Value-Dependent Types:
Efficient, Flexible Containers in Pony

Luke Cheeseman

June 13, 2016

Abstract

Parametric types offer a programmer a much greater degree of flexibility when developing programs. For example, a list may be parametric as to its contents. This list may then be instantiated to a list of numbers, or a list of student records, or any type the developer requires. Often type parametrisation is restricted to types and does not permit values. Thus we can have containers which are parametric with respect to type of the element, yet not with respect to number of elements

Further parametrisation of types on values would allow programmers to create much tighter relationships between a type and a value. It would make it possible to refer to containers of different fixed lengths as belonging to different types. We can use this distinction to ensure that operations, such as matrix multiplication, can only be used with values which are correct for the operation. This allows us to remove the run-time checks that must otherwise be executed on every execution to ensure the operation is well-defined. Parametrisation of types on values exists in various programming languages including imperative languages such as C++ as well as functional languages like Idris.

In this report I discuss the construction of Ponyta, an extension of the Pony programming language with value-dependent types. I have extended the existing Pony compiler to support types parametrised on values. I have enhanced the standard library with new data structures which utilise value-dependent types, creating efficient stores for data. And I have extended the Pony compiler so as to evaluate complex compile-time expressions.

Acknowledgements

Thanks to Sophia Drossopoulou and Sylvan Clebsch for providing much aid and support whilst supervising this project. Thanks also to Juliana Franco for providing benchmarks used to evaluate this project. Finally, thanks to George Steed for helping to consider parts of this project.

Contents

1 Introduction

Pony is an object-orientated, actor-model, capabilities-secure programming language[8]. In Pony we can create an array of elements through a parametrised class `Array[T]`; this class represents a contiguous, resizeable memory container for elements of type `T`. There is no concept of `null` in Pony and all accesses to the container are guaranteed to be safe, unlike in C where we can access past the end of an array. However, all of the information regarding the size of the container and the indices for accesses is only known at runtime, this prevents us from optimising for the size of the structure and verifying accesses at compile time. What if we wanted to define a container `Vector` that is parametrised not only by the type of its elements but also by how many elements it has, perhaps as follows:

```
class Vector[A, size: USize]
  fun apply(i: USize): A ? =>
    // code for getting ith element of the vector with sanity checks

  fun update(i: USize, value: A): A^ ? =>
    // code for updating the ith element of the vector with sanity checks
```

This defines a contiguous memory container for `size` elements of type `A`. We could then create and use an instance of a `Vector` such as in the following:

```
actor Main
  new create(env: Env) =>
    let vector: Vector[Student 4] = Vector[Student, 4].create()
    vector.update(0, Student.create("Tim"))
    let student: Student = vector.apply(0)
```

Creating a fixed size container of `Students` which is guaranteed to have 4 elements. Providing the size of the structure statically and fixing this size opens the possibility for more efficient data layouts and optimisations. For example, we can now allocate exactly enough memory for the elements of the container together with the encapsulating object.

These layouts and optimisations are not so simple with an `Array` as we may be able to resize the data, thus the elements are stored externally to the object leading to indirection in accesses. Similarly the size is only known dynamically, therefore it is non-trivial to optimise code based on the size of the `Array`.

The benefits of static information extend beyond more efficient data layouts and code generation. As we have all information regarding the size of these structures we could statically assert some properties that, if violated, cause compilation to be aborted. Consider we defined another access method in our `Vector`:

```
class Vector[A, n: USize]
  fun get[i: USize](): A =>
    # Assert(i < n)
    // code for getting ith element without sanity checks
```

Here the `get()` method is also parametrised on a `USize` and the `#` denotes an expression we want evaluated at compile-time. Now we have the ability to statically ensure accesses are valid.

```
actor Main
  new create(env: Env) =>
    let vector: Vector[Student, 4] = Vector[Student, 4].create()
    vector.get[2]()  // compiles
    vector.get[14]() // fails to compile
```

Furthermore, consider expanding parametrisation of types to include values defined by other types instead of just integers. Such as defining a `Matrix` whose dimensions are part of the type signature by using a `Vector` as defined above. A possible definition could be:

```
class Matrix[A, n: USize, dims: Vector[USize, # n] val]
  embed _data: Vector[A, # _alloc()]

  fun tag _alloc(): USize =>
    // calculates the number of elements in the matrix

  fun _calculate_address(indices: Vector[USize, # n]): USize ? =>
    // calculates the address of an element

  fun apply(indices: Vector[USize, # n]): this->A ? =>
    _data._apply(_calculate_address(indices))

  fun ref update(indices: Vector[USize, # n], value: A): A^ ? =>
    _data._update(_calculate_address(indices), consume value)
```

The `Matrix` type depends on the value of `n`, which represents the number of dimensions in the `Matrix`, and the value of `dims`, which defines the size of each dimension in the `Matrix`. Note also that the type of the `Matrix` has been defined such that the length of the `Vector` of dimensions must be `n`, therefore attempting to construct a value where this property did not hold would fail to type check. With such a definition we could then construct and use a `Matrix` as follows:

```
actor Main
  new create(env: Env) =>
    let matrix: Matrix[Student, 2, # {3, 4}] = Matrix[Student, 2, # {3,
        4}].create()
    // constructs a 3 x 4 matrix

    let bad_matrix: Matrix[Student, 2, # {7, 8, 9}] = Matrix[Student, 2,
        # {7, 8, 9}].create()
    // fails to compile as too many dimensions provided

    matrix.apply({0, 1})
    // Valid, both indices are within valid bounds

    matrix.apply({1, 6})
    // Invalid, the second index exceeds the bounds of the second
        dimension
```

```

    matrix.apply({6, 1})
    // Invalid, the first index exceeds the bounds of the first dimension
```

We can use these value-dependent types to provide even more assurances at compile-time. Recall that for the matrix multiplication operation $C = AB$, when A is a $n \times m$ matrix and B is an $m \times p$ matrix then C is an $n \times p$ matrix. We can construct a function, using value-dependent types, that ensures that we only multiply matrices of the correct dimensionality. Consider the following:

```
1  actor Main
2    fun matrix_mul[n: USize, m: USize, p: USize]
3      (a: Matrix[U32, 2, # {n, m}], b: Matrix[U32, 2, # {m, p}]):
4      Matrix[U32, 2, #{n, p}] ? =>
5
6    new create(env: Env) =>
7      let m1 = Matrix[U32, 2, # {2, 3}].undefined()
8      let m2 = Matrix[U32, 2, # {3, 5}].undefined()
9
10     let m3 = matrix_mul[2, 3, 5](m1, m2) // will compile
11
12     let m4 = matrix_mul[2, 3, 5](m1, m1) // will fail to compile
```

In this example, the call at line 12 will fail as the second `Matrix` does not have the correct dimensions. Thus we can use these value-dependent types to ensure correctness of expressions and functions.

It can be seen that we are constructing a type system where the type of data is not just a member of a fixed set of types used to ensure that operations adhere to a notion of type safety, but the type depends on the data for which it provides a type. Furthermore, these types can be used to ensure correctness of operations.

1.1 Contributions

In this project I have developed Ponyta; Pony with value-dependent types. In this project I present the following contributions:

- Development of the Pony compiler to support value-dependent types; offering Pony developers a more flexible type system. This implementation allows developers to parametrise and instantiate types using primitive values such as integers and booleans. Developers can also parametrise types on instances of any class which is possible for the compiler to represent as a compile-time object. Part of this development involved defining a notion of subtyping between value-dependent types. This subtyping relies on an equivalence between static values which has also been defined for Ponyta in this project. This development involved considering many design alternatives for the Ponyta type system, selecting those that best fit the existing language.

- Implementation of a pseudo-interpreter for evaluating complex compile-time expressions. Utilising the capabilities and the distinction between single-assignment and re-assignable names in Pony to determine what information is known at compile-time. This information has been used to define a set of rules on the values that can be used within compile-time expressions. The compiler adopts this interpreter to provide a developer with the flexibility to evaluate expressions, which adhere to the given rules, at compile-time. This interpreter works on the compile-time representations of values which have been developed during this project.
- Extension of the Pony standard library with `Vector` and `Matrix` classes which adopt the new value-dependent types. I present the flexibility available using value-dependent types by defining an n-dimensional `Matrix` class using only constructs available to any Pony developer. This report presents the results of benchmarking these new data structures. The results show that these structures can elicit equivalent or better run-times than the standard containers provided in the Pony standard library. These structures present a developer with the opportunity to utilise more efficient containers for building programs.

2 Background

In this background I will discuss the history of Pony, including the features adopted by Pony and the theory underpinning these features. I will discuss how these features are related to Ponyta and several design alternatives. I will also present existing languages and research where value-dependent types have been explored and adopted. These value-dependent types appear in functional languages such as Idris and also in more widely adopted languages such as C++. I explore these areas for inspiration and lessons which where used for building Ponyta. I finally give background on LLVM, multi-stage programming and the existing Pony compiler to provide knowledge on how value-dependent types is incorporated into the compiler.

2.1 Pony

Pony is an object-orientated, actor-model, capabilities-secure programming language[8]. The language adopts many new and existing concepts in type systems and concurrent programming to construct a powerful programming language.

2.1.1 Actor-Model Programming

In typical concurrent programming, threads serve as the unit of concurrent computation. Threads will usually share unrestricted access to the same memory. Therefore, programmers require various synchronisation constructs, such as locks and thread scheduling, to ensure accesses are safe and do not introduce data races whilst avoiding deadlocks; issues which are still difficult to avoid. When a bug is encountered in such a program, trying to resolve it is typically non-trivial as it likely depends on some particular scheduling of interactions between threads which is not easily reproducible.

Hewitt and Agha highlight the main issues in the design of programming languages for concurrent systems as being shared resources, dynamic reconfigurability and inherent parallelism[15]. That is the model upon which the language is based must address handling shared resources which may change, the creation of new objects and the communication of such and that the amount of concurrency available should be evident from the structure of a program using this model.

The actor-model uses message-passing as the basis for concurrent computation. The only way to affect an actor is to send it a communication; an actor will carry out actions, as defined in its behaviour, in response to processing these communications[15]. The delivery of messages is guaranteed in this model however the order of delivery may be non-deterministic. The actor-model is inherently concurrent and all actors in a system carry out their actions concurrently. In Pony each actor executes methods sequentially and processes messages in order, with multiple actors executing concurrently. These actors are far cheaper than typical threads as they do not require locks and context switches[8] due to the inherent concurrency of this model. Typical implementations of concurrent programming languages, such as Erlang, combine this actor model with isolated memory per process to ensure safe concurrency [22]. Using isolated memory means that message passing between actors comes with the overhead of copying messages from one memory area to another. Pony adopts the actor-model but provides a more efficient implementation of message passing by allowing actors to share memory but guarantees safety through a system of capabilities described in section 2.1.3.

The actor-model provides useful results for building parallel programs beyond the inherent concurrency of the model. An actor model can provide modularity of programs; actors only interact with each other via message passing and so the inner workings of one actor is not visible to another. Also, parallel programs built using actor systems can be composed together to build larger systems by extending message passing between these systems[15].

2.1.2 Basic Pony Programs

The program which follows is an example program written using the Pony language. To summarise the following example; we define a class `Math` which has a function that sums two values. We also define two actors, `MathServer` and `Main` which is a subtype of `MathClient`. `Main` has a `MathServer` as a field and when constructed sends `server` a message to invoke the behaviour `send()` with the arguments `(4, 7, this)`. When the `send()` behaviour executes it creates a new `Math` object, calls the method `sum()` with the provided arguments and then sends the `client` the message to invoke the `response()` behaviour with the `result` value. The `Main` actor will at some point execute `response()` and print `result` to standard output.

```
class Math
  fun sum(x: U32, y: U32) : U32 =>
    var result = x
    result + y

trait MathClient
  be response(result: U32)

actor MathServer
  be send(x: U32, y: U32, client: MathClient tag) =>
    let math: Math = Math.create()
    client.response(math.sum(x, y))

actor Main is MathClient
  let env: Env
  var server: MathServer = MathServer

  be response(result: U32) =>
    env.out.print(result.string())

  new create(env': Env) =>
```

```
22      env = env'
23      server.send(4, 7, this)
```

Pony supports named constructors. A constructor is specified by the keyword `new`; here the constructor of the actor `Main` is `create()`. The default constructor for a class or actor is the `create()` constructor and if no constructor is defined, and no `create()` method is defined, then the `create()` constructor is provided by the compiler (this is demonstrated in the `MathServer` actor). The `fun` keyword defines a method; at line 2 we define the method `sum` with two parameters of type `U32` that returns a value of type `U32`. A use of the `sum` method can be seen at line 12. Pony is expression oriented, and the return value of a function is the last expression of that function[8], here the return value of function `sum` is the value `result + y`. Note also that we can declare variables with either `let` or `var`, the former permits only a single assignment to the declared variable whereas the latter allows reassignment.

A declaration may be supplied with a type, such as the declaration `let math: Math` at line 11, or it may be left to be inferred by the compiler, for example `var result = x` at line 3. When we construct an object we specify which constructor to use, at line 11 we explicitly use the `create()` constructor. If no constructor is specified then the compiler attempts to use the `create()` constructor, this is the case when we create a new `MathServer` at line 16. Arguments can be provided to a constructor if necessary, for example `let f = Foo(4)` if the `create()` constructor of `Foo` took one argument.

Pony provides various types including `class`es, `actor`s and `primitive`s. These types are used to instantiate values which have different roles in a Pony program. As we have discussed, `actor`s are the unit of concurrency in Pony. Pony provides `class`es as in other object-orientated languages such as Java, these are used to construct objects. Pony also allows `primitive`s, these are similar to classes; the distinction is that `primitive`s do not have fields and there is only on instance of a `primitive`[8].

To construct a valid Pony program a `Main` actor must be defined and take a single `Env` argument. Like a class, an actor may have fields and methods but it may also have behaviours as described in 2.1.1. A behaviour is denoted by the `be` keyword; we can define the name and parameters of a behaviour but we do not define the return type. A behaviour cannot have a defined return type as we cannot return a value from a behaviour; a behaviour will execute asynchronously so we do not know when it will complete and as such we can not depend on a return value from the behaviour[8]. For convenience the return value of invoking a behaviour is the actor whose behaviour was called, this is to allow chaining of behaviour invocations. An example of a behaviour can be seen at line 18, the `response()` behaviour receives a `U32` a prints out the `result`

At line 11 we define a trait `MathClient` which defines a type and its methods and behaviours. Here the trait is used to define what behaviours a `MathClient` has; we later use the `is` keyword to say that `Main` is a subtype of `MathClient` (this will be discussed further in section 2.1.7). `Main` inheriting from `MathClient` allows us to call the `send()` behaviour of a `MathServer` with `this` as an argument. The `send()` behaviour of a `MathServer` accepts a `MathClient` as one of its arguments. Notice here we also have another keyword `tag`; this is an example of a capability as described in section 2.1.3. The `tag` capability in the signature of `send()` is used to type the actor and denotes that we are only permitted to invoke the behaviours of the actor[24].

2.1.3 Capabilities

Pony combines the actor model with shared memory to improve the efficiency of message passing between actors. This combination avoids the requirement of copying all messages from one actor to another. Sharing memory between actors could introduce the possibility of data races however Pony statically ensures freedom from data races through deny properties[24]. These properties make explicit what operations are permitted on another reference to the same object.

A capability indicates what operations (reads and writes) are denied on other references of the same object. These capabilities form a pair of properties which express what is denied globally and what is denied locally and from these properties the permitted operations through these aliases are derived [24]. The properties provided by each of these capabilities are detailed in table 1.

	Deny global read/ write aliases	Deny global write aliases	Allow all global aliases
Deny local read/write aliases	*iso*		
Deny local write aliases	`trn`	*val*	
Allow all local aliases	`ref`	`box`	*tag*
	(Mutable)	(Immutable)	(Opaque)

Table 1: Capability matrix from [24]. Capabilities in *italics* are sendable.

An example of one of these capabilities is `ref`. `ref` denies global read and write aliases but locally permits both, therefore `ref` is both safe to read and write locally as no other actor can cause a read or write data race. It is important to note that these capabilities are part of the type system and it is in this way that data race freedom is statically ensured.

Take the following example adapated from [24]:

```
1 class List
2   var x: U32 = 3
3
4   fun box size1(): U32 => ...
5   fun val size2(): U32 => ...
6
7 actor Main
8   new create(env: Env) =>
9     let l1: List ref = List
10    let l2: List val = List
11
12    l1.size1()
13    l1.x = 4
14    // Invalid, will fail to type check
15    l1.size2()
16
17    l2.size1()
18    l2.size2()
19    // Invalid, will fail to type check
20    l2.x = 5
```

The method `size1()` defined on line 4 includes a reference capability. The reference capability on the methods makes explicit the capability required of an object on which `size1()` is called (this is known as the receiver). The three capabilities used in this example are `ref`, described earlier, `val` which denies global and local write aliases and also `box`. `box` guarantees that if an actor has a reference to an object then no other reference to this object can be used by another actor to write to the value[24]. In `size1()` we know that no other actor can write to the object on which the method was called; this means that it is safe to read the object and enforces a read-only behaviour of the object[24]. `box` permits local write aliases which is demonstrated in the example above; we create `l1` which is of the type `List ref`. `List ref` allows us to locally read and write to the value of the the list. We go on to call `size1()` on `l1`; the call of `size1()` is safe as `ref` permits only local write aliases. As an actor executes sequentially, no read/write data races can occur due to a single actor reading and writing a value at the same time. We cannot call `size2()` on `l1` as `val` demands that we have no local or global write aliases, a weaker claim than that of `box`. Therefore, the call of `size2()` will not successfully typecheck. We repeat similarly with `l2` which is of type `List val`. We can call both `size1()` and `size2()` on `l2` as the deny properties are the same or greater than what is required, however we cannot write to the value as is attempted at line 20.

The type system contains more capabilities such as `iso` which indicates that an actor has isolated access to an object. `iso` permits local read and write aliases but denies global read and write aliases. Capabilities are further explored in [24]. The capabilities determine which values are safe to send between actors and what an actor may do with a reference to an object and so we can begin to see how they make concurrent access of shared memory safe.

Challenges and Applications for Ponyta

Capabilities were of importance in determining whether expressions were valid type parameters. We need to consider the following two definitions of `C1` and whether both a legal definitions:

```
class C1[n: Student val]

class C1[n: Student ref]
```

One answer is that we need to differentiate between the capabilities of value-dependent types. The capability needs to be known in the definition of the class so that we know in what ways it can be used, for example whether it is safe for the value to be sent to another actor. The capability also needs to be considered so that we know what operations can be performed at compile time. If we consider `val` to enforce read-only behaviour such as `const` in C[24], then this impacts what we statically know about our value, such as the values of fields in an object. However, allowing varying capabilities on these type parameters would mean that we could manipulate the value on which a type depends. For example:

```
class C1[s: Student ref]
  fun apply(age': U32) =>
    s.age = age'
```

This would mean that our static information could be mutable and we would require tracking such changes to correctly use the values.

Alternatively, we could enforce that all values provided as type arguments must have a `val` capability. This demands that a value, and transitively all their fields[24], are only readable; this would also allow us to send the values between actors. If we adopted this approach then we would only need to write definitions of the form:

```
class C1[n: Student]
```

Here the capability `val` would be implicit (this has been left as an extension). This would use the value in a more functional approach in that the information would be immutable and we would statically know everything about the value as it would not change after construction (provided we knew how it had been constructed).

In Ponyta we have adopted the approach that all value parameters must have `val` capability, we discuss this choice further in section 5.

Capabilities, and capability modifiers, also had an important role in the design of the APIs for `Vector` and `Matrix`. Consideration had to be given to which capabilities were required of receivers and arguments in these APIs to develop safe and useful containers.

2.1.4 Generics

Pony allows a programmer to define classes and actors, as well as traits and interfaces discussed in more depth in section 2.1.7. Pony's type system supports generics to allow for defining polymorphic classes and actors. These generics are used to create types parametric as to another defined type. Consider the following:

```
1  class C1[A]
2    let f: A
3
4    new create(f': A) =>
5      f = f'
6
7  actor Main
8    new create(env: Env) =>
9      let s = Student.create()
10     let c1student = C1[Student](s)
11     // Valid, will type check
12
13     let c1u32 = C1[U32](s)
14     // Invalid, will fail to type check
```

Line 1 defines the class `C1` which is parametrised on the generic type `A`. This allows us to instantiate a `C1` with any type as an argument. We instantiate these types by providing a type argument, constructing a new definition where all references to type parameters have been replaced with the respective type argument. For example, at line 9 we create `c1student` of type `C1[Student]`, this will have the field `f` of type `Student`. This field is initialised in the constructor `create()` using the `Student` argument. Conversely, the constructor call at line 13 will fail to type check as we pass an argument of type `Student` to the constructor of `C1[U32]` when an argument of type `U32` is expected.

