A Type System for Multidimensional Arrays

A Type System for Multidimensional Arrays

Theodore Liu

submitted in partial fulfillment of the requirements

for the degree of Bachelor of Arts with honors

in Computer Science

Advisor: Professor Nada Amin

Harvard University

April 2020

Acknowledgements

I would like to thank Professor Nada Amin for advising and encouraging me throughout the project. I would additionally like to thank Professors Stephen Chong and Eddie Kohler for their advising in exceptional circumstances.

I would like to thank all my friends — Richard, Dan, Nenya, Jason, Joanna, everyone who has been with me throughout my Harvard experience — for being the best, most supportive, and amazing friends I could ever ask for.

I would like to thank my parents and my sister, who have been there for me since the very beginning.

Abstract

Python is an extremely popular programming language used by many companies and academics around the world. Historically a slow language, it has gained popularity as an interface with high performance linear algebra, multidimensional array, and machine learning libraries. Initially designed as a strong, dynamically typed language, recent additions to Python have introduced a static type checker; however, the types do not provide enough information to avoid many runtime errors when typing multidimensional array operations. In this thesis, we develop a type system and extension to Python's type system that strengthens the static guarantees of the type checker over multidimensional array operations while preserving the simplicity and ease of use that is core to Python's spirit.

Contents

Chapter 1

Introduction

Python is a strongly and dynamically typed language focused on simplicity and ease of use. Since its invention in the late 1980s, Python has exploded in popularity, seeing rapid adoption in both academia and industry for its flexibility and approachable syntax. An increased interest in scientific computing and machine learning from the Python community led to the creation of packages such as `numpy` and `pytorch`, which expose high level APIs for high performance machine code, circumventing Python's slow execution and enabling rapid development of machine learning and scientific computing applications.

More recently, Python's dynamic type system has been seen as a weakness rather than a strength. Runtime errors caused by type mismatches can be annoying or potentially disastrous for people deploying Python at scale. To combat this, type annotations were added directly to Python's syntax, and a reference typechecker — `mypy` — was developed to statically analyze the annotations. `mypy` eliminates a whole class of type errors from Python code but is not powerful enough to reject multidimensional arrays with incompatible dimensions from being multiplied or applied together, resulting in runtime errors in machine learning or scientific computing code.

In this thesis, we develop a type system and extension to Python's existing type system that allows the expression of function signatures that respect the dimensions of multidimensional arrays as they are passed into functions. This contribution further strengthens the type system and makes runtime errors related to the dimensionality of arrays at function application sites impossible.

Chapter 2 presents background information on type systems, Python, `mypy`, multidimensional arrays, and related work. Chapter 3 presents the new type system for multidimensional arrays, defining syntax and semantics. Chapter 4 evaluates the type system in the context of common array and machine learning operations. Chapter 5 concludes with a reflection on this thesis's contribution.

Chapter 2

Background and Related Work

2.1 Type Systems

A type system in a programming language is a form of program analysis where labels called **types** are assigned to values and expressions. By reasoning about these types, the system can enforce properties about the execution of programs within the language. Types can be thought of as a set of values that a variable or expression can take, and type checking is the process of determining whether a value belongs to the set declared or specified. Type systems can range from the simple to the complex and are usually designed with certain properties in mind. The simply typed lambda calculus introduced by Church only had function types but guaranteed that the left sides of applications were always functions and that every expression had a normal form [10]. In contrast, Rust-lang's ownership and type system is able to not only prove that its programs invoke functions with correctly typed arguments but also that concurrent accesses to memory are safe [6].

In the remainder of this section, we will examine several dichotomies and classes of type systems that will be relevant later in this thesis.

2.1.1 Static versus Dynamic Type Systems

In a **static type system**, the types of a program are checked during the compilation of the program. The type of a function or variable is constant throughout the lifetime of program. The types can be annotated directly or inferred. A static system ensures that no runtime errors relating to the types will occur during the execution of a compiled program. Static type systems benefit greatly from this guarantee since programmers can run compiled code without fear of many errors. The type information during compilation can be used to perform optimizations. Additionally, type information can be stripped away in the final executable, reducing the size of the resulting program. However, static type systems can be difficult to implement since they can require detailed and subtle program analysis, and the performance of a static type checker slows and its complexity grows as the types it seek to support become more complex. C/C++, OCaml, and Rust are examples of languages with static type systems.

In a **dynamic type system**, the types of values are checked during the execution of a program, and variables can be reassigned to multiple different types. Dynamic type systems are good for their flexibility and ease of implementation; there is no need for type checking or inference ahead of execution. However, their lack of runtime guarantees can be difficult to handle, especially in mission-critical software. Without knowing whether a program can fail in a myriad of ways at runtime makes it hard to have confidence in its deployment. Python (traditionally), Ruby, and Javascript are languages with dynamic type systems.

2.1.2 Strong versus Weak Type Systems

A **strong type system** enforces that the type expected at a particular program location exactly matches the type found. In other words, no implicit conversions are done between types to make function calls or assignments well-typed. If there are mismatches found, an error either during compilation or runtime is raised. Strong type systems prevent users from accidentally misusing variables or performing computations they did not intend. Users need to bear the additional cost of explicitly type casting or converting whenever a type *actually* needs to be used as a different type. Python and OCaml are languages with strong type systems.

A **weak type system** attempts to perform certain type conversions implicitly if types do not match as expected. This eliminates the need for constant massaging between various types but can have disastrous consequences when unintended or unknown conversions are performed. As an example, Javascript has an infamous weak type system, leading to bizarre and unexpected results from inocuous statements like `12 + "21" == "1221"`. PHP is another example of a weakly typed language.

2.1.3 Other Type Systems

- **Duck typing** — Duck typing receives its name from the phrase: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." In programming languages, a duck type system does not care about the specific label or class a value belongs to as long as it has certain attributes or implements certain methods. As an example, if we have expression `x.f()` in a duck typing system, it does not matter whether `x` is of class A or B or a primitive type, as long as it has a callable method called `f`. Python is a duck typed language.

- **Gradual Typing** — In a gradually typed language, type annotations are optional. Wherever type annotations are present, type checking is performed. For unannotated code, type checking is either elided or a generic type of *any* is inferred [4]. Gradual type systems allow the incremental adoption of typing in a codebase. Instead of having to type every line of code in a codebase at once, small parts can be annotated and statically checked while the remaining code falls back to runtime checks. Thus, gradual typing achieves a balance between static and dynamic typing. TypeScript and Python with `mypy` are languages with gradual typing.

- **Dependent and Refinement Types** — Dependent type systems extend traditional type systems by allowing values from the language be a part of a type. This allows further specificity of the sets of values that can inhabit a type. As an example, the traditional type `List int` represents the set of all lists of any length that contain only integers. An analagous dependent type might be `List int 4`, denoting the set of integer lists which have exactly length 4. We can take this dependent type further with a function

  ```
  def f(x: int) -> List int x: ...
  ```

  which represents a function who accepts an integer and returns a list of exactly the length provided. Dependent types allow us to powerfully reason about the static properties of our language and will be useful in our reasoning about multidimensional arrays later.

