Université de Montréal

Programming tools for intelligent systems

with a case study in autonomous robotics

par

Breandan Considine

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Discipline

avril 2019

© Breandan Considine, Année de la thése

Summary

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1

Introduction

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

–Wikipedia

In computer science, we are chiefly concerned with algorithmic complexity. Computer scientists have developed powerful tools for describing algorithmic complexity using function analysis and information theory, such as big-O notation, Kolmogorov complexity and entropy. The field of software engineering is primarily interested in a different measure of complexity - the complexity of building software, the digital manifestation of algorithms on physical hardware. One sort of software complexity is the mental effort required to understand a program's source code, and which can be approximated by metrics such as cyclomatic complexity or Halstead complexity. By any measure of complexity, today's software is more intelligent than ever before, but shows few signs of becoming any more easily intelligible. As software becomes more intelligent and unpredictable, we need better tools for understanding the complexity of software systems.

Our objective is to reduce the mental effort required to build intelligent systems, using developer tools, programming language abstractions, automated testing, and containers.

One may ask, what is an intelligent system, and how is it different from ordinary software? Let us provide the following loose definition to guide our discussion:

An intelligent system is a software system which is capable of solving tasks for which it was not explicitly programmed, and which human experts were previously incapable of solving by hard-coding domain-specific rules. Typically, although not always these systems have the following properties:

(1) learn generalizable rules by processing large amounts of data

(2) possess a large number of flexible parameters (thousands to billions)

(3) are able to meet or exceed the performance of a well-trained human.

In recent years, computer science and software engineering has made large strides in building such systems. These breakthroughs were the direct result of fundamental progress in neural network algorithms and representation learning. Also key to this whole endeavor was the adoption of open source methods, largely to the credit of the software engineering community, who pioneered the development of software libraries for automatic differentiation like Theano [**2**], Torch [**8**] and Caffe [**20**], and who democratized many popular simulators, benchmarks and datasets through platforms like GitHub, Kaggle and OpenAI.

The goal of this thesis is to show it is possible to build unintelligent software tools that facilitate the process of programming intelligent systems, and which reduce the mental effort required to understand a intelligent program. First, we demonstrate an integrated development environment that assists users when writing robotics applications. This project was created to reduce the complexity of working with robotics code. Next, we demonstrate a type-safe domain specific language for differentiable programming, an emerging paradigm in deep learning. To test this application, we use a set of techniques borrowed from property-based testing [**12**]. Finally, we use Docker containers [**28**] to automate the process of building, testing and deploying reproducible robotics applications to heterogeneous hardware platforms. To demonstrate the usefulness of our toolset, we create an intelligent system comprised of a mobile autonomous vehicle and an Android mobile application.

1.1. Stages in the software development lifecycle

In traditional software engineering, the Waterfall Method is a classical software process model that comprises of five stages. We propose contributions to four: design, implementation, testing and maintenance.

1.1.1. Designing intelligent systems

In software design, designers, requirements engineers and developers must work in concert to produce a system that is well-designed, meets the needs of its users, and is highly performant. Sometimes this means compromising requirements, or redesigning a system to

become more flexible. Sometimes this means changing the requirements to accommodate engineering challenges, or refactoring the solution to remove technical debt. Software engineering is a multi-objective optimization process, and in order to produce software that approximates the criteria of all stakeholders, developers must be able to iterate on their ideas rapidly, and integrate feedback from users. Yet today's software systems are larger and more unwieldy than ever before. Some of that complexity can be managed by abstraction, but some of the complexity which developers face cannot be solved by abstraction alone.

Let us consider the mechanical process of writing software with a keyboard and a computer screen.

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 can provide completion, refactoring and source code navigation. And they can 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 the developer's attention from menial tasks. Rather than being forced to concentrate on the structure and organization of text, if developers are able manipulate code at a semantic level, they will be much happier and more productive. By automating common tasks in software development, these purely mechanical tools allow developers to focus on the fundamentals of writing and understanding programs.

In some ways, developing intelligent systems is no different than classical software engineering. The same key principles and best-practices in software engineering are also applicable to intelligent systems. The same activities that guide software engineering, from analysis, design, implementation, verification and maintenance will continue to play an important role when building intelligent systems. But in other ways, the generic tools we use for developing handwritten software will require domain-specific extensions in order for machine learning to become a truly first-class citizen in the next generation of intelligent software systems. Towards that end, we propose Hatchery, a plugin for building applications for the Robot Operating System (ROS), a popular robotics middleware. At the time of its release, Hatchery was the first ROS plugin for the IntelliJ Platform, a popular IDE for C/C++, Python and Android. While the idea is simple, its prior absence and subsequent adoption in the

community suggests these kinds of tools fill a much needed gap in the development of a complex software framework, particularly for robotics applications.

1.1.2. Implementation: Languages and compilers

When implementing intelligent systems, we need to think carefully about languages and abstractions. If developers are implementing backpropogation by hand, they will have little time to think about the high-level characteristics of these systems. This is no different from traditional software engineering - we need to choose the right abstractions for the task at hand. With machine learning, the necessity of choosing appropriate abstractions for reasoning is even more pressing.

1.1.3. Testing: Verification and validation

Most naturally arising phenomena, particularly those related to vision, planning and locomotion are high dimensional creatures. Richard Bellman famously coined this problem as the "curse of dimensionality". Our physical universe is populated by problems which are simple to pose, but impossible to solve inside of it. Claude Shannon, a contemporary of Bellman, calculated the number of unique chess games to exceed 10^{120}, more than the number of atoms in the universe by approximately 40 orders of magnitude [**36**]. At the time, it was believed that such problems would be insurmountable without a fundamental change in algorithms or computing machinery. Indeed, while Bellman or Shannon did not live to see the day, it took only half a century of progress in computer science before solutions to problems with the same order of complexity, first approximated in the Cambrian explosion 541 million years ago, were solved to a competitive margin in modern computers.

While computer science has made enormous strides in solving the common cases, Bellman's curse of dimensionality still haunts the corners of modern machine learning, particularly for distributions that are highly disperse. Because the dimensionality of many real world problems we would like to solve is intractably large, it is difficult to be certain about the behavior of a candidate solution. According to some studies, a human driver averages 1.09 fatalities per hundred million miles [**21**]. A new software build for an autonomous vehicle would need to accumulate 8.8 billion miles of real world driving in order to approximate the fatality rate of a human driver to within %20 with a 95% confidence interval. Deploying such a scheme in the real world would be logistically, not to mention ethically problematic.

Realistically, machine learning needs better ways to practice its skills and probe the effectiveness of a candidate solution within a limited computational budget, ideally without harming anyone in the process. The goal of testing is to highlight errors, but ultimately to provide feedback to the system. In software engineering, the real system under test is the ecosystem of humans and machines which provide each other's means of subsistence. Critical to this arrangement that we have an external testing mechanism which enforces a minimum bar of rigor, typically some form of user acceptance testing or simulator-based testing. If the testing system is not somehow opposed to the system under test, an intelligent system can deceive itself, which is neither in the system's nor its users' best interests.

1.1.4. Software maintenance and reproducibility

In many ways, machine learning shares the same fundamental issues as traditional software maintenance, with dependency management, source code management, documentation and so forth. We can imagine the current process of training a deep learning model as a very long compilation step that requires a great deal of trial-and-error. However, it differs in that the user-facing code for the program is a very high level meta-language that does not directly correspond to the computational performed in a particular instance, but rather configures a directed acyclic graph with randomly initialized edge weights for further optimization. With emerging techniques in meta-learning, even the structure of this computation graph is not directly specified by the user so much as loosely parameterized, and optimized by a search procedure.

