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

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 mostly concerned with algorithmic complexity. Computer scientists have developed numerous tools for describing complexity using analysis and information theory, such as Kolmogorov complexity and big-O notation. The field of software engineering is primarily interested in a different kind of complexity - the complexity of building software. One sort of software complexity is the mental effort required to understand a program, and can be approximated by metrics like cyclomatic complexity or Halstead complexity, which try to characterize the mental effort required to work with code.

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

The goal of this thesis is show it is possible to develop tools that reduce the mental burden of building intelligent software systems. 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 as raw text. 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 [**4**]. We then use Docker containers [**?**]to automate the process of building,

testing and deploying reproducible robotics applications to heterogeneous hardware platforms. Finally, we demonstrate a functional intelligent system built using the above tools which is 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 classic engineering design process model that comprises of five stages. We propose contributions to four: design, implementation, verification and maintenance.

1.1.1. Design

When designing intelligent systems, we need to iterate between requirements and design constraints. It is not sufficient for requirements engineers to hand an objective to the design team, or the design team to hand a design to the implementors. Rather, they must work in concert to produce a system that has the desired properties. Sometimes this means compromising, or redesigning the performance metrics to become more flexible.

1.1.2. Implementation

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

1.1.3. Verification

Because the space of many problems is intractably large, it is difficult to prove the correctness of a given solution. Instead, we need tools to verify the properties of a system under a given budget. In self-driving vehicles, human drivers average one fatality per hundred million miles (cite), this is incredibly difficult to test given our current methodology. Instead, we need better ways to probe the effectiveness of a candidate solution.

1.1.4. Maintenance

Maintenance of systems is radically different in machine learning. Often it is not possible to manually update the parameters of a model without starting from scratch. While there are some methods for transfer learning, if we are

Chapitre 2

Design: Programming tools for robotics systems

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

Defining the definition and structure of ROS services, messages, nodes, topics.

2.3. Foundations of a modern IDE

IDEs are more than just a souped-up text editor.

2.3.1. The parser

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. "

–Abelson & Sussman, *Structure and Interpretation of Computer Programs*

In

3.1. 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 [**10**].

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 [**5**]. 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, and guiding users . 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.

3.2. Kotlin

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

3.3. Kotlin∇

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [**6**] in Java. This result was strengthened to prove Java's type system is Turing complete [**7**]. 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, Torch, and TensorFlow. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ [**9**], DiffSharp in F# [**1**] and recently Swift [**12**]. 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.

Existing AD implementations for the JVM include Lantern [**11**], Nexus [**3**] and DeepLearning.scala [**2**], 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 backpropogation with lambdas and shift-reset continuations, following [8] and [11]. **Extension functions** support extending classes with new fields and methods and can be exposed to external callers without requiring sub-classing or inheritance.

3.4. Usage

Kotlin∇ allows users to implement differentiable programs by composing operations on 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. 3.1. 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.

Kotlin∇ treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed recursively 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)`.

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

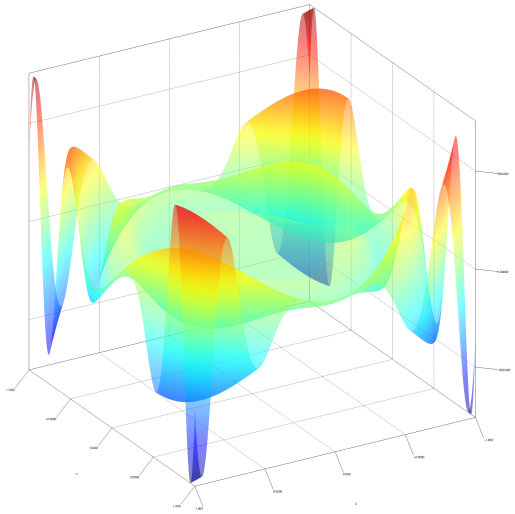

Fig. 3.2. Output generated by the program shown in Figure 3.1.

Listing 3.2. Simple code listing.

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

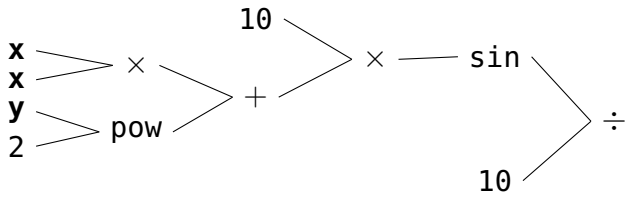

Fig. 3.3. Implicit DFG constructed by the original expression, `z`.

Kotlin∇ supports shape-shafe 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.

3.5. Type system

Describing the Kotlin∇ type system (formally).

3.6. 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 3.4. 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 3.5. 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. 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 3.6. 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. 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 and later in Backpropagation with Continuation Callbacks. Delimited continuations can be implemented using Kotlin coroutines and would be an interesting extension to this work. Please stay tuned! 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 3.7. Simple code listing.

```
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 3.8. 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.

3.7. 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 3.9. Simple code listing.

```
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>
```

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).

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. 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 3.10. Simple code listing.

```
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))
```

This allows us to put all related control flow on a single abstract class which is inherited by subclasses, simplifying readability, debugging and refactoring. 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 3.11. 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 3.12. 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 3.13. 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 3.14. 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 3.15. Simple code listing.