  Refinement types are a form of dependent types in which the set of values in a type can be filtered by predicates attached to a type. One refinement type system is Liquid Haskell (LH), which embeds refinement types in the Haskell programming language [12]. Refinement types offer a convenient way to declare specific types.

  As a demonstration of the power of refinement types (and also their potential weakness), consider the function that accepts a list and an integer for indexing into the list. In a simple type system, the type of the function might be:

  ```
  def index(lst: List[A], i: int) -> A: ...
  ```

  This type prevents improper indexing where the second argument is not an integer or where the first argument is not a list. However, programs that type check with this signature will still raise runtime errors at invocations like `index([], 1)`. This will raise an exception since `1` exceeds the empty list's maximum index.

  A stronger type — one that is possible to declare in Liquid Haskell — can statically prevent these malformed invocations. See Figure 2.1 for the declaration of the type.

```haskell
{-@ measure size @-}
{-@ size :: [a] -> {v : Int | v >= 0} @-}
size :: [a] -> Int
size [] = 0
size (h : t) = 1 + size t

{-@ index :: {l : [a] | true} -> {v : Int | v < size l && v >= 0} -> a @-}
index :: [a] -> Int -> a
index (h : t) i = if i == 0 then h else index t (i - 1)

test1 = index [1, 2, 3] 0
test2 = index [] 0 {- type error -}
```

Figure 2.1: `index` function type in Liquid Haskell

This type declaration prevents the misuse of the `index` function and statically raises a type error for the use `index [] 0`, which Python was unable to do. However, the cost of such expressiveness and specificity is increased verbosity in the type declaration.

2.2 Python

In this section, we examine some of Python's history and design philosophy, its type system, and its linear algebra and machine learning libraries.

2.2.1 History and Design Philosophy

Guido van Rossum began work on Python in 1989, and it was officially released to the public in 1991. Python is an interpreted language. It has a strong and dynamic type system. Type errors are raised at runtime, like when a string and an integer are added together. Variables can be assigned to values with different types during runtime. Python allows duck typing: if a method or line of code accesses an object's attribute or method, the runtime is happy to accept *any* object that has the appropriately named method or attribute. Python is mainly an object-oriented and imperative language although it now has aspects of functional languages as well.

Python is well-known for its simplicity, readability, and ease-of-use, and these attributes are fundamental to its spirit. In fact, Python Enhancement Proposal (PEP) 20 — Python's official format for reviewing and accepting changes to the language — makes the following aphorisms, collectively called "The Zen of Python," a part of the Python spec [14]:

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one– and preferably only one –obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!

These aphorisms are so important that they are included in every Python REPL; you can access them via `import this`.

Python's design carefully follows the above mantra. Blocks are delimited by whitespace, and its syntax closely resembles pseudocode. Variable types are never explicitly declared, memory is automatically managed and garbage collected, and returning different types from functions is allowed. Python's goal is to allow a programmer to sanely accomplish whatever they want with almost no friction. Large companies and academics — including Google, Instagram, and Dropbox — all use Python in their systems or research [5].

2.2.2 `numpy` and Other Numerical Libraries

Python unlocks a much faster speed of iteration, but it has its shortcomings. Since it is an interpreted language, it can be difficult to write performant code. Typically, Python is several orders of magnitude slower than other compiled languages.

As more and more people began to use Python for scientific computing and machine learning purposes, its speed increasingly became a barrier. This slowness led to the development of the library `numpy` which exposes a high-level API for declaring and operating on multidimensional arrays. `numpy` achieves fast performance by implementing all of its algorithms in highly optimized mahine code and then linking into Python. Machine learning libraries such as `pytorch` and `tensorflow` take similar approaches, providing high-level APIs for model training while using optimized GPU code to provide speed and parallelism [13]. In a future section, we will further explore the objects and operations that are exposed by these libraries.

The advent of these libraries has allowed engineers and researchers to leverage the flexibility of Python while not sacrificing the speed of their execution and model training.

2.2.3 `mypy` — Python Typing

Recently, dynamic typing in Python has been seen as a weakness rather than a strength, especially in large organizations. While Python's flexibility initially made development fast and easy, it did not scale as the size of Python codebases grew. Not only were runtime type errors possible, it was also difficult to look at functions and immediately determine their expectations for the type of their inputs and outputs. This emphasizes the fact that types not only function as a form of safety for a program but also as a form of self-documentation. Jukka Lehtosalo recognized these weaknesses and, with the help of Guido and others, developed a static type system and gradual type checker for Python — `mypy`. As part of PEP 484, Python was officially extended with type annotation in version 3.5 in 2015 [17].

There are now several type checkers for Python, including `pyre` from Facebook and pytype from Google. The static checkers are becoming more widespread; as of September 2019, Dropbox had reportedly typed 4 million lines of their Python code [4].

We now give an overview of the types that mypy supports.

- **Primitives** — Python's primitive types such as int, float, bool, and str.

- **Collections** — Python's standard collections like lists, dictionaries, tuples, and sets are supported. They are declared as List[Type], Dict[Type1,Type2], Tuple[Type], and Set[Type], respectively. These types are parameterized and can be instantiated with different types.

- **Classes and Objects** — all declared classes are automatically promoted to types, meaning that functions can accept parameters of certain classes. Furthermore, inheritance is automatically viewed as a form of subtyping. As an example,

  ```
  class A:
      pass

  class B(A):
      pass

  def f(x: A):
      return None

  b = B()
  f(b) # will type check
  ```

  In the example, all instances of class B are subtypes of class A, so instances of class B can stand in for class A as seen in function f and its application to b.

- **Union types** — union types are created via the Union[Type1, Type2] operator, allowing a value to be assigned to something from either of the two types.

- **Option types** — mypy also provides support for option types via Optional[Type]. Although languages with option types typically have explicit None and Some x constructors, Python simplifies this by having a None and a bare value x for the Some case.

- **Duck typing** — Python is a duck-typed language, and `mypy` makes this support explicit. For example, the type `Iterable[Type]` accepts any object which implements the `.next()` method and returns `Type`. Similarly, `Mapping[Type1, Type2]` accepts any object which implements the method `.__getitem__(x: Type1)` and returns `Type2`. In both of these examples, the exact class of the object does not matter, only that it implements a specific set of methods.

- **Function overloading** — in `mypy`, function signatures can be overloaded, meaning that functions can have multiple signatures. This is enabled via the `@overload` decorator on a function. As an example:

  ```
  @overload
  def f(x: int) -> str: ...

  @overload
  def f(x: str) -> int: ...

  def f(x: Union[int, str]) -> Union[int, str]:
      if isinstance(x, int):
          return "hello"

      return 1

  f(1) # will type check with type str
  f('hello') # will type check with type int
  ```

  Because of the `overload`, `mypy` is able to infer a more specific type for each of the function calls rather than just `Union[int, str]`.