While hardware manufacturers have been producing a variety of custom silicon, unlike general purpose programming, deep-learning is a fundamentally hardware-agnostic model of computation. As long as a computer can add and multiply, it has the capability to train and run a deep neural network. However due to the variety of hardware platforms and associated software churn, many machine learning models can be frustratingly to reproduce on different hardware, even when using the same source code and dependencies. While there are some emerging methods for a truly portable model format, if we are not careful about how we design our models, they may simply not run, or produce very different results when run on different hardware. Most open source software libraries are produced by competing industry vendors, with competing hardware platforms and incompatible standards. These

include TensorFlow, ONNX, NNEF, OpenVINO. While some have tried to leverage existing compiler frameworks such as Haskell and LLVM, there appears to be little recent convergence in the community.

At the same time, we need to ship software, and while there may be issues with

1.1.5. Case Study

Ideally, it is best when the users' interests and the engineers' interest are well-aligned, because the product tends to be much higher quality. In the best case scenario, a system's engineers are a true subset of the users (either by choice or necessity), which leads to a virtuous cycle of improvement. Termed "dogfooding" [**18**], this practice was first publicized in the dog food industry, and later adopted in the software industry due to its success. While engineers must often create products for a fundamentally different audience, or species, it is possible to create systems where both overlap.

Chapitre 2

Design: Programming tools for robotics systems

> "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."
>
> –J. C. R. Licklider, *Man-Computer Symbiosis* [**23**]

Programming tools are a bicycle for the mind. Complex systems need better tools for developers.

2.1. Software architecture of a robotics application

https://github.com/duckietown/hatchery

2.2. Structure of a ROS application

The Robot Operating System (ROS) is a popular middleware for robotics applications. At its core, ROS provides software infrastructure for distributed messaging, but more broadly encompasses a set of community-developed libraries and graphical plugins for building robotics applications. While ROS is not an operating system in the traditional sense, it extends certain operating systems features like shared memory and inter-process communication for robotics development. Unlike pure message-oriented middleware like DDS and ZMQ, in addition to the message broker, ROS provides specific features for building decentralized robotic systems, particularly those which are capable of mobility.

According to one community census in 2018, 55% of ROS applications on GitHub are written in C/C++, followed by Python at around 25% [**17**].

2.3. Foundations of a modern IDE

To build an IDE, one should have a few things. First, is an IDE, which we shall refer to as IDE_0. Assume that IDE_0 exists. In order to build a new IDE, IDE_1, load the source code from IDE_0 into IDE_0. Then, using IDE_0, modify the source code so that it compiles. Once it compiles, call it IDE_1. Iterate inside IDE_0 as necessary.

While valid, this approach has some disadvantages. First, most IDEs are already quite large and take a long time for users to download and install. Since most new functionality is small by comparison, modern IDEs have adopted a modular design, which allows them to load packages, i.e. plugins, as needed. So most developers can skip the first step, and build such a plugin, using IDE_0 directly. It is still convenient to have the source code for reference purposes, but in most cases this is read-only.

2.3.1. The parser

An IDE consists of a few important components. First is a parser. This is typically not an ordinary parser, because most of the time a user is using the IDE, the code is invalid. So this parser needs to have some special features.

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

2.3.2. Refactoring

Refactoring support is implemented.

2.3.3. Running and debugging

Assistance for running ROS applications.

2.4. More ROS Tools

Detecting and managing ROS installations.

Chapitre 3

Implementation: languages and compilers

"Programs must be written for people to read, and only incidentally for machines to execute."

–Harold Abelson and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs, Preface to the First Edition* [**?**]

In this chapter, we will discuss the theory and implementation of a type safe domain specific language for automatic differentiation (AD). AD is useful for gradient descent and has a variety of applications in numerical optimization and machine learning. The key idea behind AD is fairly simple. A small set of primitive operations form the basis for all modern computers: addition, subtraction, multiplication and division - by composing these operations over the real numbers in an orderly fashion, one can compute any computable function. In machine learning, it is often the case we are given a computable function, i.e. a program[1], that does not work properly. Perhaps the function was implemented poorly. Or perhaps the implementation was correct, but the function was not given the right set of inputs - had it been, then the program would have produced a better answer. Regardless of the implementation, we would like to determine how to change the input slightly, so as to produce a more suitable output.

[1] n.b. Not all programs are computable functions, but all computable functions are programs.

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 $P : \mathbb{R} \to \mathbb{R}$, composed of a series of nested functions:

$$P(p_0) = p_n \circ p_{n-1} \circ p_{n-2} \circ ... \circ p_1 \circ p_0 \tag{3.1.1}$$

From the chain rule of calculus, we know that:

$$\frac{dP}{dp_0} = \prod_{i=1}^{n} \frac{dp_i}{dp_{i-1}} \tag{3.1.2}$$

Likewise, if we have a function $Q(q_0, q_1, \ldots, q_n) : \mathbb{R}^n \to \mathbb{R}$, the gradient ∇Q tells us:

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

More generally, if we have a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian $\mathbf{J}$ is defined like so:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{3.1.4}$$

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 x, no matter how we update x, the output stays the same. We say that such functions have zero gradient for that input.

$$(\nabla F)(x) = \mathbf{0} \tag{3.1.5}$$

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

$$|\nabla F| = \pm\infty \tag{3.1.6}$$

It is an open question whether non-differentiable functions exist in the real world [**6**]. At the current physical (10nm) and temporal (10ns) scale of modern computing, there exist no

such functions, but modern computers are not equipped with the capability to accurately report the true value of their discrete approximations, so for all intents and purposes, programs implemented by most physical computers are discrete functions. Nevertheless, computers are capable of approximating bounded subsets of R^n to arbitrary precision given enough time and space. For most ordinary applications, a fixed precision approximation is sufficient.

There exists at the heart of machine learning a theorem which states a simple class of functions, which compute a weighted sum of non-linear functions composed with a linear function, can approximate any bounded function on $\mathbb{R}^m$ to arbitrary precision. More precisely, the universal approximation theorem **[19]** states that for all continuous functions $f : C(\mathbb{I}_m)$, where $\mathbb{I}_m = [0, 1]^m$, there exists a function F, parameterized by constants $n \in \mathbb{N}, \beta, b \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$ and $W \in \mathbb{R}^{m \times n}$:

$$\begin{aligned} F(x) &= \beta\varphi\left(W^T x + b\right) \\ \forall x \in I_m,\ &|F(x) - f(x)| < \varepsilon \end{aligned} \tag{3.1.7}$$

This theorem does not put an upper bound on the constant n, or how to find w, somewhat limiting its practical applicability. But for reasons which are not yet fully understood, empirical results suggest it is possible to obtain reasonably precise approximations to many naturally-arising functions, once thought to be intractable, in a comparatively short time by composing these non-linear and linear functions in an alternating fashion and iteratively updating w in the direction suggested by $\nabla_w F(x)$.

3.2. Differentiable programming

The renaissance of modern deep learning is widely credited to progress in three research areas: algorithms, data and hardware. Among algorithms, most research has focused on deep learning architectures and representation learning. Equally important, arguably, is the role that automatic differentiation (AD) has played in facilitating the implementation of these ideas. Prior to the adoption of general-purpose AD libraries such as Theano, PyTorch and TensorFlow, manual derivation was required. The emergence of these and similar libraries simplified and accelerated the pace of gradient-based machine learning, allowing researchers to focus on network architectures and learning representations. Some of these ideas in turn,

formed the basis for new methods in automatic differentiation, which continues to be an active area of research.

