platforms.

3.21. Conclusion

In this chapter, we have demonstrated Kotlin∇, an embedded domain specific language for differentiable programming and its implementation in the Kotlin programming language. Using our DSL as a vehicle, we explored some interesting topics in automatic differentiation and shape safe array programming. The author wishes to thank Hanneli Tavante, Alexander Nozik, Erik Meijer, Maxime Chevalier-Boisvert and Kiran Gopinathan for their valuable feedback during the development of this project.

Chapter 4

Testing intelligent systems

> "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."
>
> –Norbert Wiener [1960], *Some moral and technical consequences of automation*

Today's deep neural networks are capable of learning a broad range of functions, but have specific weaknesses. Training neural networks which can robustly transfer to new domains where the training and test distributions are highly dissimilar poses a significant challenge. These models are often susceptible to failure when presented with carefully crafted inputs. However, the same gradient-based optimization techniques used for training neural networks can also be exploited to probe their failure modes.

In software engineering, techniques for software testing are becoming increasingly automated and general-purpose. Tests help prevent regressive behavior and are a form of specification in which the developer communicates the intended result of running a program. While essential for validating a program's correctness, tests are often cumbersome to implement. Techniques in coverage-guided fuzzing have enabled developers to write fewer tests with higher coverage. This is made possible by automated testing.

In this chapter, we will explore the relationship between testing in machine learning and software engineering. We will see how the notion of adversarial testing shares a curious resemblance to fuzz testing in software engineering. In particular, we show how probabilistic sampling and constrained optimization can be seen as an extension of property-based testing (PBT) for adversarial training of differentiable programs, and propose a PBT algorithm which incorporates features of probabilistic programming and gradient-based optimization.

4.1. Unit Testing

In traditional unit testing, most tests are written in the following manner:

```
fun unitTest(subroutine: (Input) -> Output) {
    val input = Input() // Construct an input
    val expectedOutput = Output() // Construct an output
    val actualOutput = subroutine(input)
    assert(expectedOutput == actualOutput) { "Expected $expectedOutput, got $actualOutput"}
}
```

When carefully applied, unit testing can be an effective method for detecting bugs and validating the author's belief about preconditions and postconditions. The trouble is, someone needs to write a bunch of test cases for it to work. In addition, it only tests subprograms, and must be updated whenever the program changes. This has the unintended side effect of decreasing agility, discouraging refactoring, or discarding prior work when tests become obsolete.

4.2. Integration Testing

In integration testing, we are more concerned about the overall behavior of a program, rather than the specific behavior of its subroutines. Consider the following example:

```
fun <I, O> integrationTest(program: (I) -> O, inputs: Set<I>, checkOutput: (O) -> Boolean) =
    inputs.forEach { input: I ->
        try {
            val output: O = program(input)
            assert(checkOutput(output)) { "Postcondition failed on $input, $output" }
        } catch (exception: Exception) {
            assert(false) { exception }
        }
    }
```

With this strategy, there are fewer tests to write down, since we only care about end-to-end behavior. Integration testing simply checks a program for terminating exceptions and simple post conditions. For this reason, it is often too coarse-grained.

For simplicity, in the following sections, we will only consider examples of programs which are pure functions, i.e. which have no external state and produce no side effects.

4.3. Fuzz Testing

Fuzz testing is an automated testing methodology which generates random inputs to test a given program. For example, consider the following test:

```
fun <I, O> fuzzTest(program: (I) -> O, oracle: (I) -> O, rand: () -> I) =
    repeat(1000) {
        val input: I = rand()
        assert(program(input) == oracle(input)) { "Oracle and program disagree on $input" }
    }
```

The trouble is, we need an oracle, an often unreasonable assumption. This is known as the *test oracle problem*. But even if we had an oracle, since the space of inputs is often large, it can take a long time to find an output where they disagree. Since a single call to `program(i)` can be quite expensive in practice, this method can also be quite inefficient.

4.4. Property-based Testing

Property-based testing [Fink and Bishop, 1997] (PBT) attempts to mitigate the test oracle problem by using *properties*. It consists of two phases, searching and shrinking. Users specify a property over all outputs and the test fails if a counterexample can be found:

```
fun <I, O> gen(program: (I) -> O, property: (O) -> Boolean, rand: () -> I) =
    repeat(1000) {
        val randomInput: I = rand()

        assert(property(program(randomInput))) {
            val shrunken = shrink(randomInput, program, property)
            "Minimal input counterexample of property: $shrunken"
        }
    }
```

Roughly speaking, `shrink` attempts to minimize the counterexample.

```
tailrec fun <I, O> shrink(failure: I, program: (I) -> O, property: (O) -> Boolean): I =
    if (property(program(decrease(failure)))) failure // Property holds once again
    else shrink(decrease(failure), program, property) // Decrease until property holds
```

For example, given a `program: (Float) -> Any`, we might implement `decrease` like so:

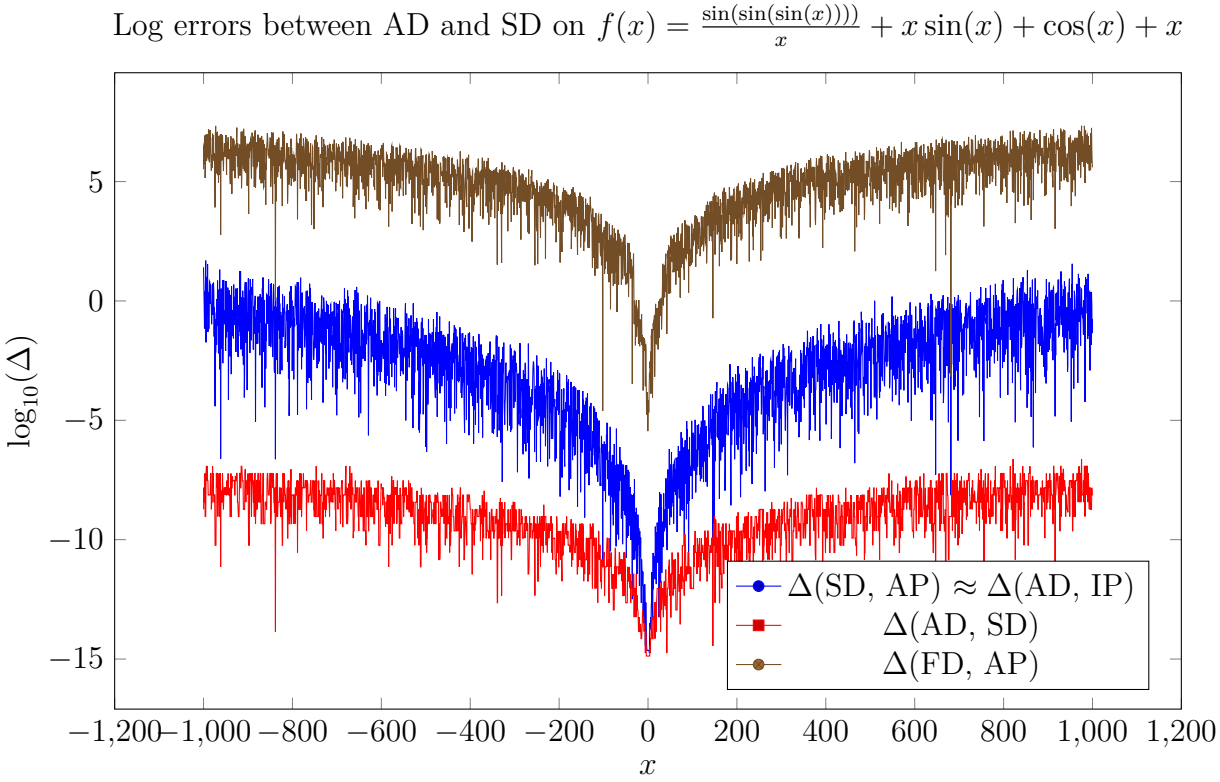


Fig. 4.1. We compare numerical drift between AD and SD over a swollen expression using fixed precision and arbitrary precision (AP). AD and SD both exhibit relative errors (i.e. with respect to each other) several orders of magnitude lower than their absolute error. These results are consistent with the findings of Laue [2019].

```
fun decrease(failure: Float): Float = failure - failure / 2
```

1

Consider Figure 4.3, which portrays the log difference between various forms of computational differentiation (evaluated using standard 32-bit precision) and AP (computed to 30 significant figures).[1] Given two algorithms for calculating the derivative, a property-based test might check whether the absolute difference is bounded over all inputs.

The trouble is, finding the right properties to test can be highly sensitive, and requires a lot of effort and domain-specific expertise. In addition, the user must specify a custom shrinker, which is unclear how to implement efficiently. What if there were a better way?

[1] To calculate AP, we symbolically derive the function and numerically evaluate it using finite difference approximation and MacLaurin series expansion of sine and cosine.

4.5. Metamorphic testing

It is often the case we would like to test the behavior of a program without providing an exhaustive specification. Many naturally-occurring generative processes exhibit a kind of local invariance – small changes to the input do not drastically change the output. We can exploit this property to design general-purpose fuzzing methods given a small set of inputs and outputs. Metamorphic testing (MT) is a property testing methodology which addresses the test oracle problem and the challenge of cheaply discovering bugs in the low-data regime. It has been successfully applied in testing driverless cars [Zhou and Sun, 2019, Pei et al., 2017, Tian et al., 2018] and other stateful deep learning systems [Du et al., 2018].

First, let us consider the following concrete example, from Tian et al. [2018]: suppose we have implemented a program which takes an image from a vehicle while driving, and predicts the simultaneous steering angle of the vehicle. Given a single image and the corresponding ground-truth steering angle from an oracle (e.g. a human driver or simulator), our program should preserve invariance under various image transformations, such as limited illumination changes, linear transformations or additive noise below a certain threshold. Intuitively, the steering angle should remain approximately constant, regardless of any single transformation or sequence of transformations applied to the original image which satisfy our chosen criteria. If not, this is a strong indication our program is not sufficiently robust and may not respond well to the sort of variability it may encounter in an operational setting.

Metamorphic testing can be expressed as follows: Given an oracle $\mathbf{P} : \mathcal{I} \to \mathcal{O}$, and a set of inputs $\mathbf{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(z)}\}$ and outputs $\mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$, a metamorphic relation (MR) is a relation $\mathcal{R} \subset \mathcal{I}^z \times \mathcal{O}^z$ where $z \geq 2$. In the simplest case, an MR is an equivalence relation $\mathcal{R}$, i.e.: $\langle \mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}' \rangle \in \mathcal{R} \Leftrightarrow \mathbf{x} \sim_{\mathcal{R}} \mathbf{x}' \Leftrightarrow \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}')$.

Suppose our MR is $\forall \varphi \in \mathcal{I} : ||\varphi|| \leq \varepsilon, \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}' = \mathbf{x} + \varphi) \approx \mathbf{y}$. Given a program $\widehat{\mathbf{P}}$ and a comparatively small set of inputs $\mathbf{X}$ and outputs $\mathbf{Y}$ from our oracle $\mathbf{P}$, the MR produces a set $\mathbf{X}', |\mathbf{X}| \ll |\mathbf{X}'|$ on which to test $\widehat{\mathbf{P}}$, without requiring corresponding outputs from $\mathbf{P}$. If we can show $\exists \mathbf{x}' \in \mathbf{X}' \mid \widehat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x})$, this implies at least one of the following:

(1) $\langle \mathbf{x}, \mathbf{P}(\mathbf{x}), \mathbf{x}', \mathbf{P}(\mathbf{x}') \rangle \notin \mathcal{R}$, i.e. our assumptions were invalid

(2) $\widehat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x}')$, i.e. the program under test is unsound

In either case, we have obtained useful information. If our assumptions were invalid, we can strengthen the invariant, $\mathcal{R}$, by removing the counterexample. Otherwise, we have detected an error and can adjust the program to ensure compliance – both are useful outcomes.

Consider the following example of an MT which uses an equivalence-based MR:

```
fun <I, O> mrTest(program: (I) -> O, mr: (I, O, I, O) -> Boolean, rand: () -> Pair<I, O>) =
    repeat(1000) {
        val (input: I, output: O) = rand()
        val tx: (I) -> I = genTX(program, mr, input, output)
        val txInput: I = tx(input)
        val txOutput: O = program(txInput)
        assert(mr(input, output, txInput, txOutput)) {
            "<$input, $output> not related to <$txInput, $txOutput> by $mr ($tx)"
        }
    }
```

The trouble is, generating valid transformations is a non-trivial exercise. We could try to generate random transformations until we find one which meets our criteria:

```
fun <I, O> genTX(program: (I) -> O, mr: (I, O, I, O) -> Boolean, i: I, o: O): (I) -> I {
    while (true) {
        val tx: (I) -> I = sampleRandomTX()
        val txInput: I = tx(i)
        val txOutput: O = program(txInput)
        if (mr(i, o, txInput, txOutput)) return tx
    }
}
```

But this would be very inefficient and depending on the type of input and output, is not guaranteed to terminate. We could handcraft a transformation, but this requires extensive domain knowledge. Instead, we should seek a more principled, computationally efficient and general purpose method of mutating an input in our dataset to discover invalid outputs.

4.6. Adversarial Testing

This leads us to adversarial testing. In the general case, we are given an input-output pair from an oracle and a program approximating the oracle, but not necessarily the oracle

itself. Our goal is to find a small change to the input of a function, which produces the largest change to its output, relative to the original output.

Imagine a function $\widehat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}$, each component $g_1, ..., g_m$ of which we seek to change by a fixed amount so as to produce the largest output value $\widehat{\mathbf{P}}(g'_1, ..., g'_m)$ directly. Suppose for each input parameter $g_1, \ldots, g_m$, we have one of three choices to make: either we can increase the value by c, decrease the value by c, or leave it unchanged. We are given no further information about $\widehat{\mathbf{P}}$. Consider the naïve solution, which tries every combination of variable perturbations and selects the input corresponding to the greatest output value:

Algorithm 1 Brute Force Adversary

procedure BfAdversary($\widehat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}$, $c : \mathbb{R}$, $g_1 : \mathbb{R}$, $g_2 : \mathbb{R}$, ..., $g_m : \mathbb{R}$): $\mathbb{R}^m$
  if $m = 1$ **then** ▷ Evaluate $\widehat{\mathbf{P}}$ and return the best variable perturbation
    return argmax$\{\widehat{\mathbf{P}}(g_1 + c), \widehat{\mathbf{P}}(g_1 - c), \widehat{\mathbf{P}}(g_1)\}$
  else ▷ Partially apply candidate perturbation and recurse
    return argmax$\{\widehat{\mathbf{P}}(g_1 + c) \circ \text{BfAdversary}(\widehat{\mathbf{P}}(g_1 + c), c, g_2, \ldots, g_m),$
      $\widehat{\mathbf{P}}(g_1 - c) \circ \text{BfAdversary}(\widehat{\mathbf{P}}(g_1 - c), c, g_2, \ldots, g_m),$
      $\widehat{\mathbf{P}}(g_1) \circ \text{BfAdversary}(\widehat{\mathbf{P}}(g_1), c, g_2, \ldots, g_m)\}$
  end if
end procedure

As we can see, algorithm Algorithm 1 is $\mathcal{O}(3^m)$ with respect to $\widehat{\mathbf{P}}$ – not a very efficient search routine, especially if we want to consider a larger set of perturbances. Clearly, if we want to find the best direction to update $\mathbf{g}$, we need to be more careful about how we perform the search.

Even if we cannot compute a closed-form derivative for $\widehat{\mathbf{P}}$, if $\widehat{\mathbf{P}}$ is differentiable almost everywhere, we can still use numerical differentiation to approximate pointwise values of its derivative. Consider algorithm Algorithm 2, a refinement of algorithm Algorithm 1 which uses the finite difference method to approximate the derivative with respect to each component of the input. This tells us how to minimally change the input to produce the largest output in reach, without needing to exhaustively check every perturbation.

Algorithm 2 Finite Difference Adversary

procedure FDADVERSARY($\widehat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}$, $c : \mathbb{R}$, $g_1 : \mathbb{R}$, $g_2 : \mathbb{R}$, ..., $g_m : \mathbb{R}$): $\mathbb{R}^m$
  if $m = 1$ **then** ▷ Compute finite (centered) difference and perform gradient ascent
    return $g_1 + \frac{\widehat{\mathbf{P}}(g_1 - c) - \widehat{\mathbf{P}}(g_1 + c)}{2c}$
  else ▷ Apply single-step gradient ascent using componentwise finite difference
    return $g_1 + \frac{\widehat{\mathbf{P}}(g_1 - c, 0, \ldots) - \widehat{\mathbf{P}}(g_1 + c, 0, \ldots)}{2c}$, FDADVERSARY($\widehat{\mathbf{P}}, c, g_1, \ldots, g_m$)
  end if
end procedure

We now have a procedure that is $\mathcal{O}(m)$ with respect to $\widehat{\mathbf{P}}$, but must be recomputed for each input – we can still do better by assuming further structure on $\widehat{\mathbf{P}}$. Furthermore, we have not yet incorporated any form of constraint on the input values. Perhaps we can combine the notion of metamorphic testing seen in § 4.5 with constrained optimization to accelerate the search for adversarial examples.