- **Literal values** — `mypy` is not a general dependent type checker, but it does have a construct `Literal[Value]` which requires that a parameter or variable be assigned to exactly the value specified in the `Literal` constructor. For example,

  ```
  x : Literal[1]

  x = 1 # will type check
  x = 2 # will not type check
  x = 3 - 2 # will not type check
  ```

Note that although in the last line above, it is "obvious" that `x` is being assigned to 1, `mypy` performs no reasoning or speculation about the result of $3 - 2$. This is limiting but greatly simplifies the task of type checking `Literal` constructors.

From above, we can see that Python's type system is fairly straightforward. Besides the `Literal` constructor, Python features no dependent types. Python invokes its own ethos as described in "The Zen of Python" and opts for a type system that is easy to check and declare rather than one that provides the most powerful guarantees.

2.3 Numerical Libraries' Objects and Operations

2.3.1 Multidimensional Arrays

The fundamental object for storing data in numerical linear algebra and machine learning libraries like `numpy` and `pytorch` is the **multidimensional** or **N-dimensional array**. In linear algebra, the fundamental elements are vectors and matrices. Vectors are arrays of numbers. Matrices are arrays of arrays of numbers. `numpy` generalizes these constructs to an arbitrary number of "arrays of arrays of arrays of ... of numbers." With this generalization, vectors are multidimensional arrays with one dimension, and matrices are multidimensional arrays with two dimensions. Solo scalars can even be represented by multidimensional arrays with zero dimensions. The number of dimensions and the size of each dimension the array is represented by a tuple of integers which is generally referred to as its **shape** [2].

To see why this representation is useful, we can look at some examples. A vector of length 10 is represented by a shape of $(10,)$. A 2×2 matrix is represented by a shape of $(2, 2)$. A dataset of 1000 256×256 resolution images, each with distinct RGB channels, could be represented by the shape $(1000, 3, 256, 256)$. The scalar 1 is also a multidimensional array with shape $(,)$ (the empty tuple).

2.3.2 Operations on Multidimensional Arrays

`numpy` and related libraries define many operations over these multidimensional arrays. We cover some of the more useful and common operations here [3].

- **Broadcasting** — a unique operation between multidimensional arrays that does not have a linear algebra interpretation. Broadcasting occurs whenever there is a binary infix operator — such as +, -, /, etc. — acting between two arrays, or there is assignment from one array to a slice of another.

  To determine whether two arrays can broadcast, we start with the *last* dimensions of both arrays. We continue pairing the dimensions of both arrays until one or both run out of dimensions to pair. For each pair of dimensions, the dimensions must be equal or at least one of the dimensions must be equal to one. If broadcasting succeeds, the resulting array has the same number of dimensions as the longer of the two original arrays. The dimensions of the final array are the maximum of the two original arrays' dimensions.

  Broadcasting is a strange but pragmatic operation. It makes it simple to multiply an entire array by a constant value, like `2 * X`, or even perform more complicated operations, like adding the same vector to each row of a matrix.

- **Generalized dot product** — the dot product definition depends on the shape of the arrays it is operating on.

  - If both arrays are vectors, it is the traditional dot product.
  - If both arrays are matrices, it is matrix multiplication.
  - If either array is scalar, it is equivalent to scalar multiplication.
  - If the first array has N dimensions and the second array has 1 dimension, the result is valid if the last dimension of the first array matches with the single dimension of the second array.

  - If the first array has N dimensions and the second array has M dimensions, the result is valid if the last dimension of the first matrix and the second-to-last dimension of the second matrix are the same.

- **Summation, Euclidean norm** — array summation and the Euclidean norm are different operators, but they reduce the dimensions of an array the same way. If given no additional arguments, summation reduces an array to a scalar or an array with zero dimensions. If given an axis, the summation only sums along that particular axis, removing the provided dimension index from the shape.

- **Zeros, Ones** — given a tuple of integers, `np.zeros` generates a multidimensional array of zeros with shape exactly equal to the tuple given. `np.ones` acts analogously.

- **Concatenation** — takes two arrays and an axis and returns the result of concatenating those arrays along the axis. For concatenation to succeed, the dimensions of the two arrays must agree for all indices except for the one on which they are concatenated.

- **Model training** [1] — in supervised model training, the model is usually fed the data, X, and a list of labels, y. In order for the training to succeed, the number of data points in X must match the number of labels in y.

These are just a small sample of the operations we can perform on multidimensional arrays, but they are exactly the operations we would like to more strongly type in Python.

2.3.3 `numpy` Operations in `mypy`

We can examine how we can type some `numpy` operations using `mypy`'s current capabilities. For the `np.dot` function described above, the most specific type we can write is

```
def dot(a: np.ndarray, b: np.ndarray) -> np.ndarray: ...
```

Although this signature will prevent nonsensical calls to `np.dot` with `str` arguments, it does not capture the most interesting part of the operation. Fundamentally, what we care about is the underlying *shape* of the multidimensional arrays and how those shapes do or do not match to correctly dot together. For any other multidimensional array functions, the signature will look very similar, with `mypy` only able to restrict to the class `np.ndarray` and no further. It will be the goal of our new type system to overcome this barrier.

2.3.4 Multidimensional Arrays in Liquid Haskell

As an experiment with the expressivity of Liquid Haskell and the practicality of refinement types, we attempt to express a strict type for multidimensional arrays. The resulting code can be found in Figure 2.2. There are a couple things to note here. First, this representation has the desired property that the shape of the multidimensional array is a part of the matrix type. Refinement and dependent types have lifted the shape value into the type, which will allow us to refer to the shape as part of our type signatures for array operations. Second, the representation is *long*. LH's power and type expressivity have allowed us to express this complicated type, but there is an explosion in verbosity and complexity as a result.

2.4 Related Work

As machine learning and linear algebra routines have become more and more popular, there has been an increased interest in statically typing matrix and multidimensional arrays.

2.4.1 Vector and Matrix Embeddings

In [11], Eaton makes similar observations about the benefit of highly optimized matrix library code, generally available via the BLAS interface. Using Haskell's type system, Eaton is able to embed the dimensions of vectors and matrices directly into their types. Thus, he is able to get type safe matrix operations. Eaton observes that the Haskell implementation outperforms Octave (another linear algebra programming language) and achieves performance on par with Matlab. However, he laments that the Haskell compiler can generate cryptic errors at

```
{-@ LIQUID "--exact-data-con" @-}
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}
import Prelude hiding (reverse, max, drop)

{-@ type Nat = {v: Int | v >= 0} @-}
{-@ measure size @-}
{-@ size :: [a] -> Nat @-}
size :: [a] -> Int
size [] = 0
size (hd : tl) = 1 + size tl

data Vector a = V {vLen::Int, elts::[a]} deriving (Eq)
{-@ data Vector a = V {vLen::Nat, elts::{v : [a] | size v == vLen}} @-}
{-@ type VectorN a N = {v : Vector a | vLen v = N} @-}

{-@ type ListNE a = {v : [a] | size v > 0} @-}

{-@ measure head1 @-}
{-@ head1 :: ListNE Nat -> Nat @-}
head1 :: [Int] -> Int
head1 (h : t) = h
{-@ measure tail1 @-}
{-@ tail1 :: ListNE Nat -> [Nat] @-}
tail1 :: [Int] -> [Int]
tail1 (h : t) = t

data Ndarray a = N {nDims::[Int], nElts::Either a (Vector (Ndarray a))}

{-@ measure ords @-}
{-@ ords :: Ndarray a -> [Nat] @-}
ords :: Ndarray a -> [Int]
ords (N nDims _nElts) = nDims

{-@ measure isLeft @-}
isLeft (Left _) = True
isLeft (Right _) = False

{-@ data Ndarray a =
 N {nDims::[Nat],
    nElts::{v : Either a (VectorN ({x : Ndarray a | (tail1 nDims) == (ords x)})
                                  {head1 nDims})
        | (isLeft v && size nDims = 0) || (not (isLeft v) && size nDims > 0) }} @-}

{-@ type NdarrayN a N = {v : Ndarray a | nDims v = N} @-}
```

Figure 2.2: Sample implementation of the `np.ndarray` type in LH

locations far removed from the actual dimensionality bugs, making it less ergonomic than desirable. Other packages available on Haskell, such as HMatrix, also provided matrix types with shapes [16].