An key aspect of deep learning is learning representations via gradient descent. For gradient descent to work, the representation must be differentiable almost everywhere. However many representations are non-differentiable in their natural domain. For example, the structure of language in its written form is not easily differentiable as small changes to a word's symbolic representation can cause large shifts to its semantic meaning. A key insight from deep learning is that many discrete problem domains can be mapped to a smoother latent space. For example, if we imagine words as a vector of real numbers, then we can learn a mapping from their discrete, character-based form to their vector-based representation such that the semantic relations between words (as measured by their statistical co-occurrence in large language corpora) are geometrically preserved in vector space [**33**]. Many classes of discrete problems can be relaxed to continuous surrogates by introducing a good representation.

As more domains found efficient vector representations, researchers observed that neural networks were part of a broader class of differentiable architectures that could be built and interpreted in a manner not unlike computer programming [1], [2]. Hence the term *differentiable programming* was born. Today, differentiable programing has found applications in a wide range of domains, including protein folding [**3**], to physics engines [**9, 10**] and from graphics rendering [**25**] to meta-learning [**24**]. These domains have well-studied models for characterizing the dynamics of various systems, with unknown parameters that can be learned via gradient descent. Traditionally, handcrafted optimization algorithms were required to tune these parameters, but differentiable programming promises to do this, more or less automatically.

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 [**34**].

Dynamically typed languages are commonly used for experimentation and prototyping. But are they scalable to production systems?

According to some studies, type errors account for over 15% of bugs [**14**]. While the causal connections between statically typed languages in general and fewer is not widely established, types are often necessary to build more powerful static analyses and tools for program understanding.

Strong, static types are important for reasoning about the behavior of complex programs. Statically typed languages offer a number of benefits to users, such as eliminating of a broad class of runtime errors by virtue of the language alone. Furthermore, a carefully designed statically typed API can eliminate the potential for incorrect API usages by enforcing certain usage patterns. Statically typed languages also provide several benefits for static code analysis, as tools can offer more relevant autocompletion suggestions, and provide early warnings for compile and probable runtime errors.

3.4. Imperative and functional languages

Most programs written today are written in imperative languages, due in part to the prevalence of the Turing Machine and Von Neumann architecture. λ-calculus provides an equivalent [2] language for computing, which we argue, is a more natural way to express mathematical functions and calculate their derivatives. In pure imperative programming the sole purpose of a function is to pass it values, but we have no way to refer to the function itself, or for example, to write a function which takes another function as input. More troubling in the case of automatic differentiation, is that imperative programs have mutable state, which is not the case in mathematics, and which requires taking extra precautions when calculating mathematical derivatives.

In functional programming, function composition is a very natural pattern. To take the derivative of a function composed by another function, we simply apply the chain rule. Since there is no mutable state, we do not require any exotic data structures or compiler tricks to keep track of the state.

For example, consider the vector function $f(l_1, l_2) = l_1 \cdot l_2$, seen in 3.1. Imperative programs, by allowing mutation, are destroying information. In order to recover that computation graph for reverse mode AD, we need to either override the assignment operator,

[2] In the sense that λ-calculus is Turing Complete.

Imperative	Functional

`fun dot(l1, l2) {`
`  if (len(l1) != len(l2))`
`  return error`
`  var sum = 0`
`  for(i in 0 to len(l1))`
`  sum += l1[i] * l2[i]`
`  return sum`
`}`
	`fun dot(l1, l2) {`
	`  return if (len(l1) != len(l2))`
	`  error`
	`  else if (len(l1) == 0) 0`
	`  else`
	`  head(l1) * head(l2) +`
	`  dot(tail(l1) + tail(l2))`
	`}`

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

or use a tape to store the intermediate values, which is quite tedious. In pure functional programming mutable variables do not exist, which makes our lives much easier.

3.5. Kotlin

Kotlin is a strong, statically typed language. It is well suited for building cross-platform applications, with implementations in native, JVM, and JavaScript.

3.6. Kotlin∇

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [**15**] in Java. This result was strengthened to prove Java's type system is Turing complete [**16**]. As a practical consequence, we can use the same technique to perform shape-safe automatic differentiation (AD) in Java, using type-level programming. A similar technique is feasible in any language with generic types. We use *Kotlin*, whose type system is less expressive, but fully compatible with Java.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano [**2**], Torch [**8**], and TensorFlow [**1**]. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ [**32**], DiffSharp in F# [**4**] and recently Swift [**22**]. However, the majority of existing automatic differentiation (AD) frameworks use a loosely-typed DSL, and few offer shape-safe tensor operations in a widely-used programming language.

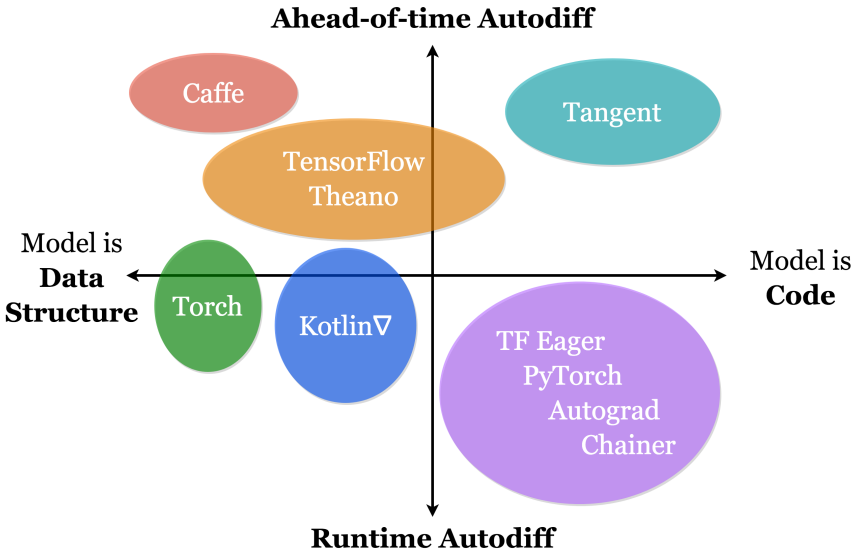

Fig. 3.1. Adapted from [**38**]. Kotlin∇ models are data structures, evaluated at runtime.

Existing AD implementations for the JVM include Lantern [**41**], Nexus [**7**] and DeepLearning.scala [**5**], 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 contains a number of desirable features for implementing a native AD framework. In addition to type-safety and interoperability, 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 and coroutines** support functional differentiable programming with lambdas and shift-reset continuations, following [**31**] and [**41**].
- **Extension functions** support extending classes with new fields and methods and can be exposed to external callers without requiring sub-classing or inheritance.

Kotlin∇ models are embedded domain specific languages (EDSLs), which are essentially just data structures masquerading as code. These structures may look and act like code, but are really just functions building an abstract sytax tree (AST), which can be evaluated eagerly or lazily depending on the developer's needs. Typically these ASTs represent simple state machines, but they can also be used to implement a full-fleged programming langauge. Popular examples include SQL/LINQ [**26**], OptiML [**37**] and other fluent interfaces [**13**]. In

a sufficiently expressive host language, one can implement any language as a library, without needing to write a lexer, parser, compiler or interpreter. And with a carefully designed type system, users will automatically receive code completion and static analysis from their favorite developer tools. Many have observed that functional programming languages are suitable host languages [**11, 35**], perhaps due to the concept of code as data and data as code[3].