During backpropagation we perform gradient descent on a differentiable function with respect to its parameters for a specific set of inputs. In gradient-based adversarial testing, we perform gradient ascent on a loss function with respect to the inputs using a fixed parameter setting. Suppose we have a differentiable vector function $\widehat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}^n$, defined as follows:

$$\widehat{\mathbf{P}}_k(\mathbf{x}; \boldsymbol{\Theta}) = \begin{cases} \widehat{\mathbf{p}}_1(\Theta_1) \circ \mathbf{x} & \text{if } k = 1 \\ \widehat{\mathbf{p}}_k(\Theta_k) \circ \widehat{\mathbf{P}}_k(\boldsymbol{\Theta}_{[1,k]}) \circ \mathbf{x} & \text{if } k > 1 \end{cases} \qquad \text{(Equation 3.1.12 revisited)}$$

In deep learning, given pairs $\mathbf{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(z)}\}, \mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ we want to find $\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} \mathcal{L}\big(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)}\big)$ which is typically achieved by performing stochastic gradient descent on the loss with respect to the model parameters:

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{1}{z} \nabla_{\boldsymbol{\Theta}} \sum_{i=1}^{z} \mathcal{L}\big(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)}\big) \qquad \text{(Equation 3.1.13 revisited)}$$

We can solve for the gradient with respect to $\boldsymbol{\Theta}$ by multiplying the Jacobians (Equation 3.1.7), $\mathcal{J}_{\mathbf{p}_1} \cdots \mathcal{J}_{\mathbf{p}_k}$. In white box adversarial learning, we are given a fixed $\boldsymbol{\Theta}$ [2] and control the value of $\mathbf{x}$, so we can rewrite $\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta})$ instead as $\widehat{\mathbf{P}}(\mathbf{x})$, and take the gradient directly with respect to $\mathbf{x}$. Our objective is to find the "worst" $\mathbf{x}$ within a small distance of

[2] In contrast with backpropagation, where the parameters $\boldsymbol{\Theta}$ are updated.

any $\mathbf{x}^{(i)}$, i.e. where $\mathbf{P}(\mathbf{x})$ least resembles $\widehat{\mathbf{P}}(\mathbf{x})$. More concretely, this can be expressed as,

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \mathcal{L}\big(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big) \text{ subject to } CS = \{\mathbf{x} \in \mathbb{R}^m \text{ s.t. } ||\mathbf{x}^{(i)} - \mathbf{x}||_p < \epsilon\} \tag{4.6.1}$$

To do so, we can initialize $\mathbf{x} \sim U[CS]$ and perform projected gradient ascent on the loss:

$$\mathbf{x} \leftarrow \mathbf{\Phi}_{CS}\Big(\mathbf{x} + \alpha \nabla_{\mathbf{x}} \mathcal{L}\big(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big)\Big), \text{ where } \mathbf{\Phi}_{CS}(\phi') = \arg\min_{\phi \in CS} \frac{1}{2}||\phi - \phi'||_2^2 \tag{4.6.2}$$

Henceforth we shall refer to $\mathcal{L}\big(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big)$ as $\mathcal{L}(\mathbf{x})$. Imagine a single test $\mathbf{T} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{B}$:

$$\mathbf{T}(\mathbf{x}, C) = \mathcal{L}(\mathbf{x}) < C \tag{4.6.3}$$

Where $C \in \mathbb{R}$. How should we find a set of inputs that break our test given a fixed computational budget (i.e. constant number of program evaluations)? More concretely:

$$\{D_{\mathbf{T}} : \mathbf{x} \in CS \mid \widehat{\mathbf{P}}(\mathbf{x}) \implies \neg\mathbf{T}\}, maximize|D_{\mathbf{T}}| \tag{4.6.4}$$

Assuming zero knowledge about the program's implementation or the data distribution, $D_{\widehat{\mathbf{P}}}$, we can do no better than random search [Wolpert and Macready, 1997]. Given access to $\widehat{\mathbf{P}}$'s implementation, we could use classical fuzzing techniques to prioritize the search for inputs more likely to violate $\mathbf{T}$. Assuming the program is differentiable, given input-output access but not the source code, we can still use zeroth order optimization techniques to approximate the gradient. Given access to its implementation, we can accelerate the search by using automatic differentiation. Furthermore, given information about the data distribution, we could re-parameterize the distribution to initialize the search in promising regions of the input space. By assuming the program has previously been tested on common inputs, we might sample from the inverse training distribution $x \sim \frac{1}{D_{\widehat{\mathbf{P}}}}$ to select low probability inputs, possibly more likely to elicit an error.

4.7. Generative Adversarial Testing

What constitutes a good adversary? For an adversary to be considered a strong adversary, a significant fraction of her candidate inputs must break the program specification. To generate plausible test cases, not only must she be able to exploit weaknesses of the program, but ideally possess a good understanding of p_{data}.

Suppose we have a program $D : \mathbb{R}^h \to \mathbb{B}$, i.e. a binary classifier. How should we test its implementation, without providing an exhaustive specification, or some prior distribution

over the inputs? One solution, known as a generative adversarial network [Goodfellow et al., 2014] (GAN), proposes a generative adversary $G : \mathbb{R}^v \to \mathbb{R}^h$. The vanilla GAN objective can be expressed as a min-max optimization problem:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\big[\log D(\mathbf{x})\big] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\big[\log\big(1 - D(G(\mathbf{z}))\big)\big] \tag{4.7.1}$$

This objective can be optimized by sampling minibatches $\mathbf{x} \sim p_{data}(\mathbf{x})$ and $\mathbf{z} \sim p_G(\mathbf{z})$, then updating the parameters of G and D using their respective stochastic gradients:

$$\theta_D \leftarrow \theta_D + \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} \Big[\log D(\mathbf{x}^{(i)}) + \log\Big(1 - D\big(G(\mathbf{z}^{(i)})\big)\Big)\Big] \tag{4.7.2}$$

$$\theta_G \leftarrow \theta_G - \nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^{m} \log\Big(1 - D\big(G(\mathbf{z}^{(i)})\big)\Big) \tag{4.7.3}$$

Albuquerque et al. [2019] propose an augmented version of this game using multiple Discriminators which each receive a fixed, random projection $P_k(\cdot)$ of the Generator's output, and solves the following multi-objective optimization problem:

$$\min \mathcal{L}_G(\mathbf{x}) = [l_1(\mathbf{z}), l_2(\mathbf{z}), \ldots, l_K(\mathbf{z})]\,, \text{ where } l_k = -\mathbb{E}_{z\sim p_z} \log D_k(P_k(G(z_k))) \tag{4.7.4}$$

This can be achieved by using a variant of hypervolume maximization:

$$\nabla_\Theta \mathcal{L}_G = \sum_{k=1}^{K} \frac{1}{\eta - l_k} \nabla_\Theta l_k \tag{4.7.5}$$

Where η is a common, fixed upper bound on every l_k. Further GAN variants such as WGAN [Arjovsky et al., 2017], MHGAN [Turner et al., 2019], et al. have proposed augmentations to the vanilla GAN to improve stability and sample diversity. GANs have been successfully applied in various domains from speech [Donahue et al., 2019] to graph synthesis [Wang et al., 2018c]. One practical extension to the latter could be applying the GAN framework to program synthesis and compiler optimization by choosing a suitable metric and following the approach proposed by e.g. Adams et al. [2019], Mendis et al. [2019].

4.8. Probabilistic Adversarial Testing

Let us consider an extension of classical fuzzing methods to differentiable functions on continuous random variables. First, we select $\mathbf{x}_j : \mathbb{R}^m \sim \mathcal{S}_m$ (e.g. uniform random prior, or using a meta-learner $\mathbf{M} : (\mathbb{R}^m \to \mathbb{R}^n) \times (\mathbb{R}^m \times \mathbb{R} \to \mathbb{B}) \to \mathbb{R}^m$ as input). If $\widehat{\mathbf{P}}(\mathbf{x}^i)$ violates $\mathbf{T}$, we can append $\mathbf{x}^i$ to $D_{\mathbf{T}}$ and repeat. Otherwise, we update $\mathbf{x}$ following $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x})$ and repeat until test failure, gradient descent convergence, or a fixed number of steps C are reached before resampling $\mathbf{x}$ from $\mathcal{S}_m$. This procedure is described in Algorithm 3.

We hypothesize that if $\widehat{\mathbf{P}}$'s implementation were indeed flawed and a counterexample to Equation 4.6.3 existed, as sample size increased, a subset of gradient descent trajectories would fail to converge at all, a subset would converge to local minima, and the remaining trajectories would discover inputs violating the program specification.

Algorithm 3 Probabilistic Generator

procedure ProbAdversary($\mathcal{L} : \mathbb{R}^m \to \mathbb{R}$, $\mathcal{S}_m$, budget: $\mathbb{Z}^+$)
  $D_{\mathbf{T}} \leftarrow \{\}, j \leftarrow 0$
  while $j \leq$ budget **do** ▷ Iterate until count exceeds our budget
    $\mathbf{x}_j \sim \mathcal{S}_m$ ▷ Sample from $\mathcal{S}_m$
    if $\mathbf{T}(\mathbf{x}_j, \mathcal{L}(\mathbf{x}_j))$ **then** ▷ Inside feasible set, perform gradient ascent
      $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}} \cup$ DiffShrink$(-\mathcal{L}, \mathbf{x}_j, \mathbf{T})$
    else ▷ Outside feasible set, perform gradient descent
      $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}} \cup$ DiffShrink$(\mathcal{L}, \mathbf{x}_j, \mathbf{T})$
    end if
    $j \leftarrow j + 1$
  end while
  return $D_{\mathbf{T}}$
end procedure

To evaluate our algorithm, we train a polynomial regressor § A.3 on input-output pairs from a set of random algebraic expressions. These expressions are produced by generating perfect binary trees of depth 5, whose leaf nodes contain with equal probability either (1) an alphabetic variable or (2) a random 64-bit IEEE 754 floating point number uniformly sampled in the range $[-1, 1]$. The internal nodes contain with equal probability a random

Algorithm 4 Differential Shrinker

procedure DIFFSHRINK($\mathcal{L} : \mathbb{R}^m \to \mathbb{R}$, $\mathbf{x}_1 : \mathbb{R}^m$)
  $i \leftarrow 1$
  $t_1 \leftarrow \mathbf{T}\big(\mathbf{x}_1, \mathcal{L}(\mathbf{x}_1)\big)$ ▷ Store initial state to detect when test flips.
  while $i \leq I_{max}$ **and** $||\mathbf{x}_i - \mathbf{x}_{i-1}||_2^2 < \epsilon$ **do** ▷ While in budget and not converged.
    $\mathbf{x}_i \leftarrow \mathbf{\Phi}_{CS}\big(\mathbf{x}_{i-1} - \alpha \nabla_{\mathbf{x}_{i-1}} \mathcal{L}(\mathbf{x}_{i-1})\big)$ ▷ PGD step (Equation 4.6.2)
    if $\mathbf{T}\big(\mathbf{x}_i, \mathcal{L}(\mathbf{x}_i)\big) \neq t_1$ **then** ▷ Boundary value was found.
      return $\{\mathbf{x}_{i-1}\}$ ▷ Return previous iterate.
    end if
    $i \leftarrow i + 1$
  end while
  return if $\neg t_1$ **then** $\{\mathbf{x}_{i-1}\}$ **else** $\varnothing$ ▷ Return last iterate or $\varnothing$ if test passed.
end procedure

operator in the set $\{+, \times\}$. Our expression generator with type $G_e : \mathbb{N}^+ \times \mathbb{Z} \to \mathbb{R}^{[1,26]} \to \mathbb{R}$ takes a depth δ, a random seed ψ, and returns a scalar-valued function.

$$G_e(\delta, \psi) = \begin{cases} \delta \sim \{a,b,..z\} & \text{if } \delta \leq 0 \text{ and } \gamma \sim_\psi \{\text{True, False}\}, \\ \chi \sim U(-1, 1) & \text{if } \delta \leq 0 \text{ and } \gamma \sim_\psi \{\text{True, False}\}, \\ G(\delta - 1, \psi) + G(\delta - 1, \psi) & \text{if } \delta > 0 \text{ and } \gamma \sim_\psi \{\text{True, False}\}, \\ G(\delta - 1, \psi) \times G(\delta - 1, \psi) & \text{if } \delta > 0 \text{ and } \gamma \sim_\psi \{\text{True, False}\}. \end{cases} \tag{4.8.1}$$

```
val sum = { left: SFun<DReal>, right: SFun<DReal> -> left + right }
val mul = { left: SFun<DReal>, right: SFun<DReal> -> left * right }
val operators = listOf(sum, mul)
val variables = ('a'..'z').map { SVar<DReal>(it) }

infix fun SFun<DReal>.wildOp(that: SFun<DReal>) = operators.random(rand)(this, that)

fun randomBiTree(height: Int): SFun<DReal> =
  if (height == 0) (listOf(wrap(rand.nextDouble(-1.0, 1.0))) + variables).random(rand)
  else randomBiTree(height - 1) wildOp randomBiTree(height - 1)
```

Our training set consists of input-output pairs produced by binding the set of free variables to values belonging to the same datatype and numerical range as the constants, evaluating the expression on input values in the range $[-1, -0.2] \cup [0.2, 1]$ and rescaling all output values in the range $[-1, 1]$ using min-max normalization, i.e. $\tilde{G}_e(\delta, \psi) = \frac{G_e(\delta,\psi)}{\max |G_e(\delta,\psi)[-1,1]|}$. For each expression in our dataset, we train a polynomial regressor to convergence using vanilla SGD with momentum. Then, for both adversarial testing and uniform sampling strategies, we compare the average number of violations detected per evaluation.

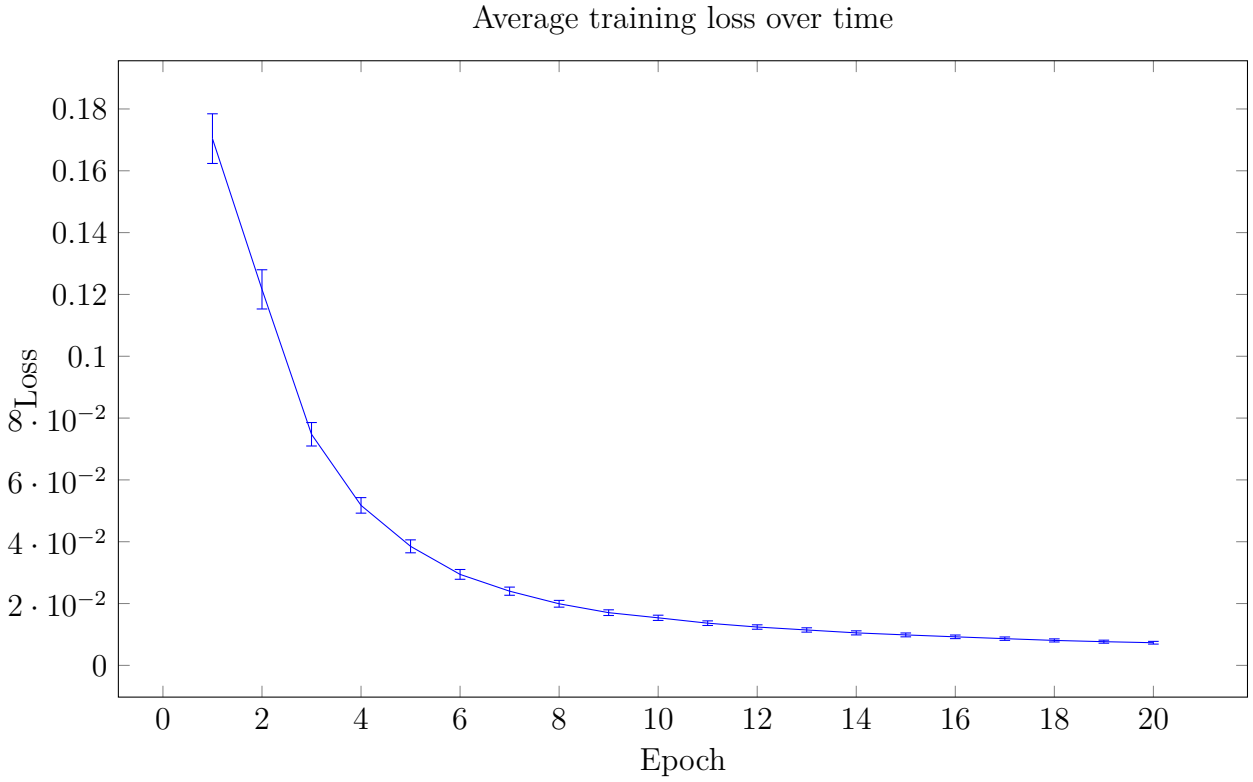


Fig. 4.2. Average loss curve of 100 trajectories of momentum SGD in parameter space.

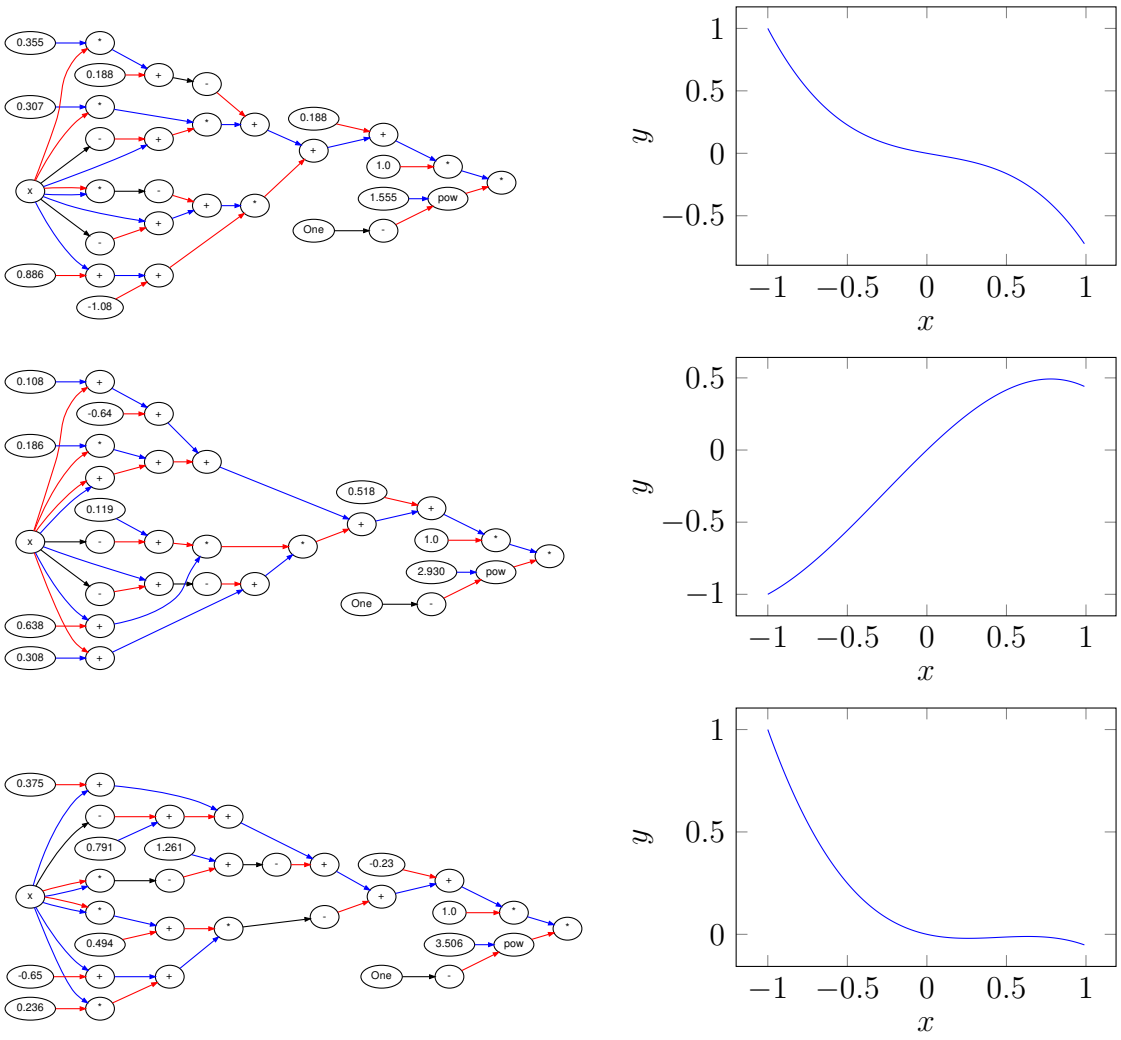


Table 4.1. Some DFGs generated by Equation 4.8.1 with accompanying 2D plots.

```
val model = Vec(D30) { x pow (it + 1) } dot weights
var update = Vec(D30) { 0.0 }

batches.forEach { i, batch ->
  val batchInputs = arrayOf(xBatchIn to batch.first, label to batch.second)
  val batchLoss = (model - label).magnitude()(*batchInputs)
  val weightGrads = (d(batchLoss) / d(weights))(*newWeights)
  update = beta * update + (1 - beta) * weightGrads
  newWeights = newWeights - alpha * update
}
```

Our adversary takes as input the trained regression model $\hat{f}$, a set of new input-output pairs from the ground truth expression, and resumes the original training procedure on $\hat{f}$ using the supplied datapoints for a fixed number of epochs, to produce a new model $\hat{f}'$. We use $\hat{f}'$ to construct a surrogate loss $\hat{\mathcal{L}}(x) = \left(\hat{f}(x) - \hat{f}'(x)\right)^2$, which can be maximized using Algorithm 4. Maximizing the surrogate loss allows us to construct adversarial examples without direct access to the oracle, an often impractical assumption in real world settings.