Pony also supports constraints for these generics type as in the following example:

```
1  class C1[A: Number]
2    // definition of C1
3
4  actor Main
5    new create(env: Env) =>
6      let c1u32 = C1[U32]
7      let c1i64 = C1[I64]
8
9      // Invalid, will fail to type check
10     let c1student = C1[Student]
```

This defines a class `C1` parametrised on a type `A` which is constrained to be a subtype of `Number`. This constraint means we can construct the values at lines 6 and 7 yet we cannot construct a value of type `C1[Student]` as we attempt to at line 10.

A programmer is allowed to provide the default type of a generic; this permits definitions of classes such as:

```
class C1[A: Number = U32]
```

Here the default type of `A` is the type `U32`. This would have allowed us to write line 6 in the above as the following:

```
let c132 = C1
```

My project is concerned with extending this parametrisation of types to further consider types which are parametrised on values. This will involve extending the syntax for defining these type parameters as well as defining the semantics of these types and how they may be used. The issues involved with this will be explored throughout this background.

2.1.5 Variance

Programming languages often support a notion of subtyping with variance in generics; for example Scala supports variance annotations for generics and Java supports wildcard types[26].

Covariant subtyping in generics allows use of a subtype of the type parameter in place of the desired type. Take the following example in Java:

```
1 class Shape {}
2 class Square extends Shape {}
3
4 ArrayList<? extends Shape> list = new ArrayList<Square>();
5
6 // this will fail to type check
7 ArrayList<Shape> list2 = new ArrayList<Square>();
```

The `ArrayList<? extends Shape>` type at line 4 demonstrates covariant subtyping; it allows us to provide an `ArrayList` where the type argument is a subtype of `Shape`; in particular we use a value of type `ArrayList<Square>`. This type checks as a type `C<U>` is a subtype of `C<? extends T>` if `U` is a subtype of `T`. We will only be able to access the element of `list` as if they were `Shape`s, without casting, but this behaviour is desirable in some contexts.

Java classes are type invariant in their use of generics; this means that line 5 in the above example will fail to type check as `ArrayList<Square>` is not a subtype of `ArrayList<Shape>`. Java expects the type arguments to be the same in the static and dynamic type. Only when we introduce wildcards into our generic parameters can we get covariant and contravariant subtyping.

Contravariant subtyping in generics allows the use of a super-type of the type parameter in place of the desired type. Again, exploring this through Java wildcards:

```
class Shape {}
class Square extends Shape {}

ArrayList<? super Square> list = new ArrayList<Shape>();

// this will fail to type check
ArrayList<Square> list = ArrayList<Shape>();
```

Line 4 displays the use of contravariant subtyping as the assignment of a value of type `ArrayList<Shape>` to a variable of type `ArrayList<? super Square>` passes type checking. This type checks because a type `C<U>` is a subtype of `C<? super T>` if `T` is a subtype of `U`, this is contravariance. Once again, compare this to the assignment without wildcards at line 5, this will fail to type check.

This use of variance in Java is called use-site variance[26] and is fairly expressive. Other languages, such as Scala, support declaration-site variance where the variance on generics is declared with the definition of the class through variance annotations. An example from [26] demonstrates their use:

```
trait Func[-A, +B]
```

The `+` and `-` annotations in Scala define covariant and contravariant type parameters of `Func`, a type `Func[C, D]` is a subtype of `Func[A, B]` if `C` is a super-type of `A` and `D` is a subtype of `B`. These annotations declare that the type `A` may only appear in covariant positions, such as method return types, and that `B` may only appear in contravariant positions, such as method parameter types [14]. This system of typing is less complex and less powerful than the use-site declarations in Java[26].

A type system with variance elicits greater expressivity yet it comes with the caveat that in general type checking becomes undecidable (as is explored in [26]); decidability can be recovered by adopting only fragments of these typing relationships and thus reducing the expressivity of the language. Pony avoids these issues as it is type invariant with subtyping of generics. A result of this is that the language has less expressive power in its typing and that we cannot have programs with assignments such as the one at line 8 in the following:

```
1 trait Shape
2
3 class Square is Shape
4
5 actor Main
6   new create(env: Env) =>
7     // Invalid, Array[Square] is not a subtype of Array[Shape]
8     arr: Array[Shape] = Array[Square].create(4)
```

In the above example, `arr` is of type `Array[Shape]` and we try and assign to it a value of the type `Array[Square]`. This is not permitted as the type system demands type invariance in the type parameters of classes. That is we require that the type on which the `Array` is parametrised must be the same in the declaration and use.

Pony overcomes this limitation in the type system through another form of subtyping which I discuss in section 2.1.7.

2.1.6 F-bounded Polymorphism

We now consider F-bounded polymorphism to motivate the requirement for exploring recursively defined types such as the following:

```
trait T1[x : T1[# x]]

class C1 is T1[# C1.create()]
```

Consider the following example from [21] which I have translated from the mathematical notation used by Canning et. al into Java:

```
public class Main {
  public static interface Movable {
    public Movable move(Integer dx, Integer dy);
  }

  public static class Point implements Movable {
    private int x;
    private int y;

    public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }

    public Movable move(Integer dx, Integer dy) {
      return new Point(x + dx, y + dy);
    }
  }

  public static class Q implements Movable { ... }

  public static class R implements Movable {
    public Movable move(Integer dx, Integer dy) {
      return new Q();
    }
  }

  public static Movable translate(Movable x) {
    return x.move(1, 1);
  }

  public static void main(String[] args) {
    Point p1 = new Point(1, 7);
    Point p2 = ((Point) translate(p1));

    R r1 = new R();
    // This will type check but the cast will fail
    // at runtime as the value is of type Q
    R r2 = ((R) translate(r2));
  }
}
```

We begin by defining an interface `Movable` which represents the class of objects which can be moved. In this class we define a method `move()`. Assume this method is supposed to define creating a new `Movable` object which is at the position of the current object moved by `dx` and `dy`. The return type of this method is `Moveable`; this is the only return type we could give this method to ensure that it was polymorphic and would work for all subclasses of `Moveable`. We go on to define three classes `Point`, `Q` and `R` all of which are subclasses of `Movable`. Then we define a method `translate()` which takes a `Movable` and `moves` the object. Notice that the only return type we could give the method is `Movable` due to the return type of `move()`.

However, consider now that we want our `move()` method and `translate()` method to be more specific. Moreover, we want them to return an object that is of the same type as the original `Moveable` object; that is when we `translate` a `Point` we want a `Point` returned. In the above example we have no alternative but to cast our return values and hope that the values are of the required type. Examples of this can be seen at lines 34 and 39 in the above, the cast of `translate(p1)` is successful yet the case at line 39 fails. This fails as `R`'s definition of `move()` returns a value of type `Q` but given the definition of `Movable` we could not have prevented this in `translate()`.

Before addressing how this can be rectified let us first look at bounded quantification[21]. Consider the following:

```
public class C1<A extends B> { ... }
```

Here, the type parameter `A` ranges over all classes which are a subclass of `B`. Therefore, `B` is the bounds of the quantification of `A`.

We now consider F-bounded quantification, a concept introduced to object-orientated languages by Canning et al. in [21]. Let us define a new interface `FMovable` and redevelop our earlier example to be:

```java
public class Main {
  public static interface FMovable<T> {
    public T move(Integer dx, Integer dy);
  }

  public static class Point implements FMovable<Point> {
    private int x;
    private int y;

    public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }

    public Point move(Integer dx, Integer dy) {
      return new Point(x + dx, y + dy);
    }
  }

  public static class Q implements FMovable<Q> { ... }

  public static class R implements FMovable<Q> {
    public Q move(Integer dx, Integer dy) {
      return new Q();
    }
  }

  public static <T extends FMovable<T>> T translate(T x) {
    return x.move(1, 1);
  }

  public static void main(String[] args) {
    Point p1 = new Point(1, 7);
    Point p2 = translate(p1);

    R r1 = new R();
    // This will fail to type check
    R r2 = translate(r1);
  }
}
```

Here, the `FMovable` interface represents the same class of objects as the `Movable` class from before. Note that `FMovable` is now parametrised on the type `T` and that `move()` is defined to return a value of the type `T`. In the `translate()` method we use an F-bound of `FMovable<T>`, the function ranges over all types `T` such that `T` is a subclass of `FMovable<T>` and types our function such that it takes an argument of type `T` and returns a value of `T`. The F-bound is the crucial point that allows us to create a function of this type, it allows us to range only over the those types such that the operation is defined for that type, these types are the within the F-bound.

We can now guarantee that `translate()` of a `Point` will return a `Point`, demonstrated on line 34. This type checks as we have defined that `Point extends FMovable<Point>`. `translate()` of an object of type `R` will cause a type error as `R` does not extend `FMovable<R>`, `R` is not within the F-bound. This gives us a powerful result that we have polymorphism where we can use a more specific type instead of just the superclass.

Furthermore, we can define classes which are parametrised on a type constrained to be within a certain F-bound. Consider the subject-observer pattern, a subject has many observers which are subscribed to it and notifies these observers when the subjects state changes, we would like to create a tight relationship between a particular subclass of an observer and subclass of a subject. Using F-bounded polymorphism we can recursively define a subject and observer to obtain such a relationship, like so:

```
1 public class Subject<S extends Subject<S, O>, O extends Observer<S, O>>{...}
2
3 public class Observer<S extends Subject<S, O>, O extends Observer<S, O>>{...}
4
5 public class MySubject extends Subject<MySubject, MyObserver>{...}
6
7 public class MyObserver extends Observer<MySubject, MyObserver>{...}
8
9 // this will fail to type check
10 public class BadObserver extends Observer<MySubject, BadObserver>{}
```

The F-bounds on the type parameters of `Subject` and `Observer` are `Subject<S, O>` and `Observer<S, O>`. These constraints create a tight relationship between a class which is a subclass of `Subject` and a class which is a subclass of `Observer`. This requires the type checking of each class to proceed in lock step; when we type check `MySubject` we require that `MySubject extends Subject<MySubject, Observer>` which we defined, we also require that `MyObserver extends Observer<MySubject, MyObserver>` which we also defined, thus type checking passes. Consider line 10, here type checking fails as we require `MySubject extends Subject<MySubject, BadObserver>`, which we do not have.

As we have seen in section 2.1.4 Pony supports generics and so we can use this notion of F-bounded polymorphism. We can construct the subject-observer pattern from above like so:

```
trait Subject[S: Subject[S, O], O: Observer[S, O]]

trait Observer[S: Subject[S, O], O: Observer[S, O]]

actor MySubject is Subject[MySubject, MyObserver]

actor MyObserver is Observer[MySubject, MyObserver]

// this will fail to type check
actor BadObserver is Observer[MySubject, BadObserver]
```

Challenges and Applications for Ponyta

This leads us to consider types which depend on a value of its own type. An example which looks reminiscent of F-bounded polymorphism is:

```
trait T1[x : T1[# x]]

class C1 is T1[# C1.create()]
```

When we were using types, the bounds on the parameter quantified the type to be within the F-bound. Here, we instead say that `T1` depends on a value `x` that is of type `T1[# x]`. When we define `C1` to be a subclass of `T1` we must provide a value of type `T1`, here we only have the option of constructing a `C1`. However, we need to know what `C1` is before we can construct one yet the definition of `C1` depends on the value of a `C1`. This begins to the raise the question of whether recursive definitions, such as `T1`, should be permitted.

Furthermore, we must give thought to how a definition in Pony, such as the following, should be handled:

```
class C1[f: C1 = # C1]

actor Main
  new create() =>
    let c1 = C1
```

This defines a class `C1` which is parametrised on a value of type `C1`. The problem here is that the default value is a new `C1`. The default argument provides a developer with the means to write an infinite type. Instantiating a `C1` with no type argument will continue to create `C1` types which are further parametrised on `C1`. If we attempt to construct this at compile time we will likely enter an infinite construction of `C1` objects and run out of memory. Also, If we tried to type check this we would result in an infinite type tree as we continue to try and type check the default argument and so we can argue that classes defined like this are ill-formed.

This looks like an easy case to recognise syntactically. However, consider cases where the infinite recursion is caused by classes mutually depending on one another as in the following:

```
class C1[c2: C2 = # C2]

class C2[c1: C1 = # C1]
```

This example encapsulates two infinite types which depend upon each other; the issue is that `C1` (and also `C2`) transitively refers to itself through class usage. This relationship between `C1` and itself needs to be derived statically so as to disallow definition of types like this. We discuss the solution to the issues raised due to these infinite types in section 5.3.4.

2.1.7 Structural and Nominal subtyping

Pony adopts two forms of subtyping relationships, structural and nominal. Nominal subtyping is explicit subtyping through names; that is for types U and T, U is a subtype of T if and only if it is declared to be[27]. In Pony nominal subtyping occurs through the use of traits. For example:

```
1  trait Shape
2    fun area(): U32
3
4  class Square is Shape
5    let length: U32 = 4
6    fun area(): U32 => length * length
7
8  class Rectangle
9    let height = 6
10   let width = 4
11   fun area(): U32 => height * width
12
13 actor Main
14   fun foo(shape: Shape) =>
15     // interacts with shape
16
17   new create(env: Env) =>
18     foo(Square.create())
19
20     // Invalid, Rectangle is not a subtype of Shape
21     foo(Rectangle.create())
```

Here a `Square` is declared explicitly to be a `Shape` so we can use an object of type `Square` when an object of type `Shape` is expected, as can be seen at line 18. `Rectangle` has the all methods provided by a `Shape` but has not been declared to be one and so cannot be used in place of one; this means the call of `foo` at line 21 will fail to compile. This form of subtyping allows explicit design intent by a programmer, which can then be verified by a type system, helping to avoid unintentional typing relationships between classes which can lead to allowing objects to be used in places where they perhaps should not have been.

In a system with structural subtyping a type U is a subtype of T if its methods and fields are a superset of T's methods and fields[27]. Pony also supports this notion of subtyping through interfaces:

```
interface ShapeI
  fun area(): U32

class Square
  let length: U32 = 4
  fun area() : U32 => length * length
  fun string(): String => length.string()

actor Main
  fun foo(shape: ShapeI) =>
    // interacts with shape

  new create(env: Env) =>
    // A Square can be used where a Shape is expected
    foo(Square.create())
```

Now `Square` is structurally a `ShapeI` and so can be used wherever a `ShapeI` is expected without the programmer having to explicitly state that a `Square` is a `ShapeI`. This allows for easy introduction of subtyping relations in the future without having to redefine classes to explicitly cater for the the new type; for example, if we were to later introduce the interface:

```
interface Stringable
  fun string(): String
```

Now, without changing the definitions of the classes we have so far, we can use any class which provides a `string()` method in place of a `Stringable`, such as a `Square`. This subtyping has the caveat that it can introduce unintentional type hierarchies. Consider an interface `Drawable` which provides a `draw()` method and the classes `Shape` and `Cowboy` both of which have a `draw()` method[27]; it is quite likely the second class's notion of draw is not what we desire when we require a `Drawable`. Nominal subtyping allows the programmer to be explicit with this relationship.

I first present the `ReadSeq[A]`, the readable interface of a sequence, as defined in the standard library:

```
interface box ReadSeq[A]
  fun size(): USize

  fun apply(i: USize): this->A ?

  fun values(): Iterator[this->A]^
```

With structural subtyping we have a useful result in that we can treat all of `Array[U64]`, `Vector[U64, 3]` and `Vector[U64, 72]` as `ReadSeq[U64]`. This has two important aspects; firstly that when we extend the Pony standard library with `Vector`s all existing code which works with `ReadSeq`s will be able to use `Vector`s. Secondly, notice how we can treat both of `Vector[U64, 3]` and `Vector[U64, 72]` as `ReadSeq[U64]`, this allows us to forget some information about the type if it is not important to us.

Structural subtyping in Pony also provides us with another important result which we now explore. Consider the following Pony program:

```
1 trait Shape
2   fun area(): U32
3
4 class Square is Shape
5   let length: U32 = 4
6   fun area() : U32 => length * length
7
8 // interface T1 which uses a type parameter to define the
9 // the type of the return value from the apply method
10 interface T1[A: Shape]
11   fun apply(): this->A
12
13 class C1[A: Shape]
14   let f: A
15   new create(f': A) => f = consume f'
16   fun apply(): this->A => f
17
18 // interface T2 which uses a type parameter to type
19 // the parameter of the apply method
20 interface T2[A: Shape]
21   fun apply(s: A)
22
23 class C2[A: Shape]
24   fun apply(s: A) => true
25
26 actor Main
27   new create(env: Env) =>
28     let c1a: T1[Shape] = C1[Shape](Square)
29     let c1b: T1[Shape] = C1[Square](Square)
30
31     // this will fail type checking
32     let c1c: T1[Square] = C1[Shape](Square)
33
34     let c2a: T2[Square] = C2[Square]
35     let c2b: T2[Square] = C2[Shape]
36
37     // this will fail type checking
38     let c2c: T2[Shape] = C2[Square]
```

We can see that at lines 29 and 35 assignments which we would expect to be permitted by a type system with subtyping with variance, this is a result of structural subtyping.

We are permitted to use a `C1[Square]` as `T1[Shape]` at line 29 as the type parameter `A` of `T1` only appears in covariant positions, namely in the return type of the method `apply`. Similarly, in line 35 we can assign a object of type `C2[Shape]` to a constant of type `T2[Square]` as the type parameter `A` of `T2` only appears in contravariant positions, namely the parameter of the `apply` method.

Challenges and Applications for Ponyta

Introducing value-dependent types into a type system with structural subtyping requires that we consider the interactions between them. For example consider a program as follows:

```
1  interface I1[n : U64]
2    fun apply(i: U64): U64 =>
3      i * n
4
5  class C1 is I1[2]
6
7  actor Main
8    fun foo(arg: I1[72]): U64 =>
9      arg.apply(12)
10
11   new create(env: Env) =>
12     let i1: I1[72] = C1
13     foo(C1.create())
```

The problem here is that the methods and fields of `I1[2]` are a superset of those of `I1[72]`, they are a superset as the return type and argument type of the `apply` method are the same. Therefore, under the current definition of structural subtyping the `I1[2]` is a subtype of `I1[72]`. Also note that `I1[72]` is a subtype of `I1[2]`

We can question as to whether such a subtyping relationship is correct. Consider what we would expect the return of the call to `apply` to be at line 9. It is likely that we would expect it to be the result of `12 * 72`, but this is not necessarily the case. As can be seen at line 13 we can pass an object of type `I1[2]` as an argument to the method `foo`. Furthermore, provided an object provides a method `apply` which returns a `U32` we could expect any value in the range of `U32`.

There are some solutions that can be proposed in order to address this problem:

- A simple solution is to accept this as the expected behaviour. This behaviour is similar to when we define an interface which is parametrised on a type but the type parameter is not used in the argument nor the return types of any method. If we were to consider such an interface to be called `I2[A]`; then for all `A`,`B` we have that `I2[A]` is a subtype of `I2[B]` and vice versa. This would mean a similar semantics between using values and types as parameters to interfaces which is desirable from a developers point of view. Although there is a subtle difference between the value and type parameters. We can use the value parameters to change the behaviour of methods (e.g. setting bounds while loop conditions) defined in an interface whereas it is more difficult to do so using type parameters.

- A simple and safe solution is to disallow parametrisation on values in interfaces to avoid the ambiguity altogether, this seems to be a fairly strict restriction but it does remove the above problem.

- The problem could be avoided by the following; when an interface is parametrised on a value then only classes which explicitly implement the interface with an equal type argument can be used as an argument. This would mean that the value argument for the interface would be derivable and we would be able to compare whether it is safe to use. So in the above we know that `C1` is a `I1[2]` so we can disallow its use. Yet this means we would lose the use of interfaces as structural subtyping in the presence of value-dependent types.

We must also consider how likely it is for such a case to arise. Notice that the above example is fairly contrived; an interface has been constructed such that it depends on a value yet this value is only used to define the body of a method. The value on which the interface depends has no effect on the argument and return types of any method. This is not typically the way in which one would use structural subtyping. Indeed we can see this quite evidently through the keyword `interface`, these are meant to define some generic interface for using objects which provide a certain API. Similarly, we can also question how useful parametrising an interface on a value is for defining an API. For example:

```
interface I1[A, n: U32]
  fun apply(v: Vector[A, # n])
```

Although quite a synthetic definition, the above does detail how parametrising an interface on a value doesn't gain us much flexibility when using an instantiation of that interface. In the above a more flexible definition would be to parametrise `apply()`. However, there may indeed be cases when such definitions are useful and so we should be hesitant to disallow them.

We discuss the solution which has been adopted later in section 5.3.2.

2.1.8 Intersection, Union and Tuple Types

Pony's type system supports further interesting features; intersection, union and tuple types. We discuss these types here to lead into an issue which can arise from inheriting from value-dependent instances of these more complex types.