3.7. Usage

Kotlin∇ allows users to implement differentiable programs by composing operations on numerical fields to form algebraic expressions. Expressions are lazily evaluated inside a numerical context, which may imported on a per-file basis or lexically scoped for finer-grained control over the runtime behavior.

Listing 3.1. Simple code listing.

```
import edu.umontreal.kotlingrad.numerics.DoublePrecision

with(DoublePrecision) { // Use double-precision protocol
  val x = variable("x") // Declare immutable vars (these
  val y = variable("y") // are just symbolic constructs)
  val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazy exp
  val dz_dx = d(z) / d(x) // Leibniz derivative notation
  val d2z_dxdy = d(dz_dx) / d(y) // Mixed higher partial
  val d3z_d2xdy = grad(d2z_dxdy)[x] // Indexing gradient
  plot3D(d3z_d2xdy, -1.0, 1.0) // Plot in -1 < x,y,z < 1
}
```

Fig. 3.2. A basic Kotlin∇ program with two inputs and one output.

Above, we define a function with two variables and take a series of partial derivatives with respect to each variable. The function is numerically evaluated on the interval $(-1, 1)$ in each dimension and rendered in 3-space. We can also plot higher dimensional manifolds (e.g. the loss surface of a neural network), projected into four dimensions, and rendered in three, where one axis is represented by time.

[3] i.e. homoiconicity, notably introduced by Lisp, one of the first functional programming languages

$$z = \sin\big(10(x * x + y^2)\big)/10,\ \texttt{plot}(\tfrac{\partial^3 z}{\partial x^2 \partial y})$$

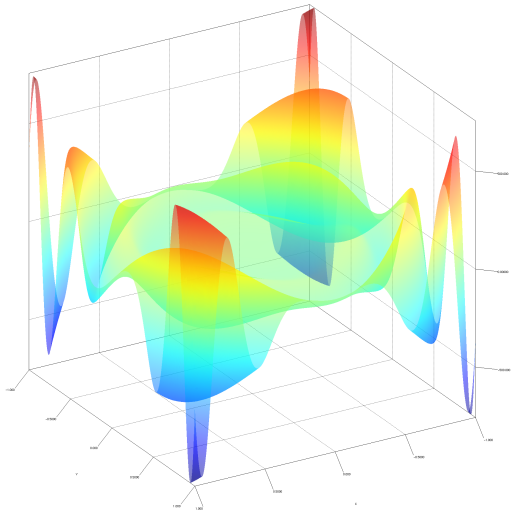

Fig. 3.3. Output generated by the program shown in Figure 3.2.

Kotlin∇ treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed to form a data-flow graph (DFG). An expression is simply a `Function`, which is only evaluated once invoked with numerical values, e.g. `z(0, 0)`. In this way, Kotlin∇ is similar to other compiled graph based approaches like TensorFlow and Theano.

Listing 3.2. Simple code listing.

```
val z = sin(10 * (x * x + pow(y, 2))) / 10
```

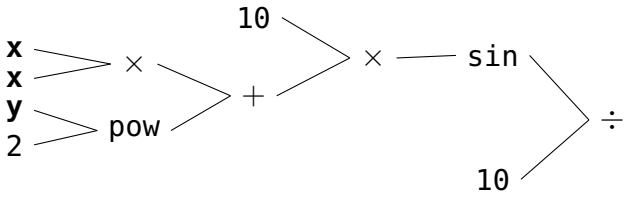

Fig. 3.4. Implicit DFG constructed by the original expression, `z`.

Kotlin∇ supports shape-safe tensor operations by encoding tensor rank as a parameter of the operand's type signature. By enumerating type-level integer literals, we can define tensor operations just once using the highest literal, and rely on Liskov substitution to preserve shape safety for subtypes.

Listing 3.3. Shape safe tensor addition for rank-1 tensors, $\forall L \leq 2$.

```
// Literals have reified values for runtime comparison
open class ‘0‘(override val value: Int = 0): ‘1‘(0)
open class ‘1‘(override val value: Int = 1): ‘2‘(1)
class ‘2‘(open val value: Int = 2) // Greatest literal
// <L: ‘2‘> will accept L <= 2 via Liskov substitution
class Vec<E, L: ‘2‘>(len: L, cts: List<E> = listOf())
// Define addition for two vectors of type Vec<Int, L>
operator fun <L: ‘2‘, V: Vec<Int, L>> V.plus(v: V) =
Vec<Int, L>(len, cts.zip(v.cts).map { it.l + it.r })
// Type-checked vector addition with shape inference
val Y = Vec(‘2‘, listOf(1, 2)) + Vec(‘2‘, listOf(3, 4))
val X = Vec(‘1‘, listOf(1, 2)) + Vec(‘3‘) // Undefined!
```

It is possible to enforce shape-safe vector construction as well as checked vector arithmetic up to a fixed `L`, but the full implementation is omitted for brevity. A similar pattern can be applied to matrices and higher rank tensors, where the type signature encodes the shape of the operand at runtime.

With these basic ingredients, we have almost all the features necessary to build an expressive shape-safe AD, but unlike prior implementations using Scala or Haskell, in a language that is fully interoperable with Java, while also capable of compiling to JVM bytecode, JavaScript, and native code.

In future work, we intend to implement a full grammar of differentiable primitives including matrix convolution, control flow and recursion. While Kotlin∇ currently implements arithmetic manually, we plan to wrap a BLAS such as cuBLAS or native linear algebra library for performance.

3.8. Type system

Describing the Kotlin∇ type system (formally).

3.9. Testing

Kotlin∇ claims to eliminate certain runtime errors, but how do we know the proposed implementation is not incorrect? One method, borrowed from the Haskell community, is called

property-based testing (PBT), closely related to metamorphic testing. Notable implementations include QuickCheck, Hypothesis and ScalaTest (ported to Kotlin in KotlinTest). PBT uses algebraic properties to verify the result of an operation by constructing semantically equivalent but syntactically distinct expressions, which should produce the same answer. Kotlin∇ uses two such equivalences to validate its AD implementation:

(1) **Symbolic differentiation**: manually differentiate and compare the values returned on a subset of the domain with AD.

(2) **Finite difference approximation**: sample space of symbolic (differentiable) functions, comparing results of AD to FD.

For example, consider the following test, which checks whether the manual derivative and the automatic derivative, when evaluated at a given point, are equal to each other within the limits of numerical precision:

Listing 3.4. Simple code listing.

```
val x = Var("x")
val y = Var("y")

val z = y * (sin(x * y) - x)            // Function under test
val ‘     zx     ‘ = d(z) / d(x)               // Automatic derivative
val manualDx = y * (cos(x * y) * y - 1) // Manual derivative

"  z  /  x  ␣should␣be␣y␣*␣(cos(x␣*␣y)␣*␣y␣-␣1)" {
    assertAll(NumericalGenerator, NumericalGenerator) {    ,     ->
    // Evaluate the results at a given seed
    val autoEval = ‘     zx     ‘(x to    , y to    )
    val manualEval = manualDx(x to    , y to    )
    // Should pass if   (adEval, manualEval) <
    autoEval shouldBeApproximately manualEval
    }
}
```

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

```
"d(sin␣x)/dx␣should␣be␣equal␣to␣(sin(x␣+␣dx)␣-␣sin(x))␣/␣dx" {
```

```
    assertAll(NumericalGenerator) {     ->
        val f = sin(x)

        val ‘ d f dx ‘ = d(f) / d(x)
        val adEval = ‘ d f dx ‘(   )

        val dx = 1E-8
        // Since     is a raw numeric type, sin => kotlin.math.sin
        val fdEval = (sin(    + dx) - sin(   )) / dx
        adEval shouldBeApproximately fdEval
    }
}
```