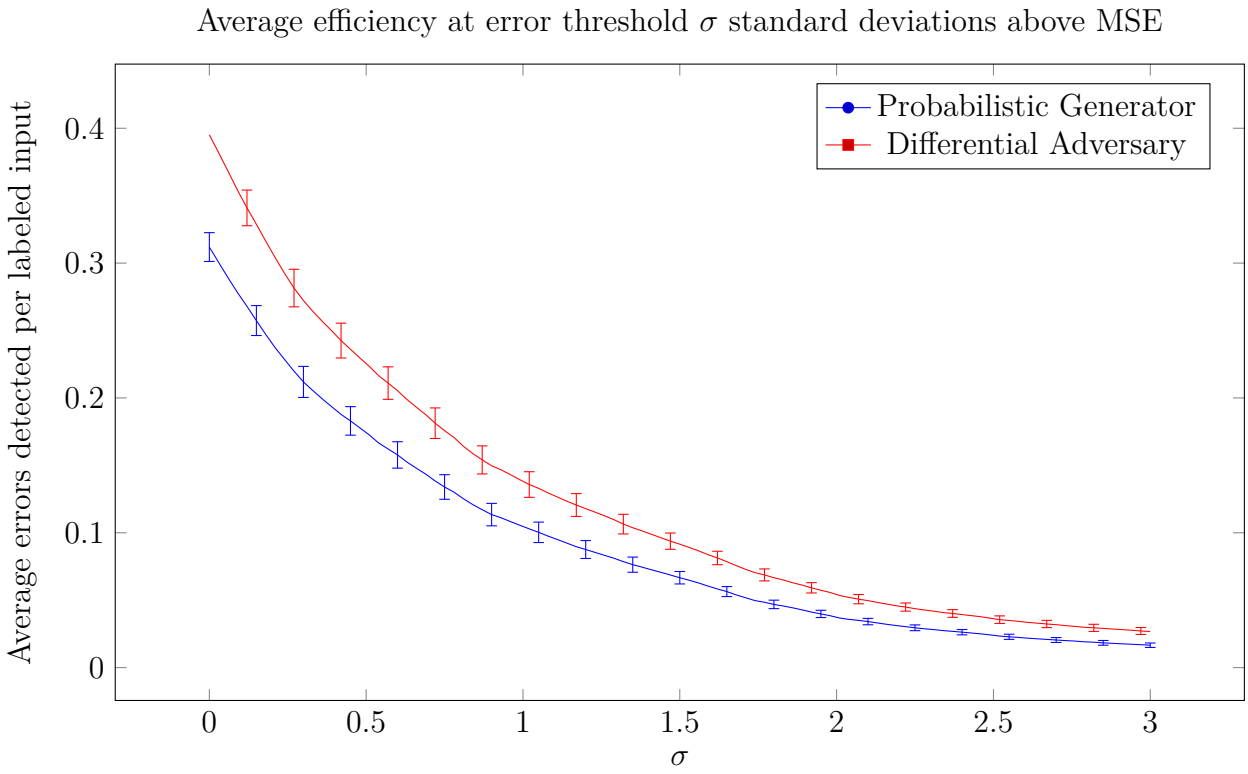


Fig. 4.3. By construction, our differential shrinker detects a greater number of errors per input-output pair than a naïve sampler which does not take the gradient into consideration.

Above, we show the number of violations exceeding error threshold σ standard deviations above the mean squared error (MSE) on the true loss $\mathcal{L}(x) = \left(f(x) - f'(x)\right)^2$. On average, our adversary exhibits a $10-15\%$ improvement over random search across all thresholds. We hypothesize that training the surrogate loss to convergence would further widen this margin, albeit potentially at the cost of generalization on other datasets.

Suppose a machine learning department has a budget B and fixed cost C for labeling a single datapoint. Given an error threshold e_{min} and access to $\frac{B}{C}$ new inputs from the oracle, our method detects on average 15% more errors than a uniform random sampling strategy

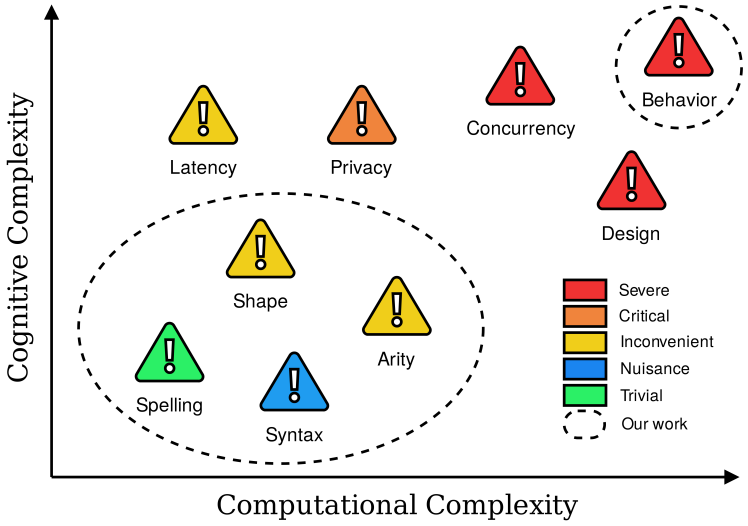


Fig. 4.4. Complexity of detecting various types of programming errors.

over the input space. Conversely, to detect the same number of errors, we require a 15% lower budget than a random sampling strategy.

4.9. Conclusion

In this chapter we have visited some interesting ideas for validating intelligent systems from the perspective of software engineering and machine learning. We have seen a curious resemblance between some new and old ideas in adversarial learning and fuzz testing. We have proposed a framework for evaluating differentiable programs in a low-cost, data-driven manner and suggest some intriguing directions for future research.

Type systems, compilers and fuzzers are all part of a broader class of validation and verification tools. The goal of these tools is to provide feedback as early as possible. Some errors (e.g. syntactical errors), are minor nuisances and can be quickly detected with a good incremental parser (§ 2.3.2). Others, as shown in Figure 4.4, are more complex but can be detected by spending computation. We argue this computational cost is often justified as undetected bugs can have catastrophic consequences. Spending computation frees up valuable cognitive resources better spent on more challenging tasks downstream. Studies have shown bugs detected early in development are more likely to be fixed [Distefano et al., 2019] – saving minutes in development could save lives during operation.

Learning capabilities will play a crucial role in autonomous systems. As today's engineers begin to incorporate learning in tomorrow's safety-critical robotic systems, we believe the increased assurance provided by intelligent validation and verification tools will be indispensable for the widespread deployment of these complex adaptive cyberphysical systems.

Chapter 5

Tools for reproducible robotics

> "An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures."
>
> –Buckheit and Donoho [1995], *WaveLab and Reproducible Research*

In this chapter, we discuss the challenge of software reproducibility and how best practices in software engineering such as continuous integration and containerization tools can help researchers mitigate the variability associated with building and maintaining robotics software. Broadly, our work attempts to isolate sources of computational variability, and does not consider notions of statistical variability arising from aleatoric or epistemic uncertainty [Diaz Cabrera, 2018]. However, minimizing the computational variability (which often impedes experimental reproducibility) is a key step in enabling researchers to more rapidly identify and diagnose these more elusive variables in robotics and machine learning.

In order to address the issue of software reproducibility, we assembled a set of tools and development workflows representing best practices in software engineering. These tools are primarily based on containerization, a widely adopted virtualization technology in the software industry. To lower the barrier of entry for new contributors and minimize variability across hardware platforms, we developed a state-of-the-art container infrastructure based on Docker [Merkel, 2014], one popular container engine. Docker allows users to set up versioned deployment artifacts which effectively freeze an entire filesystem, and manage resource constraints via a sandboxed runtime environment.

The contents of this chapter are organized as follows. In § 5.1 we introduce the problem of dependency resolution and the challenge of building reproducible software artifacts. In

§ 5.2, we describe a broad solution to this problem, software virtualization. Next, in § 5.3, we discuss a lightweight approach to virtualization, known as containerization. In § 5.4, we take a guided tour through one container implementation, called Docker. Finally, in § 5.5, we present DuckieOS, a Dockerized environment for building reproducible robotics applications for research and pedagogical use.

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 given set of software packages which must be installed, and dependencies required to install them, determining the most recent consistent version of the dependencies is as hard as the hardest problems in NP. Informally, this problem is known as *dependency hell* and becomes increasingly problematic as software projects grow and introduce new dependencies.

Dependency hell does not just arise 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 many 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 partially removed it becomes difficult to keep track of changes and their side 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 should the need arise. 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. Hardware backups would do the job, but are cumbersome to manage and are

unsuitable for development purposes. Rather, it would be convenient to have a tool which allowed applications to create 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 primitive of these are emulators, which completely simulate a foreign processor architecture, and thereby any software which runs ontop of it. Another solution are virtual machines (VMs), a kind of isolated run-time environment which use a *hypervisor* to mediate access to hardware, but usually run on bare metal. The downside of both methods is their efficiency. Virtual machines contain full-fledged operating systems and are therefor cumbersome to run and debug. This is particularly unnecessary for building and running a small application on a foreign OS. Emulators run significantly more slowly than native machine code depending on the host and target architectures.

In 2006, Linux introduced several new kernel features for controlling groups of processes, under the aegis of **cgroups** [Menage, 2007]. Collectively, these features support a form of lightweight virtualization, featuring many of the benefits of virtual machines (VMs) such as resource control and namespace isolation, without the computational overhead associated with full virtualization. These features 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 due to their light resource footprint.

5.3. Containerization

One of the challenges of distributed software development across heterogeneous platforms is the problem of variability. With today's increasing pace of software development comes the added burden of software maintenance. As hardware and software stacks evolve, source code must periodically be updated to build and run correctly. Maintaining a stable and well-documented codebase can be a considerable challenge, especially in an academic setting

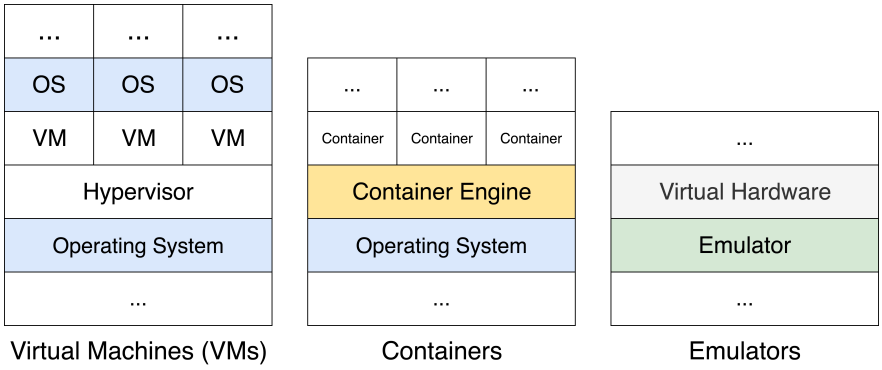


Fig. 5.1. Full virtualization is a very resource-hungry process. Containerization is cheaper, as it shares a kernel with the host OS. Emulation lets us emulate hardware as software. Any of these methods can be used in conjunction with any other.

where contributors are frequently joining and leaving a project. Together, these challenges present significant obstacles to experimental reproducibility and scientific collaboration.

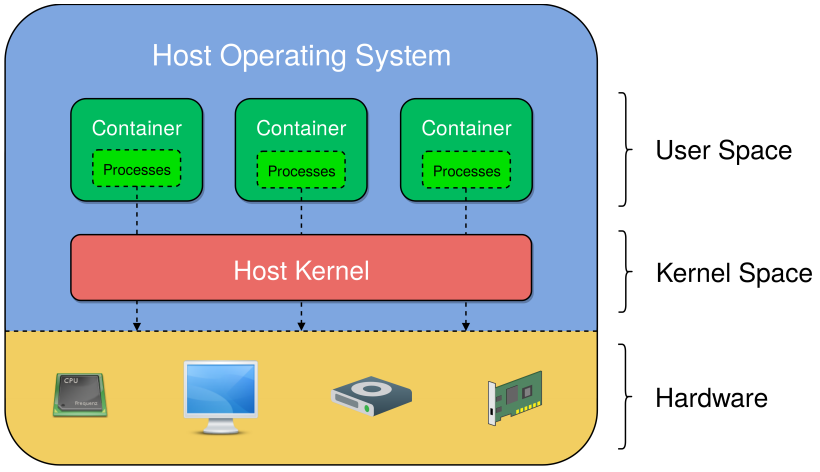


Fig. 5.2. Containers live in user space. By default they are sandboxed from the host OS and sibling containers, but unlike VMs, share a common kernel with each other and the host OS. All system calls are passed through host kernel.

Docker containers are sandboxed runtime environments that are portable, reproducible and version-controlled. Each environment fully contains its dependencies, 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 provisions a separate Linux namespace for each container, effectively isolating the network, users, and file system mounts from the host OS. Unlike virtual machines, container-based virtualization tools like Docker are suitable for

portable SBCs and can run with close to zero overhead compared to native Linux processes. A single Raspberry Pi is capable of simultaneously running hundreds of containers with no noticeable degradation in performance.[1]

While containerization considerably simplifies the process of building and deploying applications, it also introduces some additional complexity in 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 are typically constructed as a sequence of layers, each of which must periodically be updated. Care is required when designing the development pipeline to ensure that such updates do not silently break a subsequent layer, as we describe in § 5.7.

5.4. Introduction to Docker

Suppose there is a program which is known to run on some computer. It would be nice to give another computer – any computer with an internet connection – a short string of ASCII characters, press ⏎, and return to see that same program running. Never mind where the program was built or what software happened to be running at the time. This may seem trivial, but is a monumental software engineering problem. Various package managers have attempted to solve this, but even when they work as intended, only support natively compiled binaries on operating systems with the same package manager.

Docker[2] is a tool for portable, reproducible computing. With Docker, users can run any Linux program on almost any networked computing device on the planet, regardless of the underlying operating system or hardware architecture. All of the environment preparation, installation and configuration steps can be automated from start to finish. Depending on how much network bandwidth is available, it might take some time, but users will never need to intervene in the installation process.

To install Docker itself, execute the following command on a POSIX-compliant shell of any Docker-supported platform:

```
~$ curl -sSL https://get.docker.com/ | sh
```

1

[1] https://blog.docker.com/2015/09/update-raspberry-pi-dockercon-challenge/

[2] The following tutorial uses Docker, but the workflow described is similar to most container platforms.

A Docker *image* is basically a filesystem snapshot – a single file that contains everything needed to run a certain Docker container. Docker images are hosted in *registries*, similar to Git repositories or VCS servers. The following command will fetch a Docker image, e.g. `daphne/duck` from the default Docker Hub repository:

```
~$ docker pull daphne/duck
```

Every Docker image has an image ID, a name and a tag:

```
~$ docker images
REPOSITORY      TAG         IMAGE ID          CREATED        SIZE
daphne/duck     latest      ea2f90g8de9e      1 day ago      869MB
```

To run a Docker container[3], use the following command:

```
~$ docker run daphne/duck
```

The following command will verify the container is indeed running:

```
~$ docker ps
CONTAINER ID     IMAGE            ...     NAMES
52994ef22481     daphne/duck      ...     happy_hamster
```

Note how Daphne's duck container has an alphanumeric container ID, a base image, and a memorable name, `happy_hamster`. This name is an alias for the container ID, which can be used interchangeably to refer to the container.

Docker images can be created two different ways. First, in § 5.4.1, we will see how to create a Docker image by taking a snapshot from a running container, then in § 5.4.2, how to create a new Docker container using a special kind of recipe, called a `Dockerfile`.

5.4.1. Creating an image snapshot

When a Docker container writes to its own filesystem, those changes are not persisted unless committed to a new image. For example, start a container with an interactive shell:

[3]When a Docker image is running, it is referred to as a *container*.

```
~$ docker run -it daphne/duck /bin/bash
root@295fd7879184:/#
```

Note the container ID: `295fd7879184`. If we write to disk and leave the container,

```
root@295fd7879184:/# touch new_file && ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
root@295fd7879184:/# exit
```

`new_file` will not be persisted. If we re-run the same command again:

```
~$ docker run -it daphne/duck /bin/bash
root@18f13bb4571a:/# ls
root@18f13bb4571a:/# touch new_file1 && ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

It seems like `new_file` has disappeared! Notice how the container ID (`18f13bb4571a`) is now different. This is because the command `docker run daphne/duck` created a new container from the base image `daphne/duck`, rather than restarting the previous container. To see all containers on a Docker host, run the following command:

```
~$ docker container ls -a
CONTAINER ID      IMAGE            STATUS             NAMES
295fd7879184      daphne/duck      Exited (130)       merry_manatee
18f13bb4571a      daphne/duck      Up 5 minutes       shady_giraffe
52994ef22481      daphne/duck      Up 10 minutes      happy_hamster
```

It appears `295fd7879184` a.k.a. `merry_manatee` survived, but it is no longer running. Whenever a container's main process (recall we ran `merry_manatee` with `/bin/bash`) finishes, the container will stop, but it will not cease to exist. In fact, we can resume the stopped container right where it left off:

```
~$ docker start -a merry_manatee
root@295fd7879184:/# ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
```

Nothing was lost! To verify this, we can check which other containers are running:

```
~$ docker ps
CONTAINER ID       IMAGE              ...       NAMES
295fd7879184       daphne/duck        ...       merry_manatee
18f13bb4571a       daphne/duck        ...       shady_giraffe
52994ef22481       daphne/duck        ...       happy_hamster
```

Now suppose we would like to share the container `shady_giraffe` with someone else. To do so, we must first snapshot the running container, or commit it to a new image with a name and a tag. This will create a checkpoint that we may later restore:

```
~$ docker commit -m "forking daphne/duck" shady_giraffe user/duck:v2
```

To refer to the container, we can either use `18f13bb4571a` or the designated name (i.e. `shady_giraffe`). This image repository will be called `user/duck`, and has an optional version identifier, `:v2`, which can be pushed to the Docker Hub registry:

```
~$ docker push user/duck:v2
~$ docker images
REPOSITORY    TAG        IMAGE ID         CREATED          SIZE
daphne/duck   latest     ea2f90g8de9e     1 day ago        869MB
user/duck     v2         d78be5cf073e     2 seconds ago    869MB
~$ docker pull user/duck:v2
~$ docker run user/duck ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

This is a convenient way to share an image with colleagues and collaborators. Anyone with access to the repository can pull this image and continue where we left off, or create another image based on top.