The intersection of two types `A` and `B` is represented in Pony by `(A & B)`. This type characterises values which are of both types and provide all of the methods and fields of both types. In Pony this allows a class to inherit fields and methods from multiple traits and interfaces by taking the intersection of them, an example of this follows:

```
class C1 is (Hashable & Stringable)
```

`Hashable` and `Stringable` are interfaces defined in the Pony standard library. This declaration of `C1` means that it is both `Hashable` and `Stringable` and so will have both a `hash()` method and `string()` method.

The union type of the types `A` and `B` is expressed as `(A | B)`. This describes a value that may be a value of type `A` or `B` or both; we can access only the methods of a value of type `(A | B)` that appear in both `A` and `B`. This allows us to write programs such as the following:

```
1 class C1
2   fun foo(): U32 =>
3     // body of foo
4
5   fun bar(): U32 =>
6     // body of fooA
7
8 class C2
9   fun foo(): U32 =>
10    // body of foo that differs to class C1 foo
11
12 class C3
13   fun bar(value: (C1 | C2)) =>
14     // call foo on the value as C1 and C2 both have a foo method
15     value.foo()
16
17     // type error as C2 does not provide a bar method
18     value.bar()
19
20   fun baz() =>
21     // call bar with either a C1 or a C2
22     bar(C1.create())
23     bar(C2.create())
```

The method call `foo()` at line 15 type checks as `foo()` belongs to both `C1` and `C2`, however the call `bar()` at line 18 will fail to type check as is does not belong to `C1`.

Tuple types are used to type values with the product of types, for example we can have:

```
1 class C1
2 class C2
3 class C3
4
5 class C4
6   fun bar(value: (C1, C2, C3)) =>
7     let a: C1 = value._1
8     let b: C2 = value._2
9
10    // type error
11    let c: C1 = value._3
12
13    // destructures the tuple
14    (var a2, var b2, let c3) = value
```

We can access the elements of the tuple by the position of the element in the tuple, this is the operation at line 7, `value._1` gets the first element of the tuple. The type of each element corresponds to the type defined in the tuple type, for example the first element will have type `C1`, therefore the assignment at line 11 will fail to type check as `value._3` has type `C3`.

Challenges and Applications for Ponyta

An issue that needs to be considered is how the following example should be handled:

```
1 trait T1[n: U32]
2   fun apply(): U32 => n
3
4 class C1 is (T1[2] & T1[72])
```

We declared `C1` to be of type `T1[2]` and `T1[72]`, at line 4, and so it will have the methods of both traits. More precisely `C1` will have two definitions of `apply` one which returns `2` and one which returns `72`. The problem arises that we do not know which method should or will be called. We can construct a similar example in the existing language by appending the values to the trait names and reifying the body of the traits as in the following:

```
trait T1_2
  fun apply(): U32 => 2

trait T1_72
  fun apply(): U32 => 72

class C1 is (T1_2 & T1_72)
```

If we attempt to compile the above, the compiler will warn us that `C1` has multiple possible bodies for `apply` and that we must locally disambiguate the definition that we desire. This approach is useful for a developer as it avoids unintentional execution of the wrong method at runtime and so I aimed to achieve a similar result when inheriting methods from parametrised traits and interfaces. This case also arises for union types in a similar way.

This is an issue which was not noticed in the implementation of generics in Pony. Namely, we can successfully compile the following program:

```
trait T1[A]
  fun apply(): U32 =>
    let a = Array[A]

class C1 is (T1[String] & T1[U32])
```

The above will include two definitions of `apply()` in `C1`. The method which is executed is the `apply()` method provided by `T1[String]`. We discuss the implementation of this in section 5.3.3

2.2 Dependent Types

Type systems span a diverse range of methods and type theory to provide typing and checking to programming languages; in many statically typed languages, data and functions are checked with respect to a fixed set of types (this can include classes and so on that are parametrised on other types). These types do not determine the values for which they provide a type, for example statically a value of one type can be exchanged for any other value of the same type without having an effect on the outcome of type checking. To create a deeper connection between predicate logic and type systems Howard and Bruijn introduced dependent types[17]. Dependent types are an extension of traditional type systems which introduce types which depend on the data for which they provide a type, providing a tighter relationship between a value and its type. The flexible and expressive power of these types allows us to statically verify far more than what is typically achievable in a type system. As we will see later these types can be seen to be proof carrying code, allowing us to prove properties of our programs, such as the validity of array access operations.

These types have been implemented in various languages and have been the subject of research which explores what they can be used to prove at compile time. In the following sections I will further explore these kinds of types and how they can be applied to create interesting types that depend on the values of both simple and more complex types, and how these can be used to enable us to statically prove properties in Idris and C++.

2.2.1 Idris

Idris is a functional programming language, similar to Haskell, that has dependent types built in[4]. Here I further explore dependent types in Idris, taking inspiration from Conor McBride's ”Why Dependent Types Matter”[16]. For this section I will use the natural numbers as they are provided in the Idris prelude[4], that is:

```
data Nat = Z | S Nat
```

If we look first at the typical example of vectors with a known length as presented in [18]:

```
data Vect : Nat -> Type -> Type where
  Nil   : Vect Z a
  (::)  : a -> Vect k a -> Vect (S k) a
```

Here we define the data type `Vect` as being constructed from a natural number which denotes the length of the vector, and a type which details the type of the elements of the `Vect`. We use two constructors to inductively define a vector; `Nil` which has the universally quantified type `Vect Z a` for all `a` (this is implicit just as in languages like Haskell), and `(::)` which builds a value of type `Vect (S k) a` given a value of type `a` and vector of type `Vect k a`. Note that the type of a vector is defined in terms of a natural number (in particular in the `(::)` case we use the successor), instead of just the type `Nat`, that is the type of the vector depends on the value of the natural number. Creating a vector of integers of length 3 can then be done as follows:

```
intVec : Vect 3 Int
intVec = 1 :: 2 :: 3 :: Nil
```

Dependent types can be used to relate the input of a function to its output. This allows us to verify that postconditions are maintained by that function. Let us define the `(++)` function for vectors in the following way[18]:

```
6 (++) : Vect n a -> Vect m a -> Vect (n + m) a
7 (++) Nil         ys = ys
8 (++) (x :: xs)  ys = x :: (xs ++ ys)
```

Here we have that the vector that is returned as a result of the append operation is defined to be of the type `Vect (n + m) a`, i.e. a vecctor whose length is the sum of the two input lengths. Idris guarantees that the body of the function adheres to this type signature. To demonstrate that Idris verifies that our result is a vector of the desired type, consider what happens if we define the `(::)` case for the append operation (line 8 above) as follows:

```
8 (++) (x :: xs)  ys = x :: (xs ++ xs)
```

If we try and compile this definition, the Idris type system informs us that this case in the function body does not adhere to the type signature. We get the following error:

```
When checking right hand side of ++ with expected type
        Vect (S k + m) a

When checking an application of constructor :: :
        Type mismatch between
                Vect (k + k) a (Type of xs ++ xs)
        and
                Vect (plus k m) a (Expected type)

        Specifically:
                Type mismatch between
                        plus k k
                and
                        plus k m
```

The type system has statically determined that the function body does not maintain the invariant that the result is a vector whose length is the summed length of the two input vectors (defined in the type signature); here the error tells us that the types `plus k k` and `plus k m` cannot be made to unify (i.e. the lengths may differ). It is here that we can start to observe that dependent types become a form of proof carrying code. We can ignore this information and make the append more general. For example, if we defined append to have the type:

```
(++) : Vect n a -> Vect m a -> Vect x a
```

Then, we lose information about the result, we no longer say anything about the length of the resulting vector as we have replaced the length of the vector, `(n + m)`, with `x` so it can be any length. Statically, we would not detect any error if we redefined the last line to be:

```
(++) (x :: xs) ys = x :: (xs ++ xs)
```

as the postcondition of the function still holds as the result of the new functions still has a length.

If we use the earlier defined `intVec` we can see the result of performing an append (using the correct definition of append).

```
intVec ++ intVec
[1, 2, 3, 1, 2, 3] : Vect 6 Int
```

As would be expected we get a vector of type `Vect 6 Int`.

Now we can look at defining matrices as dependent types where the dimensions of the matrix are part of the type of the matrix. We will here define a matrix as a vector of vectors. We do this as defining a multi-dimensional matrix in Idris is non-trivial.

```
data Matrix : Type -> (n : Nat) -> Vect n Nat -> Type where
  MNil  : Matrix a Z Nil
  MCons : Vect m a -> Matrix a k dims -> Matrix a (S k) (m :: dims)
```

This data type is defined in a similar manner as a vector but note that the definition of the matrix is dependent on both the value of `n` (the number of rows in the matrix) as well as dependent on the value of `Vect n Nat` (the vector defining the dimension of each row) that we just defined. This demonstrates that types can depend on the value of more than just simple types such as natural numbers. Also note that the value `n` is the same in both the matrix and the vector. Using this definition we can construct the following matrix:

```
intMat : Matrix Int 2 [3, 2]
intMat = MCons [7, 9, 6] (MCons [7, 9] MNil)
```

Yet it is not possible to construct any value of the type:

```
Matrix Int 2 [1]
```

We could also define a function with the following type signature:

```
addM : Num a => Matrix a n v -> Matrix a n v -> Matrix a n v
```

That enforces properties that the two input matrices are not only of the same type but of the same dimensions and that we get a result of the same type. Here `Num a =>` states that `a` must be a member of the class `Num` so that we can perform addition with the elements of the matrices.

The above is the main link between dependent types and my project, that is representing statically known values in types. I now further discuss the power of dependent types and how they allow us to express a range of properties of programs beyond just specifying the size of data.

We can also define relations between values as dependent types themselves so that we can prove stronger postconditions of our functions and data types. The following investigates Conor McBride's approach to defining `Order` as a dependent type and using this to later define operations that assert in their type that their result is ordered. The following examples can be found in [16] written in McBride's language Epigram however here I have translated them to Idris to further explore dependent types.

Consider if we were to define a data type such as:

```
data Order = LE | GE
```

When type checking a function, the values `LE` and `GE` could be used interchangeably without causing changes to the static result[16]. Such an alteration would only manifest itself at runtime; therefore this would allow us to only express ordering between elements when we execute the program which would have to be asserted dynamically. However, by making `Order` a dependent type we can give meaning to the data statically[16]. Let us first define:

```
data Le : Nat -> Nat -> Type where
  LEZ : Le Z x
  LES : Le x y -> Le (S x) (S y)
```

Here we first construct the data type `Le` to inductively define the less than or equal relation on natural numbers as a dependent type. This would mean that `LES (LES LEZ)` would be typeable with of the types `(Le 2 2), (Le 2 3), (Le 2 4), ...` and so we cannot construct the value without some context. Note that although `Le 5 2` is a type, we cannot construct any value which has this type.

We then proceed to use this definition to express how two natural numbers are ordered in the data type `Order`:

```
data Order : Nat -> Nat -> Type where
  LE : (Le x y) -> Order x y
  GE : (Le y x) -> Order x y

order : (x: Nat) -> (y: Nat) -> Order x y
order    Z      y = LE LEZ
order (S x)     Z = GE LEZ
order (S x) (S y) with (order x y)
  | LE xley = LE (LES xley)
  | GE ylex = GE (LES ylex)
```

Compare this revised definition to the original definition of `Order`, we have now given meaning to the numbers in the constructors of `Order`. Finally, we define the function `order` which given two natural numbers `x`, `y` gives a result which is of the type `Order x y`; guaranteeing that the result is ordered.

It is then apparent that as we can express the local ordering between pairs of elements in the type `Order`, we can do as McBride displayed[16] which is that we can express that a list is ordered in its data type.

```
data OList : Nat -> Type where
  ONil  : OList b
  OCons : (x : Nat) -> Le b x -> OList x -> OList b
```

In the above definition, `b` represents the lower bound of the list i.e. the least value that can be appended to the front of the list. To demonstrate this data type in use, we can construct the value:

```
OCons 3 (LES (LES LEZ)) ONil
```

but not the value

```
OCons 1 (LES (LES LEZ)) ONil
```

as 1 is less than the lower bounds of the list, namely 2 represented by `(LES (LES LEZ))`, so we cannot add it to the front of the list.

We can now express that the resulting type of a merge of two ordered lists is an ordered list.

```
merge : OList b -> OList b -> OList b
merge ONil ys = ys
merge xs ONil = xs
merge (OCons x blex xs) (OCons y bley ys) with (order x y)
  | LE xley = OCons x blex (merge xs (OCons y xley ys))
  | GE ylex = OCons y bley (merge (OCons x ylex xs) ys)
```

Utilising these we can construct a very crude sorting algorithm that takes a list of natural numbers and provides the postcondition that the result is ordered.

```
sort : List Nat -> OList 0
sort = (foldl1 merge) . map (\x => OCons x LEZ ONil)
```

McBride details a more involved merge sort algorithm that better explores sorting the list, however the above algorithm demonstrates the point I wish to make; our function carries a proof of sorting a list and statically provides meaning to its operation. Take for example the execution of:

```
sort [3, 1, 2]
```

We will get a value which is guaranteed to be ordered. Namely we get the following:

```
OCons 1 LEZ (OCons 2 (LES LEZ) (OCons 3 (LES (LES LEZ)) ONil))
```

Which if were to "flatten" this structure into a list would be:

```
[1, 2, 3]
```

Thus we have related the input data with the output data and we have used our type system to construct a proof that the postcondition regarding the resulting list (i.e. the ordered property of the lists) is guaranteed by performing the operation.

A point worth noting here as that the ordering example is only defined for the natural numbers, as such we can only have `OList` of natural numbers. It would be ideal to extend this so that we could have `OLists` of any member of a type class like `Ord` (the class of totally ordered data types). The `OList` data type depends on `Le` so we would also have to make this data type polymorphic in some way. Also `order` is used in `merge` and this function too depends on `Nats` due to the definition of the data type `Order` which depends on `Le`. Here `Le` is coupled tightly to the structure of `Nat` making defining `Le`, `Order` or `OList` polymorphically difficult.

It is fairly evident from the above that obtaining this information in a type can become quite involved. The vector is fairly straightforward to define in terms of dependent types and we get some nice simple results from its use, even the matrix example is reasonably straightforward. However, when we begin to define the orderings of natural numbers, our types and functions which interact with them become notably more complex. McBride's language Epigram expresses these data types in a natural deduction format[16] which makes them clearer than when defining them in Idris. Observe that the merge operation requires the two ordered lists to have the same lower bound, we would perhaps like to instead have that the new lower bound is the least of the two input lists but this would require expressing "minimum" as a dependent type also and then including this in the type of `merge`, further increasing the complexity of our typing. This demonstrates how these dependent types, although very powerful, can quickly become unwieldy to construct and use.

It is not only the programmer who faces a more complex task when using dependent types, they are free to use them to whatever expressiveness they wish, but the type system has to become more advanced to support these types. The type system needs to be able to ensure that the type `Vect (1 + 2) Int` is the same as the type `Vect 3 Int` as well as being able to infer the values of dependent types in some cases.

In fact type checking in the presence of dependent types can be undecidable, therefore a type system which caters for them must implement some method to handle this, this is addressed in Epigram by using general recursion in types to ensure that type checking terminates. Alongside the increase in complexity of types is the increase in complexity of code, for example we have to handle more complex data structures when writing functions, this can be seen in the `merge` function above.

2.2.2 C++

C++ does not support the rich system of dependent types provided by Idris, however it does support templating on non-type parameters as well as type parameters [12]. These templates can be used to construct both templated classes and functions. Consider the following example:

```
template<uint32_t lo, uint32_t hi>
uint32_t extract(uint32_t bits) {
  static_assert(hi >= lo, "hi must be greater than or equal to lo");
  constexpr uint32_t mask = ((1 << (hi - lo + 1)) - 1) << lo;
  return (bits & mask) >> lo;
}

int main() {
  extract<1, 4>(3);

  // this will fail to compile
  extract<5, 1>(17);
}
```

In the above we define a templated function for extracting the bits from a number between the range of `lo` and `hi`. This is template meta-programming; we defined a general way of performing an operation, by writing the call `extract<1, 4>(3)`. This call will create a function with `lo` set to `1` and `hi` set to `4` that will then be called with the argument `3`. We have also used `constexpr`, this tells the compiler that the value of `mask` is a compile time expression. To ensure that the operation is defined on a range that makes sense we used `static_assert`; this checks our expression at compile time and if it fails then so does compilation, an example of this is at line 12[12]. These templates have some important properties; firstly, they allow us to define all functions for extracting bits from value in a general way, but we could have done this by passing `hi` and `lo` as arguments. Secondly, using templates gives us a faster function at runtime as we do not have the overhead of passing and looking up arguments, also we are able to push some of the cost of evaluating expressions into the compiler, such as the evaluation of `mask`. At runtime the function that is executed at the point of the call at line 9 is the following:

```
uint32_t extract<1, 4>(uint32_t bits) {
  uint32_t mask = 30;
  return (bits & mask) >> 1;
}
```

Utilising these templates, we can once again construct the vector with a statically known length as before:

```
template <typename T, size_t S>
class vector
{
  T arr[S];
public:
  vector(T(&array)[S]) {
    for (size_t i = 0; i < S; ++i)
      arr[i] = array[i];
  }

  T& operator[] (size_t nIndex) {
    assert(nIndex < S && "Index exceeds dimensions of vector");
    return arr[nIndex];
  };

  /* remaining definition of vector */
};
```

We can then define functions which statically validate accesses to the vector (these could have been member functions of a vector):

```
template <size_t nIndex, typename T, size_t S>
T get(vector<T, S> &v) {
  static_assert(nIndex < S, "Index exceeds dimensions of vector");
  return v[nIndex];
}

template <size_t nIndex, typename T, size_t S>
void set(vector<T, S> &v, T e) {
  static_assert(nIndex < S, "Index exceeds dimensions of vector");
  v[nIndex] = e;
}
```

As is often the case when we move to imperative programming from functional programming, the checks we perform are made explicit by the programmer. Here `static_asserts` check that the accesses are within the bounds of the vector at compile time. Compare the vector accesses which are validated statically against those which are validated dynamically. Take the following vector:

```
int a[2] = {1, 2};
vector<int, 2> v(a);
```

We can access the vector in one of the two following ways:

```
get<1>(v);
v[1];
```

In this case the accesses are valid and will compile and execute as expected. Now compare this to the example where the access is out of the vectors bounds:

```
get<42>(v);
v[42];
```

Both accesses are invalid and will fail an assertion, however the first access will violate an assertion statically and will fail to compile. The second example will successfully compile and the access will only trigger the assertion when executed. It is not always simple or possible to statically verify accesses and so we need to use the static information in conjunction with the dynamic information. For example, consider the following `append` operation:

```
template<typename T, size_t S1, size_t S2>
vector<T, S1+S2> append(vector<T, S1> &v1, vector<T, S2> &v2) {
  vector<T, S1+S2> b;
  for (size_t i = 0; i < S1; ++i)
    b[i] = v1[i];
  for (size_t i = 0; i < S2; ++i)
    b[S1+i] = v2[i];
  return b;
}
```

This provides the same guarantees about the size of the return vector as we had in Idris, however all accesses here are validated dynamically.

Although constructing the vector example is rather straightforward, producing the matrix class is less so. This is due to restrictions in C++ on non-type template parameters. We cannot simply create a class such as:

```
template<typename T, size_t s, vector<s, T> dims>
class Matrix {
  /* definition of a matrix */
}
```

We can construct such a definition in Ponyta as we will see later in section 7.

To obtain a definition like the above we use a pointer as the template parameter. C++ templates do not allow templating on an object; it is only possible to template on a pointer to an object (as in the listing below) or a reference to an object. We also need to create a global static `Vector` to pass as a type argument when constructing a `Matrix`. This looks like the following:

```
template<typename T, size_t s, Vector<s, T>(*dims)>
class Matrix {
  /* definition of a matrix */
}

static int sa[2] = {7, 4}
static Vector<int, 2> sv(sa);

int main {
  Matrix<int, 2, &sa> mat;
}
```

Pony does not distinguish between a value and a pointer to a value, as everything is implicitly a pointer to a value. Therefore, we do not need to create this restriction in an implementation of value-dependent types for Ponyta. When we parametrise a type on a value in Pony we will be able to use a type as the constraint for the value and the type will imply whether a pointer to a value is provided, in the case of objects, or a literal value is used, as in the case of primitive literals.

Observe in the vector example above the internal representation of vector was not `const` and so our `get()` method cannot return a `constexpr` value. This would prevent us using any of the elements of the static `Vector` to perform access validation in the `Matrix`. This is a limitation with how we defined a vector, another representation of `Vector` can be seen in Appendix A. This approach represents the vector as a linked list and combines C++ templates with the meta-programming library to traverse the data structure and obtain values, as all nodes are `const` we can know these values at compile time. Using the linked list definition it is possible to create the `Matrix` as above.