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

3.10. Shape safety

Shape safety is an important concept in Kotlin∇. There are three broad strategies for handling shape errors:

(1) Hide the error somehow by implicitly reshaping or broadcasting arrays

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

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

In Kotlin∇, we use the last strategy to check the shape of tensor operations. Consider the following program:

```
// Inferred type: Vec<Int, ‘2‘>
val a = Vec(1.0, 2.0)
// Inferred type: Vec<Int, ‘3‘>
val b = Vec(1.0, 2.0, 3.0)

val c = b + b
// Does not compile, shape mismatch
// a + b
```

Attempting to sum two vectors whose shapes do not match will fail to compile, and they must be explicitly resized.

```
// Inferred type: Mat<Double, '1', '4'>
val a = Mat('1', '4', 1.0, 2.0, 3.0, 4.0)
// Inferred type: Mat<Double, '4', '1'>
val b = Mat('4', '1', 1.0, 2.0, 3.0, 4.0)

val c = a * b
// Does not compile, inner dimension mismatch
// a * a
// b * b
```

Similarly, attempting to multiply two tensors whose inner dimensions do not match will fail to compile.

```
val a = Mat('2', '4',
    1.0, 2.0, 3.0, 4.0,
    5.0, 6.0, 7.0, 8.0
)
val b = Mat('4', '2',
    1.0, 2.0,
    3.0, 4.0,
    5.0, 6.0,
    7.0, 8.0
)

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

Explicit types are optional but encouraged. Type inference helps preserve shape information over long programs.

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

```
fun someMatFun(m: Mat<Double, `2`, `2`>) = ...
```

When writing a function, it is mandatory to declare the input type(s), but the return type may be omitted. Shape-safety is currently supported up to rank-2 tensors, i.e. matrices.

While first-class dependent types are useful for ensuring arbitrary shape safety (e.g. when concatenating and reshaping matrices), they are unnecessary for simple equality checking (such as when multiplying two matrices)[4]. When the shape of a tensor is known at compile time, it is possible to encode this information using a less powerful type system, as long as it supports subtyping and parametric polymorphism. In practice, we can implement a shape-checked tensor arithmetic in languages like Java, Kotlin, C++, C or Typescript, which accept generic type parameters. In Kotlin, whose type system is less expressive than Java, we use the following strategy.

First, we enumerate a list of integer type literals as a chain of subtypes, so that `0 <: 1 <: 2 <: 3 <: ... <: C`, where `C` is the largest fixed-length dimension we wish to represent. Using this encoding, we are guaranteed linear growth in space and time for subtype checking. C can be specified by the user, but they will need to rebuild this project from scratch.

```
open class `0`(override val i: Int = 0): `1`(i) { companion object: `0`(), Nat<`0`> }
open class `1`(override val i: Int = 1): `2`(i) { companion object: `1`(), Nat<`1`> }
open class `2`(override val i: Int = 2): `3`(i) { companion object: `2`(), Nat<`2`> }
open class `3`(override val i: Int = 3): `4`(i) { companion object: `3`(), Nat<`3`> }
//...This is generated
sealed class `100`(open val i: Int = 100) { companion object: `100`(), Nat<`100`> }
interface Nat<T: `100`> { val i: Int } // Used for certain type bounds
```

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

[4]Many less powerful type systems are still capable of performing arbitrary computation in the type checker. As specified, Java's type system is known to be Turing Complete. It may be possible to emulate a limited form of dependent types in Java by exploiting this property, although this may not computationally tractable due to the practical limitations noted by Grigore [**16**].

```
infix operator fun <C: ‘100‘, V: Vec<Float, C>> V.plus(v: V): Vec<Float, C> =
    Vec(length, contents.zip(v.contents).map { it.first + it.second })
```

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

```
open class Vec<E, MaxLength: ‘100‘> constructor(
    val length: Nat<MaxLength>,
    val contents: List<E> = listOf()
) {
    operator fun get(i: ‘100‘): E = contents[i.i]
    operator fun get(i: Int): E = contents[i]
    companion object {
        operator fun <T> invoke(t: T): Vec<T, ‘1‘> = Vec(‘1‘, listOf(t))
        operator fun <T> invoke(t0: T, t1: T): Vec<T, ‘2‘> = Vec(‘2‘, listOf(t0, t1))
        operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, ‘3‘> = Vec(‘3‘, listOf(t0, t1, t2))
        //...
    }
}
```

The initializer may be omitted in favor of dynamic construction, although this may fail at runtime. For example:

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

A similar syntax is possible for matrices and higher-rank tensors. For example, Kotlin∇ can infer the shape of multiplying two matrices, and will not compile if their inner dimensions do not match:

```
// Inferred type: Mat<Int, ‘4‘, ‘4‘>
val l = Mat(‘4‘, ‘4‘,
1, 2, 3, 4,
5, 6, 7, 8,
9, 0, 0, 0,
9, 0, 0, 0
```

```
)
// Inferred type: Mat<Int, ‘4‘, ‘3‘>
val m = Mat(‘4‘, ‘3‘,
1, 1, 1,
2, 2, 2,
3, 3, 3,
4, 4, 4
)

// Inferred type: Mat<Int, ‘4‘, ‘3‘>
val lm = l * m
// m * m // Does not compile
```

A similar technique is possible in Haskell, which is capable of a more powerful form of type-level computation, type arithmetic. Type arithmetic makes it easy to express convolutional arithmetic and other arithmetic operations on shape variables (say, splitting a vector in half), which is currently not possible, or would require enumerating every possible combination of type literals.

3.11. Operator overloading

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

For example, suppose we have an interface `Group`, which overloads the operators + and ×, and is defined like so:

Listing 3.5. Simple code listing.

```
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 quantification to ensure that operations return the concrete type variable T, rather than something

more generic like `Group`. Imagine a class `Expr` which has implemented `Group`. It can be used as follows:

Listing 3.6. Simple code listing.

```
fun <T: Group<T>> cubed(t: T): T = t * t * t
fun <E: Expr<E>> twiceExprCubed(e: E): E = 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

With 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 (typed FP). A number of recent papers have demonstrated the expressiveness of this paradigm for automatic differentiation.

In Kotlin∇, all expressions can be treated as functions. For example:

Listing 3.7. Simple code listing.

```
fun <T: Group<T>> makePoly(x: Var<T>, y: Var<T>) = x * y + y * y + x * x

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

Currently, it is only possible to represent functions where all inputs and outputs share a single type. In future iterations, it is possible to extend support for building functions with varying input/output types and enforce constraints on both, using covariant and contravariant type bounds.

3.13. Coroutines

Coroutines are a generalization of subroutines for non-preemptive multitasking, typically implemented using continuations. One form of continuation, known as shift-reset a.k.a. delimited continuations, are sufficient for implementing reverse mode AD with operator overloading alone (without any additional data structures) as described by Wang et al. in Shift/Reset the Penultimate Backpropagator [**40**] and later in Backpropagation with Continuation Callbacks [**39**]. Delimited continuations can be implemented using Kotlin coroutines and they are currently a work in progress.

3.14. Extension Functions

Extension functions augment external classes with new fields and methods. Via context oriented programming, Kotlin can expose its custom extensions (e.g. in `DoublePrecision`) to consumers without requiring subclasses or inheritance.