5.4.2. Writing an image recipe

The second way to create a Docker image is to write a recipe, called a `Dockerfile`. A `Dockerfile` is a text file that specifies the commands required to create a Docker image, typically by modifying an existing container image using a scripting interface. They also have special keywords (which are conventionally `CAPITALIZED`), like `FROM`, `RUN`, `ENTRYPOINT` and so on. For example, create a file called `Dockerfile` with the following content:

```
FROM daphne/duck      # Defines the base image
RUN touch new_file1   # new_file1 will be part of our snapshot
CMD ls -l             # Default command run when container is started
```

Now, to build the image, we can simply run:

```
~$ docker build -t user/duck:v3 .
```

The `.` indicates the current directory, which should be the same one containing our `Dockerfile`. This command should produce something like the following output:

```
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM daphne/duck
--- ea2f90g8de9e
Step 2/3 : RUN touch new_file1
--- e3b75gt9zyc4
Step 3/3 : CMD ls -l
--- Running in 14f834yud59
Removing intermediate container 14f834yud59
--- 05a3bd381fc2
Successfully built 05a3bd381fc2
Successfully tagged user/duck:v3
```

The command, `docker images` should display an image called `user/duck:v3`:

```
~$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
daphne/duck   latest    ea2f90g8de9e   1 day ago        869MB
user/duck     v2        d78be5cf073e   5 minutes ago    869MB
user/duck     v3        05a3bd381fc2   2 seconds ago    869MB
```

This procedure is identical to the snapshot technique performed in § 5.4.1, but the result is cleaner. Rather than maintaining a 869 MB image, we can just store the 4 KB text file instead. To run the resulting image, we can simply use the same command as before:

```
~$ docker run -it user/duck:v3
total 0
-rw-r--r-- 1 root root 0 May 21 21:35 new_file1
```

Notice that as soon as we run the container, Docker will execute the `ls -l` command as specified by the `Dockerfile`, revealing that `new_file1` was indeed stored in the image. However, this default command can be overridden by supplying a custom command:

```
~$ docker run -it user/duck:v3 [custom_command]
```

5.4.3. Layer Caching

Layers are an important concept to understand when working with Docker. One way to think of a layer is like a Git commit – a set of changes to a previous image or layer, uniquely identified by a hash code. In a `Dockerfile`, layers begin with a keyword.

```
FROM daphne/duck
RUN touch new_file1                          # Defines a new layer
RUN mkdir config && mv new_file1 config      # Layers can chain commands
RUN apt-get update && apt-get install -y wget  # Install a dependency
RUN wget https://get.your.app/install.sh     # Download a script
RUN chmod +x install.sh && ./install.sh      # Run the script
```

To build this image, we can run the following command:

```
~$ docker build -t user/duck:v4 .
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM daphne/duck
---> cd6d8154f1e1
...
Removing intermediate container 8fb56ef38bc8
---> 3358ca1b8649
Step 5/6 : RUN wget https://get.your.app/install.sh
---> Running in e8284ff4ec8b
...
2018-10-30 06:47:57 (89.9 MB/s) - 'install.sh' saved [13847/13847]
Removing intermediate container e8284ff4ec8b
---> 24a22dc2900a
Step 6/6 : RUN chmod +x install.sh && ./install.sh
---> Running in 9526651fa492
# Executing install script, commit: 36b78b2
...
Removing intermediate container 9526651fa492
---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged user/duck:v4
```

Layers are conveniently cached by the Docker daemon. Should we need to run the same command twice, Docker will use the cache instead of rebuilding the entire image:

```
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM daphne/duck
---> cd6d8154f1e1
Step 2/6 : RUN touch new_file1
---> Using cache
---> 0473154b2004
...
Step 6/6 : RUN chmod +x index.html && ./index.html
---> Using cache
```

```
---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged user/duck:v4
```

If we need to make a change to the `Dockerfile`, Docker will only rebuild the image starting from the first modified step. Suppose we were to add a new `RUN` command at the end of our `Dockerfile` and trigger a rebuild like so:

```
~$ echo 'RUN echo "Change here!"' >> Dockerfile
~$ docker build -t user/duck:v4 .
Sending build context to Docker daemon  2.048kB
...
Step 6/7 : RUN chmod +x index.html && ./index.html
---> Using cache
---> a8be23fea573
Step 7/7 : RUN echo "Change here!"
---> Running in 80fc5c402304
Change here!
Removing intermediate container 80fc5c402304
---> c1ec64cef9c6
Successfully built c1ec64cef9c6
Successfully tagged user/duck:v4
```

If Docker had to rerun the entire `Dockerfile` from top to bottom each time it was rebuilt, this would be slow and inconvenient. Instead, Docker caches the unmodified steps by default, and only reruns the minimum set of steps necessary to rebuild. This can sometimes introduce unexpected results, especially when the cache is stale. To ignore the cache and force a clean rebuild, use the `--no-cache` flag when building a `Dockerfile`.

What does Docker consider when deciding whether to use the cache? First is the `Dockerfile` itself – when a step in a `Dockerfile` changes, both it and any subsequent steps will need to be rerun during a build. Docker also checks the build context for changes. When `docker build -t TAG .` is written, the `.` indicates the build context, or path where the build should occur. Often, this path contains build artifacts. For example:

```
FROM daphne/duck
COPY duck.txt .
RUN cat duck.txt
```

Now if we add some message in `duck.txt` and rebuild our image, the file will be copied into the Docker image, and its contents will be printed:

```
~$ echo "Make way!" > duck.txt && docker build -t user/duck:v5 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM daphne/duck
---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
---> e0e03d9e1791
Step 3/3 : RUN cat duck.txt
---> Running in 590c5420ce29
Make way!
Removing intermediate container 590c5420ce29
---> 1633e3e10bef
Successfully built 1633e3e10bef
Successfully tagged user/duck:v5
```

As long as the first three lines of the `Dockerfile` and `duck.txt` are unmodified, these layers will be cached and Docker will not rebuild them. If the contents of the file `duck.txt` are subsequently modified, this will trigger a rebuild to occur. For example, if we append to the file and rebuild, only the last two steps will need to be executed:

```
~$ echo "Thank you. Have a nice day!" >> duck.txt
~$ docker build -t user/duck:v5 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM ubuntu
---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
---> f219efc150a5
Step 3/3 : RUN cat duck.txt
```

```
---> Running in 7c6f5f8b73e9
Make way!
Thank you. Have a nice day!
Removing intermediate container 7c6f5f8b73e9
---> e8a1db712aee
Successfully built e8a1db712aee
Successfully tagged user/duck:v5
```

A common mistake when writing `Dockerfile`s is to `COPY` more files than are strictly necessary to perform the following build step. For example, if `COPY . .` is written at the beginning of the `Dockerfile`, whenever any file is changed within the build context, this will trigger a rebuild of all subsequent build steps. In order to maximize cache reusability and minimize rebuild time, users should be as conservative as possible and only `COPY` the minimum set of files necessary to accomplish the following build step.

5.4.4. Volume Sharing

There is a second method of depositing data into a container, which does not require baking it into the parent image at compile-time. This method is more appropriate for data which is required at runtime, but non-essential for the build. It takes the following form:

```
~$ docker run user/duck:v6 -v HOST_PATH:TARGET_PATH
```

Suppose we have a `Dockerfile` which provides a default `CMD` instruction:

```
FROM daphne/duck
CMD /bin/bash -c "/launch.sh"
```

If we built this image and tried to run it, the file `launch.sh` would be missing:

```
~$ docker build -t user/duck:v6 && docker run user/duck:v6
bash: /launch.sh: No such file or directory
```

Instead, when running the container, we need to share the file via the Docker CLI:

```
~$ echo -e '#!/bin/bash\necho Launching...' >> launch.sh && \
   chmod 775 launch.sh && \
   docker run user/duck:v6 -v launch.sh:/launch.sh
Launching...
```

This way, the local file `launch.sh` will be available to use from within the container at the designated path, `/launch.sh`.

5.4.5. Multi-stage builds

Docker's filesystem is additive, so each layer will only increase the size of the final image. For this reason, it is often necessary to tidy up unneeded files after installation. For example, when installing dependencies on Debian-based images, it is a common practice to run:

```
RUN apt-get update && apt-get install ... && rm -rf /var/lib/apt/lists/*
```

This ensures the package list is not baked into the image (Docker will only checkpoint the layer after each step is complete). Builds can often consume several steps, despite only producing a single artifact. Instead of chaining together several commands and cleaning up changes in a single step, multi-stage builds let us build a series of images inside a `Dockerfile`, and copy resources from one to another, discarding all intermediate build artifacts:

```
FROM user/duck:v3 as template1

FROM daphne/duck as template2
COPY --from=template1 new_file1 new_file2
FROM donald/duck as template3
COPY --from=template2 new_file2 new_file3
CMD ls -l
```

Now we can build and run this image as follows:

```
~$ docker build . -t user/duck:v4
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM user/duck:v3 as template1
--- e3b75ef8ecc4
```

```
Step 2/6 : FROM daphne/duck as template2
--- ea2f90g8de9e
Step 3/6 : COPY --from=template1 new_file1 new_file2
---> 72b96668378e
Step 4/6 : FROM donald/duck:v3 as template3
---> e3b75ef8ecc4
Step 5/6 : COPY --from=template2 new_file2 new_file3
---> cb1b84277228
Step 6/6 : CMD ls
---> Running in cb1b84277228
Removing intermediate container cb1b84277228
---> c7dc5dd63e77
Successfully built c7dc5dd63e77
Successfully tagged user/duck:v4
~$ docker run -it user/duck:v4
total 0
-rw-r--r-- 1 root root 0 Jul  8 15:06 new_file3
```

One application of multi-stage builds is compiling a project dependency from its source code. In addition to all the source code, the compilation process could introduce gigabytes of build artifacts and transitive dependencies, just to build a single binary. Multi-stage builds allow us to build the file, and copy it to a fresh layer, unburdened by intermediate files.

5.5. ROS and Docker

White and Christensen [2017] previously explored Dockerizing ROS, whose work forms the basis for our own, which extends their implementation to the Duckietown platform [Paull et al., 2017], a more hardware- and domain-specific set of ROS applications.

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 either target architecture.

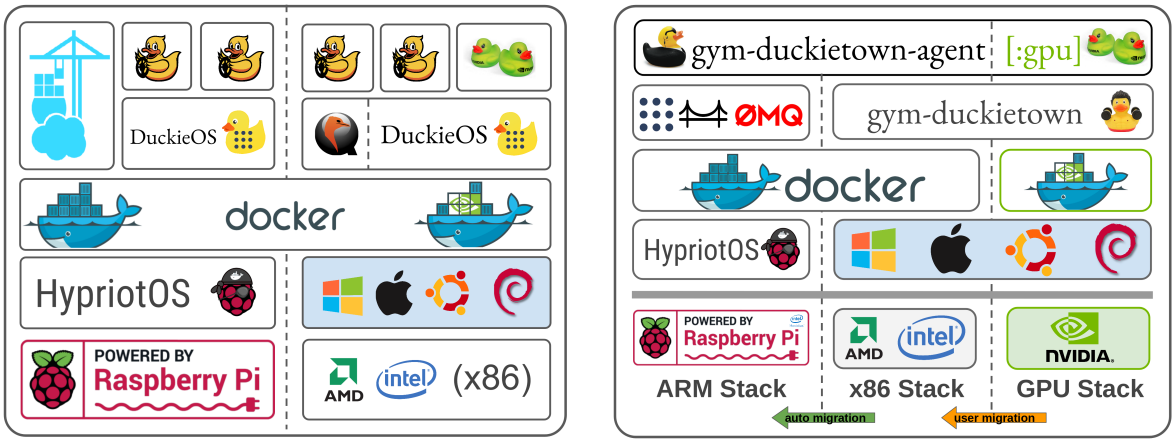


Fig. 5.3. Container infrastructure. **Left**: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we build ARM artifacts on x86 using QEMU [Bellard, 2005]. **Right**: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models may be also be run on an ARM device using an accelerator.

Runtime emulation of foreign artifacts is also possible, using a similar technique.[4] For performance and simplicity, we only use emulation where necessary (e.g., on x86 devices). On ARM-native, the base operating system is HypriotOS, a lightweight Debian distribution for the Raspberry Pi and other ARM-based SBCs, with native 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. Since both ROS and Docker have extensive command line interfaces, a unified interface, the Duckietown Shell (`dts`), is provided to wrap their functionality and perform common tasks.

5.6. Duckiebot Development using Docker

Software development for the Duckietown platform requires the following physical objects:

(1) Duckiebot (including custom hat, camera, wheels, and Raspberry Pi 3B+)[5]

(2) Micro SD card (16GB+ recommended)

(3) Personal computer

[4]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]Full materials list can be located at the following URL: https://get.duckietown.org/

(4) Internet-enabled router

(5) MicroSD card adapter

In addition, we assume the following software dependencies have been installed on (3):

(*) Docker CE

(*) POSIX-compliant shell

(*) `dts`, the Duckietown shell[6]

(*) Web browser (e.g. Chrome or Firefox)

(*) `wget`/`curl`

The following workflow has been tested extensively on Linux hosts running Ubuntu 16.04 (and to a lesser extent, Mac OS X and VMs). No other dependencies are assumed or required.

5.6.1. Flashing a bootable disk

One of the first steps in the Duckietown manual requires users to manually install a custom operating system onto bootable media, a tedious and time-consuming process. The following installation script was written to automate this process, allowing users to setup a reproducible software environment more easily:

```
~$ bash -c "$(wget -O- h.ndan.co)"
```

1

Now, with the Duckietown Shell, the following command is all that is needed:

```
dt> init_sd_card [--hostname "DUCKIEBOT_NAME"] [--wifi "username:password"]
```

Users must insert an SD card and follow the instructions provided. When complete, the card is removed and inserted into the SD card slot on the Raspberry Pi. On first boot, care must be taken to ensure the device is powered continuously for a minimum of ten minutes in order to allow installation to complete and avoid filesystem corruption.

5.6.2. Web interface

To access the DuckieOS web interface, users can visit the following URL in any JavaScript-enabled web browser: http://DUCKIEBOT_NAME:9000/. If the installation process successfully completed and the network is properly configured, the web application displayed

[6] May be obtained at the following URL: https://github.com/duckietown/duckietown-shell

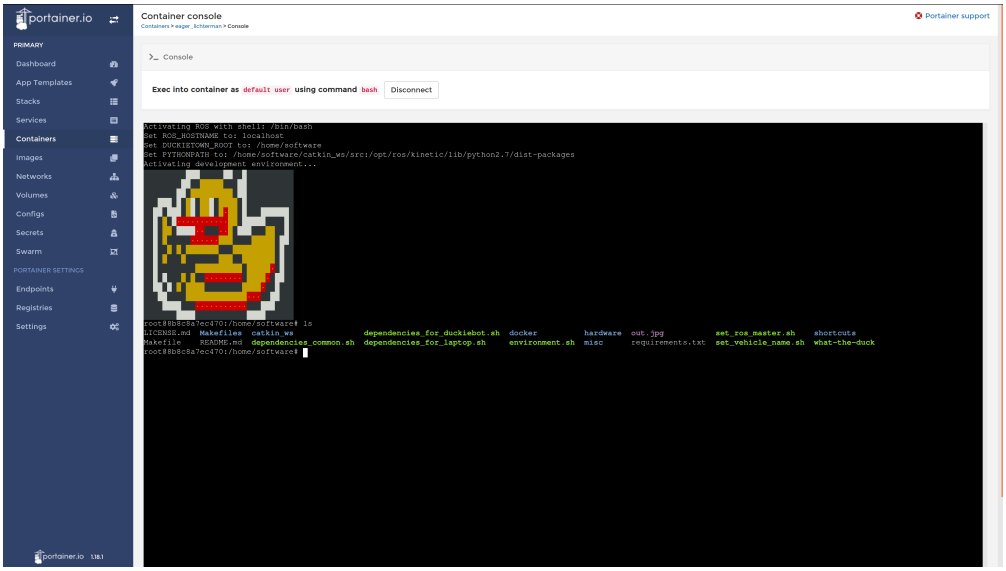


Fig. 5.4. Browser interface for individual Duckiebots. It is provided by Portainer, a RESTful web dashboard, which wraps the Docker CLI and offers support for container management, configuration, networking and terminal emulation (shown above). http://DUCKIEBOT_NAME:9000/#/container/container_name "Console"

in Figure 5.4 should be accessible. This application allows users unfamiliar with the CLI to manage containers on their Duckiebots from within the browser.