```
// 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
```

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

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.

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.

3.8. 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 like Theano, PyTorch and TensorFlow, gradients needed to be programmed manually. The emergence of these and other libraries specifically tailored to deep learning simplified and accelerated the pace of machine learning, allowing researchers to focus on gradient-based optimization, 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.

In deep learning, research primarily focuses on differentiable representations. In contrast, many problems we would like to solve are non-differentiable in their natural domain. For example, the structure of character-based representations are not easily differentiable as small changes to a word's symbolic representation can cause large differences to its semantic meaning. A key insight from deep learning research is that many problems require posing a good representation. For example, if we can represent words as a vector of real numbers, so that relations between words are spatially preserved by the vector representation, then we can perform gradient descent on such a space. Many classes of discrete problems can be relaxed to continuous surrogates by introducing a good representation.

As more domains found efficient representations, researchers observed 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.

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

How do we test a machine that learns?

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 (cf. no free lunch). However, if we know something about the input distribution, we could re-parameterize the distribution to incorporate our knowledge. Assuming the program already works on most common inputs (e.g. from a training set), we might want to sample $x \sim \frac{1}{D_P}$ for inputs that are infrequent, although this strategy is not guaranteed to yield any additional failures. If we knew how P were implemented, we might be able 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. Finally, we could train a neural network to predict inputs that were likely to cause a program to fail a given specification. As input, the network would take the function and test cases, and as its output, produce values that were likely to violate T.

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:

[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 G.D. steps C are reached before resampling x^{i+1} from the initial sampling strategy S to ensure each G.D. 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 these systems.

Chapitre 5

Case study: application for autonomous robotics

Here, we implement a mobile application using the above toolchain.

5.1. Design

Designed with Hatchery.

5.2. Implementation

Implementation includes Kotlin∇

5.3. Verification

Verified using property-based testing.

5.4. Maintenance

Deployed and CI-tested using Docker.

Chapitre 6

Software reproducibility

In this chapter, we will discuss the challenges of software reproducibility and how containers, CI/CD workflow can help researchers remove this source of variability. Docker [**?**] is one popular container solution we will address.

6.1. Operating systems and virtualization

In 2006, Linux introduced a kernel feature called cgroups. This enabled a kind of lightweight virtualization for containers, where a fully virtualized machine was no longer necessary to get many of the benefits of VMs. This paved the way for the tools that are now known as containers. Unlike VMs, containers do not run their own operating systems, but rather share resources with the host operating system, while remaining isolated from the host and sibling containers. While VMs heavy resource requirements often limits their deployment to server-class hardware, containers' lightweight footprint enabled them to run on a far broader class of mobile and embedded platforms.

6.2. Dependency management

Packages are one source of variability. Dependency hell is NP Complete: https://research.swtch.com/version-sat

6.3. 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.

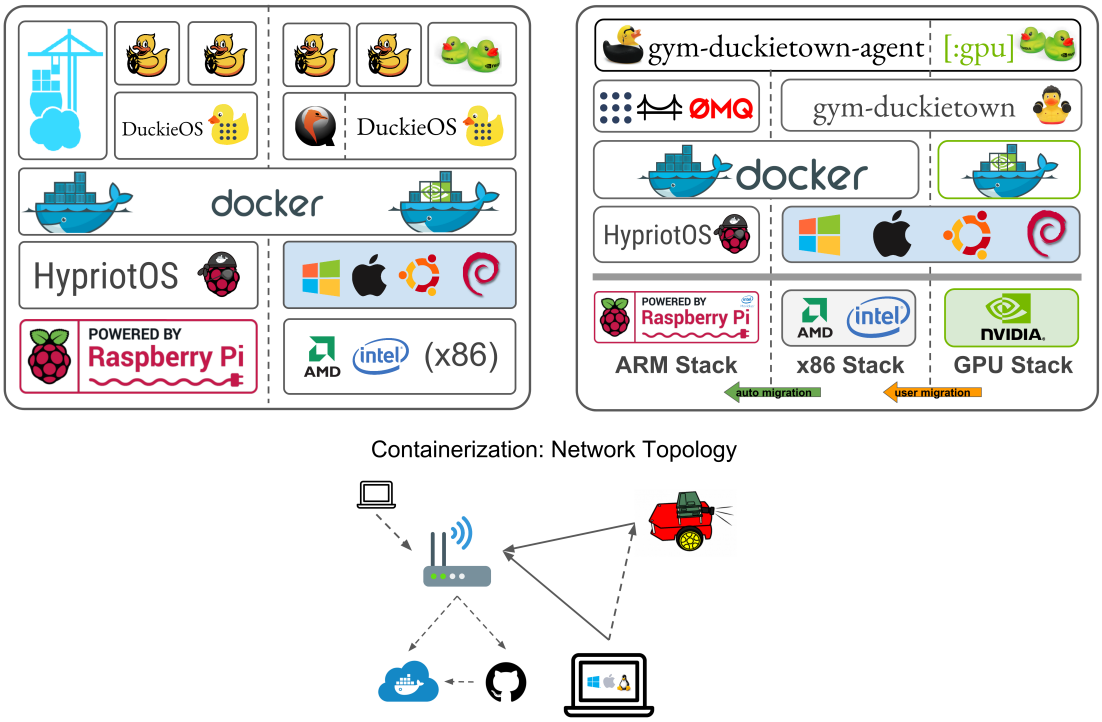

Fig. 6.1. AI-DO container infrastructure. Left: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are typically trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their specific architecture, may be run on an ARM device using an Intel NCS.

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, 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 OS for build purposes. In this case, while submissions may be reproducible, they are not easily modified or inspected. The second method is to

[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/`.

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 also has the benefit of reproducibility and diagnostics. If a competitor requires assistance, troubleshooting becomes much easier when 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 environment periodically, copying sources into the image.

6.4. Docker and ROS

Prior work in Dockerization of ROS [**?**]

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...

Tranditional 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 reimagining 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

In CI/CD, software artifacts are "continously" updated. Similarly, intelligent systems must continously adapt to their environment and this trend will only accelerate.

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] Richard Wei. First-class automatic differentiation in S wift: A manifesto. 2018.

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