Listing 3.8. Using extension functions we can provide numerical conversions for common data types, wrapped by a Context.

```
data class Const<T: Group<T>>(val number: Double) : Expr()
data class Sum<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
data class Prod<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()

class Expr<T: Group<T>>: Group<Expr<T>> {
    operator fun plus(addend: Expr<T>) = Sum(this, addend)
    operator fun times(multiplicand: Expr<T>) = Prod(this, multiplicand)
}

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

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

Listing 3.9. Simple code listing.

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

Extensions can also be defined in another file or context and imported on demand. This approach was borrowed from KMath [**29**], another mathematical library for Kotlin.

3.15. Algebraic data types

Algebraic data types (ADTs) in the form of sealed classes (a.k.a. sum types) allows creating a closed set of internal subclasses to guarantee an exhaustive control flow over the concrete types of an abstract class. At runtime, we can branch on the concrete type of the abstract class. For example, suppose we have the following classes:

Listing 3.10. Users are forced to handle all subclasses when branching on the type of a sealed class, as incomplete control flow will not compile (instead of say, failing silently at runtime).

```
sealed class Expr<T: Group<T>>: Group<Expr<T>> {
   fun diff() = when(expr) {
       is Const -> Zero
       // Smart casting allows us to access members of a checked typed without explicit casting
       is Sum -> e1.diff() + e2.diff()
       // Product rule: d(u*v)/dx = du/dx * v + u * dv/dx
       is Prod -> e1.diff() * e2 + e1 * e2.diff()
       is Var -> One
       // Since the subclasses of Expr are a closed set, no ‘else -> ...‘ is required.
    }

    operator fun plus(addend: Expr<T>) = Sum(this, addend)
    operator fun times(multiplicand: Expr<T>) = Prod(this, multiplicand)
}

data class Const<T: Group<T>>(val number: Double) : Expr()
data class Sum<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
data class Prod<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
class Var<T: Group<T>>: Expr()
class Zero<T: Group<T>>: Const<T>
class One<T: Group<T>>: Const<T>
```

Smart-casting allows us to treat the abstract type Expr as a concrete type, e.g. Sum after performing an is Sum check. Otherwise, we would need to write (expr as Sum).e1

in order to access its field, e1. Performing a cast without checking would throw a runtime exception, if the type were incorrect. Using sealed classes helps avoid casting, thus avoiding ClassCastExceptions.

3.16. Multiple Dispatch

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

Building on the previous example, a common task in AD is to simplify a graph. This is useful in order to minimize the number of calculations required, or to improve numerical stability. We can eagerly simplify expressions based on algebraic rules of replacement. Smart casting allows us to access members of a class after checking its type, without explicitly casting it:

Listing 3.11. 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.

```
override fun times(multiplicand: Function<X>): Function<X> =
    when {
        this == zero -> this
        this == one -> multiplicand
        multiplicand == one -> this
        multiplicand == zero -> multiplicand
        this == multiplicand -> pow(two)
        this is Const && multiplicand is Const -> const(value * multiplicand.value)
        // Further simplification is possible using rules of replacement
        else -> Prod(this, multiplicand)
    }

val result = Const(2.0) * Sum(Var(2.0), Const(3.0))
//         = Sum(Prod(Const(2.0), Var(2.0)), Const(6.0))
```

3.17. Shape-safe Tensor Operations

While first-class dependent types are useful for ensuring arbitrary shape safety (e.g. when concatenating and reshaping matrices), they are unnecessary for simple equality checking

(such as when multiplying two matrices).* When the shape of a tensor is known at compile time, it is possible to encode this information using a less powerful type system, as long as it supports subtyping and parametric polymorphism (a.k.a. generics). In practice, we can implement a shape-checked tensor arithmetic in languages like Java, Kotlin, C++, C or Typescript, which accept generic type parameters. In Kotlin, whose type system is less expressive than Java, we use the following strategy.

First, we enumerate a list of integer type literals as a chain of subtypes, so that 0 <: 1 <: 2 <: 3 <: ... <: C, where C is the largest fixed-length dimension we wish to represent. Using this encoding, we are guaranteed linear growth in space and time for subtype checking. C can be specified by the user, but they will need to rebuild this project from scratch.

Listing 3.12. Simple code listing.

```
open class `0`(override val i: Int = 0): `1`(i) { companion object: `0`(), Nat<`0`> }
open class `1`(override val i: Int = 1): `2`(i) { companion object: `1`(), Nat<`1`> }
open class `2`(override val i: Int = 2): `3`(i) { companion object: `2`(), Nat<`2`> }
open class `3`(override val i: Int = 3): `4`(i) { companion object: `3`(), Nat<`3`> }
//...This is generated
sealed class `100`(open val i: Int = 100) { companion object: `100`(), Nat<`100`> }
interface Nat<T: `100`> { val i: Int } // Used for certain type bounds
```

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

Listing 3.13. Simple code listing.

```
infix operator fun <C: `100`, V: Vec<Float, C>> V.plus(v: V): Vec<Float, C> =
    Vec(length, contents.zip(v.contents).map { it.first + it.second })
```

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

Listing 3.14. Simple code listing.

```
open class Vec<E, MaxLength: ‘100‘> constructor(
    val length: Nat<MaxLength>,
    val contents: List<E> = listOf()
) {
    operator fun get(i: ‘100‘): E = contents[i.i]
    operator fun get(i: Int): E = contents[i]

    companion object {
        operator fun <T> invoke(t: T): Vec<T, ‘1‘> = Vec(‘1‘, arrayListOf(t))
        operator fun <T> invoke(t0: T, t1: T): Vec<T, ‘2‘> = Vec(‘2‘, arrayListOf(t0, t1))
        operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, ‘3‘> = Vec(‘3‘, arrayListOf(t0, t1,
             t2))
        //...
    }
}
```

The initializer may be omitted in favor of dynamic construction, although this may fail at runtime. For example:

Listing 3.15. Simple code listing.

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

A similar syntax is possible for matrices and higher-rank tensors. For example, Kotlin can infer the shape of multiplying two matrices, and will not compile if their inner dimensions do not match:

Listing 3.16. Further examples are provided for shape-safe matrix operations such as addition, subtraction and transposition.

```
// Inferred type: Mat<Int, ‘4‘, ‘4‘>
val l = Mat(‘4‘, ‘4‘,
    1, 2, 3, 4,
    5, 6, 7, 8,
    9, 0, 0, 0,
    9, 0, 0, 0
```

```
)
// Inferred type: Mat<Int, ‘4‘, ‘3‘>
val m = Mat(‘4‘, ‘3‘,
    1, 1, 1,
    2, 2, 2,
    3, 3, 3,
    4, 4, 4
)