5.6.3. Testing ROS

To verify Docker is working properly, launch a remote container, interactively, like so:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host \
   duckietown/rpi-ros-kinetic-base:master18
```

The `-H` flag indicates a remote Docker host on the local area network where the Docker command should be executed. For the `DUCKIEBOT_NAME` address to work, mDNS must be properly configured in the network settings, otherwise an IP address is required.

5.6.4. Build and deployment

Docker images can be cross-compiled by enclosing the ARM-specific portion of the `Dockerfile` with the `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` instructions. The following command can be used for deployment:

```
~$ docker save TAG_NAME | ssh -C duckie@DUCKIEBOT_NAME docker load
```

Alternately, it is possible to build directly on ARM devices by creating a file named `Dockerfile.arm`, adding a base image and build instructions, then running the command:

```
~$ docker build --file=FILE_PATH/Dockerfile.arm [--tag TAG_NAME] .
```

5.6.5. Multi-architecture support

As of Docker version 18.09.6, ARM-specific `Dockerfile`s will not build on x86 machines. [7], and attempting to build one will produce the following error when running `docker build`:

```
standard_init_linux.go:175: exec user process caused "exec format error"
```

In order to circumvent this restriction, ARM-specific `Dockerfile`s can be ported to run on x86 by using the `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` directives, after the `FROM` and before the `CMD` instructions. See § B.1.1 for further details.

All Duckietown Docker images ship with the QEMU [Bellard, 2005] emulator – this allows us to run ARM images on x86 directly. To run a pure compute ROS node (i.e. one that does not require any camera or motor access) on an x86 platform, developers must supply a custom entrypoint to Docker when running the image using the entrypoint flag as follows:

```
~$ docker run ... --entrypoint=qemu3-arm-static IMAGE [RUN_COMMAND]
```

Here, `RUN_COMMAND` may be a shell such as `/bin/bash` or another command such as `/bin/bash -c "roscore"`. The entrypoint refers to the ARM emulator packaged within the base image, `duckietown/rpi-ros-kinetic-base`, which allows ARM binaries to be run on x86 hosts.

5.6.6. Running a simple HTTP file server

All persistent data is stored in `/data`. To serve this directory, a web server is provided:

[7] With the exception of the Mac OS Docker client, which offers multi-architecture support. Further details on multiarch support can be found here: https://docs.docker.com/docker-for-mac/multi-arch/ More recent versions of Docker Desktop for Mac OS and Windows have introduced native ARM emulation: https://engineering.docker.com/2019/04/multi-arch-images/

```
~$ docker -H DUCKIEBOT_NAME run -d -v /data:/data -p 8082:8082 \
   duckietown/rpi-simple-server:master18
```

To then access this directory, visit the following URL: http://DUCKIEBOT_NAME:8082/

5.6.7. Camera testing

The following command can be used to test the camera is working properly. By default, images will be hosted at: http://DUCKIEBOT_NAME:8081/figures/image.jpg

```
~$ docker -H DUCKIEBOT_NAME run -d --privileged -v /data:/data -p 8081:8081
duckietown/rpi-docker-python-picamera:master18
```

Like most commands, a Python-based shell is provided for the user's convenience:

```
dt> duckiebot demo --demo_name camera --duckiebot_name DUCKIEBOT_NAME
```

5.6.8. Graphical User Interface tools

To use GUI tools, one must first allow incoming X connections from the host. On Linux hosts, this can be done by running `xhost +` outside Docker.[8] A container with common ROS GUI plugins can be started with following command:

```
~$ docker run -it --rm --net host \
   --env ROS_MASTER_URI=http://DUCKIEBOT_IP:11311 \
   --env ROS_IP=LAPTOP_IP \
   --env="DISPLAY" \
   --env="QT_X11_NO_MITSHM=1" \
   --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
   duckietown/rpi-gui-tools
```

Packaged within this image are common ROS plugins which can be run on graphical environments. A shell wrapper is also provided for convenience:

```
dt> start_gui_tools DUCKIEBOT_NAME rqt_image_view
```

[8] See https://wiki.ros.org/docker/Tutorials/GUI#The_safer_way for a more secure alternative.

The above command opens a ROS shell that will connect to the `DUCKIEBOT`'s ROS master node. To test the ROS connection works, run `roswtf`.

5.6.9. Remote control

The following container launches the joystick demo (USB joystick must be connected):

```
~$ docker -H DUCKIEBOT_NAME run --privileged --net host -v /data:/data \
   duckietown/rpi-duckiebot-joystick-demo:master18
```

```
dt> duckiebot demo --demo_name joystick --duckiebot_name DUCKIEBOT_NAME
```

```
dt> duckiebot keyboard_control DUCKIEBOT_NAME
```

5.6.10. Camera calibration

The following container will launch the extrinsic calibration procedure:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data
duckietown/rpi-duckiebot-calibration:master18
```

Passing `-v /data:/data` is necessary so that all calibration settings will be preserved. When placed on the calibration pattern, the following commands will initiate an interactive calibration sequence for the camera.

```
dt> duckiebot calibrate_extrinsics DUCKIEBOT_NAME
```

```
dt> duckiebot calibrate_intrinsics DUCKIEBOT_NAME
```

5.6.11. Wheel calibration

To calibrate the gain and trim of the wheel motors, the following commands are needed:

```
dt> duckiebot demo --demo_name base --duckiebot_name DUCKIEBOT_NAME
```

```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_gain --GAIN
```

```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_trim --TRIM
```

5.6.12. Lane following

Once calibrated, the lane following demo can be launched as follows:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data
duckietown/rpi-duckiebot-lanefollowing-demo:master18
```