In [8], Abe and Sumii achieve similar matrix and vector bindings for OCaml. What is incredible about their bindings is that they use the standard OCaml system to embed the dimension of matrices into their types. OCaml does *not* have any form of dependent typing, so such an embeding is a remarkable achievement.

Eaton and Abe and Sumii's results are interesting but fall short of the general goal of typing multidimensional arrays since their methods only work for vectors and matrices. This is an inherent limitation of their representation. Each dimension of their vector or matrix is passed as a separate argument to their type rather than all dimensions being passed in a single array.

2.4.2 Multidimensional Array Embeddings

Work has been done for the more general case of multidimensional arrays. In [15], Rink represents the type of multidimensional arrays as lists of integers — analogous to the shape discussed in Section 2.3.1. Rink also identifies common operations between the shapes of multidimensional arrays. He includes swapping dimensions, dropping dimensions if they are equal, and concatenating the shapes of two arrays. However, the paper requires that the shapes contain only concrete integers — no abstract values like N — and the arguments to the operators he declares must also be integers. Furthermore, there is no concept of array types which accept an arbitrary number of dimensions.

[9] embeds a multidimensional array type in the Scala type system, leveraging Scala's built-in heterogenous list (HList) type. Like Rink, Chen defines and implements several operations over the shape types, including inserting a new dimension and contracting all dimensions that are equal. Unlike Rink, Chen's dimensions are abstract and do not require concrete integers. Also, Chen's dimensions allow for an arbitrary number of dimensions by

having the shape be a generic subtype of an HList. Overall, Chen's embedding most closely matches the embedding we desire in Python's type system. It still lacks support for the atypical broadcast operator, and arithmetic between the dimensions of the arrays does not seem to be allowed. It is not possible to specify exact integers for the dimensions of an array if desired. Also, it appears that the use of user-provided integers is not possible in the embedding.

Chapter 3

A Type System for Multidimensional Arrays

Having seen the semantics of multidimensional arrays and various embeddings in other languages, how can we check the dimensions of arrays within Python while maintaining its ethos? The goal is to increase the strength of the type checker while not exploding the complexity of the syntax. We should be able to annotate many, if not all, of the functions previously described in Section 2.3.2. With these goals in mind, we present a new syntax and extension to Python's existing type system for declaring and checking multidimensional array functions and values.

3.1 Syntax

We present the syntax extension and give intution for the various constructs. The new major type is the **Ndarray**. This type represents a multidimensional array with certain shape. It accepts a list of *dimensions* to represent the concrete or abstract dimensions of the array. *dimensions* have several constructs.

- **Id** — declares or references an abstract dimension of an array, one that can be of any

```
⟨type⟩ ::= ...
  |  Nparray ⟨dimension⟩*

⟨dimension⟩ ::= Id ⟨string⟩
  |  Int ⟨int⟩
  |  Add (⟨dimension⟩, ⟨dimension⟩)
  |  Mul (⟨dimension⟩, ⟨dimension⟩)
  |  Spread ⟨string⟩
  |  Drop (⟨string⟩, (⟨string⟩ | ⟨int⟩)*)
  |  Keep (⟨string⟩, (⟨string⟩ | ⟨int⟩)*)
  |  Broadcast ⟨string⟩

⟨functionarg⟩ ::= (⟨string⟩, ⟨type⟩)

⟨functiontype⟩ ::= (⟨functionarg⟩*, ⟨type⟩)
```

Figure 3.1: Full syntax for matrix types

non-negative length. It can even refer to a user provided parameter name as long as the type of that parameter is a **LiteralInt**.

- **Int** — declares a dimension that must be exactly a certain size.

- **Add** — declares a dimensions that is the result of adding the sizes of previously declared dimensions. **Add** can contain other **Add**, **Mul**, **Id**, and **Int** dimensions. In Python syntax, it can be represented by the `+` operator.

- **Mul** — declares a dimensions that is the result of multiplying the sizes of previously declared dimensions. **Add** can contain other **Add**, **Mul**, **Id**, and **Int** dimensions. In Python, it can be represented by the `*` operator.

- **Spread** — declares or references an arbitrary number of dimensions that have arbitrary sizes. In Python, it can be represented as `*A` where `A` could be an arbitrary variable name.

- **Drop** — references an arbitrary number of dimensions declared by a **Spread** operation and allows the dropping of one or more of the captured dimensions. The dimensions to be dropped can be provided as direct integers or as IDs referencing user parameters as long as those parameters are of type **LiteralInt**.

- **Keep** — references an arbitrary number of dimensions declared by a **Spread** operation and allows keeping one or more of the captured dimensions, discarding the rest. The dimensions to be kept can be provided as direct integers or as IDs referencing user parameters as long as those parameters are of type **LiteralInt**.
- **Broadcast** — references an arbitrary number of dimensions mapped by a **Spread** operation. This constructor ensures that there is a prefix of the argument dimensions that broadcasts with the dimensions.

We represent a function parameter type is a pair between a string — the name of the parameter — and a type. A function type is then a pair between a list of function parameter types and a return type.

See Figure 3.1 for the formal syntax.

3.2 Simplifications

We choose not to parameterize the **Ndarray** type with the type of its contents and instead assume that all **Ndarray**s are filled with integers. Since our main concern is the shape of the arrays, we elided this parameterization, but it should not be difficult to add in the future.

For arguments to these functions, we restrict to **Ndarray** types which have a finite and known number of dimensions, disallowing **Spread**, **Drop**, and **Keep** dimension constructors. This restriction is necessary to enable application type checking and will be further discussed later in this thesis.

3.3 Semantics

We now provide a formal semantics for type checking the types defined above. There are three stores in the context: Γ, Σ, and Π. They map single dimension variables to the argument's dimensions, spread variables to a list of argument dimensions, and parameter names to the arguments' types, respectively.