// Inferred type: Mat<Int, ‘4‘, ‘3‘>
val lm = l * m
// m * m // Does not compile
```

A similar technique is possible in Haskell, which is capable of a more powerful form of type-level computation, type arithmetic. Type arithmetic makes it easy to express convolutional arithmetic and other arithmetic operations on shape variables (say, splitting a vector in half), which is currently not possible, or would require enumerating every possible combination of type literals.

Math	Infix	Prefix	Postfix	Type
$A + B$	`a + b` `a.plus(b)`	`plus(a, b)`		$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^\pi) \to (\mathbb{R}^? \to \mathbb{R}^\pi)$
$A - B$	`a - b` `a.minus(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)`		$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*n}, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{n*p}) \to (\mathbb{R}^? \to \mathbb{R}^{m*p})$
$\frac{A}{B}$ AB^{-1}	`a / b` `a.div(b)`	`div(a, b)`		$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*n}, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{p*n}) \to (\mathbb{R}^? \to \mathbb{R}^{m*p})$
$-A$ $+A$		`-a` `+a`	`a.unaryMinus()` `a.unaryPlus()`	$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi) \to (\mathbb{R}^\tau \to \mathbb{R}^\pi)$
A+1 A-1	`a + 1` `a - 1`	`++a` `-a`	`a++, a.inc()` `a-, a.dec()`	$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*m}) \to (\mathbb{R}^\tau \to \mathbb{R}^{m*m})$

sin(a)
cos(a)
tan(a)
		`sin(a)`
		`cos(a)`
		`tan(a)`
			`a.sin()`
			`a.cos()`
			`a.tan()`
				$(\mathtt{a} : \mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$

$\ln(A)$		`ln(a)` `log(a)`	`a.ln()` `a.log()`	$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*m}) \to (\mathbb{R}^\tau \to \mathbb{R}^{m*m})$
$\log_b A$	`a.log(b)`	`log(a, b)`		$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*m}, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{m*m}) \to (\mathbb{R}^? \to \mathbb{R})$
A^b	`a.pow(b)`	pow(a, b)		$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*m}, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}) \to (\mathbb{R}^? \to \mathbb{R}^{m*m})$
$\sqrt{a}$ $sqrt[3](a)$	`a.pow(1.0/2)` `a.root(3)`	`a.pow(1.0/2)` `a.root(3)`	`a.sqrt()` `a.cbrt()`	$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m*m}) \to (\mathbb{R} \to \mathbb{R}^{m*m})$
$\frac{da}{db}$ $a'(b)$	`a.diff(b)`	`grad(a)[b]`	`d(a) / d(b)`	$(\mathtt{a} : C(\mathbb{R}^m)^*, \mathtt{b} : \mathbb{R} \to \mathbb{R}) \to (\mathbb{R}^m \to \mathbb{R})$
∇a		`grad(a)`	`a.grad()`	$(\mathtt{a} : C(\mathbb{R}^m)^*) \to (\mathbb{R}^m \to \mathbb{R}^m)$

Chapitre 4

Verification and validation

> "If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere... then we had better be quite sure the purpose put into the machine is the purpose which we really desire."
>
> –Norbert Weiner, *Some moral and technical consequences of automation* **[43]**

4.1. Adversarial test case generation

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

4.2. Background

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

$$P(x) = p_n \circ p_{n-1} \circ p_{n-2} \circ ... \circ p_1 \circ p_0 \tag{4.2.1}$$

From the chain rule of calculus, we know that:

$$\frac{dP}{dp_0} = \prod_{i=1}^{n} \frac{dp_i}{dp_{i-1}} \tag{4.2.2}$$

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

$$T : \forall x \in (0, 1), P(x) < C \tag{4.2.3}$$

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

$$D_T : \{x^i \sim \mathbb{R}(0,1) \mid P(x^i) \implies \neg T\}, maximize|D_T| \tag{4.2.4}$$

If we have no information about the program implementation or its input distribution, D_P, we can do no better than random search [**44**]. However, if we know something about the input distribution, we could re-parameterize the distribution to incorporate our knowledge. Assuming the program has been tested on common inputs, we might consider sampling $x \sim \frac{1}{D_P}$ for inputs that are infrequent. If we knew how P were implemented, we could prioritize our search in input regions leading towards internal discontinuities (e.g. edge cases in software testing). However for functions that are continuous and differentiable, these heuristics are almost certainly insufficient.

Another strategy, independent of how candidate inputs are selected, is to use some form of gradient based optimization in the search procedure. For example in (3) we could have a loss function:

$$\mathcal{L}(P, x) = C - P(x) \tag{4.2.5}$$

The gradient of the loss w.r.t. x (assuming P is fixed[1]) is:

$$\nabla\mathcal{L}(P, x) = -\frac{dP}{dx} \tag{4.2.6}$$

Where the vanilla gradient update step is defined as:

$$x^i_{n+1} = x^i_n - \alpha\frac{dP}{dx} \tag{4.2.7}$$

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

[1]In contrast with backpropogation, where the parameters are updated.

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

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

$D_T = []$;

evalCount = 0;

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

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

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

    append x^i *to* D_T

  else

    $n = 0$;

    $x_n^i = x^i$;

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

      $n{+}{+}$;

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

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

        append x_n^i *to* D_T

        break;

      end

      evalCount++;

    end

  end

  evalCount++; i++;

end

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

4.3. Regression testing: a tool to prevent catastrophic forgetting

An endemic problem in modern deep learning is the problem of forgetting. In order to combat this issue, we turn to a classic software testing tool: regression testing.

Regular regression testing gives clear diagnostics about the behavior of intelligent systems.

Chapitre 5

Software Maitenance and Reproducibility

In this chapter, we will discuss the challenges of software reproducibility and how best practices in software engineering like continuous integration and delivery can help researchers mitigate the variability associated with building and running software. Docker [**28**] is one technical solution we will discuss. Our work is roughly related to determinism, and does not consider the variability associated with training data and statistical variability.

One of the challenges of building intelligent systems and programming at large, is the problem of reproducibility. Software reproducibility has a number of subtle aspects, including hardware compatibility, operating systems, file systems, build reproducibility, and runtime determinism. While at one point in the history of computing, it may have been sufficient to write down a program and feed it directly into a computer, the source code of a modern program is far too removed from its mechanical implementation to be meaningfully executed in isolation. Today's handwritten programs are like the schematics for a traffic light. Built by compilers, interpreted by virtual machines, communicating with other programs, running inside an operating system, they are essentially meaningless abstractions without a factory to build them, a city-worth of infrastructure, cars, and traffic laws.

As necessary in any good schematic, much of the information required to build a program is divided into layers abstractions. Most of the instructions executed by a computer between when a programmer starts a program and the output becomes visible are not intended to be read by the programmer, and have long since been automated and forgotten. In a modern programming language like Java, C# or Python, the total information required to compile and run a simple program numbers in the trillions of bits. A portion of the 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. And for most programs, such as those found in a typical GitHub repository, a vanishingly small fraction corresponds to the handwritten program itself.

Researchers would often like to reproduce the work of other researchers, but the mental effort of re-implementing their abstractions can sometimes be tedious and detrimental towards scientific progress. It is often necessary to utilize programs written by other researchers, and it would be convenient if we had better tools for reproducibility and incremental development. This is the same problem software developers have been attempting to solve for many years, via the open source community. But source control alone is insufficient, since these tools are primarily intended for text. While text-based representations may be stable for a time, as software is updated and rebuilt, important details about the original development environment can be misplaced. To reproduce the program in its entirety, we need a snapshot of all digital information present on the computer at the time of its execution. Short of that, the minimal set of dependencies for running a program is required.

5.1. Operating systems and virtualization