Lessons for Ponyta

The C++ approach to templates is closer to what has been adopted in Ponyta. The concerns for the templates in Ponyta and C++ involve what we define to be a value type parameter. In C++ these values are restricted to integral types, pointer types, pointer to member types, enumeration types and lvalue reference types (taking a variable by reference as the template parameter)[12]. This means we cannot always use compile-time known values, also known as literal types[1], directly as template arguments. In C++ template arguments must be immutable, although one can template on an immutable pointer to a mutable value. The requirement of immutable template arguments is demonstrated in the following C++ program:

```
1 class C1
2 {
3 public:
4   const int x;
5   constexpr C1(int x): x(x) {}
6 };
7
8 template<const C1 &c> class C2 {};
9
10 static constexpr C1 c1(12);
11
12 int main()
13 {
14   C2<c1> c2;
15 }
```

In the above, C++ demands that the template argument `c1` be `constexpr`. If we did not provide a `constexpr` constructor for `C1` and did not make `c1` `constexpr` then the instantiation at line 14 would fail.

It is also worth considering what constitutes as a compile time expression. In C++ a programmer makes explicit what they want and expect to be statically known or statically computable through the `constexpr` keyword. The statically known values are referred to as literal types[1]; these may be assigned to `constexpr` variables as well as being returned from `constexpr` functions. There are rules which define what can be a literal type in C++, this includes scalar types, reference types and arrays of literal types. A programmer can extend this set of types by defining a class with a `constexpr` constructor and using only `const` members. A programmer can also define functions as `constexpr`, these are functions where the result is be computable at compile time, this imposes some restrictions on how the function can be defined (for example looping constructs could not be used in C++11). In Ponyta we attempt to impose few restrictions, allowing values that we can track through and between compile-time expressions. We will discuss the rules in Ponyta more in section 4.

We must also consider whether a programmer should make explicit the expressions they want evaluated at compile-time; this is the approach taken in C++ and also in Ponyta. An alternative is to implement the compiler such that it can infer which expressions can be evaluated at compile-time. There are trade-offs in either solution. If we adopt the inference approach then we avoid an over use of keywords as well as making the use of the language easier for a programmer. Inferring compile-time expression may also allow unintentional use of these compile-time expressions by a programmer, possibly leading to the compiler reducing code complexity is some cases.

Inferring compile-time expressions comes with the overhead of the compiler inspecting every expression to determine whether it can be evaluated at compile-time. Enforcing that a programmer marks expressions that they want evaluated at compile-time does not result in such an overhead. Marking expressions also ensures a programmer knows as early as possible which compile-time expression failed. Consider a programmer relying on some value being known at compile-time to compute some other value, yet a change in the first prevents the use of the second. These effects from a distance issues do not make for useful diagnostics for a developer.

We can also observe that in C++ we need the values used to instantiate a template to be global, static values. We require them to be global as they may be used across multiple class instantiations and so all objects of such type must have access to the value. Here static defines static storage duration, this means that the storage for the object is allocated when the program begins and deallocated when the program ends and that only one instance of the object exists [11]. These are properties of objects used for defining types that have been considered when implementing types dependent on objects in Pony. We have achieved this by using global constants provided by LLVM, we discuss these further in section 4.1.3.

2.2.3 Index Types

Xi and Pfenning introduced the dependently typed functional language DML (Dependent ML) with parametric types [29]. Types may be declared to depend on a value which is constrained to be of a type drawn from a different set of types, only *int* and *bool* are presented in [29]. The value on which a type depends are not constrained to be only of some type but may also be constrained to satisfy some predicate. Consider an equivalent comparison with a Ponyta program:

```
class C1[n: {s: USize | s > 10}]
```

Here the class `C1` depends on a `USize` which must be greater than 10. The parametric types are used to track properties of a type such as the length of a list, and how additions and removals from a list effect the length of the resulting list.

Campos and Vasconcelos extended these parametric types to imperative languages[20, 19]. These types are referred to as index types, an example of such a type is `int<7>` which types the value `7`. The novelty of these index types is that the type may change over the lifetime of a program. For example a variable `i` incremented on each iteration of a loop may initially have the type `int<0>`, then `int<1>` and so on[20]. The index types can then be used to track the mutation of an object through a method. For example, tracking that a list object which has type `list<n>` has type `list<n + 1>` upon completion of a `append()` method. [19]

These types provide a very rich and novel means for providing types to values. However, both [20, 19] conclude that an issue which must be addressed in any implementation of these types is a careful tracking of aliases of a value. Without careful handling of aliases we could have two references to, say, a list where each reference claims the list has a different length. To handle this, an implementation must ensure that old aliases to a value are no longer usable.

In Pony we can destroy old aliases with a `consume` expression (used to destructively read a value). However, Pony does not enforce that old aliases are destroyed on reads of values by default. We could add extra support to the language to ensure we do not have multiple aliases to a value with an index type. We would also have to add support to Pony to adopt these types that change (however only the value-dependent types may change). Ultimately, I favour the dependent types presented in languages such as Idris adn C++, where the type of a value is fixed as this feels like a better fit to the existing language. Also, as can be seen in section 6, the implementation of the `Vector` should mean that the type of the value never changes as it may lead to issues with respect to memory accesses. The predicates that a type argument must satisfy to be used to create a legal instantiation of a type would be a interesting addition to Ponyta and are considered as an extension to the work presented in this report.

2.3 Multi-Stage programming

In [28], Taha explores the programming concept of multi-stage programming. Taha describes Meta-Ocaml, a language which allows a programmer to annotate how they would like to generate syntactically well-formed OCaml code at runtime. This code is then run using an OCaml interpreter. These annotations represent a kind of template function that allow generating specialised functions for input parameters at runtime. This runtime code generation allows programmers to write general programs that avoid the overhead of recursion by generating the necessary body for the function. Taha explores this in the context of a factorial function which generates all multiplications in a single expression without requiring recursive calls to compute sub-goals.

Links to Ponyta

Consider the C++ templates described in section 2.2.2. A templated function is generated in the intermediate representation of the program when we supply it with type arguments. Generating code may in turn lead to further generation of code if the body of the function contains templated functions or classes. Generating code at compile time means that if we violate the rules of the templates or we violate a `static_assert` then our program will not be built into anything that can be executed or linked against. The approach adopted by C++ is similar to that which is used in the Pony Compiler.

The concept of multi-stage programming as detailed in[28] is closely related to Ponyta as we will be creating types parametrised on values. Providing arguments to a parametrised type will not generate Pony code but will instead generate the intermediate representation for a concrete instance of the type. Generation involves replacing all instances of a parameter with the respective provided value. The difference between meta-programming in Pony and MetaOCaml is that the annotations are used to generate code at compile time in Pony instead of run time, as in MetaOCaml.

2.4 LLVM

The Pony compiler adopts the LLVM compiler infrastructure for backend optimisations and code generation. By adopting the LLVM infrastructure the Pony compiler can focus on providing a front-end for the Pony language and the necessary powerful type-system. Many of the design decisions made during this project are strongly related to the underlying LLVM framework. In this section we will discuss LLVM and introduce some of the syntax so that we may discuss, in depth, some of the design choices in later sections.

The LLVM project provides a range of open-source compiler and toolchain technologies[5]. This toolchain includes the LLVM core libraries which operate on a target-independent IR (intermediate representation) of programs. These libraries involve target-independent and target-dependent optimisations for programs as well as providing multiple backends to lower the IR to work on a variety of architectures.

LLVM uses a static single assignment (SSA) representation for representing programming languages; this representation is used throughout LLVM tools. The IR has three forms. One representation is an in-memory compiler IR used for analysis and optimisation within the LLVM tools. Another representation is a bitcode representation; this representation is used for tools which build upon LLVM. Finally, the IR has a human readable form which we will use throughout this report to explore the design and implementation of value-dependent types in the Pony compiler.

When compiling a program, the Pony compiler instantiates all reachable types and builds descriptions for these types in terms of LLVM `struct`s. The compiler also generates IR for each reachable method. Take the following Pony program:

```
1 class C1
2   let field: U32
3   new create(field': U32) => field = field'
4
5 actor Main
6   new create(env: Env) =>
7     let c = C1(23)
```

Compiling this program generates about 3500 lines of LLVM IR. We will inspect certain aspects of this IR to understand the LLVM syntax and semantics. Let us examine some of the LLVM IR generated for `C1(23)`, at line 7, in the above.

```
1 %C1_Desc = type { i32, i32, i32, i32, i32, void (i8*, %__object*)*, void (i8*, %__object*)*, void (i8*, %__object*)*, void (i8*, %__object*, %__message*)*, void (%__object*)*, i32, [0 x i32]*, [0 x { i32, %__Desc* }]*, [1 x i8*] }
2
3 @C1_Desc = internal constant %C1_Desc { i32 23, i32 16, i32 0, i32 0, i32 8, void (i8*, %__object*)* null, void (i8*, %__object*)* null, void (i8*, %__object*)* null, void (i8*, %__object*, %__message*)* null, void (%__object*)* null, i32 -1, [0 x i32]* null, [0 x { i32, %__Desc* }]* null, [1 x i8*] [i8* bitcast (%C1* (%C1*, i32)* @C1_ref_create_Io to i8*)] }
4
5 %C1 = type { %C1_Desc*, i32 }
```

Here we can see two forms of identifiers. The identifiers with a leading `@` denote global variables [6] whilst those with a leading `%` denote local variables. In the above we have that line 1 defines a local variable (`%C1_Desc`) which describes the structure of the `C1` descriptor header. This header consists of information for an object such as the number of fields, the trace function to use when tracing the object, the virtual table for the object and more information.

At line 4, we construct a global constant instance of `%C1_Desc` which is named `@C1_Desc`, this instance assigns values to all elements of the `%C1_Desc` structure. Finally we create another local variable at line 7, `%C1`, this is another local variable which describes the type of a `C1`.

The `C1` type contains only two elements, the first element is the descriptor header we mention earlier, the second is the `i32` field. We can now look at the IR generate for the `create()` constructor for a `C1 ref`. The generated IR has been reproduced in the following:

```
1 define fastcc %C1* @C1_ref_create_Io(%C1* dereferenceable(16) %this, i32
    %"field'") {
2 entry:
3   %this1 = alloca %C1*
4   store %C1* %this, %C1** %this1
5   %"field'2" = alloca i32
6   store i32 %"field'", i32* %"field'2"
7   %0 = getelementptr inbounds %C1, %C1* %this, i32 0, i32 1
8   %1 = load i32, i32* %0
9   store i32 %"field'", i32* %0
10  ret %C1* %this
11 }
```

Let us first detail the function signature; we defined the global method `@C1_ref_create_Io` which takes as arguments a `%C1*` and an `i32`. The argument types are a pointer to a `C1` object and a `U32` value respectively. The first argument is the receiver on which the method will operate, the second argument is the `U32` argument `field'` that was defined on line 3 of the original Pony program. The `dereferenceable` syntax here states that our pointers may be dereferenced and the size which appears after the `dereferenceable` key words is the size of the pointee type[6].

We now proceed to detail the `create()` constructor. On line 3 we allocate a new `C1*` which obtains a `C1[String val]**` value which is assigned to `%this1`. On line 5, the address of the receiver object (`%this`) is then stored in the memory location represented by `%this1`. On lines 5 and 6 we perform similarly to obtain a reference to the `U32` `field'` argument. The `getelementptr` instruction on line 7 calculates an address to the `field` field in `%this`, the `0` argument looks through the pointer and the `1` argument is used as an offset into the object to get the address of the `field` field. On line 8, the pointer to the `U32` is loaded from the calculated address and assigned to `%1` (this is the used for destructive reads in Pony, however here we will not use the result). On line 9, the `i32` argument `%"field'"` is stored at the address calculated using the `getelementptr` instruction. Finally, on line 10 we return the receiver.

This example illustrates much of the LLVM IR that will be used throughout the rest of this report to explain the code generation aspects in developing Ponyta.

2.5 Pony and Ponyta

The Pony compiler, `ponyc`, is a compiler (written in C) for the Pony language currently under development. It support actors, capabilities, generics, F-bounded polymorphism, structural and nominal subtyping and union, intersection and tuple types as we have discussed so far. On top of this, the Pony compiler supports features such as:

- Delegates
- Exceptions
- Lambda functions
- Pattern matching
- Partial application
- Consume
- Recovery
- Viewpoint Adaptation
- Aliases
- C FFI

The compiler is a multi-pass compiler; it incrementally builds an abstract syntax tree (AST) intermediate representation (IR) from source code and performs static checking on this representation. The AST representation of the program is then lowered to the LLVM IR. The Pony compiler has been extended to support Ponyta.

2.5.1 Multi-Pass Compilation

The Pony compiler takes a Pony program as input and passes over the program multiple times before the final executable is generated. These passes include generating the AST, expanding syntactic sugar, type checking expressions and generating LLVM IR. A summary of the compiler passes, which are of interest for this project, can be found in fig. 1. Here I omit the scope, import, flatten and docs passes.

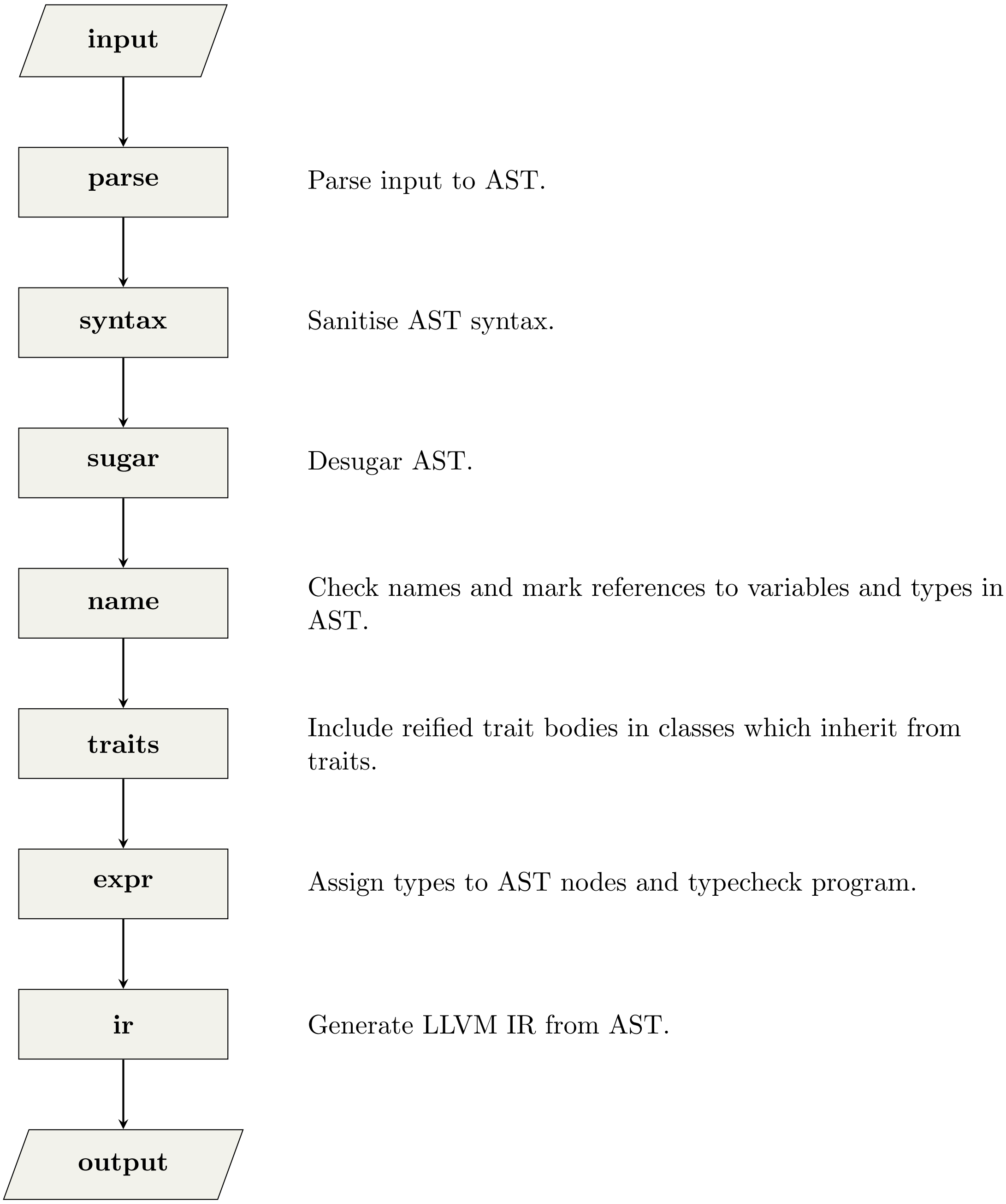

Figure 1: Summary of Pony compiler passes.

The following describes, in detail, the passes of the compiler that have been changed to accommodate Ponyta.

Parse

This pass parses the input program checking that the program adheres to the defined parse rules; It is in this pass that the initial AST representation is constructed.

Changes for Ponyta

This pass has been extended to incorporate new parse rules for parsing compile-time expressions and permitting values to appear within type arguments.

Syntax

The AST constructed by the parse pass represents only a program that adheres to parse rules, this does not necessarily mean that the program which was successfully parsed used valid syntax. For example, a convention adopted in Pony is that identifiers for types must begin with an upper case letters whereas identifiers for variables must begin with lowercase letters. Take the following program which violates this convention:

```
actor Main
  fun foo(arg: u32) =>
    // definition of foo
```

This program would cause no error if only the parse pass of the compiler was used, however when enabling the syntax pass an error would be raised that `u32` is not a valid type identifier.

Changes for Ponyta

This pass has been altered so that type parameters now permit value identifiers as well as type identifiers. Initially this pass had extra restrictions to ensure only type identifiers could appear as parameters. This restriction existed as we could only defined types parametric with respect to other types.

This pass will also be changed to understand the new required syntax of value-dependent types such as enforcing that constraints of type parameters must themselves be type identifiers and not value identifiers.

Sugar

The Pony language defines many abbreviations to allow a programmer to write more concise programs. The abbreviations are expanded in the sugar pass. This pass updates the AST with new nodes which represent the expansions. For example one such rewriting involves calling the default `create()` constructor when no constructor is provided. One could think of this as rewriting the following program:

```
let c: C1 = C1
```

to instead be:

```
let c: C1 = C1.create()
```

Such a process is also referred to as de-sugaring. This pass has not been adapted but it is worth detailing as we will use many abbreviations throughout this report.

Name

This stage is concerned with the names of types that are used in a Pony program. This involves verifying that a type that is used in a program is defined in the program, for example:

```
class C1
  let x: A
```

The error that `A` is undefined will be detected at this stage. This stage also manipulates the representation of parametrised types in the AST. Consider the following example:

```
1 class C1[A]
2   fun id(x: A) => x
```

Parsing this example will build an AST that represents that `x` is of type `A`, where `A` is a type in the program not the type defined by the type parameter. In this stage the definition of the parametrised type is traversed and the uses of the parameter type are replaced with new nodes that make this reference explicit. This setup is used in later stages when type arguments are supplied so that a concrete implementation of methods and fields can be created for subtypes and values. Also, if a class `A` already exists then the above example, specifically line 1, will raise an error during this stage as the parameter will shadow the existing type.

Changes for Ponyta

So as to follow how type parameters are handled so far, this stage has been augmented to incorporate the addition of value-dependent types. For example in the following:

```
1 class C2[n: U32]
2   fun get(): U32 =>
3     n
```

the type parameter `n` on line 3 needs to be distinguished from a variable `n`; this pass will handle replacing these nodes in the AST in much the same way as for types.

Traits

Consider the following:

```
trait T1[A: Number]
  fun id(x: A): A => x

class C1 is T1[U32]
```

The class `C1` here inherits the methods of `T1`; the traits pass is concerned with determining these methods and adding them to the definition of `C1`. When adding these methods the type arguments, here `U32`, are used to replace the type parameters of `T1` to construct an instantiated version of the trait. The instantiated methods get added to the AST of the class definition, therefore the node for `C1` has a subtree representing that it has the method:

```
fun id(x: U32): U32 => x
```

Changes for Ponyta

This pass has been altered to replace value type parameters with the supplied value argument. This replacement will need to ensure that replacing occurrences of references to the type parameter result in the same value being used. If we do not ensure this then we will face the issue presented in the following Ponyta program:

```
trait T1[s: Student]
  fun foo(): Student => s

  fun bar(): Student => s

class C1 is T1[# Student.create()]

actor Main
  new create(env: Env) =>
    let c = C1
```

We rewrite the methods of `c` to be:

```
class C1
  fun foo(): Student => Student.create()

  fun bar(): Student => Student.create()
```

While this above reification would still provide the same result we would not be recognising the type argument provided as the value which is used to reify the class. A way of handling this is to evaluate the expression provided as an argument and replace all occurrences of the type parameter in the definition of the class with a reference to this value. Alternatively, ensuring that the same expression returns the same value (for expressions marked with `#`) will also obtain a working implementation which respects the type argument supplied. We discuss this method more in section 8.