At a high level, checking falls into three steps:

1. **Application checking** — this checks a *function type* against a list of *arg types*. For each parameter in the function type, the type of the parameter is checked against the type of the corresponding argument. If each of the parameter types checks, then we can form the return type from the context. It is represented by the relation

$$\Gamma, \Sigma, \Pi \vdash_A (param\_types, ret\_type), args : \Gamma', \Sigma', \Pi', \tau$$

2. **Parameter/argument checking** — this checks a specific parameter type against the provided argument type. If the parameter type is an integer, the type checking is straightforward. We simply check that the argument type is an integer, literal integer, or **Ndarray** with 0 dimensions (which is equivalent to a scalar integer). If the parameter is a literal integer, we check that the argument type is also a literal integer of the same value. If the parameter type is an **Ndarray**, after verifying that the argument type is also an **Ndarray**, we need to do further checking of the dimensions of the parameter and argument type. It is represented by the relation

$$\Gamma, \Sigma, \Pi \vdash_P (type, arg\_type) : \Gamma', \Sigma', \Pi'$$

3. **Dimension checking** — this is the "lowest" level of the type checking. Here we will establish equalities and mappings between the dimensions required by the type and those provided in the arguments. For each of the dimension constructs, there are

different typing rules. Let *arg_dims* represent the dimensions of the argument type. It is represented by the relation

$$\Gamma, \Sigma, \Pi \vdash_D (\mathit{dimension}, \mathit{arg\_dims}) : \Gamma', \Sigma', \Pi', \mathit{rem\_dimensions}$$

For any proofs about equality between integers, we defer to an SMT solver (Z3). We now present informal rules for checking each of the dimension types. Let *hd* be the standard function that gets the head of a list.

- **Id** i — if i is not in Γ, then map $\Gamma[i \mapsto \mathit{hd}(\mathit{arg\_dims})]$ and continue type checking the remaining argument dimensions. If i is in Γ, then prove that $\Gamma(i) = \mathit{hd}(\mathit{arg\_dims})$. Alternatively, if i is in Π and $\Pi(i) =$ **LiteralInt** n, then prove that $n = \mathit{hd}(\mathit{arg\_dims})$.
- **Int** i — prove that $i = \mathit{hd}(\mathit{arg\_dims})$.
- **Add** (d_1, d_2) — build arithmetic expressions from d_1 and d_2 resulting in e_1 and e_2. Prove that $e_1 + e_2 = \mathit{hd}(\mathit{arg\_dims})$.
- **Mul** (d_1, d_2) — build arithmetic expressions from d_1 and d_2 resulting in e_1 and e_2. Prove that $e_1 * e_2 = \mathit{hd}(\mathit{arg\_dims})$.
- **Spread** *var* — if *var* is not in Σ, then find a split of *arg_dims* into lists of types *front* and *back* such that $\Sigma[\mathit{var} \mapsto \mathit{front}]$ allows the remainder of type checking to succeed. If *var* is in Σ, then split *args* into *front* and *back* where $\mathit{len}(\mathit{front}) = \mathit{len}(\Sigma(\mathit{var}))$. For each element of *front* and $\Sigma(\mathit{var})$, prove that they are equal.
- **Drop** (*var*, *indices*) — require that *var* is in Σ. Convert *indices* to a list of integers and drop those indices from $\Sigma(\mathit{var})$. The indices can be pulled from Π as long as Π maps them to **LiteralInt**s.
- **Keep** (*var*, *indices*) — require that *var* is in Σ. Convert *indices* to a list of

$$\textsc{CheckAppEmpty}\ \frac{\Gamma, \Sigma, \Pi \vdash_R ret : \tau}{\Gamma, \Sigma, \Pi \vdash_A ([], ret), [] : \Gamma, \Sigma, \Pi, \tau}$$

$$\textsc{CheckApp}\ \frac{\Gamma, \Sigma, \Pi[p \mapsto h_2] \vdash_P h_1, h_2 : \Gamma', \Sigma', \Pi' \qquad \Gamma', \Sigma', \Pi' \vdash_A (t_1, ret), t_2 : \Gamma'', \Sigma'', \Pi'', \tau}{\Gamma, \Sigma, \Pi \vdash_A ((p, h_1) :: t_1, ret), h_2 :: t_2 : \Gamma'', \Sigma'', \Pi'', \tau}$$

Figure 3.2: Formal semantics of application checking

integers and keep those indices in $\Sigma(var)$. The indices can be pulled from Π as long as Π maps them to **LiteralInt**s.

- **Broadcast** *var* — require that *var* is in Σ. Find a split of *arg_dims* into lists of dimensions *front* and *back* such that *front* broadcasts with $\Sigma(var)$.

The formalized semantics for the type checking are provided in the following figures. Figure 3.2 provides semantics for function application. Figure 3.3 provides semantics for parameter checking. Figure 3.4 provides semantics for dimension checking. Figure 3.5 provides semantics checking for return types. Figure 3.6 provides semantics for various utilities used throughout type checking. In the semantics of dimension checking, the functions *drop* and *keep* refer to the functions that take a list and a list of integers and drop or keep those indices in the list, respectively.

3.4 Implementation

We implement the above typing rules in OCaml. We opted not to directly augment the `mypy` checker because it was a massive and difficult-to-interpret code base. OCaml lends itself nicely to implementing inference rules because of its first class support for algebraic data types. The code can be found online at `https://github.com/theodoretliu/thesis`.

$$\textsc{CheckParamScalarArrArr}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{Ndarray}\ [], \mathbf{Ndarray}\ []) : \Gamma, \Sigma, \Pi}$$

$$\textsc{CheckParamScalarArrInt}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{Ndarray}\ [], \mathbf{Int}) : \Gamma, \Sigma, \Pi}$$

$$\textsc{CheckParamScalarArrLitInt}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{Ndarray}\ [], \mathbf{LiteralInt}\ i) : \Gamma, \Sigma, \Pi}$$

$$\textsc{CheckParamScalarIntArr}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{Int}, \mathbf{Ndarray}\ []) : \Gamma, \Sigma, \Pi}$$

$$\textsc{CheckParamScalarLitInt}\ \frac{i = j}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{LiteralInt}\ i, \mathbf{LiteralInt}\ j) : \Gamma, \Sigma, \Pi}$$

$$\textsc{CheckParamArray}\ \frac{\begin{array}{c}\Gamma, \Sigma, \Pi \vdash_D h, \ell_2 : \Gamma', \Sigma', \Pi', rem \\ \Gamma', \Sigma', \Pi' \vdash_P (\mathbf{Ndarray}\ \ell_1, \mathbf{Ndarray}\ rem) : \Gamma'', \Sigma'', \Pi''\end{array}}{\Gamma, \Sigma, \Pi \vdash_P (\mathbf{Ndarray}\ h :: \ell_1, \mathbf{Ndarray}\ \ell_2) : \Gamma'', \Sigma'', \Pi''}$$