```
dt> duckiebot demo --demo_name lane_following --duckiebot_name DUCKIEBOT_NAME
```

5.7. Retrospective

One problem encountered during the development of Duckietown's Docker infrastructure was the matter of whether to store source code inside or outside the container (e.g. as described in § 5.4.4). If stored externally, a developer can still load source code in a shared volume and rebuild on container startup. Both approaches can produce reproducible artifacts if properly versioned, but Docker images launch more quickly when images are fully prebuilt and tend to be more inspectable with sources included.

Initially, we made the explicit decision to ship user source code directly inside the image. As a consequence, any modifications to the source code would trigger a subsequent rebuild, tying the sources and Docker image together. While including sources enables easier troubleshooting and diagnostics, it also adds some friction during development, which caused users to struggle with environment setup and Docker configuration issues.

The root cause of this friction was a product of imprecise versioning and over-automation. As version tags were initially omitted, all images were built and pulled from latest commit on the mainline development branch. The auto-build feature of the CI server caused upstream modifications to cascade to downstream images. Our short-term solution was to disable auto-building, and push local builds to the server manually, however fixing it required us to rethink the role of versioning and testing Docker builds in the CI toolchain.

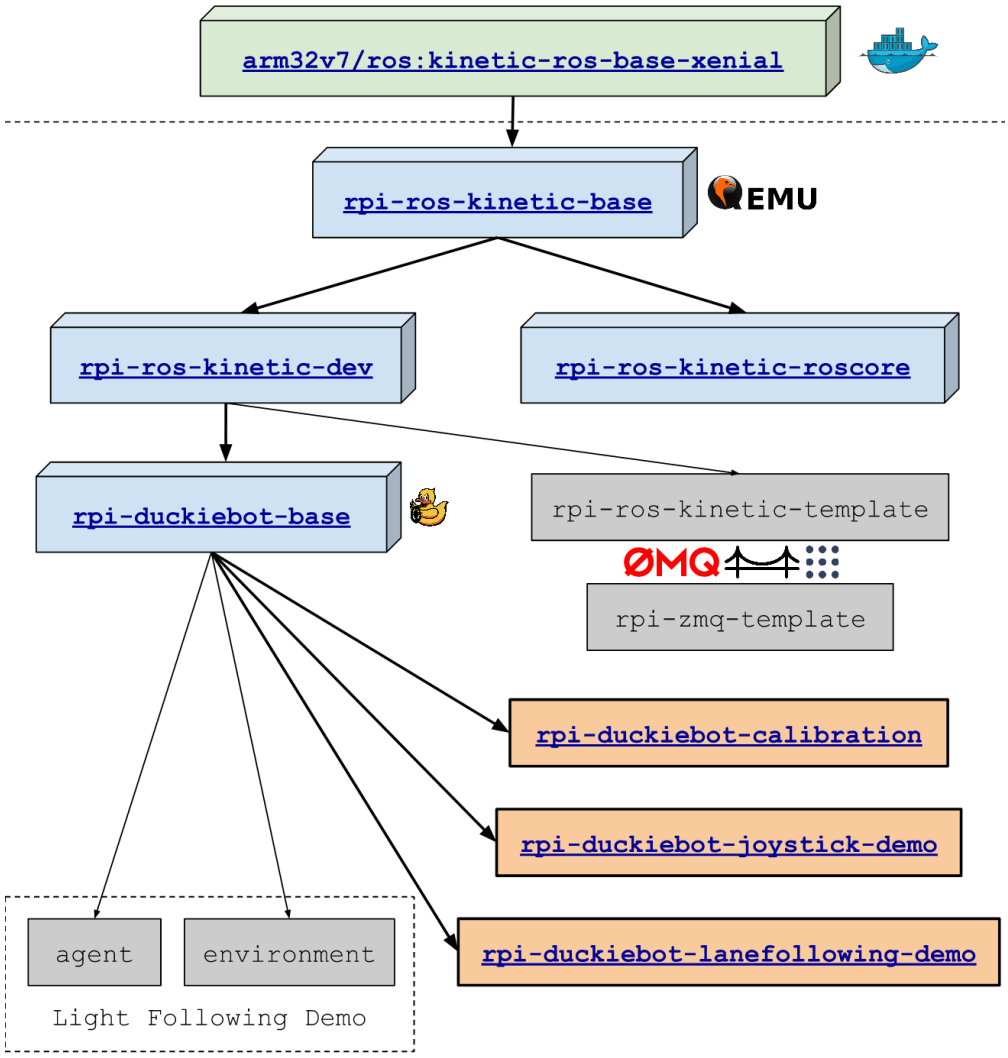


Fig. 5.5. Early prototype of the Docker image hierarchy. Chaining unversioned autobuilds without disciplined unit testing creates a potential domino effect which allows breaking changes to propagate downstream, resulting in a cascade of silent failures.

Fig. 5.6. The AI Driving Olympics, a primary use case for the system described above.

A more stable solution is to store all sources on the local development environment and rebuild the image only when its upstream dependencies change. The image only contains its compiled upstream dependencies and is only paired with source code at runtime.

One of the primary use cases for the Duckietown container infrastructure is a biannual autonomous robotics competition called the AI Driving Olympics [Zilly et al., 2019] (AIDO). To participate, competitors must submit a Docker image (various templates are provided for reinforcement learning, imitation learning and classical robotics). The submitted image, together with a Git repository and a commit hash, constitutes an AIDO submission. The submission is retrieved by the organizers and evaluated on a random map in Duckietown's simulator [Chevalier-Boisvert et al., 2018]. This evaluation produces a numerical score in several categories. Valid submissions may also be run on a physical *robotarium*. The highest ranking submissions are evaluated in a final round at NeurIPS and ICRA.

5.7.1. Remarks on security

An unfortunate technical shortcoming of the Docker system is its reliance on superuser privileges. While Docker takes a variety of preventative measures to ensure container inhabitants cannot gain escalated privileges, numerous breakout attacks have been discovered [Martin et al., 2018] in the wild. Any process which can circumvent container security gains unfettered access to the host OS, making Docker especially unsuitable for deployment on cloud, grid, and cluster computing environments.

Furthermore, Docker provides a mechanism to bypass its own security measures, allowing container applications to run as if they were root processes on the host OS: the `--privileged` flag. This feature, alongside the fact that most Docker users are unqualified to audit upstream images, which are liable to include packages of dubious provenance [Martin et al., 2018], makes Docker particularly troublesome for shared-computing environments.

Docker's unnecessarily high privileges and susceptibility for misuse are serious issues. While operator error may be partly at fault, these vulnerabilities are primarily the result of poor implementation choices. Docker's flagrant violation of the principle of least privilege [Saltzer and Schroeder, 1975] effectively compromises the entire Linux security model.

To address these issues, various container platforms, including Shifter [Gerhardt et al., 2017] and Singularity [Kurtzer et al., 2017], have emerged and gained traction in the scientific

computing community, owing to their lower privileges and compatibility with legacy Linux distributions used by many academic computing environments. Since then, Docker has also introduced a rootless mode, but it remains experimental at the time of writing this thesis.

5.8. Conclusion

In this chapter we have taken a guided tour through the process of containerization and demonstrated the effectiveness of containers for building reproducible robotics software – a key step in the broader quest for experimental reproducibility. We offer a set of best practices and lessons learned during the design, development and deployment of Docker containers for the Duckietown [Paull et al., 2017] platform. The author wishes to thank Rusi Hristov for his invaluable technical assistance during the preliminary stages of this project.

Chapter 6

Conclusion

> "We are all shaped by the tools we use, in particular: the formalisms we use shape our thinking habits, for better or for worse, and that means that we have to be very careful in the choice of what we learn and teach, for unlearning is not really possible."
>
> –Edsger W. Dijkstra [2000], *Answers to questions from students of Software Engineering*

In this work, we explored four different programming tools from software engineering for the development of intelligent systems, broadly addressing cognitive complexity arising in four phases of Royce's Waterfall method Figure 1.1. These tools have varying degrees of practicality, from highly theoretical (e.g. adversarial testing of differentiable programs Chapter 4) to more pragmatic (e.g. containerization Chapter 5). In each chapter, we provide some motivating examples and use cases which demonstrate key deficiencies in state-of-the-art programming tools for intelligent systems and propose candidate solutions which address a few of those shortcomings. While we certainly hope that intelligent system programmers (e.g. roboticists and machine learning practitioners) may derive some value from the tools themselves, our intention is to be *illustrative* rather than *prescriptive*.

In building tools and validating their effectiveness for toy applications, we hope that middleware and tools developers will carefully consider the cognitive complexity which software tools can introduce and the importance of notational and denotational design. Notation forces the author to think carefully about their abstractions, prevents logical errors, and allows the reader to more easily understand its implications. In addition, we hope that the programming tools illustrated in this thesis will inspire developers to re-imagine the potential for computer-aided programming in the design of intelligent systems.

By complementing the cognitive abilities of human programmers – who excel at creative problem solving and high-level abstract reasoning – with the low-level symbolic processing

capabilities of programming tools, we can accelerate the design, development and validation of intelligent systems in real-world applications. This process, we argue, deserves more specific tools than general-purpose programming due to the opportunities and challenges which intelligent systems present and the unique interplay between human and machine intelligence.

As we start to engineer autonomous systems which take increasingly human decisions, programmers will play a critical role in shaping the behavior and dynamics of these systems. In order to build trustworthy autonomous systems, tools which enable humans to reason about the behavior of autonomous systems are essential [Famelis et al., 2012]. Doing so requires us to actively rethink the programming model in machine learning to incorporate human knowledge, e.g. using differentiable programming and type theory (Chapter 3) or building tools which provide automated reasoning and visualization capabilities (e.g. customized run and debugging assistance Chapter 2). Ensuring software artifacts are reproducible will require sound build systems and best practices for reproducible software installation and configuration (Chapter 5).

Traditional software engineering prescribes a rigorous process model and testing methodology Figure 1.1 which has guided generations of software projects. To become a true engineering discipline, the machine learning community will need to re-imagine this paradigm for systems which continously adapt to their environment. Intelligent systems are trained on *objective functions*, which are typically one- or low-dimensional functions for measuring the performance of a system, typically outputting a scalar value known as *error* or *loss*. In practice, we care about a multiobjective set of criteria [Censi, 2015], including energy efficiency [Paull et al., 2010], memory [Mitliagkas et al., 2013], usability [Breuleux and van Merriënboer, 2017], predictability [Turner and Neal, 2017], latency [Ravanelli et al., 2018], robustness [Pineau et al., 2003], explainability [Turner, 2016], traceability [Guo et al., 2017, Tsirigotis et al., 2018], certainty [Diaz Cabrera, 2018], trustworthiness [Xu, 2017], transferability [Mehta et al., 2019], scalability [Luan et al., 2019] and other criteria.

In traditional software engineering, it is reasonable to assume the people who are implementing a new system have some implicit domain knowledge and are well-intentioned human beings working towards a common goal – given a coarse description, they can fill in the blanks. When building an intelligent system, it is more accurate to assume the entity

implementing our requirements is a naïve but crafty genie. Given some data and an optimization metric, it will take every available shortcut to satisfy the desired criteria. If we are not careful about constructing the requirements correctly, this entity can create a solution that simply does not work (in the best case), or appears to work but is actually cursed in a subtle manner.

When building an intelligent system developers must carefully ask, "What are the behavioral requirements of the system?" This question is often very troublesome, for the requirements cannot be fuzzy specifications, but precise constraints on the solution set. Specifying the requirements is often indistinguishable from implementing the system – with the right language abstractions (e.g. declarative programming), requirements and implementation can even take the same notation (e.g. SQL, Prolog). But how can we be assured the system meets those requirements? Humans can drive a car, but have difficulty describing the algorithm for driving. Labeling the data by hand is too expensive. Exhaustive verification is right out the window. However fuzz testing remains an attractive alternative. As we show in § 4.8, by making practical assumptions about the data and oracle, we can spend the available computational budget more wisely to detect more severe errors with lower fiscal and computational overhead.

For example, in the design of a web-based advertisement recommendation system, we can optimize for various criteria such as click rate, engagement, and sales conversion. So long as we can measure these parameters, modern function approximators can optimize for any single criterion or combination thereof (Equation 4.7.4). Much of the work involved in machine learning is designing representations which are suitable for downstream tasks and designing loss functions which accurately capture those tasks. For example, by optimizing for click rate, we create an artificial market for click bots. Similarly, in self-driving vehicles, we often want to optimize for passenger safety. However, by doing so naïvely can train a vehicle that never moves, or always yields to passing vehicles. Building representations and loss functions which capture the full range of objectives can be a painstaking process.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://dl.acm.org/citation.cfm?id=3026877.3026899.

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.

Peter Abeles. Efficient Java Matrix Library, 2010. URL http://ejml.org/.

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. ACM Trans. Graph., 38(4):121:1–121:12, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL https://doi.acm.org/10.1145/3306346.3322967.

Ashish Agarwal. Static automatic batching in TensorFlow. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36$^{\text{th}}$ International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 92–101, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/agarwal19a.html.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In Proceedings of the 2nd SysML Conference, 2019. URL https://www.sysml.cc/doc/2019/88.pdf.

Isabela Albuquerque, Joao Monteiro, Thang Doan, Breandan Considine, Tiago Falk, and Ioannis Mitliagkas. Multi-objective training of Generative Adversarial Networks with multiple discriminators. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 202–211, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/albuquerque19a.html.

Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. bioRxiv, 2018. doi: 10.1101/265231. URL https://www.biorxiv.org/content/early/2018/08/29/265231.

Mario Alvarez-Picallo and C.-H. Luke Ong. Change actions: Models of generalised differentiation. volume abs/1902.05465, 2019. URL https://arxiv.org/abs/1902.05465.

Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation: Derivatives of fixpoints, and the recursive semantics of Datalog. volume abs/1811.06069, 2018. URL https://arxiv.org/abs/1811.06069.

Nada Amin and Ross Tate. Java and Scala's type systems are unsound: The existential crisis of null pointers. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984004. URL https://doi.acm.org/10.1145/2983990.2984004.

Andrew W. Appel and Jens Palsberg. Modern Compiler Implementation in Java. Cambridge University Press, USA, 2nd edition, 2003. ISBN 052182060X.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein Generative Adversarial Networks. In Doina Precup and Yee Whye Teh, editors, Proceedings of the 34th International Conference on Machine Learning, volume 70 of Proceedings of Machine Learning Research, pages 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/arjovsky17a.html.

John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs, volume 21. ACM, New York, NY, USA, August 1978. doi: 10.1145/359576.359579. URL https://doi.acm.org/10.1145/359576.359579.

Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open Neural Network Exchange, 2019. URL https://github.com/onnx/onnx.

A. G. Baydin and B. A. Pearlmutter. Automatic differentiation of algorithms for machine learning. In Proceedings of the AutoML Workshop at the International Conference on Machine Learning (ICML), Beijing, China, June 21–26, 2014, 2014. URL https://arxiv.org/abs/1404.7456.

Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. CoRR, abs/1502.05767, 2015a. URL https://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 https://arxiv.org/abs/1511.07727.

Fabrice Bellard. QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track, volume 41, page 46, 2005.

R. E. Bellman, H. Kagiwada, and R. E. Kalaba. Wengert's numerical method for partial derivatives, orbit determination and quasilinearization. Commun. ACM, 8(4):231–232, April 1965. ISSN 0001-0782. doi: 10.1145/363831.364886. URL https://doi.acm.org/10.1145/363831.364886.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. Trans. Neur. Netw., 5(2):157–166, March 1994. ISSN 1045-9227. doi: 10.1109/72.279181. URL https://doi.org/10.1109/72.279181.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, et al. Theano: a CPU and GPU math expression compiler. In Proceedings of the Python for scientific computing conference (SciPy), volume 4. Austin, TX, 2010. URL http://deeplearning.net/software/theano/.

G. Berry and R. Sethi. From regular expressions to deterministic automata. Theor. Comput. Sci., 48(1):117–126, December 1986. ISSN 0304-3975. URL https://dl.acm.org/citation.cfm?id=39528.39537.

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In European Conference on Object-Oriented Programming, pages 257–281. Springer, 2014.

Richard F Blute, J Robin B Cockett, and Robert AG Seely. Differential categories. Mathematical structures in computer science, 16(6):1049–1083, 2006.

Richard F Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential categories. Theory and Applications of Categories, 22(23):622–672, 2009.

Yang Bo. DeepLearning.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 Sampedro, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. 2018. URL https://arxiv.org/abs/1709.07857.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, Verification, Model Checking, and Abstract Interpretation, pages 427–442, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-31622-0. URL http://theory.stanford.edu/~arbrad/papers/arrays.pdf.

Mikio L Braun, Johannes Schaback, Matthias L Jugel, Nicolas Oury, et al. jBlas: Linear algebra for Java, 2011. URL http://jblas.org/.

Olivier Breuleux and Bart van Merriënboer. Automatic differentiation in myia. 2017. URL https://github.com/mila-udem/myia.

Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the Simply Typed Lambda-calculus with Linear negation. 2020. URL https://arxiv.org/abs/1909.13768.

Janusz A. Brzozowski. Derivatives of regular expressions. J. ACM, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL https://doi.acm.org/10.1145/321239.321249.

Jonathan B. Buckheit and David L. Donoho. WaveLab and Reproducible Research, pages 55–81. Springer New York, New York, NY, 1995. ISBN 978-1-4612-2544-7. doi: 10.1007/978-1-4612-2544-7_5. URL https://doi.org/10.1007/978-1-4612-2544-7_5.

Roman V Buniy, Stephen DH Hsu, and Anthony Zee. Is Hilbert space discrete? Physics Letters B, 630(1-2):68–72, 2005. URL https://doi.org/10.1016/j.physletb.2005.09.084.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In Proceedings of the 35$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL https://doi.acm.org/10.1145/2594291.2594304.

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99392. URL https://doi.acm.org/10.1145/99370.99392.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. volume 109, pages 4–56, Duluth, MN, USA, February 1994. Academic Press, Inc. doi: 10.1006/inco.1994.1013. URL https://dx.doi.org/10.1006/inco.1994.1013.

Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. pages 179–191, 2011.

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. Journal of Statistical Software, 76(1), 1 2017. doi: 10.18637/jss.v076.i01.

Andrea Censi. A mathematical theory of co-design. arXiv preprint arXiv:1512.08055, 2015.

J-M Champarnaud, J-L Ponty, and Djelloul Ziadi. From regular expressions to finite automata. International journal of computer mathematics, 72(4):415–431, 1999. URL https://doi.org/10.1080/00207169908804865.

Émilie Charlier, Narad Rampersad, and Jeffrey Shallit. Enumeration and decidable properties of automatic sequences. Lecture Notes in Computer Science, page 165179, 2011. ISSN 1611-3349. doi: 10.1007/978-3-642-22321-1_15. URL https://dx.doi.org/10.1007/978-3-642-22321-1_15.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017. doi: 10.1145/3136000.3136001. URL https://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, Department of Computer Science, Hong Kong, 1998. URL https://www.cse.ust.hk/~scc/publ/CS98-01-metamorphictesting.pdf.

Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. pages 299–310, 2012. doi: 10.1145/2254064.2254100. URL https://doi.acm.org/10.1145/2254064.2254100.

Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for OpenAI Gym. https://github.com/duckietown/gym-duckietown, 2018.

Bruce Christianson. A Leibniz notation for automatic differentiation. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, Recent Advances in Algorithmic Differentiation, volume 87 of Lecture Notes in Computational Science and Engineering, pages 1–9. Springer, Berlin, 2012. ISBN 978-3-540-68935-5. doi: 10.1007/978-3-642-30023-3_1. URL https://uhra.herts.ac.uk/bitstream/handle/2299/8933/904722.pdf.

Alonzo Church. The Calculi of Lambda-conversion. Annals of Mathematics Studies. Princeton University Press, 1941. ISBN 9780691083940. URL https://books.google.ca/books?id=yWCYDwAAQBAJ.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. pages 268–279, 2000. doi: 10.1145/351240.351266. URL https://doi.acm.org/10.1145/351240.351266.

James Clift and Daniel Murfet. Derivatives of Turing machines in Linear Logic. arXiv preprint arXiv:1805.11813, 2018. URL https://arxiv.org/abs/1805.11813.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Idiap-RR Idiap-RR-46-2002, IDIAP, 2002.

George F Corliss and Andreas Griewank. Operator overloading as an enabling technology for automatic differentiation. Technical report, Argonne National Laboratory, 1993. URL http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93431.pdf.

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. URL https://hal.inria.fr/hal-01257471/document.

Sébastien Crozet et al. nalgebra: a linear algebra library for Rust, 2019. URL https://nalgebra.org.

H.B. Curry and R. Feys. Combinatory Logic. Number v. 1 in Combinatory Logic. North-Holland Publishing Company, 1958. URL https://books.google.ca/books?id=fEnuAAAAMAAJ.

Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. CoRR, abs/1801.08058, 2018. URL https://arxiv.org/abs/1801.08058.

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 https://arxiv.org/abs/1611.01652.

T. J. Dekker. A floating-point technique for extending the available precision. Numer. Math., 18(3):224–242, June 1971. ISSN 0029-599X. doi: 10.1007/BF01397083. URL https://dx.doi.org/10.1007/BF01397083.

Commons Math Developers. Apache Commons Math. Forest Hill, MD, USA: The Apache Software Foundation, 2012. URL https://commons.apache.org/proper/commons-math/.

Manfred Ramon Diaz Cabrera. Interactive and Uncertainty-aware Imitation Learning: Theory and Applications. PhD thesis, Concordia University, 2018. URL https://spectrum.library.concordia.ca/984373/1/Diaz_MSc_F2018.pdf.

Edsger W Dijkstra. Answers to questions from students of software engineering. Circulated privately, 2000. URL http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. Commun. ACM, 62(8):62–70, July 2019. ISSN 0001-0782. doi: 10.1145/3338112. URL https://doi.acm.org/10.1145/3338112.

Sergey Dmitriev. Language oriented programming: The next programming paradigm. JetBrains onBoard, 1(2):1–13, 2004. URL http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf.

Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In ICLR, 2019. URL https://github.com/chrisdonahue/wavegan.

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. URL https://arc.aiaa.org/doi/abs/10.2514/3.25422.

Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. Deepcruiser: Automated guided testing for stateful deep learning systems. CoRR, abs/1812.05339, 2018. URL https://arxiv.org/abs/1812.05339.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285, 2016. URL https://arxiv.org/abs/1603.07285.

A. Edalat and A. Lieutier. Domain theory and differential calculus (functions of one variable). In Proceedings 17$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science, pages 277–286, July 2002. doi: 10.1109/LICS.2002.1029836.

Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. CoRR, abs/1606.01642, 2016. URL https://arxiv.org/abs/1606.01642.

Thomas Ehrhard and Laurent Regnier. The differential λ-calculus. Theor. Comput. Sci., 309 (1):1–41, December 2003. ISSN 0304-3975. doi: 10.1016/S0304-3975(03)00392-X. URL https://dx.doi.org/10.1016/S0304-3975(03)00392-X.

Conal Elliott. The simple essence of automatic differentiation. Proc. ACM Program. Lang., 2(ICFP):70:1–70:29, July 2018. ISSN 2475-1421. doi: 10.1145/3236765. URL https:

//doi.acm.org/10.1145/3236765.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. Journal of Functional Programming, 13(2), 2003. URL http://conal.net/papers/jfp-saig/.

Conal M. Elliott. Beautiful differentiation. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09, pages 191–202, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596579. URL https://doi.acm.org/10.1145/1596550.1596579.

Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869625. URL https://doi.acm.org/10.1145/1869542.1869625.

Michais Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In 2012 34th International Conference on Software Engineering (ICSE), pages 573–583. IEEE, 2012.

Michalis Famelis, Naama Ben-David, Alessio Di Sandro, Rick Salay, and Marsha Chechik. Mu-Mmint: An IDE for model uncertainty. In Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, pages 697–700, Piscataway, NJ, USA, 2015. IEEE Press. URL https://dl.acm.org/citation.cfm?id=2819009.2819141.

George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. SIGSOFT Softw. Eng. Notes, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL https://doi.acm.org/10.1145/263244.263267.

Bryan Ford. Parsing Expression Grammars: A recognition-based syntactic foundation. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011. URL https://doi.acm.org/10.1145/964001.964011.

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, ICSE '17, pages 758–769, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.75. URL https://doi.org/10.1109/ICSE.2017.75.

Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for HPC. Journal of Physics: Conference Series, 898:082021, oct 2017. doi: 10.1088/1742-6596/898/8/082021. URL https://doi.org/10.1088%2F1742-6596%2F898%2F8%2F082021.

Carlo Ghezzi and Dino Mandrioli. Incremental parsing. ACM Trans. Program. Lang. Syst., 1(1):58–70, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357066. URL https://doi.acm.org/10.1145/357062.357066.

Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. 56:10:1–10:27, 2016. ISSN 1868-8969. doi: 10.4230/LIPIcs.ECOOP.2016.10. URL http://drops.dagstuhl.de/opus/volltexte/2016/6104.

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

C. F. Goldfarb. A generalized approach to document markup. In Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, pages 68–73, New York, NY, USA, 1981. ACM. ISBN 0-89791-050-8. doi: 10.1145/800209.806456. URL https://doi.acm.org/10.1145/800209.806456.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press. URL https://dl.acm.org/citation.cfm?id=2969033.2969125.

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. Probabilistic programming with densities in SlicStan: Efficient, flexible, and deterministic. Proc. ACM Program. Lang., 3(POPL):35:1–35:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290348. URL https://doi.acm.org/10.1145/3290348.

Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In Complexity in numerical optimization, pages 128–162. World Scientific, 1993. URL http://ftp.mcs.anl.gov/pub/tech_reports/reports/P355.pdf.

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

P. R. 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, IFL '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4273-5. doi: 10.1145/2897336.2897341. URL https://doi.acm.org/10.1145/2897336.2897341.

Radu Grigore. Java generics are Turing Complete. pages 73–85, 2017. doi: 10.1145/3009837.3009871. URL https://doi.acm.org/10.1145/3009837.3009871.

Martin Guenther. Are serious things done with ROS in Python?, 2018. URL https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6.

Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 3–14. IEEE, 2017. URL https://www.cs.mcgill.ca/~jguo/resources/papers/ICSE2017_JIN_Preprint.pdf.

Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802046. URL https://doi.acm.org/10.1145/800055.802046.

Pieter Hintjens. ZeroMQ: messaging for many applications. O'Reilly Media, Inc., 2013.

Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. Journal of Object Technology, 7(3):125–151, 2008.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. Neural Netw., 2(5):359–366, July 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8. URL https://dx.doi.org/10.1016/0893-6080(89)90020-8.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation. CoRR, abs/1910.00935, 2019. URL https://arxiv.org/abs/1910.00935.

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. URL

https://link.springer.com/chapter/10.1007/978-3-540-45226-3_44.

Alekseĭ Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic predicting devices. CCM Information Corporation, 1965. URL https://books.google.ca/books?id=FhwVNQAACAAJ.

Kenneth E Iverson. Math for the layman, 1999. URL http://www.cs.trinity.edu/About/The_Courses/cs301/math-for-the-layman/.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. volume abs/1712.05877, 2017. URL https://arxiv.org/abs/1712.05877.

C. Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, In Computing: the Australasian Theory Seminar, Proceedings, 1997.

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, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL https://doi.acm.org/10.1145/2647868.2654889.

M. Kac. On some connections between probability theory and differential and integral equations. In Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, pages 189–215, Berkeley, Calif., 1951. University of California Press. URL https://projecteuclid.org/euclid.bsmsp/1200500229.

W. Kahan. Further remarks on reducing truncation errors. Commun. ACM, 8(1):40–, January 1965. ISSN 0001-0782. doi: 10.1145/363707.363723. URL https://doi.acm.org/10.1145/363707.363723.

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.

Robert Kelly, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Evolving the incremental λ calculus into a model of forward automatic differentiation (AD). CoRR, abs/1611.03429, 2016. URL https://arxiv.org/abs/1611.03429.

Andrew Kennedy. Dimension types. In European Symposium on Programming, pages 348–362. Springer, 1994.

Andrew Kennedy. Types for Units-of-Measure: Theory and Practice, pages 268–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17685-2. doi: 10.1007/978-3-642-17685-2_8. URL https://doi.org/10.1007/978-3-642-17685-2_8.

Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pages 21–40, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094814. URL https://doi.acm.org/10.1145/1094811.1094814.

Andrew John Kennedy. Programming languages and dimensions. Technical report, University of Cambridge, Computer Laboratory, 1996. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf.

Oleg Kiselyov. Number-parameterized types. The Monad Reader, 5:73–118, 2005.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. April 2009. URL https://www.microsoft.com/en-us/research/publication/fun-type-functions/.

Gerwin Klein, Steve Rowe, and Régis Décamps. JFlex-the fast scanner generator for Java. 2001. URL http://www.jflex.de.

Rainer Koppler. A systematic approach to fuzzy parsing. Softw. Pract. Exper., 27(6): 637–649, June 1997. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3. URL https://dx.doi.org/10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3.

Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. PLOS ONE, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459. URL https://doi.org/10.1371/journal.pone.0177459.

Leslie Lamport. A discussion with Leslie Lamport, August 2002. URL https://www.microsoft.com/en-us/research/publication/discussion-leslie-lamport/.

Chris Lattner and Jacques Pienaar. MLIR primer: A compiler infrastructure for the end of Moores law, 2019. URL https://ai.google/research/pubs/pub48035.

Chris Lattner and Richard Wei. Swift for TensorFlow. 2018. URL https://github.com/tensorflow/swift.

Sören Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. CoRR, abs/1904.02990, 2019. URL https://arxiv.org/abs/1904.02990.

Gary T Leavens and Todd D Millstein. Multiple dispatch as dispatch on tuples. ACM SIGPLAN Notices, 33(10):374–387, 1998. URL http://web.cs.ucla.edu/~todd/research/oopsla98.pdf.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521(7553): 436–444, 5 2015. ISSN 0028-0836. doi: 10.1038/nature14539.

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. ACM Trans. Graph., 37(4):139:1–139:13, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201383. URL https://doi.acm.org/10.1145/3197517.3201383.

J. C. R. Licklider. Man-computer symbiosis. IEEE Ann. Hist. Comput., 14(1):24–, January 1992. ISSN 1058-6180. URL https://dl.acm.org/citation.cfm?id=612400.612433.

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. BIT Numerical Mathematics, 16(2):146–160, Jun 1976. ISSN 1572-9125. doi: 10.1007/BF01931367. URL https://doi.org/10.1007/BF01931367.

Barbara Liskov. Keynote address - data abstraction and hierarchy. SIGPLAN Not., 23(5): 17–34, January 1987. ISSN 0362-1340. doi: 10.1145/62139.62141. URL https://doi.acm.org/10.1145/62139.62141.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. CoRR, abs/1806.09055, 2018. URL https://arxiv.org/abs/1806.09055.

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, KDD '05, pages 641–647, New York, NY, USA, 2005. ACM. ISBN 1-59593-135-X. doi: 10.1145/1081870.1081950. URL https://doi.acm.org/10.1145/1081870.1081950.

Sitao Luan, Mingde Zhao, Xiao-Wen Chang, and Doina Precup. Break the ceiling: Stronger multi-scale deep graph convolutional networks. In Advances in Neural Information Processing Systems, pages 10943–10953, 2019.

David R. MacIver. Hypothesis, 2018. URL https://github.com/HypothesisWorks/hypothesis.

Dougal Maclaurin. Modeling, Inference and Optimization with Composable Differentiable Procedures. PhD thesis, Harvard University, April 2016. URL https://dash.harvard.edu/handle/1/33493599.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in NumPy. In ICML 2015 AutoML Workshop, 2015. URL https://github.com/HIPS/autograd.

J.R. Magnus and H. Neudecker. Matrix differential calculus with applications in statistics and econometrics. Wiley series in probability and mathematical statistics. Wiley, 1988. ISBN 0471915165. Pagination: xvii, 393.

Dhruv C. Makwana and Neelakantan R. Krishnaswami. NumLin: Linear Types for Linear Algebra. In Alastair F. Donaldson, editor, 33rd European Conference on Object-Oriented Programming (ECOOP 2019), volume 134 of Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.14. URL http://drops.dagstuhl.de/opus/volltexte/2019/10806.

Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem–vulnerability analysis. Computer Communications, 122:30–43, 2018.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. 2001.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. In Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08, pages 287–295, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328474. URL https://doi.acm.org/10.1145/1328438.1328474.

Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active domain randomization, 2019. URL https://arxiv.org/abs/1904.04762.

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, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552. URL https://doi.acm.

org/10.1145/1142473.1142552.

Paul B Menage. Adding generic process containers to the Linux kernel. In Proceedings of the Linux Symposium, volume 2, pages 45–57, 2007.

Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 14598–14609. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/9604-compiler-auto-vectorization-with-imitation-learning.pdf.

Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, 2014(239), March 2014. ISSN 1075-3583. URL https://dl.acm.org/citation.cfm?id=2600239.2600241.

Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11, pages 189–195, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034801. URL https://doi.acm.org/10.1145/2034773.2034801.

Ioannis Mitliagkas, Constantine Caramanis, and Prateek Jain. Memory limited, streaming pca. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 2886–2894. Curran Associates, Inc., 2013. URL http://papers.nips.cc/paper/5035-memory-limited-streaming-pca.pdf.

Aaron Moss. Derivatives of Parsing Expression Grammars. In Proceedings 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017., pages 180–194, 2017. doi: 10.4204/EPTCS.252.18. URL https://doi.org/10.4204/EPTCS.252.18.

Maurice Naftalin and Philip Wadler. Java generics and collections. O'Reilly Media, 2007.

Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: A fluent API generator. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, pages 199–211, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5524-7. doi: 10.1145/

3136040.3136041. URL https://doi.acm.org/10.1145/3136040.3136041.

Yuri Nesterov. Gradient methods for minimizing composite functions. Mathematical Programming, 140(1):125–161, Aug 2013. ISSN 1436-4646. doi: 10.1007/s10107-012-0629-5. URL https://doi.org/10.1007/s10107-012-0629-5.

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 language for science and kMath library. AIP Conference Proceedings, 2163(1):040004, 2019. doi: 10.1063/1.5130103. URL https://aip.scitation.org/doi/abs/10.1063/1.5130103.

Yu Nureki. JAutoDiff: A pure Java library for automatic differentiation, 2012. URL https://github.com/uniker9/JAutoDiff.

Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. SIAM J. Sci. Comput., 26(6):1955–1988, June 2005. ISSN 1064-8275. doi: 10.1137/030601818. URL https://dx.doi.org/10.1137/030601818.

Christopher Olah. Neural networks, types, and functional programming. 2015. URL https://colah.github.io/posts/2015-09-NN-Types-FP.

Gerardo Pardo-Castellote. OMG Data-Distribution Service: Architectural overview. In 23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings., pages 200–206. IEEE, 2003.

Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience, 25(7):789–810, 1995.

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. URL https://openreview.net/pdf?id=BJJsrmfCZ.

Liam Paull, Howard Li, and Liuchen Chang. A novel domestic electric water heater model for a multi-objective demand side management program. Electric Power Systems Research, 80(12):1446–1451, 2010.

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.

David J Pearce and James Noble. Implementing a language with flow-sensitive and structural typing on the jvm. Electronic Notes in Theoretical Computer Science, 279(1):47–59, 2011.

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. ISSN 1439-7358. doi: 10.1007/978-3-540-68942-3_8. URL http://www.bcl.hamilton.ie/~barak/papers/sound-efficient-ad2008.pdf.

Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, pages 165–168, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. doi: 10.1145/2500828.2500846. URL https://doi.acm.org/10.1145/2500828.2500846.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132785. URL https://doi.acm.org/10.1145/3132747.3132785.

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, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://www.aclweb.org/anthology/D14-1162.

Kaare Brandt Petersen et al. The Matrix Cookbook. 2012. URL https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf.

Leonardo Piñeyro, Alberto Pardo, and Marcos Viera. Structure verification of deep neural networks at compilation time using dependent types. In Proceedings of the XXIII Brazilian

Symposium on Programming Languages, SBLP 2019, pages 46–53, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7638-9. doi: 10.1145/3355378.3355379. URL https://doi.acm.org/10.1145/3355378.3355379.

Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Policy-contingent abstraction for robust robot control. In Proceedings of Conference on Uncertainty in Articifical Intelligence (UAI), pages 477 – 484, August 2003.

Gill A Pratt. Is a Cambrian explosion coming for robotics? Journal of Economic Perspectives, 29(3):51–60, 2015. URL https://www.aeaweb.org/full_issue.php?doi=10.1257/jep.29.3#page=53.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In ICRA workshop on open source software, volume 3, page 5. Kobe, Japan, 2009.

Mirco Ravanelli, Dmitriy Serdyuk, and Yoshua Bengio. Twin regularization for online speech recognition. Interspeech 2018, Sep 2018. doi: 10.21437/interspeech.2018-1407. URL http://dx.doi.org/10.21437/Interspeech.2018-1407.

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 https://doi.acm.org/10.1145/3126905.

Norman A. Rink. Modeling of languages for tensor manipulation. CoRR, abs/1801.08771, 2018. URL https://arxiv.org/abs/1801.08771.

Mikael Rittri. Dimension inference under polymorphic recursion. In Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95, pages 147–159, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224197. URL https://doi.acm.org/10.1145/224164.224197.

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new IR for machine learning frameworks. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2018, pages 58–68, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211348. URL https://doi.acm.org/10.1145/3211346.3211348.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL https://doi.acm.org/10.1145/1868294.1868314.

Frank Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958. URL https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. CoRR, abs/1805.00907, 2018. URL https://arxiv.org/abs/1805.00907.

W. W. Royce. Managing the development of large software systems: Concepts and techniques. In Proceedings of the 9th International Conference on Software Engineering, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0. URL https://dl.acm.org/citation.cfm?id=41765.41801.

Claudio Ruch, Sebastian Hörl, and Emilio Frazzoli. AMoDeus, a simulation-based testbed for autonomous mobility-on-demand systems. 2018 21st International Conference on Intelligent Transportation Systems (ITSC), pages 3639–3644, 2018. URL https://www.amodeus.science/.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. pages 696–699, 1988. URL https://dl.acm.org/citation.cfm?id=65669.104451.

Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975. URL http://web.mit.edu/Saltzer/www/publications/rfc/csr-rfc-060.pdf.

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

Moses Schönfinkel. Über die bausteine der mathematischen logik. Mathematische annalen, 92(3):305–316, 1924.

F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson. Analyzing errors with the boolean difference. IEEE Trans. Comput., 17(7):676–683, July 1968. ISSN 0018-9340. doi: 10.1109/TC.1968.227417. URL https://doi.org/10.1109/TC.1968.227417.

C. E. Shannon. Computer chess compendium. pages 2–13, 1988. URL https://dl.acm.org/citation.cfm?id=61701.67002.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. Higher-Order and Symbolic Computation, 21(4):361–376, Dec 2008. ISSN 1573-0557. doi: 10.1007/s10990-008-9037-1. URL https://doi.org/10.1007/s10990-008-9037-1.

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. URL http://www.r6rs.org/final/html/r6rs/r6rs.html.

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.

Bernd Steinbach and Christian Posthoff. Boolean differential calculus. Synthesis Lectures on Digital Circuits and Systems, 12(1):1–215, 2017.

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 28$^{\text{th}}$ International Conference on Machine Learning (ICML-11), pages 609–616, 2011. URL https://stanford-ppl.github.io/website/papers/icml11-sujeeth.pdf.

Gerald J. Sussman and Guy L. Steele, Jr. Scheme: An interpreter for Extended Lambda Calculus. Technical report, Cambridge, MA, USA, 1975.

Norihisa Suzuki and David Jefferson. Verification decidability of Presburger array programs. J. ACM, 27(1):191–205, January 1980. ISSN 0004-5411. doi: 10.1145/322169.322185. URL https://doi.acm.org/10.1145/322169.322185.

AD Talantsev. On the analysis and synthesis of certain electrical circuits by means of special logical operators. Avt. i Telem., 20(7):898–907, 1959. URL http://www.mathnet.ru/links/bafa0b55349439b7d01a805198178a20/at12783.pdf.

Ross Tate. Mixed-site variance. In FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages, 2013. URL http://www.cs.cornell.edu/~ross/publications/mixedsite/.

H. C. A. Tavante, Benedito Donizeti Bonatto, and Maurilio Pereira Coutinho. Open source implementations of electromagnetic transient algorithms. 2018 13th IEEE International Conference on Industry Applications (INDUSCON), pages 825–828, 2018. URL https://github.com/hannelita/PyTHTA.

Eclipse Deeplearning4j Development Team. DL4J: Deep Learning for Java. 2016a. URL https://github.com/eclipse/deeplearning4j.

Eclipse Deeplearning4j Development Team. ND4J: Fast, scientific and numerical computing for the JVM. 2016b. URL https://github.com/eclipse/deeplearning4j.

A. Thayse and M. Davio. Boolean differential calculus and its application to switching theory. IEEE Trans. Comput., 22(4):409–420, April 1973. ISSN 0018-9340. doi: 10.1109/T-C.1973.223729. URL https://dx.doi.org/10.1109/T-C.1973.223729.

Andre Thayse. Boolean calculus of differences, volume 101 of ser-LNCS. 1981. ISBN 0-387-10286-8 (paperback). URL https://www.springer.com/gp/book/9783540102861.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pages 303–314, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180220. URL https://doi.acm.org/10.1145/3180155.3180220.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS), 2015. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

Christos Tsirigotis, Xavier Bouthillier, François Corneau-Tremblay, Peter Henderson, Reyhane Askari, Samuel Lavoie-Marchildon, Tristan Deleu, Dendi Suhubdy, Michael Noukhovitch, Frédéric Bastien, et al. Oríon: Experiment version control for efficient hyperparameter optimization. 2018. URL https://openreview.net/pdf?id=r1xkNLPixX.

Ryan Turner. A model explanation system. In 2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP), pages 1–6. IEEE, 2016.

Ryan Turner and Brady Neal. How well does your sampler really work? arXiv preprint arXiv:1712.06006, 2017. URL https://arxiv.org/pdf/1712.06006.pdf.

Ryan Turner, Jane Hung, Eric Frank, Yunus Saatchi, and Jason Yosinski. Metropolis-Hastings Generative Adversarial Networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 6345–6353, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/turner19a.html.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22, 2011.

Bart van Merriënboer. Sequence-to-sequence learning for machine translation and automatic differentiation for machine learning software tools. PhD thesis, Université de Montréal, September 2018. URL http://hdl.handle.net/1866/21743.

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 31, pages 6256–6265, 2018. URL https://arxiv.org/abs/1711.02712.

Bart van Merriënboernboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. Automatic differentiation in ML: where we are and where we should be going. CoRR, abs/1810.11530, 2018. URL https://arxiv.org/abs/1810.11530.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018. URL https://arxiv.org/abs/1802.04730.

Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. Software Language Engineering, SLE, 16(3), 2010. URL https://pdfs.semanticscholar.org/5adb/c633179cd7d32e1c1840225b2890ddeb32a5.pdf.

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, Software Language Engineering, pages 41–61, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11245-9. URL https://gsd.uwaterloo.ca/sites/default/files/2014-sle-projectional.pdf.

Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97, pages 31–43, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. URL https://doi.acm.org/10.1145/258915.258920.

Timothy A. Wagner. Practical Algorithms for Incremental Software Development Environments. PhD thesis, EECS Department, University of California, Berkeley, Mar 1998. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html.

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 31, pages 10180–10191, 2018a. URL https://www.cs.purdue.edu/homes/rompf/papers/wang-nips18.pdf.

Fei Wang, Xilun Wu, Gregory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. CoRR, abs/1803.10228, 2018b. URL https://arxiv.org/abs/1803.10228.

Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. GraphGAN: Graph representation learning with Generative Adversarial Nets. 2018c. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16611.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, pages 7686–7695, 2018d. URL https://dl.acm.org/citation.cfm?id=3327757.3327866.

Xie Wang, Huaijin Wang, Zhendong Su, et al. Global optimization of numerical programs via prioritized stochastic algebraic transformations. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, pages 1131–1141, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00116. URL https://doi.org/10.1109/ICSE.

2019.00116.

Richard Wei, Vikram S. Adve, and Lane Schwartz. DLVM: A modern compiler infrastructure for deep learning systems. CoRR, abs/1711.03016, 2017. URL https://arxiv.org/abs/1711.03016.

R. E. Wengert. A simple automatic derivative evaluation program. Commun. ACM, 7 (8):463–464, August 1964. ISSN 0001-0782. doi: 10.1145/355586.364791. URL https://doi.acm.org/10.1145/355586.364791.

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. ISSN 0036-8075. doi: 10.1126/science.131.3410.1355. URL https://science.sciencemag.org/content/131/3410/1355.

Virginia Vassilevska Williams. Multiplying matrices in $\mathcal{O}(n^{2.373})$ time. 2014. URL https://people.csail.mit.edu/virgi/matrixmult-f.pdf.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. Trans. Evol. Comp, 1(1):67–82, April 1997. ISSN 1089-778X. doi: 10.1109/4235.585893. URL https://doi.org/10.1109/4235.585893.

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. CoRR, abs/1802.04680, 2018. URL https://arxiv.org/abs/1802.04680.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277732. URL https://doi.acm.org/10.1145/277650.277732.

Anqi Xu. Efficient Collaboration with Trust-seeking Robots. PhD thesis, McGill University Libraries, 2017.

Wojciech Zaremba. Learning Algorithms from Data. PhD thesis, New York University, 2016. URL https://cs.nyu.edu/media/publications/zaremba_wojciech.pdf.

Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1, NIPS'14, pages 1278–1286, Cambridge, MA, USA, 2014. MIT Press. URL https://dl.acm.org/citation.cfm?id=2968826.2968969.

Christoph Zenger. Indexed types. Theoretical Computer Science, 187(1-2):147–165, November 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(97)00062-5. URL https://dx.doi.org/10.1016/S0304-3975(97)00062-5.

Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. Commun. ACM, 62 (3):61–67, February 2019. ISSN 0001-0782. doi: 10.1145/3241979. URL https://doi.acm.org/10.1145/3241979.

Julian G. Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F. Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, A. Kirsten Bowser, Matthew R. Walter, Ruslan Hristov, Sunil Mallya, Emilio Frazzoli, Andrea Censi, and Liam Paull. The AI driving olympics at NeurIPS 2018. CoRR, abs/1903.02503, 2019. URL https://arxiv.org/abs/1903.02503.

Appendix A

Type-safe differentiable programming

A.1. Grammar

Below is an approximately complete BNF grammar for Kotlin∇:

⟨*type*⟩ ::= `Double` | `Float` | `Int` | `BigInteger` | `BigDouble`

⟨*nat*⟩ ::= `1` | ... | `99`

⟨*output*⟩ ::= `Fun<`⟨*type*⟩`Real>` | `VFun<`⟨*type*⟩`Real,`⟨*nat*⟩`>` | `MFun<`⟨*type*⟩`Real,`⟨*nat*⟩`,`⟨*nat*⟩`>`

⟨*int*⟩ ::= `0` | ⟨*nat*⟩⟨*int*⟩

⟨*float*⟩ ::= ⟨*int*⟩`.`⟨*int*⟩

⟨*num*⟩ ::= ⟨*type*⟩`(`⟨*int*⟩`)` | ⟨*type*⟩`(`⟨*float*⟩`)`

⟨*var*⟩ ::= `x` | `y` | `z` | `ONE` | `ZERO` | `E` | `Var()`

(A.1.2)

$$
\begin{aligned}
\langle signOp \rangle &::= \texttt{+} \mid \texttt{-} \\
\langle binOp \rangle &::= \langle signOp \rangle \mid \texttt{*} \mid \texttt{/} \mid \texttt{pow} \\
\langle trigOp \rangle &::= \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \mid \texttt{asin} \mid \texttt{acos} \mid \texttt{atan} \mid \texttt{asinh} \mid \texttt{acosh} \mid \texttt{atanh} \\
\langle unaryOp \rangle &::= \langle signOp \rangle \mid \langle trigOp \rangle \mid \texttt{sqrt} \mid \texttt{log} \mid \texttt{ln} \mid \texttt{exp} \\
\langle exp \rangle &::= \langle var \rangle \mid \langle num \rangle \mid \langle unaryOp \rangle \langle exp \rangle \mid \langle var \rangle \langle binOp \rangle \langle exp \rangle \mid \texttt{(} \langle exp \rangle \texttt{)} \\
\langle expList \rangle &::= \langle exp \rangle \mid \langle exp \rangle \texttt{,} \langle expList \rangle \\
\langle linOp \rangle &::= \langle signOp \rangle \mid \texttt{*} \mid \texttt{dot} \\
\langle vec \rangle &::= \texttt{Vec(} \langle expList \rangle \texttt{)} \mid \texttt{Vec} \langle nat \rangle \texttt{(} \langle expList \rangle \texttt{)} \\
\langle vecExp \rangle &::= \langle vec \rangle \mid \langle signOp \rangle \langle vecExp \rangle \mid \langle exp \rangle \texttt{*} \langle vecExp \rangle \mid \langle vec \rangle \langle linOp \rangle \langle vecExp \rangle \\
&\quad \langle vecExp \rangle \texttt{.norm(} \langle int \rangle \texttt{)} \\
\langle mat \rangle &::= \texttt{Mat} \langle nat \rangle \texttt{x} \langle nat \rangle \texttt{(} \langle expList \rangle \texttt{)} \\
\langle matExp \rangle &::= \langle mat \rangle \mid \langle signOp \rangle \langle matExp \rangle \mid \langle exp \rangle \langle linOp \rangle \langle matExp \rangle \mid \\
&\quad \langle vecExp \rangle \langle linOp \rangle \langle matExp \rangle \mid \langle mat \rangle \langle linOp \rangle \langle matExp \rangle \\
\langle anyExp \rangle &::= \langle exp \rangle \mid \langle vecExp \rangle \mid \langle matExp \rangle \mid \langle derivative \rangle \mid \langle invocation \rangle \\
\langle bindings \rangle &::= \langle exp \rangle \texttt{ to } \langle exp \rangle \mid \langle exp \rangle \texttt{ to } \langle exp \rangle \texttt{,} \langle bindings \rangle \\
\langle invocation \rangle &::= \langle anyExp \rangle \texttt{(} \langle bindings \rangle \texttt{)} \\
\langle derivative \rangle &::= \texttt{d(} \langle anyExp \rangle \texttt{) / d(} \langle exp \rangle \texttt{)} \mid \langle anyExp \rangle \texttt{.d(} \langle exp \rangle \texttt{)} \mid \langle anyExp \rangle \texttt{.d(} \langle expList \rangle \texttt{)} \\
\langle gradient \rangle &::= \langle exp \rangle \texttt{.grad()}
\end{aligned}
$$

A.2. Linear regression

Recall the matrix equation for linear regression, where $\mathbf{X} : \mathbb{R}^{m \times n}$ and $\mathbf{\Theta} : \mathbb{R}^{n \times 1}$:

$$\widehat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta}) = \mathbf{X}\mathbf{\Theta} \tag{A.2.1}$$

Imagine we are given the following dataset:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} = \begin{pmatrix} 1 & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ 1 & \dots & x_{mn} \end{pmatrix}, \mathbf{Y} = \begin{bmatrix} y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{A.2.2}$$

Our goal in ordinary least squares (OLS) linear regression is to minimize the loss, or error between the data and the model's prediction:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta}) = ||\mathbf{Y} - \widehat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta})||^2 \tag{A.2.3}$$

$$\mathbf{\Theta}^* = \underset{\mathbf{\Theta}}{\text{argmin}} \mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta}) \tag{A.2.4}$$

A.2.1. Finite difference method

First, we consider the scalar case, where $\widehat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta}) = \hat{f}(x; \theta_2, \theta_1) = \theta_2 x + \theta_1$. Since $\mathbf{X}, \mathbf{Y}$ are considered to be fixed, we can rewrite $\mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta})$ as simply:

$$\mathcal{L}(\mathbf{\Theta}) = \mathcal{L}(\theta_2, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (y_i - (\theta_2 x_i + \theta_1))^2 \tag{A.2.5}$$

To find the minimizer of $\mathcal{L}(\mathbf{\Theta})$, we need $\nabla_{\mathbf{\Theta}} \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial \theta_2}, \frac{\partial \mathcal{L}}{\partial \theta_1}]$. There are various ways to compute this. First, let's see FDM with centered differences:

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \lim_{h \to 0} \frac{\sum_{i=1}^{m} \left(y_i - (\theta_2 x_i + \theta_1 + h)\right)^2 - \sum_{i=1}^{m} \left(y_i - (\theta_2 x_i + \theta_1 - h)\right)^2}{2hm} \tag{A.2.6}$$

$$= \lim_{h \to 0} \frac{1}{2hm} \sum_{i=1}^{m} \left(y_i - (\theta_2 x_i + \theta_1 + h)\right)^2 - \left(y_i - (\theta_2 x_i + \theta_1 - h)\right)^2 \tag{A.2.7}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \lim_{h \to 0} \frac{\sum_{i=1}^{m} \left(y_i - ((\theta_2 + h) x_i + \theta_1)\right)^2 - \sum_{i=1}^{m} \left(y_i - ((\theta_2 - h)\, x_i + \theta_1)\right)^2}{2hm} \tag{A.2.8}$$

$$= \lim_{h \to 0} \frac{1}{2hm} \sum_{i=1}^{m} \left(y_i - ((\theta_2 + h)\, x_i + \theta_1)\right)^2 - \left(y_i - ((\theta_2 - h)\, x_i + \theta_1)\right)^2 \tag{A.2.9}$$

Using computer algebra, the above equations can be simplified considerably:

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \lim_{h \to 0} \frac{1}{2hm} \sum_{i=1}^{m} (4h(\theta_1 + \theta_2 x_i - y_i)) \tag{A.2.10}$$

$$= \boxed{\frac{2}{m} \sum_{i=1}^{m} (\theta_1 + \theta_2 x_i - y_i)} \tag{A.2.11}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \lim_{h \to 0} \frac{1}{2hm} \sum_{i=1}^{m} (4hx_i(\theta_2 x_i + \theta_1 - y_i)) \tag{A.2.12}$$

$$= \boxed{\frac{2}{m} \sum_{i=1}^{m} (x_i)(\theta_2 x_i + \theta_1 - y_i)} \tag{A.2.13}$$

Equation A.2.10: https://www.wolframalpha.com/input/?i=(y_i-((%CE%B8_2%2Bh)x_i%2B%CE%B8_1))%5E2-(y_i-((%CE%B8_2-h)x_i%2B%CE%B8_1))%5E2
Equation A.2.12: https://www.wolframalpha.com/input/?i=(y_i-(%CE%B8_2*x_i%2B%CE%B8_1%2Bh))%5E2%E2%88%92(y_i-(%CE%B8_2*x_i%2B%CE%B8_1-h))%5E2

A.2.2. Partial differentiation

Alternatively, we can calculate the partials analytically, by applying the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial}{\partial \theta_1} \frac{1}{m} \sum_{i=1}^{m} (y_i - (\theta_2 x_i + \theta_1))^2 \tag{A.2.14}$$

$$= \frac{1}{m} \sum_{i=1}^{m} 2(y_i - (\theta_2 x_i + \theta_1)) \frac{\partial}{\partial \theta_1} (y_i - (\theta_2 x_i + \theta_1)) \tag{A.2.15}$$

$$= \frac{2}{m} \sum_{i=1}^{m} (y_i - (\theta_2 x_i + \theta_1))(-1) \tag{A.2.16}$$

$$= \boxed{\frac{2}{m} \sum_{i=1}^{m} (\theta_2 x_i + \theta_1 - y_i)} \tag{A.2.17}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial}{\partial \theta_2} \frac{1}{m} \sum_{i=1}^{m} (y_i - (\theta_2 x_i + \theta_1))^2 \tag{A.2.18}$$

$$= \frac{1}{m} \sum_{i=1}^{m} 2(y_i - (\theta_2 x_i + \theta_1)) \frac{\partial}{\partial \theta_2} (y_i - (\theta_2 x_i + \theta_1)) \tag{A.2.19}$$

$$= \frac{2}{m} \sum_{i=1}^{m} (y_i - (\theta_2 x_i + \theta_1))(-x_i) \tag{A.2.20}$$

$$= \boxed{\frac{2}{m} \sum_{i=1}^{m} (x_i)(\theta_2 x_i + \theta_1 - y_i)} \tag{A.2.21}$$

Notice how analytical differentiation gives us the same answer as the finite difference method (this is not by accident), with much less algebra. We can rewrite these two solutions in gradient form, i.e. as a column vector of partial derivatives:

$$\nabla_{\boldsymbol{\Theta}} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \frac{\partial \mathcal{L}}{\partial \theta_2} \end{bmatrix} = \frac{2}{m} \begin{bmatrix} \sum_{i=1}^{m} (\theta_2 x_i + \theta_1 - y_i) \\ \sum_{i=1}^{m} (x_i)(\theta_2 x_i + \theta_1 - y_i) \end{bmatrix} \tag{A.2.22}$$

A.2.3. Matrix solution

Having reviewed the scalar procedure for linear regression, let us now return to the general form of $\mathcal{L}(\boldsymbol{\Theta})$. Matrix notation allows us to simplify the loss considerably:

$$\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{m} (\mathbf{Y} - \mathbf{X}\boldsymbol{\Theta})^\intercal (\mathbf{Y} - \mathbf{X}\boldsymbol{\Theta}) \tag{A.2.23}$$

$$= \frac{1}{m} (\mathbf{Y}^\intercal \mathbf{Y} - \mathbf{Y}^\intercal \mathbf{X}\boldsymbol{\Theta} - \boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{Y} + \boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{X}\boldsymbol{\Theta}) \tag{A.2.24}$$

$$= \frac{1}{m} (\mathbf{Y}^\intercal \mathbf{Y} - 2\boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{Y} + \boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{X}\boldsymbol{\Theta}) \tag{A.2.25}$$

Matrix notation allows us to derive the gradient and requires far less algebra:

$$\nabla_{\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{m} (\nabla_{\boldsymbol{\Theta}} \mathbf{Y}^\intercal \mathbf{Y} - 2\nabla_{\boldsymbol{\Theta}} \boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{Y} + \nabla_{\boldsymbol{\Theta}} \boldsymbol{\Theta}^\intercal \mathbf{X}^\intercal \mathbf{X}\boldsymbol{\Theta}) \tag{A.2.26}$$

$$= \frac{1}{m} (0 - 2\mathbf{X}^\intercal \mathbf{Y} + 2\mathbf{X}^\intercal \mathbf{X}\boldsymbol{\Theta}) \tag{A.2.27}$$

$$= \boxed{\frac{2}{m} (\mathbf{X}^\intercal \mathbf{X}\boldsymbol{\Theta} - \mathbf{X}^\intercal \mathbf{Y})} \tag{A.2.28}$$

For completeness, and to convince ourselves the matrix solution is indeed the same:

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}1 & \dots & 1\\ x_1 & \dots & x_m\end{bmatrix}}_{\mathbf{X}^\intercal}\underbrace{\begin{bmatrix}1 & x_1\\ \vdots & \vdots\\ 1 & x_m\end{bmatrix}}_{\mathbf{X}}\underbrace{\begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix}}_{\mathbf{\Theta}} - \underbrace{\begin{bmatrix}1 & \dots & 1\\ x_1 & \dots & x_m\end{bmatrix}}_{\mathbf{X}^\intercal}\underbrace{\begin{bmatrix}y_1\\ \vdots\\ y_m\end{bmatrix}}_{\mathbf{Y}}\right) \tag{A.2.29}$$

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}m & \sum_{i=1}^m x_i\\ \sum_{i=1}^m x_i & \sum_{i=1}^m x_i^2\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}}\underbrace{\begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix}}_{\mathbf{\Theta}} - \underbrace{\begin{bmatrix}\sum_{i=1}^m y_i\\ \sum_{i=1}^m x_i y_i\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{Y}}\right) \tag{A.2.30}$$

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}m\theta_1 + \sum_{i=1}^m \theta_2 x_i\\ \sum_{i=1}^m \theta_1 x_i + \sum_{i=1}^m \theta_2 x_i^2\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}\mathbf{\Theta}} - \underbrace{\begin{bmatrix}\sum_{i=1}^m y_i\\ \sum_{i=1}^m x_i y_i\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{Y}}\right) \tag{A.2.31}$$

$$= \boxed{\frac{2}{m}\underbrace{\begin{bmatrix}\sum_{i=1}^m \theta_2 x_i + \theta_1 - y_i\\ \sum_{i=1}^m (x_i)(\theta_2 x_i + \theta_1 - y_i)\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}\mathbf{\Theta} - \mathbf{X}^\intercal\mathbf{Y}} = \begin{bmatrix}\frac{\partial \mathcal{L}}{\partial \theta_1}\\ \frac{\partial \mathcal{L}}{\partial \theta_2}\end{bmatrix} = \nabla_{\mathbf{\Theta}}\mathcal{L}(\mathbf{\Theta})} \tag{A.2.32}$$

Notice how we recover the same solution obtained from partial differentiation and finite difference approximation, albeit in a more compact form. For a good introduction to matrix calculus, the textbook by Magnus and Neudecker [1988] is an excellent guide, of which Petersen et al. [2012] offer a review of important identities.

OLS linear regression is a convex optimization problem. If $\mathbf{X}^\intercal\mathbf{X}$ is invertible, i.e. full-rank, this implies a unique solution $\mathbf{\Theta}^*$, which we can solve for directly by setting $\nabla_{\mathbf{\Theta}}\mathcal{L} = \mathbf{0}$:

$$\mathbf{0} = \mathbf{X}^\intercal\mathbf{X}\mathbf{\Theta} - \mathbf{X}^\intercal\mathbf{Y} \tag{A.2.33}$$

$$\mathbf{\Theta} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{Y} \tag{A.2.34}$$

Solving this requires computing $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ which is at least $\mathcal{O}(n^{2.373})$[Williams, 2014] to the best of our knowledge, i.e. quadratic with respect to the number of input dimensions. Another way to find $\mathbf{\Theta}^*$ is by initializing $\mathbf{\Theta} \leftarrow \mathbf{0}$ and repeating the following procedure until convergence:

$$\mathbf{\Theta}' \leftarrow \mathbf{\Theta} - \alpha\nabla_{\mathbf{\Theta}}\mathcal{L}(\mathbf{\Theta}) \tag{A.2.35}$$

Typically, $\alpha \in [0.001, 0.1]$. Although hyperparameter tuning is required to find a suitable α (various improvements like Nesterov momentum [Nesterov, 2013] and quasi-Newton methods also help to accelerate convergence), this procedure is guaranteed to be computationally more efficient than matrix inversion for sufficiently large m and n. In practice, the normal equation is seldom used unless m is very small.

A.3. Polynomial regression

Polynomial regression is a straightforward application of linear regression, where the weights are the coefficients for each term in a polynomial. Consider the univariate case:

$$y_i \;=\; \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m = \sum_{j=0}^{m} \beta_j x_i^j \tag{A.3.1}$$

We can rewrite this function in matrix form as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} \tag{A.3.2}$$

$$\widehat{\mathbf{f}}_{PR}(\mathbf{X}; \boldsymbol{\beta}) = \mathbf{X}\boldsymbol{\beta} \tag{A.3.3}$$

The resemblance to Equation A.2.1 should be clear. To find $\boldsymbol{\beta}$ minimizing $\mathcal{L}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\beta})$, we can use the same method described in § A.2.3.

Appendix B

Tools for reproducible robotics

B.1. Useful Docker resources

The following resources have proven particularly helpful during the development of Duckietown's container infrastructure.

B.1.1. Balena

Balena is a very good source of base images for ARM devices. The best part of using Balena images, is that they can be rebuilt on x86 devices, such as a laptop or cloud server. Baked into every Balena image is a shim for the shell that will allow users to run ARM binaries on x86 from inside a container. To use this feature, the following `Dockerfile` template is provided:

```
FROM balena/BASE_IMAGE # e.g. raspberrypi3-python
RUN [ "cross-build-start" ]
# ARM-specific code goes here...
RUN [ "cross-build-end" ]
CMD <DEFAULT_START_COMMAND>
```

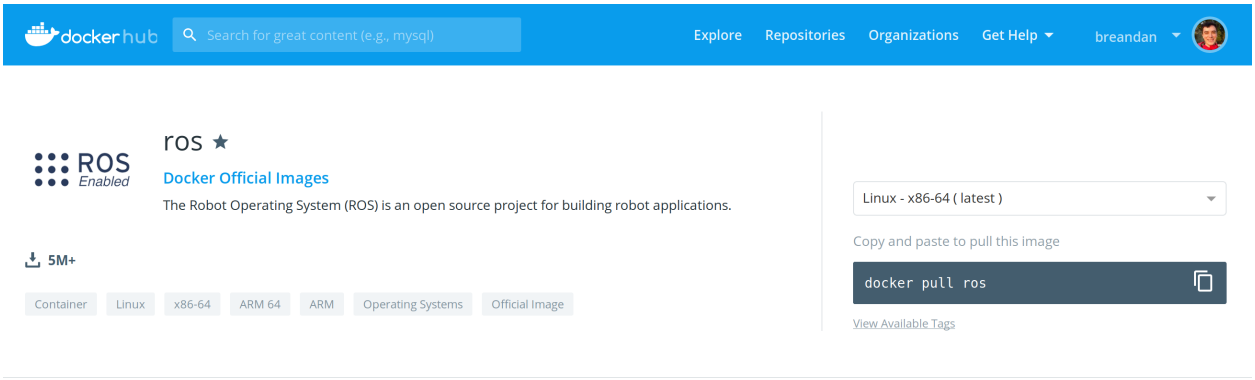
Balena uses QEMU [Bellard, 2005] to cross-build images.[1] When running an ARM image, simply use the `qemu-arm-static` binary as a custom entrypoint:

```
~$ docker run --entrypoint=qemu-arm-static -it your/arm-image bash
```

[1] https://www.balena.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/

B.1.2. ROS Docker Images

ROS.org builds nightly ARM and x86 images for robotics development. For each distribution, there are packages like `core`, `base`, `perception` (including OpenCV), `robot` and others.



B.1.3. Hypriot

Hypriot, a base OS for RPi and other ARM devices, includes support for Docker straight out of the box. Hypriot is a lightweight Raspbian-based Linux distribution which builds from the latest Raspberry Pi kernels and Raspbian releases.

B.1.4. PiWheels

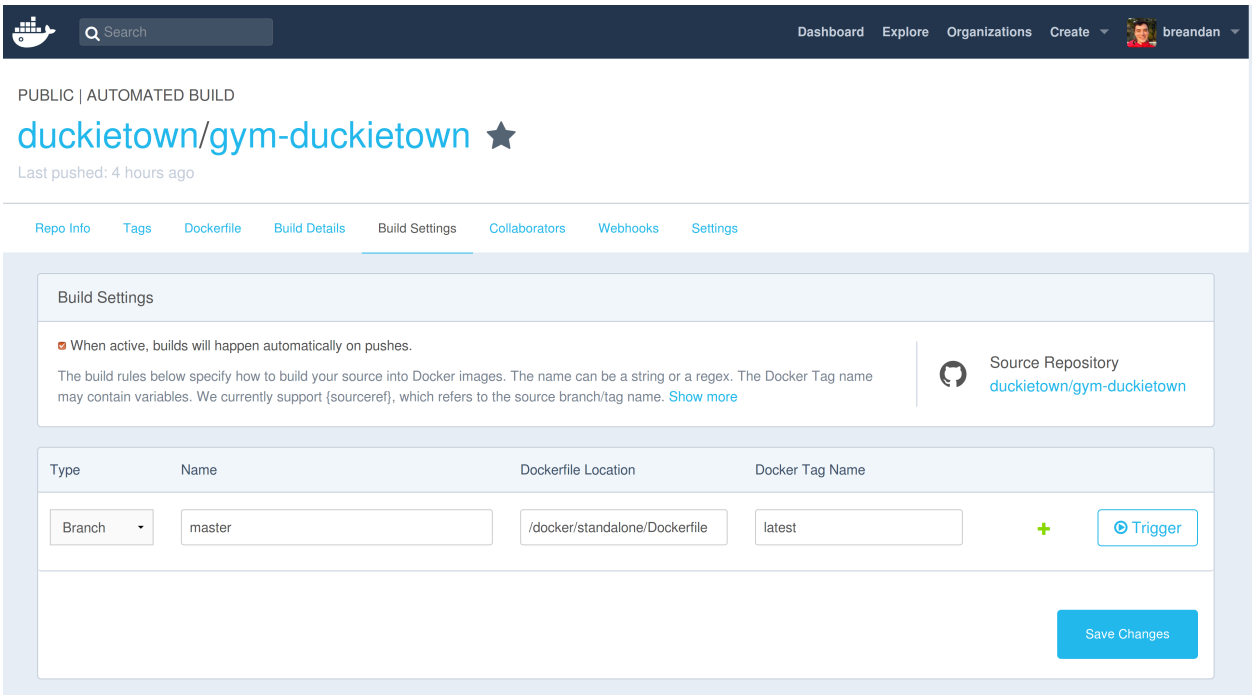
Not all Python packages (especially if they wrap a native library) can be run on all platforms. One might be tempted to build some package from its sources (and in rare cases, they might need to do so). But there is a good chance the package has already been compiled for Raspberry Pi on PiWheels. By using the following command (either in a `Dockerfile` or via the CLI), various Python packages may be installed, e.g. `opencv-python`:

```
~$ pip install opencv-python --index-url https://www.piwheels.org/simple
```

B.1.5. Docker Hub

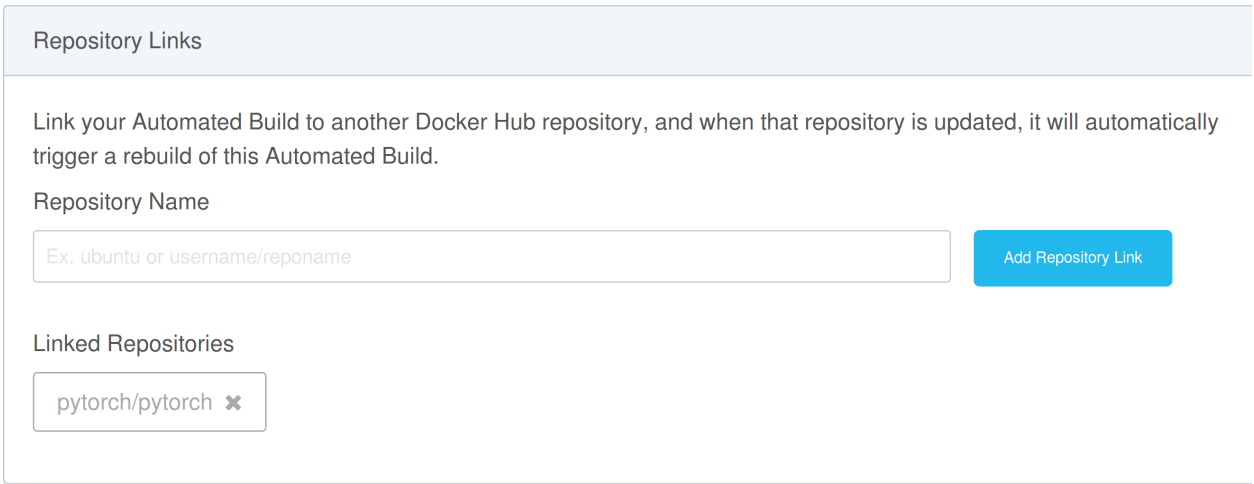
Docker Hub is the central repository for Docker Images. Unless a separate registry has been configured, whenever users pull a Docker image tag, it will first query the Docker Hub for a matching image. The Docker Hub can be used to upload Docker images, and configure

automated builds from GitHub (with a two hour build timeout). Docker Hub does not support layer caching of any kind, so the build will always take a fixed amount of time.



Docker Hub auto-builds support linking a `Dockerfile` in a GitHub repository, and whenever that `Dockerfile` changes, the Docker image will be updated.

The Docker Hub also has features for configuring repository links and build triggers. These will automatically rebuild downstream Docker images whenever some event occurs.



Repository links allow support chaining builds together across Docker Hub repositories. Whenever a linked repository is updated, the dependent image will be rebuilt.

B.1.6. Docker Cloud

Docker Cloud is a Docker registry which is fully integrated with the Docker Hub. Builds are automatically published from Docker Cloud to Docker Hub. Notifications for email and Slack, as well as longer build timeouts (up to 4-hours) are supported. Docker Cloud also supports more advanced build options than Docker Hub, such as a configurable build context and cache settings.