Expr

This pass in the compiler is concerned with type checking and type inference. This pass is also responsible for some transformations of the AST such as distinguishing between a reference to a function and a constructor. The expr pass is responsible for raising errors in programs such as in the following:

```
1 actor Main
2   new create(env: Env) =>
3     let x: String = 3
```

A type error for line 3 will be raised during the expr pass. It is also in this pass that we instantiate classes and type check generics. Consider the following:

```
class C1[A: Stringable]

actor Main
  new create(env: Env) =>
    let c = C1[Bool]
```

The expr pass will construct instantiated definitions of `C1[String]` and `C1[Bool]`. Part of this instantiation involves checking that `Bool` adheres to the constraints of the parameter. Namely, the expr pass ensures that `Bool` is a subtype of `Stringable`.

Changes for Ponyta

This pass required a large amount of work to incorporate types which depend on values as types. This pass has been extended to check the types of values provided as type arguments. We also had to develop the subtyping relationships for Ponyta. This subtyping relationship involved adding logic so that the compiler can test equality between two compile-time values. Consider the following example:

```
1 class C1[s: Student]
2
3 actor Main
4   new create(env: Env) =>
5     let x = C1[# Student.create()]
6     let y = C1[# Student.create()]
7     let z: C1[# Student.create()] = C1[# Student.create()]
```

We needed to decide whether `x` and `y` are of the same type. One solution is to say that they are not, the arguments provided are not the same object and so `x` and `y` are not of the same type. However, this may mean that line 7 represents an invalid assignment. Another solution is to use a notion of equality instead of sameness, this could be in terms of a recursive structural equality (i.e. do two compile-time objects fields have equal values) or could perhaps request a user defined equality method to be provided. We will discuss the solution in detail in section 5.2.

Codegen

This pass is responsible for generating LLVM IR from the AST representation of the provided program. My contributions to this pass of the compiler are in developing generation of LLVM IR for new data structures for Pony as well as generating code that interacts with and uses compile-time values (such as statically constructed objects).

2.5.2 Parser

Pony has an LL(1) grammar. This means that the parser uses 1 token to look ahead when parsing input to decide which parse rule is applicable[25]. LL(1) grammars are both decidable and unambiguous, making the design of tools to work with the grammar much simpler. These grammars also lend themselves to fast parsing algorithms[25]. However, these grammars have the impact that alternate rules have to be unambiguous in their first token; to take an example from the Pony grammar (here given as BNF parse rules), ⟨*typeargs*⟩ are defined to be:

⟨*typeargs*⟩ ::= '[' ⟨*typearg*⟩ (',' ⟨*typearg*⟩)* ']'

⟨*typearg*⟩ ::= ⟨*type*⟩

This lets us write a type arguments of the form:

```
C1[U32]
```

In Pony, an upper-case identifier represents a type and a lower-case identifier represents a value. Therefore, the above expresses providing the type argument `U32` to the type `C1`.

Consider that we now want to express that a type argument may be a value (not a type), such as the following:

```
1 C1[Student.create()]
```

Line 1 above represents us passing a new `Student` object as an argument to the type `C1`. We need to alter the parse rule ⟨*typearg*⟩ to permit parsing of this syntax, we can use a ⟨*postfix*⟩ (the definition of which can be found in section 3) to represent the value. Now we face the issue that the rules for ⟨*type*⟩ and ⟨*postfix*⟩ can both begin with an ID token. Therefore, to make the grammar unambiguous (and LL(1)) we introduce a token, here '#', to differentiate between the two cases:

⟨*typearg*⟩ ::= ⟨*type*⟩ | '#' ⟨*postfix*⟩

In fact, as the token for a literal value, e.g. `2`, cannot clash with a type we can provide a slight convenience to a developer by defining ⟨*typearg*⟩ as follows:

⟨*typearg*⟩ ::= ⟨*type*⟩ | ⟨*literal*⟩ | '#' ⟨*postfix*⟩

The desired meaning can now be obtained but we must write:

```
C1[ # Student.create()]
```

The effects of this choice in grammar are taken into consideration when extending the syntax of the language so as to design an LL(1) grammar, adopting appropriate keywords where necessary.

3 Language Extension

We will first discuss the value-dependent types extension to Pony by considering how we will extend the syntax. Initially we will consider extending the abstract syntax used in the Pony formal model [24]; this syntax has been reproduced in figs. 2 and 3.

$$
\begin{array}{rcccl}
\mathtt{P} & \in & \mathit{Program} & ::= & \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \\
\mathtt{CT} & \in & \mathit{ClassDef} & ::= & \mathtt{class}\,\mathtt{C}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \\
\mathtt{AT} & \in & \mathit{ActorDef} & ::= & \mathtt{actor}\,\mathtt{A}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \\
\mathtt{S} & \in & \mathit{TypeID} & ::= & \mathtt{A} \mid \mathtt{C} \\
\mathtt{T} & \in & \mathit{Type} & ::= & \mathtt{S}\,\kappa \\
\mathtt{ET} & \in & \mathit{ExtType} & ::= & \mathtt{T} \mid \mathtt{S}\,(\mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref})\circ \\
\mathtt{F} & \in & \mathit{Field} & ::= & \mathtt{var}\,\mathtt{f} : \mathtt{T} \\
\mathtt{K} & \in & \mathit{Ctor} & ::= & \mathtt{new}\,\mathtt{k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{M} & \in & \mathit{Func} & ::= & \mathtt{fun}\,\kappa\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e} \\
\mathtt{B} & \in & \mathit{Behv} & ::= & \mathtt{be}\,\mathtt{b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{n} & \in & \mathit{MethID} & ::= & \mathtt{k} \mid \mathtt{m} \mid \mathtt{b} \\
\kappa & \in & \mathit{Cap} & ::= & \mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref} \mid \mathtt{val} \mid \mathtt{box} \mid \mathtt{tag} \\
\mathtt{e} & \in & \mathit{Expr} & ::= & \mathtt{this} \mid \mathtt{x} \mid \mathtt{x} = \mathtt{e} \mid \mathtt{null} \mid \mathtt{e}; \mathtt{e} \\
 & & & \mid & \mathtt{e.f} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{recover}\,\mathtt{e} \\
 & & & \mid & \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e.b}(\overline{\mathtt{e}}) \mid \mathtt{S.k}(\overline{\mathtt{e}}) \\
\mathtt{E}[\cdot] & \in & \mathit{ExprHole} & ::= & \mathtt{x} = \mathtt{E}[\cdot] \mid \mathtt{E}[\cdot]; \mathtt{e} \mid (\mathtt{E}[\cdot]) \mid \mathtt{E}[\cdot].\mathtt{f} \\
 & & & \mid & \mathtt{e.f} = \mathtt{E}[\cdot] \mid \mathtt{E}[\cdot].\mathtt{f} = \mathtt{z} \mid \mathtt{E}[\cdot].\mathtt{n}(\overline{\mathtt{z}}) \\
 & & & \mid & \mathtt{e.n}(\overline{\mathtt{z}}, \mathtt{E}[\cdot], \overline{\mathtt{e}}) \mid \mathtt{recover}\,\mathtt{E}[\cdot]
\end{array}
$$

Figure 2: Syntax

$$
\begin{array}{rclrcl}
\mathtt{C} & \in & \mathit{ClassID} & \mathtt{k} & \in & \mathit{CtorID} \\
\mathtt{A} & \in & \mathit{ActorID} & \mathtt{m} & \in & \mathit{FuncID} \\
\mathtt{f} & \in & \mathit{FieldID} & \mathtt{b} & \in & \mathit{BehvID} \\
\mathtt{this}, \mathtt{x} & \in & \mathit{SourceID} & \mathtt{n} & \in & \mathit{CtorID} \cup \mathit{BehvID} \\
\mathtt{t} & \in & \mathit{TempID} & \mathtt{y}, \mathtt{z} & \in & \mathit{LocalID}
\end{array}
$$

Figure 3: Identifiers

We now go on to extend this syntax to include parametrised classes, actors and methods. The changes to the abstract syntax have been highlighted in figs. 4 and 5.

P	$\in$	*Program*	::=	$\overline{\texttt{CT}}\ \overline{\texttt{AT}}$
CT	$\in$	*ClassDef*	::=	class C $\overline{\texttt{TP}}$ $\overline{\texttt{F}}\ \overline{\texttt{K}}\ \overline{\texttt{M}}$
AT	$\in$	*ActorDef*	::=	actor A $\overline{\texttt{TP}}$ $\overline{\texttt{F}}\ \overline{\texttt{K}}\ \overline{\texttt{M}}\ \overline{\texttt{B}}$
S	$\in$	*TypeID*	::=	A | C
T	$\in$	*Type*	::=	S κ
ET	$\in$	*ExtType*	::=	CN | S (iso | trn | ref)$\circ$
CN	$\in$	*Constraint*	::=	T | I | C$\overline{\texttt{ta}}$
TC	$\in$	*TypeConstraint*	::=	I : CN
VC	$\in$	*ValueConstraint*	::=	v : CN
TP	$\in$	*TypeParam*	::=	TC | VC
ta	$\in$	*TypeArg*	::=	CN | e
F	$\in$	*Field*	::=	var f : CN
K	$\in$	*Ctor*	::=	new k $\overline{\texttt{TP}}$ ($\overline{\texttt{x}}$: $\overline{\texttt{CN}}$) $\Rightarrow$ e
M	$\in$	*Func*	::=	fun κ m $\overline{\texttt{TP}}$ ($\overline{\texttt{x}}$: $\overline{\texttt{CN}}$) : ET $\Rightarrow$ e
B	$\in$	*Behv*	::=	be b $\overline{\texttt{TP}}$ ($\overline{\texttt{x}}$: $\overline{\texttt{CN}}$) $\Rightarrow$ e
n	$\in$	*MethID*	::=	k | m | b
κ	$\in$	*Cap*	::=	iso | trn | ref | val | box | tag
e	$\in$	*Expr*	::=	this | x | v | x = e | null | e; e
			|	e.f | e.f = e | recover e
			|	e.m $\overline{\texttt{ta}}$ ($\overline{\texttt{e}}$) | e.b $\overline{\texttt{ta}}$ ($\overline{\texttt{e}}$) | S.k $\overline{\texttt{ta}}$ ($\overline{\texttt{e}}$)
E[·]	$\in$	*ExprHole*	::=	x = E[·] | E[·]; e | (E[·]) | E[·].f
			|	e.f = E[·] | E[·].f = z | E[·].n($\overline{\texttt{z}}$)
			|	e.n($\overline{\texttt{z}}$, E[·], $\overline{\texttt{e}}$) | recover E[·]

Figure 4: Extended Syntax

C	$\in$	*ClassID*	k	$\in$	*CtorID*
A	$\in$	*ActorID*	m	$\in$	*FuncID*
f	$\in$	*FieldID*	b	$\in$	*BehvID*
this, x	$\in$	*SourceID*	n	$\in$	*CtorID* $\cup$ *BehvID*
t	$\in$	*TempID*	y, z	$\in$	*LocalID*
I	$\in$	*TypeParamID*	v	$\in$	*ValueParamID*

Figure 5: Extended Identifiers

Note that, while the Pony compiler supports generic types parametrised on other types, the formal model does not yet support these generic types[24]. Thus, we extend the syntax to consider types parametrised on both types and values.

We first consider the extensions required to accommodate for type parameters. We introduce a new syntax rule for defining $Constraints$; these are used to constrain the type parameters such that the supplied parameter is a subtype of the constraint. These constraints may be concrete types or they may be other parameters as can be seen in the $TypeConstraint$ rule. The $TypeConstraint$ allows us to constrain a type parameter to be a subtype of the type provided upon declaration of the generic type. These constraints are part of the definition of $TypeParam$, a rule which is used to augment $ClassDef$, $ActorDef$, $Ctor$, $Func$ and $Behv$ such that we can construct parametric definitions of classes, actors and methods. Part of these parametric definitions require that our fields can now be of the type defined by the parameter. Thus we change the $Field$ rule to permit such definitions. Finally, to allow us to instantiate these templates, we define $TypeArg$ which may be a $Constraint$. These type arguments can be seen to be used in $Expr$ as we can now also supply a type to an actor, class or method definition so that we can reify them to obtain a concrete definition.

The extended syntax gives some indication as to which parts of the type system and compiler will need to be altered to support value dependent types as a new language feature. Firstly, we need to extend type parameters to permit value parameters; this can be seen in the definition of $TypeParam$. We permit a $TypeParam$ to be either a $TypeConstraint$ or a $ValueConstraint$, allowing us to distinguish between which kind of parameter we expect. A $ValueConstraint$ allows us to expect a value parameter of a certain type. Note that we could not omit the type constraint of the value parameter otherwise we would not know the type of the value in the template definition and thus we could not type check the template. It would be possible to omit the constraint of $TypeConstraint$ (although here we do not) as this would denote that we permit any type.

In a similar manner to parameters we must also extend type arguments so that we can instantiate types using values. This can be seen in the definition of $Expr$; this definition has been extended such that we can provide type arguments to methods, behaviours and constructors. Notice that the definition of $TypeArg$ allows us to provide either a $Constraint$ or an $Expr$ as a type argument; $Expr$ allowing us to provide values as type arguments.

Also, observe that $Expr$ has been extended so that we can use $ValueParamID$s in expressions, these can be used with classes and methods which are parametrised on values. To utilise parametric definition will require replacement of identifiers by their respective provided arguments.

To utilise parametrised types will also require evaluation of value argument expressions as we will require statically known values to utilise these types. This highlights that we will need to include evaluation of expressions within our type system.

We can surmise from these extensions that we will have to extend the compiler to perform at least the following:

- Extended syntax to incorporate value parameters and arguments for types.
- Reification of definitions; replacing parameter references with argument values.
- Static evaluation of expressions.

We aim to make such changes that are useful and intuitive to a developer, including clear syntax and semantics when evaluating compile-time expressions which match their runtime counterpart.

We now go on to detail a reduced subset of the Pony BNF adopted by the compiler in fig. 6. This syntax extends beyond the abstract syntax presented in fig. 2, for example in fig. 6 we distinguish between 'var', 'let' and 'embed' (rule ⟨*pattern*⟩). Figure 6 includes the extensions necessary for value dependent types (which have been highlighted). The rules which are of note are ⟨*typearg*⟩ and ⟨*term*⟩; I have extended both of these rules to include an alternative of $'\#'\langle postfix\rangle$. The extensions to these rules relate back to fig. 4 in that we can now provide value arguments to parametrised types. This reflects the similar change to the *Expr* rule in fig. 4. In this BNF description we make no distinction between a $TypeParamID$ and $ValueParamID$. This can be seen in the definition of ⟨*typeparam*⟩ which uses an ID to capture both cases. The lack of a distinction between a type identifier and value identifier does not present ambiguity in the compiler. Pony avoids the ambiguity by adopting the convention that all types must begin with an uppercase letter and variables may not begin with an uppercase letter. This convention means we can always distinguish between the two kinds of identifiers.

We use the $'\#'$ to denote compile-time expressions. This syntax is used to indicate that the expression should be evaluated by the compiler (possibly failing as the expression was not possible to evaluate at compile-time). A developer should be aware that the $'\#'$ compiler directive behaves like an operator with the weakest precedence. Therefore an expression such as `# (C1.create(2)).string()` behaves like `# ((C1.create(2)).string())`, parsing the entire expression as a compile-time expression.

Finally, note that, for convenience, I have also extend the syntax to allow programmers to write literal `Vector`s. This syntax can be seen in the highlighted rule in ⟨*atom*⟩. We will discuss `Vector`s further in section 6.

Now that we have discussed the syntax for this extension, we go on to explore how value-dependent types have been supported in the compiler. We will look, first, at compile-time expressions in section 4 and then how they are used to extend Pony's type system with value-dependent types in section 5. We then look at how we have adopted value-dependent types to extend the Pony standard library with a new `Vector` class in section 6.

```
⟨class_def⟩      ::= ('interface' | 'trait' | 'class' | 'actor') ⟨cap⟩? ID ⟨typeparams⟩? ⟨members⟩

⟨members⟩        ::= ⟨field⟩* ⟨method⟩*

⟨field⟩          ::= ('var' | 'let') ID ':' ⟨type⟩ ('=' ⟨infix⟩)?

⟨method⟩         ::= ('fun' | 'be' | 'new') ⟨cap⟩? ID ⟨typeparams⟩?
                     '(' ⟨params⟩? ')' (':' ⟨type⟩)? ('=>' ⟨exprseq⟩)?

⟨exprseq⟩        ::= ⟨assignment⟩ (⟨exprseq⟩)?

⟨assignment⟩     ::= ⟨infix⟩ ('=' ⟨assignment⟩)?

⟨infix⟩          ::= ⟨term⟩ (⟨binop⟩)*

⟨binop⟩          ::= ('+' | '-') ⟨term⟩

⟨term⟩           ::= 'if' ⟨exprseq⟩ 'then' ⟨exprseq⟩ (⟨elseif⟩ | ('else' ⟨exprseq⟩))? 'end'
                   | 'while' ⟨exprseq⟩ 'do' ⟨exprseq⟩ ('else' ⟨exprseq⟩)? 'end'
                   | ⟨pattern⟩
                   | '#' ⟨postfix⟩

⟨elseif⟩         ::= 'elseif' ⟨exprseq⟩ 'then' ⟨exprseq⟩ (⟨elseif⟩ | ('else' ⟨exprseq⟩))?

⟨pattern⟩        ::= ('var' | 'let' | 'embed') ID (':' ⟨type⟩)?
                   | ⟨parampattern⟩

⟨parampattern⟩   ::= ('not' | '-') ⟨parampattern⟩
                   | ⟨postfix⟩

⟨postfix⟩        ::= ⟨atom⟩ (⟨dot⟩ | ⟨typeargs⟩ | ⟨call⟩)*

⟨call⟩           ::= '(' ⟨positional⟩? ')'

⟨dot⟩            ::= '.' ID

⟨atom⟩           ::= ID | 'this' | ⟨literal⟩
                   | '[' ('as' ⟨type⟩ ':')? ⟨exprseq⟩ (',' ⟨exprseq⟩)* ']'
                   | '{' ('as' ⟨type⟩ ':')? ⟨exprseq⟩ (',' ⟨exprseq⟩)* '}'

⟨postional⟩      ::= ⟨exprseq⟩ (',' ⟨exprseq⟩)*

⟨type⟩           ::= ID ⟨typeargs⟩? ⟨cap⟩? ('^' | '!')?

⟨cap⟩            ::= 'iso' | 'trn' | 'ref' | 'val' | 'box' | 'tag'

⟨typeargs⟩       ::= '[' ⟨typearg⟩ (',' ⟨typearg⟩)* ']'

⟨typearg⟩        ::= ⟨type⟩ | ⟨literal⟩ | '#' ⟨postfix⟩

⟨typeparams⟩     ::= '[' ⟨typeparam⟩ (',' ⟨typeparam⟩)* ']'

⟨typeparam⟩      ::= ID (':' ⟨type⟩)? ('=' ⟨typearg⟩)?

⟨params⟩         ::= ⟨param⟩ (',' (⟨param⟩)*

⟨param⟩          ::= (⟨parampattern⟩ | '_') (':' ⟨type⟩)? ('=' infix)?

⟨literal⟩        ::= 'true' | 'false' | INT | FLOAT | STRING
```

Figure 6: Reduced BNF

4 Compile Time-Expressions

We must consider the rules that allow us to know that a expression is a compile-time expression. We also need to consider which capabilities we allow a compile-time expression to have and incorporate. In the following sections I will discuss the rules which must be followed in compile-time expressions. I will also describe how I have implemented a pseudo-interpreter which is used by the compiler to evaluate compile-time expressions.

4.1 Evaluation of Compile-Time Expressions

To develop value-dependent types requires a means by which the compiler can know, statically, values which are passed to instantiate a class. To allow greater flexibility in these types the compiler also requires the ability to evaluate expressions which involve such statically known values. Consider the return type of `add` from the `Vector` API which can be found in section 6.4:

```
Vector[this->A!, #(_size + _size')]^
```

The type in the above requires us to be able to statically evaluate `#(_size + _size')`.

We need not restrict these compile time expressions to appear within type arguments only. We can see in fig. 6 that the Pony syntax has been extended to permit compile-time expressions wherever a ⟨*term*⟩ is expected. This allows a developer to request that some of their code is evaluated at compile-time.

Developing a means for evaluating expressions at compile-time relied on two observations:

1. Pony programs are parsed to an intermediate AST representation on which the compiler performs all of its passes.
2. Evaluation of compile-time expressions is, in essence, rewriting parts of the AST to a value that the expression would evaluate to at runtime.

From these observations we can conclude that we can build a pseudo-interpreter which operates on the AST representation of expressions and rewrites expressions to their evaluated equivalent. We may then embed this pseudo-interpreter into the compiler and use it to evaluate compile-time expressions during compilation.

Alongside the concerns of how to build an interpreter that evaluates compile-time expressions, we must also consider at what stage of compilation we attempt evaluation of such expressions. An implementation should be able to detect type errors and not attempt evaluation of expressions with such errors. Take the following expressions which are not type safe:

```
let x: U32 = #(1 and false)
let y: U32 = #(U32(16) and true)
```

Both of the above compile-time expressions have type errors, therefore the compiler should not attempt to evaluate the expressions.