Figure 3.3: Formal semantics of parameter checking

$$\textsc{CheckDimIdNotFound}\ \frac{n \notin \Gamma}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Id}\ n, h :: t) : \Gamma[n \mapsto h], \Sigma, \Pi, t}$$

$$\textsc{CheckDimIdFound}\ \frac{n \in \Gamma \qquad \Gamma(n) = h}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Id}\ n, h :: t) : \Gamma, \Sigma, \Pi, t}$$

$$\textsc{CheckDimIdFoundInParams}\ \frac{n \in \Pi \qquad \Pi(n) = \textbf{LiteralInt}\ i \qquad i = h}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Id}\ n, h :: t) : \Gamma, \Sigma, \Pi, t}$$

$$\textsc{CheckDimInt}\ \frac{h = i}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Int}\ i, h :: t) : \Gamma, \Sigma, \Pi, t}$$

$$\textsc{CheckDimAdd}\ \frac{\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2 \qquad h = e_1 + e_2}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Add}\ (d_1, d_2), h :: t) : \Gamma, \Sigma, \Pi, t}$$

$$\textsc{CheckDimMul}\ \frac{\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2 \qquad h = e_1 \times e_2}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Mul}\ (d_1, d_2), h :: t) : \Gamma, \Sigma, \Pi, t}$$

$$\textsc{CheckDimSpreadNotFound}\ \frac{n \notin \Sigma}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Spread}\ n, \mathit{front}@\mathit{back}) : \Gamma, \Sigma[n \mapsto \mathit{front}], \Pi, \mathit{back}}$$

$$\textsc{CheckDimSpreadFound}\ \frac{n \in \Sigma \qquad \mathit{len}(\mathit{front}) = \mathit{len}(\Sigma(n)) \qquad \mathit{front} = \Sigma(n)}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Spread}\ n, \mathit{front}@\mathit{back}) : \Gamma, \Sigma, \Pi, \mathit{back}}$$

$$\textsc{CheckDimDrop}\ \frac{n \in \Sigma \qquad \Pi \vdash_L \ell \rightsquigarrow \ell' \qquad \mathit{drop}(\Sigma(n), \ell') = \mathit{front}}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Drop}\ (n, \ell), \mathit{front}@\mathit{back}) : \Gamma, \Sigma, \Pi, \mathit{back}}$$

$$\textsc{CheckDimKeep}\ \frac{n \in \Sigma \qquad \Pi \vdash_L \ell \rightsquigarrow \ell' \qquad \mathit{keep}(\Sigma(n), \ell') = \mathit{front}}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Keep}\ (n, \ell), \mathit{front}@\mathit{back}) : \Gamma, \Sigma, \Pi, \mathit{back}}$$

$$\textsc{CheckDimBroadcast}\ \frac{n \in \Sigma \qquad \Sigma(n) =_B \mathit{front}}{\Gamma, \Sigma, \Pi \vdash_D (\textbf{Broadcast}\ n, \mathit{front}@\mathit{back}) : \Gamma, \Sigma, \Pi, \mathit{back}}$$

Figure 3.4: Formal semantics of dimension checking

$$\textsc{RetInt}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Int} : \textbf{Int}}$$

$$\textsc{RetLiteralInt}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_R \textbf{LiteralInt}\ i : \textbf{LiteralInt}\ i}$$

$$\textsc{RetNdarrayEmpty}\ \frac{}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ [] : \textbf{Ndarray}\ []}$$

$$\textsc{RetNdarrayFullId}\ \frac{i \in \Gamma \qquad \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Id}\ i :: t) : \textbf{Ndarray}\ (\textbf{Id}\ i)}$$

$$\textsc{RetNdarrayFullInt}\ \frac{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Int}\ i :: t) : \textbf{Ndarray}\ ()\textbf{Int}\ i :: t')}$$

$$\textsc{RetNdarrayFullAdd}\ \frac{\begin{array}{c}\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2 \\ \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'\end{array}}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Add}\ (d_1, d_2) :: t) : \textbf{Ndarray}\ (\textbf{Add}\ (e_1, e_2) :: t')}$$

$$\textsc{RetNdarrayFullMul}\ \frac{\begin{array}{c}\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2 \\ \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'\end{array}}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Mul}\ (d_1, d_2) :: t) : \textbf{Ndarray}\ (\textbf{Mul}\ (e_1, e_2) :: t')}$$

$$\textsc{RetNdarrayFullSpread}\ \frac{n \in \Sigma \qquad \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Spread}\ n :: t) : \Sigma(n) @ t'}$$

$$\textsc{RetNdarrayFullDrop}\ \frac{n \in \Sigma \qquad \Pi \vdash_L \ell \rightsquigarrow \ell' \qquad \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Drop}\ (n, \ell) :: t) : \mathit{drop}(\Sigma(n), \ell') @ t'}$$

$$\textsc{RetNdarrayFullKeep}\ \frac{n \in \Sigma \qquad \Pi \vdash_L \ell \rightsquigarrow \ell' \qquad \Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ t : \textbf{Ndarray}\ t'}{\Gamma, \Sigma, \Pi \vdash_R \textbf{Ndarray}\ (\textbf{Keep}\ (n, \ell) :: t) : \mathit{keep}(\Sigma(n), \ell') @ t'}$$

Figure 3.5: Formal semantics of return types

$$\textsc{TransformId}\ \frac{d \in \Gamma}{\Gamma, \Pi \vdash_T d \to \Gamma(d)}$$

$$\textsc{TransformIdParams}\ \frac{d \in \Pi \qquad \Pi(d) = \textbf{LiteralInt}\ i}{\Gamma, \Pi \vdash_T d \to i}$$

$$\textsc{TransformAdd}\ \frac{\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2}{\Gamma, \Pi \vdash_T \textbf{Add}\ (d_1, d_2) \to e_1 + e_2}$$

$$\textsc{TransformMul}\ \frac{\Gamma, \Pi \vdash_T d_1 \to e_1 \qquad \Gamma, \Pi \vdash_T d_2 \to e_2}{\Gamma, \Pi \vdash_T \textbf{Mul}\ (d_1, d_2) \to e_1 \times e_2}$$

$$\textsc{TransformListEmpty}\ \frac{}{\Pi \vdash_L [] \rightsquigarrow \ell}$$

$$\textsc{TransformListInt}\ \frac{\Pi \vdash_L \ell \rightsquigarrow \ell'}{\Pi \vdash_L i :: \ell \rightsquigarrow i :: \ell'}\ i \in \mathbb{Z}$$

$$\textsc{TransformListId}\ \frac{s \in \Pi \qquad \Pi(s) = \textbf{LiteralInt}\ i \qquad \Pi \vdash_L \ell \rightsquigarrow \ell'}{\Pi \vdash_L s :: \ell \rightsquigarrow i :: \ell'}$$

$$\textsc{BroadcastLeftEmpty}\ \frac{}{[] =_B \ell}$$

$$\textsc{BroadcastRightEmpty}\ \frac{}{\ell =_B []}$$

$$\textsc{BroadcastNonempty}\ \frac{t_1 = t_2 \lor t_1 = 1 \lor t_2 = 1 \qquad h_1 =_B h_2}{h_1 @ [t_1] =_B h_2 @ [t_2]}$$

Figure 3.6: Formal semantics of type checking utilities

Chapter 4

Evaluation

We will now evaluate how well the new **Ndarray** type can annotate the functions we defined in Section 2.3.2. From these annotations, we will observe the strengths and areas for improvement of the extension. We can then develop plans for future work.