In 2006, Linux began introducing a variety of new kernel features for controlling groups of processes, under the aegis of **cgroups** [**27**]. Collectively, these features supported a kind of lightweight virtualization, where a fully virtual machine was no longer necessary to get many of the benefits of VMs, primarily resource control and namespace isolation. 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 the host OS and sibling containers. While the overhead of running VMs often limits their deployment to server-class hardware, container's are comparatively lightweight, which enables them to run on a far broader class of mobile and embedded platforms.

5.2. Dependency management

One common source of software variability are packages. In fact, operating systems have dealt with this is Dependency hell is NP Complete: https://research.swtch.com/version-sat

5.3. Containerization

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

In order to address the issue of software reproducibility, we developed a set of tools and development workflows that draw on best practices in software engineering. These tools are primarily built around containerization, a widely adopted virtualization technology in the software industry. In order to lower the barrier of entry for participants and minimize variability across platforms (e.g. simulators, robotariums, Duckiebots), we provide a state-of-the-art container infrastructure based on Docker, a popular container engine. Docker allows us to construct versioned deployment artifacts that represent the entire filesystem and to manage resource constraints via a sandboxed runtime environment.

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

Docker containers are sandboxed runtime environments that are portable, reproducible and version controlled. Each environment contains all the software dependencies necessary to run the packaged application(s), but remains isolated from the host OS and file system. Docker provides a mechanism to control the resources each container is permitted to access,

[1]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/`.

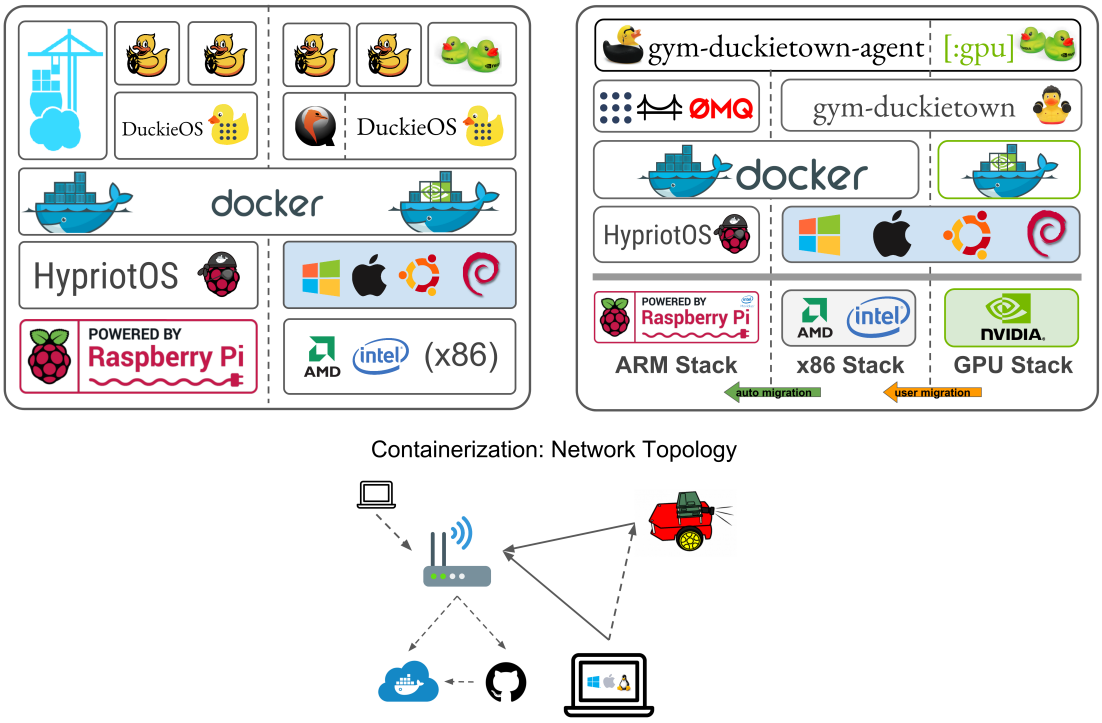

Fig. 5.1. AI-DO container infrastructure. Left: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are typically trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their specific architecture, may be run on an ARM device using an Intel NCS.

and a separate Linux namespace for each container, isolating the network, users, and file system mounts. Unlike virtual machines, container-based virtualization like Docker only requires a lightweight kernel, and can support running many simultaneous containers with close to zero overhead. A single Raspberry Pi is capable of supporting hundreds of running containers.

While containerization considerably simplifies the process of building and deploying applications, it also introduces some additional complexity to the software development lifecycle. Docker, like most container platforms, uses a layered filesystem. This enables Docker to take an existing "image" and change it by installing new dependencies or modifying its functionality. Images may be based on a number of lower layers, which must periodically be updated.

Care must be taken when designing the development pipeline to ensure that such updates do not silently break a subsequent layer as described earlier in Sec. **??**.

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

5.4. Docker and ROS

Prior work has explored the Dockerization of ROS containers [**42**]. This work forms the basis for our own, which is extended specifically for the Duckietown platform [**30**].

Chapitre 6

Case study: application for autonomous robotics

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

6.1. Design

Designed with Hatchery.

6.2. Implementation

Implementation includes Kotlin∇

6.3. Verification and validation

Verified using property-based testing.

6.4. Containerization

Deployed and CI-tested using Docker.

Chapitre 7

Conclusion

7.1. Future work

7.1.1. Requirements Engineering

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

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

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

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

When building intelligent system developers must first ask, what are the requirements of the system? This process is often the most troublesome part, because the requirements must not be fuzzy specifications like traditional software engineering, but precise, programmable directives. "I would like the system to be fast," is not sufficiently precise. These kinds of requirements must be translated into statistical loss functions, so we must be very precise about how we specify our requirements. If we simply say, "The system must produce a valid response as quickly as possible, in less than 100ms," is better, but leaves open the possibility of returning an empty response.

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

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

7.1.2. Continuous Delivery and Continual Learning

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

Incremental updates will grow increasingly smaller, until the program starts to alter itself after every input it processes.

We need tools to more effectively harness the stochasticity of these learning systems.

7.1.3. Developers, Operations, and the DevOps toolchain

Software engineers have begun to realize the value of bespoke tools that facilitate the process of shipping software, in addition to the software itself.

Teams building software are cybernetic systems, and require meta-programs for building code and organizational processes which enable them to ship code more efficiently.

Bibliography

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

[2] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

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

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: Automatic differentiation library. *CoRR*, abs/1511.07727, 2015.

[5] Yang Bo. Deep L earning.scala: A simple library for creating complex neural networks. 2018.

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

[7] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017.

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

[9] 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.

[10] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. *CoRR*, abs/1611.01652, 2016.

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

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

[13] M. Fowler. Fluent interface, 2005.

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

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

[16] Radu Grigore. Java generics are T uring C omplete. pages 73–85, 2017.

[17] Martin Guenther. Are serious things done with ros in python? - discourse.ros.org. `https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6`. (Accessed on 04/12/2019).

[18] Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006.

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[21] 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.

[22] Chris Lattner and Richard Wei. Swift for tensorflow. 2018.

[23] Joseph Carl Robnett Licklider. Man-computer symbiosis. *IRE transactions on human factors in electronics*, (1):4–11, 1960.

[24] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[25] Matthew M Loper and Michael J Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

[26] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[27] Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium*, volume 2, pages 45–57. Citeseer, 2007.

[28] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[29] Alexander Nozik. Kotlin - new language for scientific programming. In *Proceedings of 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Mar 2019.

[30] 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.

[31] Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: L ambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

[32] Barak A Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make automatic differentiation sound and efficient. pages 79–90, 2008.

[33] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[34] 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.

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

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

[37] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language

for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

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

[39] Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *Advances in Neural Information Processing Systems*, pages 10180–10191, 2018.

[40] Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228*, 2018.

[41] Fei Wang, Xilun Wu, Gr é gory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018.

[42] Ruffin White and Henrik Christensen. Ros and docker. In *Robot Operating System (ROS)*, pages 285–307. Springer, 2017.

[43] Norbert Wiener. Some moral and technical consequences of automation. *Science*, 131(3410):1355–1358, 1960.

[44] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.