Finally, we impose the restriction that all compile-time expression must result in a `val` capability value. The expression within a compile-time expression does not necessarily have to be be of `val` capability however the evaluated result must be recoverable to a `val`. This recovery means that the capability of the value must be able to seen as a `val` capability. We will discuss this restriction further in sections 4.1.3 and 5

4.1.1 Primitives

I have considered the primitive literal integer, floating-point, boolean and string values to be compile-time expressions as their value is known statically. Each of these literals has an internal representation in the compiler. integers are represented using 128-bit lexical unsigned integers (a struct of 2 64-bit unsigned integers). Floating-point values are represented using double-precision floating-point values. Boolean values are represented using two identifier tokens, namely `TK_TRUE` and `TK_FALSE`. Finally, strings are represented using C-style strings.

The values are used to represent literals for the classes such as `U32`, `I64`, `F32`, `Bool` and `String` (among others) which appear in the Pony standard library. The class for each of these values define a set of methods used to perform operations using these values. I also consider the methods defined on these classes to be compile-time expressions.

To evaluate these methods, I have implemented that the interpreter maintains a method-lookup table which maps a pair of type name and function name to a C function. These bespoke C functions inspect the AST which represents the expression and builds a new node with the appropriate result. The original expression is then replaced with the resulting node.

This means that we can write programs such as the following:

```
let x: U32 = #(7 + 42)
```

The expression is evaluated during compilation rewriting the AST as follows:

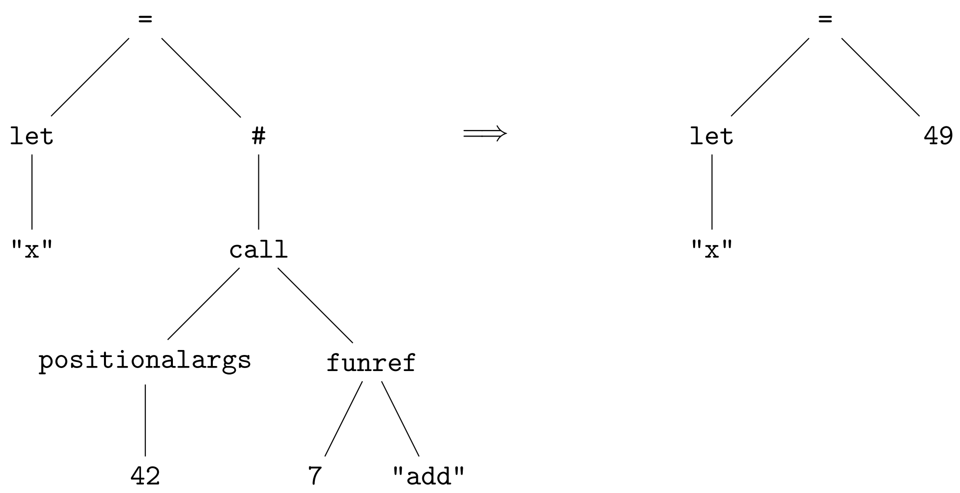

Figure 7: AST rewriting

and so we generate code for the following:

```
let x: U32 = 49
```

An issue with implementing this is demonstrated in the following example:

```
class C1
  fun apply(): U32 =>
    #(1 + 2)
```

Type-checking is performed from the leaves of the AST and progresses up towards the root. This has the effect that we do not know the type of neither `1` nor `2` as we see these nodes before seeing the context in which they are used; these literals could have any literal `Number` type (e.g. `I32`, `U64`, etc.).

A solution to this lies in the assumption that all of these integer values work on the same underlying representation, the 128-bit lexical integer (floats and integers use different tokens and so we can distinguish which representation is to be used during evaluation). This integer representation can be used to perform arithmetic and bitwise operations to obtain evaluated expressions. One issue that arose from this is demonstrated in the following example:

```
1 let x: I32 = -1
2 let y: I32 = #(-1)
```

The assignment in line 1 works as the type of `1` is inferred to be an `I32`, which is within the range of an `I32`, after which the negation operation is executed on `1`. However, the assignment at line 2 is subtly different as the negation is applied to `1`. The negation results in a node which is the largest 128-bit value, which is out of range for an `I32`. This required manipulation of the lexical integers so that the interpreter could track whether a value was believed to be negative or not. This allowed the compiler to check whether a value or its negated counterpart (if necessary) was within range of the data type.

4.1.2 Compile-Time Variables

It is also possible to track the value of some variables throughout a program; I now define which variables can be used within compile-time expressions. Take the following example:

```
1 let x: U32 = # 4
2 let z: U32 = # x
3
4 var y: U32 = # 17
5 y = 4
6 let p: U32 = # y
```

Recall from section 2.1.2 the distinction between `let` and `var`; `let` constants may not be reassigned, whereas we may reassign to a `var` variable as we see fit.

The assignments at lines 1 and 2 should succeed as both `4` and the value of `x` are known at compile time. These values are known at compile-time as a result of two facts:

1. `x` is a `let` constant and therefore cannot be reassigned
2. The right-hand side of both expressions are compile-time expressions and therefore the compiler knows their values.

Contrast line 2 with the assignment at line 6; evaluation of `# y` will fail as `y` is a `var` variable. Indeed we can see that `y` is reassigned at line 3 and so the compiler would no longer know the value of `y`. Although determining the value of `y` in the above is simple, We impose the restriction that `var` variables are not permitted in compile-time expression so that the compiler is not required to track all assignments to these variables. Consider the following:

```
var i: U32 = # 0
while i < 10 do
  i = #(i + 1)
end
```

Here, the AST could not simply be written to evaluate `#(i + 1)` at compile-time as the value of `i` changes on each iteration of the loop. Disallowing `var` variables prevents us having to support this case.

However, it is possible to track the value of `var` variables in certain contexts. Consider the following:

```
# (var i: USize = 3
   i = i * i
   i = i + 10)
```

In the above example the declaration and all assignments to `i` are encapsulated within the compile-time expressions, thus such expressions should be legal.

Note that the following expressions is disallowed,

```
# (var i: USize = 3
  i = i * i
  i = i + 10)
let z: USize = i
```

This is because the `i` variable will not exist when we assign the value to `z`. The first expression will have been re-written to its result, thus removing the variable `i` from the program. Therefore the reference to `i` is an error. This shows that compile-time expressions require their own scopes to avoid the issue described.

I now define some rules on the variables we can use within compile-time expressions;

- `let` constants which have been assigned compile-time expressions are usable within compile-time expressions.
- `var` variables declared within compile-time expressions are usable within the same compile-time expression.

These rules encapsulate the circumstances in which the compiler can track the value of constants and variables.

The compiler uses frames for type-checking. These handle scoping and provide a symbol table for each scope so as to track the definition of variables and their assigned types. I have extended this so that a symbol can also include a value; the extra value allows the symbol table to map a variable or constant to a given value. A mapping is only created or updated when the variable or constant is permitted in compile-time expressions under the above rules.

4.1.3 Objects and Constructors

We can extend the values that can exist in a compile-time representation beyond just primitive values and also develop compile-time objects. Take the following Pony program:

```
class C1
  let x: U32
  let y: Bool
  new val create(x': U32, y': Bool) =>
    x = x'
    y = y'

class C2
  let c: C1 val
  new val create(c': C1 val) => c = c'

actor Main
  new create(env: Env) =>
    let c = # C2(C1(12, true))
```

In this example we define two classes `C1` and `C2`; `C1` has `U32` and `Bool` fields and `C2` has a `C1` field. At line 11 we then construct two compile-time objects, a `C1` and `C2` object.

I have constructed an internal representation of objects which is an AST node that stores the name of the object and the value of each member. Also stored with the object is a symbol table whose purpose will described shortly. The AST representation of the compile-time object on line 14 in the example above can be seen in fig. 8.

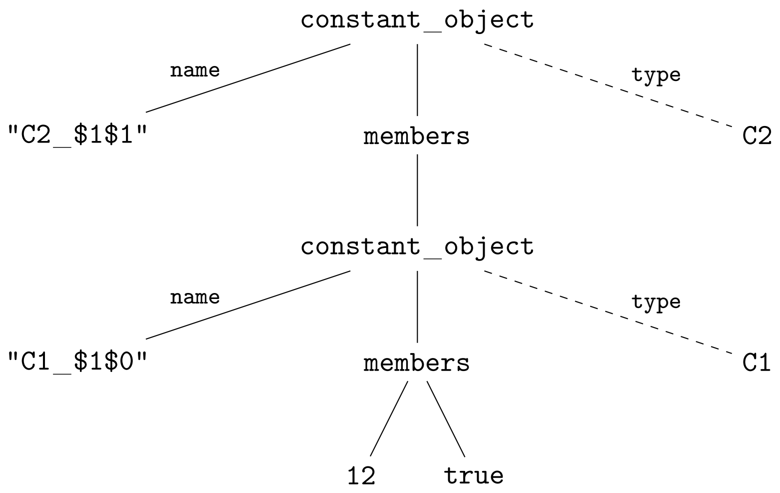

Figure 8: AST representation of `# C2(C1(12, true))evaluated`

Each object is given a hygienic name; a name that is guaranteed not to clash with any other name. This is constructed by mangling together the type and a counter kept for each package. We provide a name for the objects to allow for identity equivalence of objects. Also, we require a name so that we can refer to the constructed globals in the LLVM IR. An example of this hygienic naming is visible in fig. 8. The `C1(12, true)` object has the name `C1_$1$0`.

The fields of compile-time objects are obtained by evaluating the requested constructor for the object. Assigning a value to a field in a constructors maps the field name to the field value in the symbol-table of the object. After calling the constructor the fields are inspected and their values appended to the members node of the object. There is no duplication of data here as the nodes are referenced to by pointers, the members as children is an implementation choice to avoid orphan nodes as this can cause issues during compilation when traversing the AST. Storing the symbol tables allows us to evaluate field lookups in compile-time expressions by looking up the value mapped to by the requested field.

These compile-time objects are lowered into the LLVM IR as global constants, these are statically initialised values which are placed in the read-only section of memory and as such can never be modified [6]. Constructing these values as global constants is necessary as these objects may be used when constructing a dependent type, such values need to be known statically to construct a definition of the class thus they must be global. We are also safe in making the values appear in the read-only data section (and thus immutable) as these objects are guaranteed to have `val` capability and therefore will never change during runtime.

The generated LLVM IR for the example described above is reproduced in the following:

```
@"C1_$1$0"=internal constant %C1 { %C1_Desc* @C1_Desc, i32 12, i1 true }
@"C2_$1$1"=internal constant %C2 { %C2_Desc* @C2_Desc, %C1* @"C1_$1$0" }
```

Here we statically define two objects, the first is `@"C1_$1$0"` with the field values `12` and `true`. Similarly, the object `@"C2_$1$1"` is defined with the member `@"C1_$1$0"`.

The capabilities involved in compile-expressions must also be considered; the capabilities of both the value returned as a result of a compile time expression and the capabilities of values we can use within the expression. We explore this in the context of compile time objects.

Take the following example:

```
1 class Student
2   let age: U32
3   new val create(age': U32) => age = age'
4
5 actor Main
6   new create(env: Env) =>
7     let s1: Student val = # Student(13)
8     let age = # s1.age
```

Here we first define a class `Student` which has a single `U32` field. We also provide a constructor, `create()`, which returns a `Student` of `val` capability.

At line 7 we request a compile-time `Student`, we then read the `Student`'s age in the compile-time expression at line 8. The expression at line 8 demonstrates why compile-time values are required to be of `val` capability; for the compiler to to know the value of `s1.age` at compile-time, it must be the case that the value has not changed since its definition. In other words, we require that compile-time values are immutable, compare this with both the functional immutable objects in Idris and `constexpr` variables in C++.

For the purposes of this exploration, consider that `ref` capability values were permitted in compile-time expressions and we constructed the following program:

```
1 class Student
2   let age: U32
3   new ref create(age': U32) => age = age'
4
5 actor Main
6   new create(env: Env) =>
7     let s1: Student ref = # Student(13)
8     s1.age = 47
9     let age = # s1.age
```

In this example we alter the value of the age field and then attempt to read this value in the compile-time expression at line 9. Similarly to `var` variables, the compiler would have to track all writes to the `s1` object to obtain the correct result (i.e. 47). This motivates the conclusions that the result of a compile-time expressions should have `val` capability.

We do not make any restriction on the capabilities which may appear inside of the compile-time expressions. We are free to provide this freedom as we only permit compile-time variables to be used across different compile-time expressions, the capability of which must be `val` and are therefore immutable. All other objects must have been constructed within the current compile-time expression and therefore we can track their values. Currently Ponyta does not support mutable data structures, disallowing compile-time object with `var` fields. Mutable compile-time objects is presented as an extension in section 11.3.

4.1.4 Method Calls

We extend compile-time expressions to include evaluation of method calls. Methods on primitive values are handled by defining methods internal to the compiler which know how to appropriately rewrite the AST as discussed in section 4.1.1.

The evaluation of compile-time methods extends also to user defined methods. Take the following program:

```
actor Main
  fun foo(x: U32, y: U32, z: U32): U32 =>
    (x * y) + z

  new create(env: Env) =>
    let m = # foo(2, 8, 3)
```

First note how we have so far detailed a method for evaluating compile-time expressions which works on the AST representation of an expression. Secondly note how we have an AST representation of the method `foo`. Therefore, to extend the Pony interpreter to support method calls involves mapping parameters to values and evaluating the body of the function like any other compile-time expression.

One can also call methods on compile-time values such as in the following:

```
class C1
  let x: U32

  new create(x': U32) => x = x'

  fun apply(): U32 => x + x

actor Main
  new create(env: Env) =>
    # C1(42).apply()
```

In this example, the receiver is a newly constructed `C1` object with a `U32` field assigned the value `42`. The method evaluation continues as described earlier, however in the interpreter we first note what the receiving object is so that we can look up the value of fields that are used within method calls.

The interpreter does not currently support methods which manipulate the state of an object, this is why we impose the restriction that compile-time objects may not have `var` fields.

4.1.5 Compile-Time Errors and Static Assertions

Assertions in languages such as Java and C++ result in an aborting call from the program if the predicate to the assertion results in false. For example:

```
int main()
{
  assert(false); // this will cause the program to abort at runtime
}
```

C++ also provides a `static_assert` which must pass at compile time for a program to successfully compile. An example follows:

```
int main()
{
  static_assert(false); // this will fail compilation
}
```

There is no notion of assertions in Pony however we are able to throw errors from certain contexts and use these errors for control flow and error handling. These errors and control flow are demonstrated in the following:

```
primitive Assert
  fun apply(b: Bool) ? => if not b then error end

actor Main
  new create(env: Env) =>
    let i: USize = 13
    try
      Assert(i < 10) // this expression will throw an error
    else
      env.err.print("Error!")
    end
```

In line 1 we define a primitive `Assert` which takes a single `Bool` argument and if the argument is not true then an error is thrown. The fact that `apply()` can result in `error` requires the method to be partial, denoted by `?`. Calling the `apply()` method at line 8 requires that we surround the call in a `try` expression.

We now consider the result of attempting to evaluate `error` expressions statically. If a compile-time expression encounters an `error` expression during compilation then this result is propagated up to the enclosing expression and if there is any mechanism designed to handle this (i.e. a `try` expression) then this is used. If no mechanism has been provided to control flow through the expression then `error` will be the result of the compile-time expression. If a compile-time expression results in `error` then compilation is aborted and an error reported detailing the origin of the error.

This approach allows to makes static assertions; take the following example:

```
actor Main
  new create(env: Env) =>
    let i: USize  = # 13
    # Assert(i < 10) // this expression will cause compilation to fail
```

Here we define `i` as a compile-time constant and statically assert that `i < 10`. This expression will cause compilation to be aborted as the expression will result in an error. Note also that we are able to omit the try expression surrounding the call to `apply()`, this is as the call to `apply()` has been evaluated at compile time and thus cannot throw an error dynamically.

4.1.6 Other Compile-Time Expressions

I have detailed in this report the considerations of most compile-time expressions, focusing on those which are of most interest. It should be noted that the interpreter supports more expressions such as sequencing and control flow structures (e.g `if` and `while` expressions) that can be used within compile-time expressions. These have been implemented to permit using the same methods both statically and dynamically more flexibly by a developer. We do not explore these expressions further for conciseness.

4.1.7 Caching Evaluations

To ensure that the same expression results in the same object on every evaluation and also as an optimisation to obtain better compilation times, the interpreter caches the results of evaluating expressions. This caching maps an expression to its evaluated result. When the interpreter attempts to evaluate a compile-time expression the cache is first queried to test if the expression has been mapped to a result, if so that cached result is returned.

This can be used to give memoisation in recursive functions. Consider the following definition of the Fibonacci sequence:

```
fun fib(n: USize): USize =>
  if n < 2 then
    n
  else
    fib(n - 1) + fib(n - 2)
  end
```

If we call this function as `# fib(3)`, we will invoke the call `fib(2)` which in turn will invoke the call `fib(0)` and `fib(1)`, caching each result. To evaluate `fib(3)` we will also invoke `fib(1)`, however we have previously evaluated this call so we can simply return the cached result.

This makes the evaluation of calls such as `fib(50)` cheaper when compared with the dynamic call as we have memoisation of calls.

Caching the result of expressions means that when we evaluate two expressions which are the same (based on AST equivalence) we will get the same result. It is in this way that we ensure that compile-time expressions which are duplicated (such as when reifying a trait to include the definition of methods in a class) have neither the issue of repeated execution nor evaluation to different objects.

4.2 Rules Summary

We can summarise the rules for compile-time expressions as follows:

- Primitive literal values such as integers, floating-point values, boolean and strings are compile-time values.
- Basic (e.g. `add`, `and`) methods defined for these primitives are compile-time expressions.
- `let` constants which have been assigned compile-time expressions are usable within compile-time expressions.
- `var` variables declared within compile-time expressions are usable within compile-time expressions.
- Only classes defined without using `var` fields may be instantiated as compile-time objects.
- Compile-time expressions must be recoverable to `val` capability values.
- Field lookups on compile-time objects are compile-time expressions.
- Methods built using these rules, and using compile-time values for arguments, can be used within compile-time expressions.
- If the result of a compile-time expression is an `error` then compilation fails.
- Actors and behaviours cannot be used in compile-time expressions.

5 Value-Dependent Types

With the notion of compile-time expressions in-place we can begin discussing extending Pony's type system to incorporate value-dependent types. We discussed the additional syntax both abstractly and using the BNF adopted by the compiler in section 3. Consider an example of such syntax in the following:

```
1 class CStatic[n: USize]
2   fun apply(): USize => n
3
4 actor Main
5   new create(env: Env) =>
6     let c : CStatic[2] = CStatic[2]
```

Figure 9: Value-Dependent class `CStatic`

Line 1 describes the class `CStatic` which depends on a value of type `USize`. We refer to the value on which `CStatic` has been parametrised as `n` in the definition of the class, this can be seen on line 2. The definition of `CStatic` looks very similar to if we had defined `n` to be a field of `CStatic` which was initialised in the constructor. For example:

```
class CDynamic
  let x: USize

  new create(x': USize) => x = x'

  fun apply(): USize => x

actor Main
  new create(env: Env) =>
    let c : CDynamic = CDynamic(2)
```

Figure 10: Class `CDynamic` with a field

There is an important distinction between fig. 9 and fig. 10, namely the difference between static and dynamic information. In fig. 9 the definition of `n` is provided statically on line 6. In fact the definition of `CStatic` in fig. 9 acts as a template for a definition of a class which will be constructed when the template is instantiated with a value replacing `n`. We reify the class `CStatic` using the type arguments, replacing the type parameters, therefore in fig. 9 we reify `CStatic[2]` to create a definition with `2` replacing `n`, giving us a definition of `CStatic[2]` which is presented in the following:

```
class CStatic[2]
  fun apply(): USize => 2
```

Here we can see that we do not require any field for `n` as in fig. 10; we do not require the field as `n` has been replaced in the body of `apply()`. This replacement of a reference by a value to construct a new definition of a type is the meaning we give to the term value-dependent type in Pony. The type that we construct, its fields and methods depend on the value which we provide.