4.1 Typing Multidimensional Array Operators

We can go one by one over the functions of interest in Section 2.3.2.

4.1.1 Broadcast

The broadcast operator is very special. We can now write the signature of the broadcast function as

```
def broadcast(x: Ndarray[*A], y: Ndarray[Broadcast[A]]) -> Ndarray ?: ...

x: Ndarray[A, B, C]
y: Ndarray[B, C]
z: Ndarray[1, C]
a: Ndarray[2, C]

broadcast(x, y) # will type check
broadcast(x, z) # will type check
broadcast(y, z) # will type check
broadcast(x, a) # will not type check
```

Our "hard-coded" dimension constructor for broadcasting makes this signature easy to declare. We can see that our type checker is able to determine exactly the cases that *should* broadcast and rejects the ones that do not. However, we still do not have a proper construct for the *return* type of broadcast. None of the constructors we have created can represent the return type of a broadcast.

4.1.2 Generalized Dot Product

Recall that the generalized dot product has different behaviors depending on the shapes of the arguments it is provided. We can use the `overload` construct in `mypy` to declare the different function types one by one. For vector dot products:

```
@overload
def dot(x: Ndarray[A], y: Ndarray[A]) -> int: ...

x : Ndarray[A]
y : Ndarray[A]
z : Ndarray[4]
a : Ndarray[5]

dot(x, y) # will type check
dot(y, z) # will not type check
dot(z, a) # will not type check
```

With our new types, we can perfectly declare the type of this function. There is some ambiguity in the return type; we could also have returned type `Ndarray[]`, a multidimensional array with no dimensions, which is also an integer scalar. For matrix multiplication:

```
@overload
def dot(x: Ndarray[A, B], y: Ndarray[B, C]) -> Ndarray[A, C]: ...

X: Ndarray[A, B, C]
Y: Ndarray[A, B]
Z: Ndarray[B, C]
W: Ndarray[C, D]

dot(X, Y) # will fail, number of dimensions incorrect
dot(Y, Y) # will fail, inner dimensions don't work
res = dot(Y, Z) # will succeed
```

```
dot(res, W) # will succeed, res inferred to have type Ndarray[A, C]
```

Again, we are able to declare and check the type of the second form of dot product. We can even infer the result of the return type and propagate that type information into another call to dot without explicit type annotations. Continuing with the scalar cases:

```
@overload
def dot(x: Ndarray[], y: Ndarray[*A]) -> Ndarray[*A]: ...

@overload
def dot(x: Ndarray[*A], y: Ndarray[]) -> Ndarray[*A]: ...

X: Ndarray[A, B, C]
r: int

dot(X, r) # will type check, Ndarray[A, B, C]
dot(r, X) # will also type check, Ndarray[A, B, C]
```

Again, we see the ambiguity here with `Ndarray[]` which could also be replaced by type `int`. For the last two cases of dot:

```
@overload
def dot(x: Ndarray[*A, B], y: Ndarray[B]) -> Ndarray[*A]: ...

@overload
def dot(x: Ndarray[*A, B], y: Ndarray[*C, B, D]) -> Ndarray[*A, *C, D]: ...

X: Ndarray[A, B, C]
Y: Ndarray[C]
Z: Ndarray[C, D]

dot(X, Y) # will check, Ndarray[A, B]
dot(X, Z) # will check, Ndarray[A, B, D]
```

Again, the new types seem to fit very well here and can accurately annotate and check the type of these generalized dot products. Overall, the generalized dot product is very well annotated by these new types.

4.1.3 Summation, Euclidean Norm

Summation is also broken into two distinct function signatures, which we overload. We have

```
def sum(x: Ndarray[*A]) -> int: ...

x : Ndarray[1, 2, 3]
y : Ndarray[A]

sum(x) # type checks, int
sum(y) # type checks, int
```

The first case is very simple. The next case is interesting:

```
def sum(x: Ndarray[*A], y: int) -> Ndarray[Drop[A, [y]]]: ...

x: Ndarray[1, 2, 3]
y: Ndarray[A, B, C]

sum(x, 1) # type checks, Ndarray[1, 3]
sum(y, 2) # type checks, Ndarray[A, B]
sum(x, -1) # type checks, Ndarray[1, 2]
sum(x, 5) # fails, 5 exceeds max index
sum(y, 3 - 2) # fails, type of second argument is not
              # determined to be LiteralInt
```

This case is the first time we have reason to use our **Drop** constructor to remove dimensions from a multidimensional array. If integer literals are provided as the parameter `y`, the type checker is able to use that integer literal to drop the correct axis. However, if `y` is the result of some computation, its type cannot be inferred as integer literal and the type checking fails.

4.1.4 Zeros, Ones

The current type system extension does not have support for tuples, so a fully correct version of zeros cannot be implemented. However, using some overloads, we can get part of the way there:

```
@overload
def zeros(k: int) -> Ndarray[k]: ...

@overload
def zeros(k1: int, k2: int) -> Ndarray[k1, k2]: ...
```

```
zeros(2) # type checks, Ndarray[2]
zeros(3, 4) # type checks, Ndarray[3, 4]
zeros(3 + 2) # does not check
```

Even though we do not have direct access to tuples, this approximation of the zeros function allows zeros to potentially take multiple arguments and return the **Ndarray** that has the same dimension as the integer provided. Again, the integer must be inferred to be of type integer literal or the typing will not succeed.

4.1.5 Concatenation

As with the zeros function, we cannot represent all possibilities of concatenate, but we can approximate it with overloading.

```
@overload
def concatenate(x: Ndarray[A], y: Ndarray[B]) -> Ndarray[A + B]: ...

@overload
def concatenate(x: Ndarray[A, B],
                y: Ndarray[A, C],
                axis: Literal[1]) -> Ndarray[A, B + C]: ...

x: Ndarray[1]
y: Ndarray[2]

a: Ndarray[2, 4]
b: Ndarray[2, A]

concatenate(x, y) # type checks, Ndarray[3]
concatenate(x, x) # type checks, Ndarray[2]

concatenate(a, b, 1) # type checks, Ndarray[2, A + 4]
concatenate(x, a) # fails
```

The inability to completely capture all possible cases of concatenation is a cause for concern.

4.1.6 Model Training

Model training can be annotated well:

```
def fit(X: Ndarray[N, *A], y: Ndarray[N, B]) -> None: ...

data: Ndarray[N, 5]
labels: Ndarray[N, 1]

fit(data, labels) # type checks

data2: Ndarray[N, 3, 256, 256]
labels2: Ndarray[N, 5]

fit(data2, labels2) # type checks
```

The type annotations capture the variation we might have in our dataset and training label shapes.

4.1.7 More Types

As a further test of the types' flexibility, we can declare the types of other interesting functions.

- **k-means clustering** — Given a set of N d-dimensional points, return the k centroids.

  ```
  def kmeans(X: Ndarray[N, d], k: int) -> Ndarray[k, d]: ...
  ```

- **Singular value decomposition** — Given a matrix M of size $m \times n$, decompose it into $U\Sigma V^*$.

  ```
  def svd(M: Ndarray[m, n]) -> Tuple[Ndarray[m, m],
                                     Ndarray[m, n],
                                     Ndarray[n, n]]: ...
  ```

4.2 Strengths

These type annotations excel in situations where the function types will rely on a finite number of dimensions and use the dimension variables or dimension integers to express equality between the arguments' dimensions. In the examples in the previous section, the annotations seemed to capture the complete behavior of a function best when it did not excessively or esoterically use the **Spread** operator.

The annotations are also extremely intuitive to read and declare for programmers who are familiar with the shape attribute of multidimensional arrays.

The use of these type annotations in machine learning and linear algebra algorithms is particularly promising. In machine learning, the maximum number of dimensions in the data array is typically small. As seen in the model training example, the type annotations can assure that the number of data points is equal to the number of data labels. As an example, the k-means and SVD algorithms were expressed succinctly and accurately with the new type annotations.

4.3 Future Work and Remaining Improvements

We now review the areas for improvement in the type system, starting with the most problematic, and propose future steps to mitigate the issue.

4.3.1 `mypy` Integration

A type system is only useful if people actually *use* it. While we have defined the semantics for type checking and implemented a checker in OCaml, it still remains for this syntax to be approved and integrated into `mypy`, the official Python type checker. Until this integration happens, this project will never bring the benefits of its type system to real users. It will be our goal to incorporate this project into the main `mypy` source tree, even if in a more limited form, so that people can start typing real code and real use cases. This will further unlock more insights into the type annotations that people actually want, need, and use for multidimensional array typing.

4.3.2 Restrictions on Argument Types

As previously stated in Section 3.2, we restrict the types of arguments to the finite length dimension constructors. To see the problem with removing such a restriction, consider the following function type and application:

```
def f(x: Ndarray[*A, *B]) -> Ndarray[*A]: ...

y: Ndarray[*C]
f(y) # ???
```

In the example above, what should we map `A` to? What should we map `B` to? There are infinitely many solutions and mappings possible. We could split up `C` in infinite ways since we do not generally know anything about its structure.

This issue with **Spread** and related dimension constructors in argument positions also prevents certain function body checking. Consider if the above declaration for `f`, and we also add

```
def g(y: Ndarray[*C]) -> Ndarray[*C]:
    # y has type Ndarray[*C] in the body of g
    ...
    f(y) # if we encounter this invokation, we cannot do anything
    ...
```

A challenge and improvement going forward would be solving this type checking problem, either by developing appropriate semantics for type checking or more elegantly circumventing the issue. We would have to exercise caution when developing semantics because we would not want to accidentally cause a complexity explosion, making the type checker run slowly and/or making the surface syntax more complicated.

Even without directly fixing this, function body checking where the parameters of the function are fixed dimension arrays is still possible. Also, `mypy` supports a `.pyi` file format where function types are declared, but the function body is completely omitted, so function body checking might not even be that important in the first place.

4.3.3 Strict Dimension Checking

When presented with two dimensions, the type checker attempts to *prove* that the two dimensions are equal. In certain cases, there is not information to prove that an abstract dimension is equal to a particular integer and type checking will fail. As a simple example:

```
def f(x: Ndarray[2]): ...

y: Ndarray[A]

f(y) # will fail, can't prove that A == 2
```

One of the original goals of this thesis was for the new type system to align with Python's ethos. If the type checker is this strict, will it make it difficult to use these type annotations? Future work would either relax this constraint or determine that the constraint is not an impediment to the use of the annotations.

4.3.4 Supporting Zeros, Concatenate Fully

As demonstrated in the examples in Section 4.1, the current type annotations are unable to support the zeros and concatenate functions fully. If we extend the annotations to support the full functions, we can reason more powerfully about the dimensions of arrays in our system.

That being said, it might be the case that people rarely make zero arrays of more than five or so dimensions, and people rarely concatenate arrays of more than four dimensions. If that were true, then we could actually get away with simply overloading the function a few times to capture all the practical cases that people use. There is precedent for this in `mypy`. In the typing stubs for the `zip` function in Python — which can actually take an arbitrary number of arguments — the function is simply overloaded six times to support receiving one through six arguments [7].

4.3.5 Building Primitives for Broadcast Outputs

As noted in Section 2.3.2, the broadcast is one of the more unusual but also useful operations available on multidimensional arrays. We should incorporate another primitive to the dimension language to represent the "max" operation that occurs during broadcasting. This would enable the return types of broadcasts to be correctly inferred, which will strengthen our type system.

4.3.6 Parameterizing the Ndarray Type

We noted in Section 3.2 that we elided a type parameter for the **Ndarray** which would parameterize the type of data stored in the array. We should add this type parameter to fully capture the nature of multidimensional arrays and allow us to better understand the return types of reductions on arrays of integers versus floats.

Chapter 5

Conclusion

In this thesis, we have introduced a new type system for Python that enables type checking for multidimensional arrays in a compact and intuitive syntax. Although there still remain several areas for improvement within the system, it already has shown itself capable of annotating and checking the application of useful array, machine learning, and linear algebra operations. Ultimately, the improvement and use of this multidimensional array type system will allow engineers and academics to worry less about debugging the dimensions of their datasets and transformations and worry more about their models and ideas, leading to faster iteration and innovation.

Bibliography

[1] Linearregression. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.fit`. [Online; accessed 10-April-2020].

[2] The n-dimensional array. `https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html`. [Online; accessed 10-April-2020].

[3] Numpy manual. `https://docs.scipy.org/doc/numpy/index.html`. [Online; accessed 10-April-2020].

[4] Our journey to type checking 4 million lines of python. `https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python`. [Online; accessed 08-April-2020].

[5] Quotes about python. `https://www.python.org/about/quotes/`. [Online; accessed 10-April-2020].

[6] The rust programming language. `https://doc.rust-lang.org/book/`. [Online; accessed 09-April-2020].

[7] Typeshed. `https://github.com/python/typeshed/`. [Online; accessed 10-April-2020].

[8] Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. In *ML/OCaml*, 2015.

[9] Tongfei Chen. Typesafe abstractions for tensor operations. *ArXiv*, abs/1710.06892, 2017.

[10] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5:56–68, 1940.

[11] Frederik Eaton. Statically typed linear algebra in haskell. In *Haskell '06*, 2006.

[12] Ranjit Jhala. Refinement types for haskell. In *PLPV '14*, 2014.

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary Devito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[14] Tim Peters. Pep 20 – the zen of python. `https://www.python.org/dev/peps/pep-0020/`. [Online; accessed 08-April-2020].

[15] Norman A. Rink. Modeling of languages for tensor manipulation. *ArXiv*, abs/1801.08771, 2018.

[16] Alberto Ruiz. Hmatrix. `https://hackage.haskell.org/package/hmatrix-0.20.0.0/docs/Numeric-LinearAlgebra-Data.html#t:Matrix`. [Online; accessed 10-April-2020].

[17] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Pep 484 – type hints. `https://www.python.org/dev/peps/pep-0484/`. [Online; accessed 09-April-2020].