I will now described how I have implemented this in the compiler. We first have an AST which represents the generic class `C1Static[n: USize]` which is detailed in fig. 11

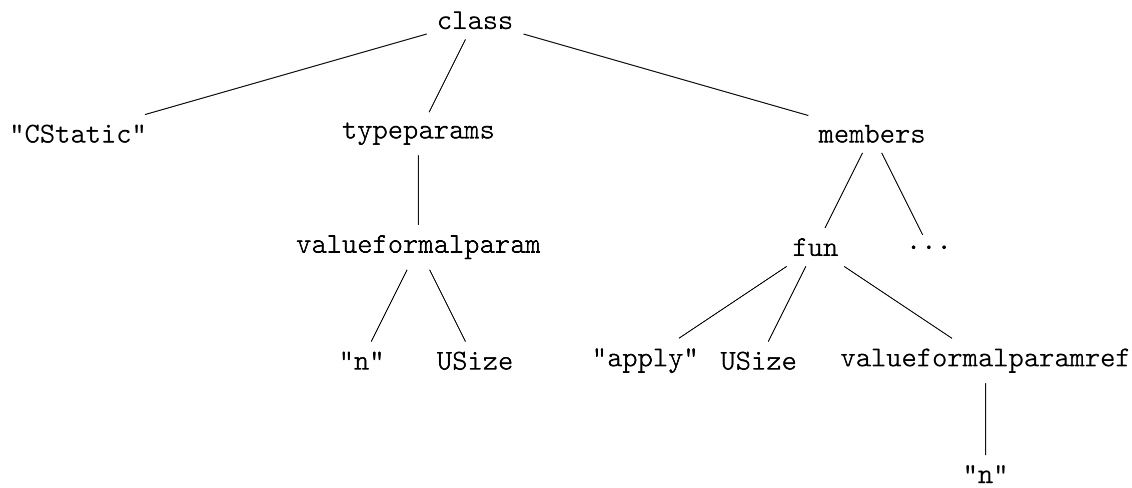

Figure 11: AST representation of `CStatic[n: USize]`

In fig. 11 we can see that we have a `valueformalparam` node, this describes a value parameter that is required to instantiate a templated class of function. The `valueformalparam` node tracks both the name of the parameter and its constraint (here `"n"` and `USize` respectively). We can also observe that class AST has a `fun` subtree which has child nodes detailing the name, return type and body of the method. The body subtree has a single node `valueformalparamref` which is used to mark a reference to the value parameter (here `"n"`). These `valueformalparamref` nodes are initially constructed in the "names" pass of compilation by looking up the definition of a reference and transforming the node accordingly.

We can construct a instance of the `CStatic[n: USize]` class such as follows:

```
CStatic[2].create().apply()
```

Calling methods and constructors with instantiated types leads to reifiying the methods using the type arguments supplied. For example, the `apply()` method will be reified to the AST shown in fig. 12. This reification replaces all references to `n` with the provided type argument

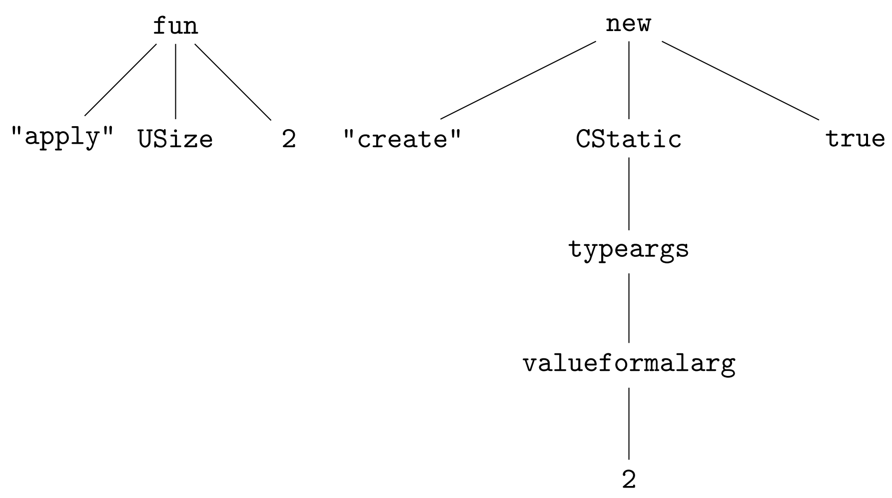

Figure 12: AST representation of reified `apply()` and `create()`

These reified methods are used for subtyping to check validity of arguments and return types. This can be seen in the AST representation of `create()`, the middle subtree represent the return type of the constructor. Here we can see that the return type of the constructor has been reified to `CStatic[2]`.

The code generated for the value-dependent types reflects the value on which they depend on. Observe the the following two instantiations of `CStatic` and their respective calls to `apply()`:

```
actor Main
  new create(env: Env) =>
    CStatic[8].apply()
    CStatic[63].apply()
```

Here we have instantiated `CStatic[n: USize]` with two different type argument, namely `8` and `63`. We then call `apply()` on each of these constructed objects, It is important to note that this will generate two different definitions of the function `apply`, one where `n` has been reified to `8` and another where `n` has been reified to `63`.

As we have already discussed, these two instantiations reify to two distinct definitions of `CStatic[8]` and `CStatic[63]`. We generate code for these two classes, constructing unique names and definitions by mangling the value argument into the type name. These two distinct types can be seen by observing the following generate LLVM IR:

```
%CStatic_8 = type { %CStatic_8_Desc* }
%CStatic_63 = type { %CStatic_63_Desc* }
```

Note then that as we have replaced all references to the value parameters in the AST representation of methods, we may use all of the existing code generation. Therefore we interact with value-dependent types in the same way as any other data at the LLVM IR level. The two definitions of `apply()` are shown in the following LLVM IR:

```
define fastcc i64 @CStatic_8_val_apply_Z(%CStatic_8* nocapture readnone dereferenceable(8) %this) {
entry:
  ret i64 8
}

define fastcc i64 @CStatic_63_val_apply_Z(%CStatic_63* nocapture readnone dereferenceable(8) %this) {
entry:
  ret i64 63
}
```

I now introduce an example of a type which depends on an object value:

```
1  class C1
2    let x: U32
3    new create(x': U32) => x = x'
4
5  class C2[c: C1 val]
6    fun apply(): C1 val => c
7
8  actor Main
9    new create(env: Env) =>
10     let c2 = C2[# C1(48)]
```

At line 9 in the above we instantiate `C2` with an object of type `C1`. Notice that the provided argument is a compile-time object. We will now briefly discuss the capabilities of value parameters. For similar reasons to those discussed in section 4.1.3, we require that all value parameters are of `val` capability. This is similar to how in C++ template arguments must be statically know and immutable (although in C++ we may instantiate an immutable pointer to a mutable value). Notice that we also get this guarantee in Idris for free as the language is functional and thus all data is immutable. We enforce this restriction as we require that data on which a type depends is immutable, a guarantee provided by the `val` capability. We require that the data is immutable so that we can statically use the type argument in the compile-time expressions defined in the class, we see an example of this in section 7. We could allow mutable references to type arguments but we would not statically known any information about this value. Furthermore, we would have to ensure that we used the correct viewpoint adaptation rules of value parameters to ensure we do not invalidate Pony's guarantee of no data races. Otherwise, we could introduce types which provide access to mutable values in multiple actors. As mutable values do not provide much use as value parameters, no more than dynamically passing the value, and to ensure that values remain constant we require that value parameters have `val` capability.

Recall from section 4 that the result of a compile-time expression must be `val`, therefore as all type arguments must be compile-time expressions we can only instantiate a type with `val` type arguments.

We return again to our example. Any instantiation of C2 depends on an object of type C1; in fact,it depends on a particular instance of a C1 object. This is reflected once again in the reified AST and generated code:

```
@"C1_$1$0" = internal constant %C1 { %C1_Desc* @C1_Desc, i32 48 }
%"C2_C1_$1$0" = type { %"C2_C1_$1$0_Desc"* }
```

The first line, in the above, is the static object `@"C1_$1$0"`, The second line is the type of `C2[# C1(48)]`, where the type name has be mangled together with the name of the object constructed by `# C1(48)`, namely `@"C1_$1$0"`. Note here that, as mentioned earlier, C2 does not have the object has part of the structure, the type has only a description header. The object has already been used to reify the methods as necessary. This reification can be seen in the following:

```
1 define fastcc %C1* @"C2_C1_$1$0_ref_apply_o"(%"C2_C1_$1$0"*
      dereferenceable(8) %this) {
2 entry:
3   %this1 = alloca %"C2_C1_$1$0"*
4   store %"C2_C1_$1$0"* %this, %"C2_C1_$1$0"** %this1
5   ret %C1* @"C1_$1$0"
6 }
```

Line 5 is the line which is of interest here. We directly refer to the global `@"C1_$1$0`, clearly distinguishing the difference between value-dependent types and values passed as arguments.

We now go on to consider how the type system incorporates these new types.

5.1 Equivalent Value-Dependent Types

A very important consideration for value-dependent types is what it means for two value-dependent types to be equivalent. We consider two value-dependent types to be equivalent using the same notion of equivalent types used for generics in Pony. This is that the template type and all type arguments are the same; recall from section 2.1.5 that Pony is type invariant. We introduce the extra constraint on value-dependent types which is that value arguments must be equal. Consider the following program:

```
class C1[n: USize]

actor Main
  new create(env: Env) =>
    let c: C1[4] = C1[4] // successfully compiles
    let c: C1[4] = C1[72] // fails to compile as 4 != 72
```

There are subtleties in a notion of equality between value arguments, one such subtlety is that two expressions which are not the same may in fact result in the same value. For example:

```
actor Main
  new create(env: Env) =>
    let c: C1[4] = C1[# (1 + 3)] // successfully compiles
```

In the above assignment the two instantiations of `C1` use different expressions to denote the same value. Therefore, before we check equivalence between values we first attempt evaluation of expressions to reduce expressions to their result.

However, evaluating expressions to a value is not always possible. We may be type-checking a template definition. In the template definition we may be working with unknown values that will be provided later. I now go on to describe the notion of equality between value arguments and type-checking value-dependent classes and functions.

5.2 Equality of Compile-Time Expressions

We defined subtyping of value-dependent types to be based on equality of values. We must be specific about the meaning of equality and when such equality applies. Consider the following example:

```
1 class C1[n: U32]
2
3 class C2[n: U32, m: U32]
4   let c: C1[n] = C1[m]
5
6 actor Main
7   new create(env: Env) =>
8     let c1 = C2[1, 1]
9     let c2 = C2[72, 13]
```

We must consider how to handle such a program. Consider first the definition of `c1` at line 8. We instantiate the class `C2` with the arguments `[1, 1]`, thus the reified class will look like:

```
class C2[1, 1]
  let c: C1[1] = C1[1]
```

This looks to be a valid and well-formed class definition that should be accepted at type checking. Consider now the definition of `c2` at line 9. Here we instantiate `C2` with different arguments, namely `[72, 13]`. Again, let us observe the reified class:

```
class C2[72, 13]
  let c: C1[72] = C1[13]
```

Clearly, such a definition should be rejected by the type checker as `C1[13]` is not a subtype of `C1[72]`.

This leads us to consider how permissive we should be with type checking and here we propose some possible solutions for handling the above issue.

5.2.1 Type Checking on Reification

One of the most permissive type checking approaches is obtained by only type checking the class/method upon reification. The templated method is not fully type checked and permits the assignment:

```
let c: C1[n] = C1[m]
```

We can instantiate the class, which constructs a reified definition, as in the following:

```
class C2[1, 1]
  let c: C1[1] = C1[1]
```

The reified definition is type checked again and found to be type safe. If only this reification were used then the program would compile. In other words, the following program would be permitted.

```
actor Main
  new create(env: Env) =>
    let c1 = C2[1, 1]
```

Conversely, the following program would be disallowed:

```
1 actor Main
2   new create(env: Env) =>
3     let c2 = C2[72, 13]
```

Due to the reification at line 3 creating a definition which would fail type checking.

This style of typechecking is inspired by the C++ lazy type checking of templates. Take the following C++ class:

```
template <size_t n>
class C1 {}

template <size_t n, size_t m>
class C2
{
  C1<n> c = C1<m>();
}
```

Without any call to the constructor of `C2` the above definition would not only be allowed but would in fact never be type checked, we would be permitted to define `C2` as follows:

```
template <size_t n, size_t m>
class C2
{
  C1<n> c = C1<m>();
  bool x = 5;
}
```

A C++ compiler would only report an error when we attempted to construct an object in the following way (for example):

```
C2<72, 13> c;
```

This method has the benefits of being the most permissive for programmers as well as only requiring equality between known values instead of unknowns type parameters. However, a class or method body will require type checking on every instantiation. Such an approach could have a noticeable impact on compilation time. Also, consider the practical applications for permitting such definitions, the above Pony example will only pass for a very limited number of instantiations (compare with the number of possible instantiations) and a developer does not gain much from such a definition as they must statically know both parameters. These arguments lead to a more restrictive approach as is described in the following section.

5.2.2 Syntactic Equality of Expressions

Syntactic equality of expressions requires only type checking of the template definition and does not require subsequent checks upon each reification. We allow subtyping between values if the two values are syntactically the same. Thus, we would never allow the definition from our earlier example due to the following assignment:

```
let c: C1[n] = C1[m]
```

This is disallowed as `m` and `n` are syntactically different arguments. As we disallow such definitions at the template level, we know that the body of classes and methods are type safe when we reify them, thus we need only replace the parameters with their arguments.

This approach is similar to a class defined as follows:

```
class C1[A, B]
  let array: Array[A] = Array[B]
```

In Pony, the above definition is disallowed as no guarantee can be made about the types of `A` and `B`.

This notion of equality leads to a similar type-checking system with generics. This approach also requires the fewest passes for type-checking whilst not causing too much restriction to a developers flexibility in defining value-dependent types. For these reasons we adopt this notion of equality between type arguments.

5.2.3 Semantic Equality of Expressions

Semantic equality of expressions is an extension of the previous solution; we could go on to test whether two expressions are considered to be equivalent, for example `n + m` and `m + n`. Note that this approach is only required when we type check only the template definition and not the reified class, otherwise we could simply evaluate two reified expressions and test that we get the same value.

5.2.4 Equality of Compile-Time Objects

Under the syntactic equality of expressions we also need to consider what it means for two compile-time objects to be equivalent. Consider the following examples:

```
1 class C1
2
3 class C2[c: C1]
4
5 class C3
6   let c2: C2[# C1] = C2[# C1]
```

This example prompts us to consider whether the assignment at line 6 is valid. This becomes a question of how we consider two object values to be equal. Two possible solutions to equivalence follow.

Object Structural Equivalence

This would consider two values to be equal if all of their fields were considered to be equal, this would mean that the example above would be considered valid as the two `C1` objects would be equivalent.

Object Identity Equivalence

This is more a notion of whether two objects are the same object. This would suggest that the example above would not be valid and we would instead have to write something like:

```
class C3
  let c1 = C1.create()
  let c2: C2[# c1] = C2[# c1]
```

This is a natural progression from section 5.2.2 where we check whether two expressions are syntactically the same. Recall from section 4.1.3 that compile time objects are represented with a special AST node which contains the name of a compile-time object, thus we simply compare the names of two objects.

I have adopted identity equivalence in my implementation of Ponyta. I chose this approach as the result of compile-time expressions are cached. Therefore programs with value-dependent types, such as the one presented when introducing object equivalence, will result in the same object being used in each expression. Furthermore, this avoids requiring to recurse into the AST to determine equality. Finally, this feels like a closer relation to the generated code. If we consider the same structure to be equivalent then a type which depends on a value could be used any different (yet equivalent) object. This will still give the same result as the values are equivalent but would provide a slightly different meaning to value-dependent types.

5.3 Subtyping Value-Dependent Types

I now present some of the issues and applications that the existing Pony type system presents for Ponyta and how these have been resolved.

5.3.1 Nominal Subtyping

As discussed in section 2.1.7, Pony supports nominal subtyping through `traits`. This subtyping does not change with the introduction of value-dependent types. Therefore we can write programs such as the following:

```
trait T1[n: USize]
class C1 is T1[0]

actor Main
  fun foo(t: T1[0]) => true

  new create(env: Env) => foo(C1)
```

5.3.2 Structural Subtyping

In section 2.1.7 we discussed possible issues, and some solutions, with structural subtyping of value-dependent types. In keeping with the extension so far, we aim to provide some semantics which are clear and useful to a developer but also follow the current implementation. If we first consider the following Pony program:

```
1 interface I1[A]
2   fun apply() =>
3     let a = Array[A]
4
5 class C1 is I1[String]
6
7 actor Main
8   fun foo(i: I1[Bool]) =>
9     i.apply()
10
11   new create(env: Env) =>
12     foo(C1)
```

In line 12 of this example we pass a value of type `C1` when an `I1[Bool]` is expected. This raises the question of whether `I1[String]` is a subtype of `I1[Bool]`. The type parameter `A` does not appear in either the parameter types nor the return type of any method in `I1`. Therefore, `I1[Bool]` and `I1[String]` are structurally equivalent. This equivalence means that the above program is correct.

We will use this example for generics to help us define structural subtyping for value-dependent types. Take the following Example:

```
interface I1[A, n : U32]
  fun apply(i: U32)

class C1 is I1[Student, 2]

actor Main
  fun bar(i: I1[Student, 47])

  new create(env: Env) =>
    let c: C1 = C1.create()
    bar(c)
```

In this example, we are presented with a similar question to that of structural subtyping with generics. Namely, is `I1[Student, 2]` a subtype of `I1[Student, 47]`. As in the previous example, the value parameter does not appear in a parameter or return type of any method in `I1`. We conclude that `I1[Student, 2]` is a subtype of `I1[Student, 47]`.

Note that these values can indeed affect the structure of an interface. We can affect the structure by including the values within the types of parameters and return types, such as in the following:

```
interface I1[n: USize]
  fun apply(): Vector[U32, n] =>
    Vector[U32, n].undefined()

class C1 is I1[12]
class C2 is I1[2]

actor Main
  fun foo(i: I1[2]) =>
    i.apply()

  new create(env: Env) =>
    foo(C2) // legal
    foo(C1) // will error at compile-time
```

Now, as the values alter the types of the entities within `I1`, we have that `I1[12]` is not a subtype of `I1[2]` nor is `I1[2]` a subtype of `I1[12]`.

5.3.3 Intersection and Union Types

We discussed possible issues with respect to ambiguity introduced when using the intersection or union of value-dependent types in section 2.1.8. The example which posed an issue is the following:

```
trait T1[n: U32]
  fun apply(): U32 => n

class C1 is (T1[2] & T1[73])
```

We handle this in a similar way to how we check subtyping between value-dependent types. The method `apply()` from `T1[2]` is added to `C1`, this is then followed by an attempt to add `apply()` from `T1[73]` to `C1`. Before adding the second definition of `apply()` we check to see whether a definition for `apply()` already exists. We will find that a definition already exists and so we test whether this introduces an ambiguous definition.

To check for an ambiguous definition first involves determining if the definition of `apply()` comes from the same trait; if not, then we will certainly introduce an ambiguous definition. If the definition comes from the same trait, as it does in the example above, we proceed to test for type argument equality based on AST equivalence. If the AST equivalence succeeds, then we will not introduce any ambiguity as we have only used the same method twice, otherwise the compiler will error stating that an ambiguous definition of `apply()` in `C1` has been found. This implementation has also been applied to prevent ambiguous method bodies when inheriting from two instantiations of the same generic trait.

5.3.4 F-Bounded Polymorphism and Infinite types

We discussed F-Bounded polymorphism in section 2.1.6 and suggested that there may be issues giving meaning to programs such as the following:

```
trait T1[x : T1[# x]]
  fun apply(): T1[# x] => x

class C1 is T1[# C1.create()]
  new val create() => true
```

The issue being that the type of `x` depends on `x` itself. A possible solution is to consider this to be an infinite type that cannot be constructed, the compiler can then catch this and inform the developer that this type definition causes an error.

To explain an alternative to this requires noting the difference between a value and a reference to that value. If we now rewrite the above definition using `reference` to denote references to a value we get the following:

```
trait T1[x: T1[# reference(x)]]
```

Now we assign types to the two values we have in this definition, namely `x` and `# reference (x)`. we assign both of these values the type `T1[# reference(x)]`, noting here that we leave the value `# reference(x)` in the type to not have any type (attempting to assign a type to this value would result in defining an infinite type).

We go on to instantiate this type, namely:

```
class C1 is T1[# C1.create()]
```

Assume that `# C1.create()` evaluates to `C1_$1$1`. The subtyping of `T1[# C1_$1$1]` would include the following methods in `C1`:

```
fun apply(): T1[# C1_$1$1] => C1_$1$1
```

This definition would provide all `C1` objects with access to some global `C1` object through the interface defined by `T1`. The semantics of this and the benefits that it provides to a programmer are unclear. More research is required to see if this is a useful to include in Ponyta. For now, we conclude that such definitions of types are not supported in Ponyta. Currently my implementation does not warn or forbid against such definitions however compilation will not succeed if a class like the above is defined.

5.4 Trace Function

The Pony runtime includes a garbage collector used to deallocate objects that are no longer reachable and thus unusable. The garbage collector is based on each actor maintaining a reference count of each reachable object.[23] These reference counts are maintained through trace functions. One concern is whether we should trace the values provided as static parameters for types. Take the following example:

```
class C1[s: Student]
  fun apply: U32() => s.age

actor Main
  new create(env: Env) =>
    let s = # Student(42)
    let c1 = C1[# s]
```

Recall from section 4.1.3 that this will construct a LLVM global constant object. These constant objects should never be garbage collected as we may construct types which depend on these values. We may have the case that no object has the type of a value-dependent type, however this does not mean that we cannot have a such a value at any point in the future. We may later construct an object typed using a value-dependent type. Therefore we should ensure that we do not remove these types otherwise we could have definitions of methods which will behave incorrectly at runtime, attempting to use deallocated objects.

This does mean that any compile-time object will remain in memory for the lifetime of a program even if it is unused. Possible analysis could be made to see which objects are used to define types, removing those which are not used to define types after they have been used. However, the LLVM optimiser has many powerful optimisations which can be applied to global constants that means they will in many cases be removed before constructing the executable if possible.

6 Vector Class

We now consider a new class for the Pony standard library which adopts such value-dependent types. The `Vector` class represents a data structure of elements whose size is both fixed and known statically. The `Vector` class was designed to be similar to Idris style vectors where we know how many elements are in the vector and the vector does not grow or shrink. The references to the elements of the `Vector` are embedded within the `Vector` object, differing from `Array` objects which we will now discuss.

6.1 Layout

The `Vector` class is similar to the existing `Array` built-in for Pony. One difference is that the allocated memory for an `Array` is determined at run-time and is dynamically managed, increasing and reducing the used memory as functions are called. A result of this dynamic memory management is that a `Array` manages both how many members it stores and also the number of elements for which memory has been allocated.

A `Vector` is a fixed, statically known, sized memory store whose memory is allocated once. The `Vector` represents something similar to the vectors found in Idris. Defining the `Vector` class in this way means that the vector does not require means to track the allocated memory nor the number of elements in the vector.

Another difference between an `Array` and `Vector` is the layout of the class in memory. An `Array` maintains three members; the number of elements, the amount of allocated memory and a `Pointer` (which acts as an interface to memory) which points to where the elements exist in memory. A `Vector`s size is known before allocation and, once allocated, will remain the same size. These properties allows us to optimise the structure of the object and store the elements embedded within the encapsulating object. This fundamental difference means that a `Vector[String, 2]` is of a different size to that of a `Vector[String, 7]`.

We further discuss the difference in object layout between the following `Array[Student]` and `Vector[Student, 4]` by utilising the following Pony snippet:

```
let vector: Vector[String, 4] = {"A", "B", "C", "D"}
let array: Array[String] = ["A", "B", "C", "D"]
```

The two objects can be considered as having the following memory layouts displayed in fig. 13. Note in particular that the `Array` class must store the contents externally.

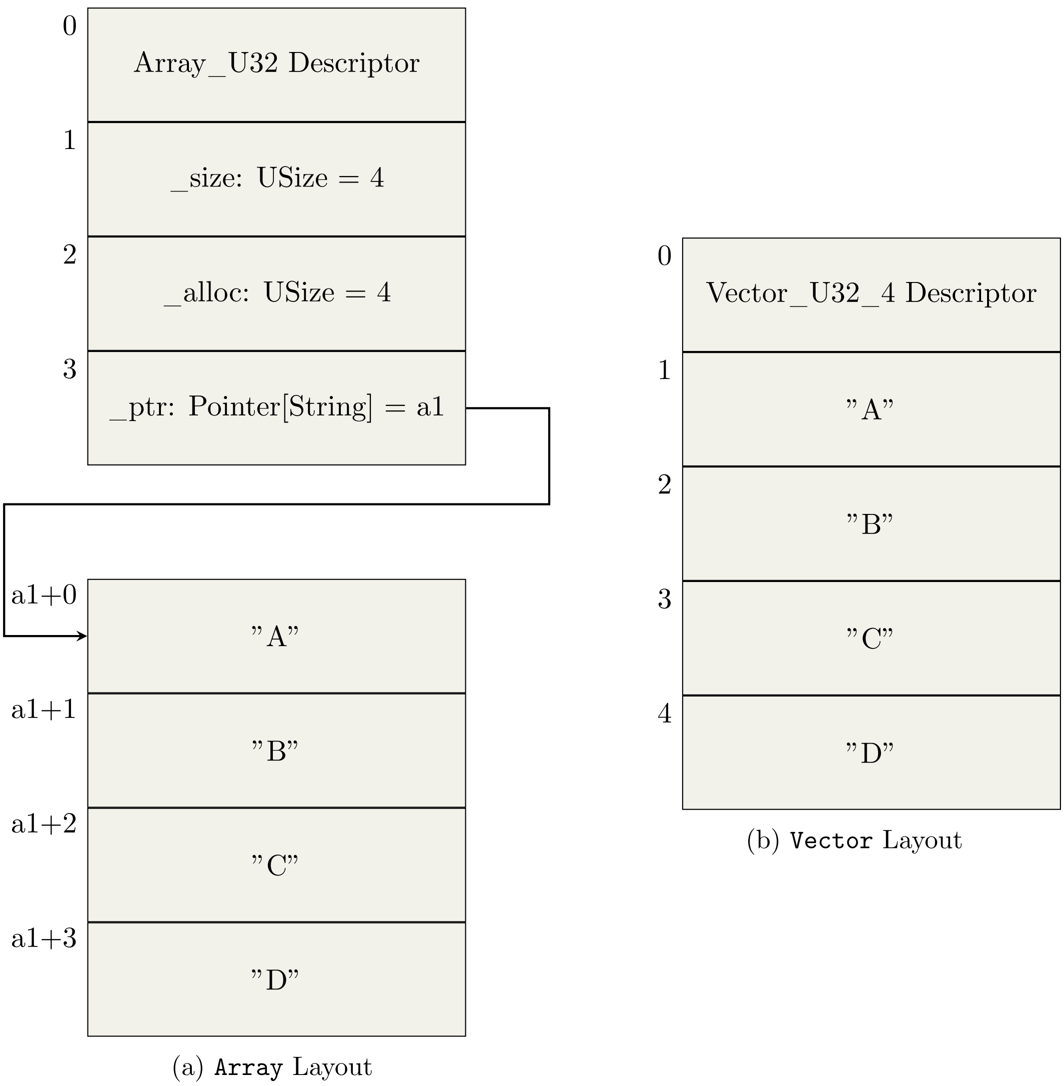

Figure 13: Comparing `Array` and `Vector` layouts

6.2 Implementation

Consider the layout of an `Array[A]` object as described in fig. 13a. The elements of an `Array` are stored through the indirection of the `Pointer` member. The `Array[A]` has 3 fields, `_size`, `_alloc` and `_ptr`. When we allocate an `Array[A]` it will always have 3 members.

However, the class `Vector[A, _size: USize]` defines a class where the number of members is unknown until instantiation of the type. Upon instantiation, say `Vector[String, 7]`, the `Vector` object will have `7 String` members. We can, therefore, observe that when we allocate a `Vector[String, 7]`, we must allocate at least an object large enough to accommodate 7 elements of type `String`. This requires compiler support to know how to allocate such an object as well as support for interacting with objects of type `Vector[String, 7]`.

There were multiple possible solutions for building the `Vector[A, _size: USize]`, these are as follows:

6.2.1 Embedding a `Pointer[A]`

This approach was inspired by the implementation of the `Array` class; the `Array` class uses a `Pointer` as a means to interface with memory. Reproduced here is part of the `Array` class to demonstrate such an implementation.

```
1  class Array[A] is Seq[A]
2    """
3    Contiguous, resizable memory to store elements of type A.
4    """
5    var _size: USize
6    var _alloc: USize
7    var _ptr: Pointer[A]
8
9    new create(len: USize = 0) =>
10     """
11     Create an array with zero elements, but space for len elements.
12     """
13     _size = 0
14     _alloc = len
15     _ptr = Pointer[A]._alloc(len)
16
17   fun apply(i: USize): this->A ? =>
18     """
19     Get the i-th element, raising an error if the index is out of bounds.
20     """
21     if i < _size then
22       _ptr._apply(i)
23     else
24       error
25     end
```

As can be seen at line 15, the `_ptr` is initialised through a `Pointer[A].alloc(len)`, representing a pointer to an area of allocated memory. This allocated memory is external to the `Array[A]` object as explained in fig. 13a. However, we only allocate this memory externally in `Array[A]` as we do not know how large the `Array` will be at compile time and therefore it must be able to grow and shrink in size.

In the case of `Vector`s we know the number of elements contained in the `Vector`, therefore we can allocate the exact amount of memory required and store the elements internally, avoiding a level of indirection to the elements. In the `Vector` case we also do not require tracking the number of elements in the `Vector` nor the allocated amount of memory, thus giving us a more compact representation. In summary, this representation saves us 3 fields (machine words) and a level of indirection.

We now consider the `embed` keyword in Pony. This keyword changes the representation of an object in memory. Consider the following example where a field is embedded in a class:

```
1 class C1
2   let x: U32 = 2
3
4 class C2
5   embed c: C1 = C1
```

The embedded field at line 5 represents that an object of type `C1`, instead of a reference to an object of type `C1`, is stored within an object of type `C2`. This is the effect that we want in our layout of a `Vector`. Consider a definition of a `Vector` as follows:

```
1  class Vector[A, _size]
2    """
3    Contiguous, fixed memory to store elements of type A.
4    """
5    embed _ptr: Pointer[A]
6
7    new init(seq: Seq[A]) =>
8      """
9      Create an vector with zero elements, but space for len elements.
10     """
11     true
12
13   fun apply(i: USize): this->A ? =>
14     """
15     Get the i-th element, raising an error if the index is out of bounds.
16     """
17     if i < _size then
18       _ptr._apply(i)
19     else
20       error
21     end
```

Note at line 5 we have embedded the `_ptr`, the effect of this is that the pointer should point to a element within the `Vector` object. All element accesses are then handled through the `_ptr`.

While this approach enables re-use of the `Pointer[A]` class and the logic for memory manipulation, we are faced with some issues:

Allocating the `Pointer[A]`

We can see that `_ptr` is not initialised in the definition of `Vector`. The pointer should not be allocated when the constructor is called as we will have already allocated the memory when we request a new `Vector`. This means that the pointer should not be initialised in any constructor, instead the pointer should be set to point to the beginning of the `Vector`. An impact of this would be that the `_ptr` field would be left uninitialised in a constructor (as the compiler would handle the initialisation), requiring extra support by the type system to ignore a violation of Pony's requirement of no undefined or null values.

Semantics of embedding a `Pointer[A]`

To embed one object within another requires that the object has some structure at runtime, for most classes such a structure will contain a descriptor header and any fields of the object. However, the `Pointer[A]` class is a special class which has no structure at runtime, the pointer is represented using an `LLVMPointerType`. The result of this means that embedding the pointer has no observable runtime effect on the layout of the object but instead tells the compiler to arrange that the pointer points to the beginning of the object.

Use of resizable memory

Enabling the embedding of the memory in the vector object would mean one would have to be careful to not call resizing operations. Although such calls are restricted to the `builtin` package (a package which uses a number of unsafe operations that are presented to developers using a wrapper method which sanity checks arguments provided), the restriction are made by making the `_ptr` field private (as is the case here). We can still consider the repercussions of using resizing operations on a `Vector` object. A high level concern with allowing resizing a `Vector` object is that we only have static information regarding the size of the `Vector`, we do not have any field to dynamically track how large a `Vector` is as we never expect it to change. This would mean that methods which use this length would use inaccurate information, this could mean copying of too few or many elements for example (such as in the `add` method) leading to using unallocated memory and thus bugs.

Conclusions of approach

The biggest issues present in this approach are the large amounts of manipulation of the compiler to support the approach, alongside the complicated semantics of these changes. As such this solution was deemed to not be particularly useful and did not seem like the best implementation for the `Vector`.

6.2.2 Elements as Fields

To avoid the use of a `Pointer` field the following alternative was considered. When constructing a `Vector[A, _size]` object, space would be reserved for `_size` elements of type `A`. The structure of this object would consist of a pointer to the descriptor table and `_size` fields which are accessed with special `_apply()` and `_update()` compiler intrinsics. Each element of the `Vector` would be stored as an embedded field in the object. The method which is described in the next section is similar to this approach. This approach presented in the following section was preferred due to being able to adopt an `LLVMArrayType` data structure. This approach would have required embedding `_size` number of copies of the element type structure in the `Vector`.

6.2.3 Elements as an Array

This approach is similar to the previous implementation possibility. Here, instead of implicitly adding `_size` fields to an object of type `Vector[A, _size: USize]` we observe the following; we statically know `_size`, we know that all elements will appear contiguously in the object layout and finally LLVM provides a structure to represent such structures, namely `LLVMArrayType`.

Therefore, to construct a `Vector[A, _size: USize]` we represent the structure of the elements as a `LLVMArrayType` with `_size` elements of type `A`. Consider the following:

```
let v = Vector[U32, 4].undefined()
```

Let us consider the structure of such an object in terms of the LLVM IR:

```
%"Vector_U32_val_$value_4"
    = type { %"Vector_U32_val_$value_4_$desc"*, [4 x i32] }
```

Here we can see that the structure is a struct containing two elements, the descriptor table and an array of 4 `i32`s. The latter is an example og the `LLVMArrayType`. Such an implementation is hidden from the user as the definition of the `Vector` class would be:

```
class Vector[A, _size]
  """
  Contiguous, fixed memory to store elements of type A.
  """

  fun _apply(i: USize): this->A ? =>
    """
    Get the i-th element.
    """
    compile_intrinsic

  fun _update(i: USize, value: A): A^ =>
    """
    Change the i-th element.
    """
    compile_intrinsic

  fun apply(i: USize)
    """
    Get the i-th element, raising an error if the index is out of bounds.
    """
    if i < _size then
      _apply(i)
    else
      error
    end
```

Note that at line 10 and 16 we use `compile_intrinsics` to handle interfacing with the elements of the `Vector`. These are expressions which denote that the body of this method will be supplied by the compiler. These expressions are reserved for when it is not possible or very difficult to write the body in Pony. These compile intrinsics will be replaced with the appropriate LLVM IR, given the type of the elements in the `Vector`, for interacting with the `LLVMArrayType` element.

Take the following instantiation, update and access of a `Vector`:

```
let v: Vector[String, 2] = Vector[String, 2].init(["A", "B"])
v.update(1, Student("C"))
let s: String= v.apply(1)
```

We can look at the LLVM IR which is generated in place of the `compile_intrinsic` for the `_update()` and `_apply()` methods. First we look at the code generated for the `_apply()` method:

```
1 define fastcc %String* @"Vector_String_val_$value_2__apply"
2   (%"Vector_String_val_$value_2"* dereferenceable(24), i64) {
3 entry:
4   %2 = getelementptr inbounds
5         %"Vector_String_val_$value_2"* %0, i32 0, i32 1, i64 %1
6   %3 = load %String** %2
7   ret %String* %3
8 }
```

We can see on line 2 that this method takes two arguments; the first argument is the vector that we wish to access and the second is the index of the element which we would like to return. Line 4 is where we calculate the address of the element using the `getelementptr` instruction; this instruction takes the vector as its first argument and then the subsequent 3 arguments are indices for getting the pointer to the element. The `getelementptr` instruction gets the address `%0 + 0 + 1 + %1`; In other words, the address of the vector plus `0` as the `getelementptr` instruction requires all calculations for dereferencing. Thus we must provide an argument which looks through the pointer to the vector. We then add 1 to obtain the array element of the vector, finally adding the value `%1` (the index argument to the function) which tells us which element we want. Once we have the pointer to our element we load it and return the value.

Next we go on to look at the `_update()` method:

```
define fastcc %String* @"Vector_String_val_$value_2__update"
  (%"Vector_String_val_$value_2"* dereferenceable(24), i64,
   %String* dereferenceable(32)) {
entry:
  %3 = getelementptr inbounds
        %"Vector_String_val_$value_2"* %0, i32 0, i32 1, i64 %1
  %4 = load %String** %3
  store %String* %2, %String** %3
  ret %String* %4
}
```

The code which is presented here closely resembles the _apply() method. The difference between _apply() and _update() is that we expect a third argument to _update(), this is the new value that we wish to store. We again calculate the address of the element and load the value, however after the load we replace the value with the new value (the new value being %3) using a store. Finally, we return the old value so as to perform an operation similar to Pony's destructive reads.

The leading _ in Pony marks a method or field as private to a class, therefore neither the _apply() nor _update() methods are visible outside of the builitins package (where this class can be found). We make these private as their arguments are not sanity checked and they cannot throw any errors. We use these primitives and wrap them with safe versions of these calls as can be seen in the apply() method at line 18. In this method we first check that the index is inbounds before we access the elements, throwing an error if it is not.

It is this solution which has been adopted as the implementation for Vectors. This approach was selected as it used LLVM built-in types to represent the structure of the type, also removing a level of indirection. This was also selected over embedding a pointer as this implementation is simpler to implement in the compiler. We also did not require most of the behaviour of the Pointer, such as resizing the memory allocated; this implementation avoids being able to use such methods.

6.3 Trace Function

We discussed in trace functions in section 5.4 and that we would not require adding extra tracing for the static values. For the Vector we do require a bespoke trace function as we are adopting a new structure of objects of Vector[A, _size: USize] (for some given type A and value _size).

There are two cases for the trace function for Vectors. One case is that the elements do not need tracing (for example for integer primitives); in which case we do not trace the elements and we can immediately return from the trace method. The second case is when the elements do need tracing (for example object elements); in the case tracing is required, calling the trace function on each element in turn and then returning from the function.

We are able to test the Vector trace function using three actors which collaborate in the following way:

1. Actor one sends a message of 3 Strings to actor two
2. Actor two receives the Strings and constructs a Vector from them. Actor two sends the constructed Vector to actor three.
3. Actor three tests the contents of the Vector against the original known values asserting that they are as expected.

At each step we invoke the Pony garbage collector,. If the trace function for the `Vector` did not trace the elements then the original `String`s would be garbage collected and so the final actor's assertions would (likely) fail. The test which has been detailed here can be found in Appendix D.

The test which we have described only checks that the `String`s were not deallocated during the test. It is more difficult to ensure that objects with no references are deallocated; this would require some telemetary data indicating how much memory and how many objects had been allocated and deallocated. Currently Pony supports telemetry data indicating how much data and how many objects had been allocated but not deallocated. Other telemetry data that is possible to use is the amount of time spent in trace functions, this would indicate that an object had been traced but not whether it was indeed garbage collected.

6.4 API

We can now further discuss how we use these `Vectors` and how they use value-dependent types. Reproduced in fig. 14 is part of the API for the `Vector` class. We attempt to provide an API which provides similar functionality to the `Array` class. In this API we are able to remove at least one argument in all of the constructors when compared with the `Array`. The argument we do not require is the argument which states the size of the `Array` to be allocated. The construction of this API has also lead to consideration in altering the `Array` API, for example providing an equivalent of the `generate()` constructor. In this API we are also able to construct further value-dependent types in methods signatures using the information available. We now go on to discuss such methods.

6.4.1 `add` method

We define the `add` method (line 48 in fig. 14) on the `Vector` class to have the following signature:

```
class Vector[A, _alloc: USize]
  fun add[_size': USize](vector: Vector[this->A, m]):
    Vector[this->A!, #(_alloc + _size')]^ =>
    // definition of add
```

Addressing each component of this signature individually as follows:

Class Definition: `Vector[A, _alloc: USize]`

This defines the `Vector` class to be parametrised on a `USize`, this represents the number of elements in the vector in the type, and a type `A` which denotes the type of the elements in the `Vector`.

```
class Vector[A, _size: USize]
  new init(from: Iterator[A^]) ?
    """
    Create a vector, initialised from the given iterator.
    """

  new generate(f: {ref(USize): A^ ?} ref) ?
    """
    Create a vector initiliased using a generator function
    """

  new undefined[B: (A & Real[B] val & Number) = A]()
    """
    Create a vector of len elements, populating them with random memory.
    This is only allowed for a vector of numbers.
    """

  fun filter(f: {val(box->A): Bool} val): Array[this->A!]
    """
    Return an array of elements that satisfy the predicate f
    """

  fun copy_to(dst: Seq[this->A!])
    """
    Copy the contents of this Vector to another Sequence
    """

  fun string(f: {val(box->A!): String} val): String ref^
    """
    Return a string representation of the vector
    """

  fun size(): USize
    """
    Return the parameterised size
    """

  fun apply(i: USize): this->A ?
    """
    Get the i-th element, raising an error if out of bounds.
    """

  fun ref update(i: USize, value: A): A^ ?
    """
    Change the i-th element, raising an error if out of bounds.
    """

  fun add[_size': USize](that: Vector[this->A, _size']):
        Vector[this->A!, #(_size + _size')]^
    """
    Build a vector from the contents of the original vectors.
    """
```

Figure 14: Vector API

Method Type Parameters: `add[_size': USize]`

The `add` method for vectors is parametrised on a value `_size'` of type `USize`. The argument that is passed for this parameter is used so that the type system knows the size of the vector argument that is passed for the `add` method.

Method Arguments: `(vector: Vector[this->A, _size'])`

Here the argument is defined to be a second `Vector` of size `_size'`, this must match the size on which the `add` method was parametrised. Also, the contents of the `Vector` must be the same as how the receiver views objects of type `A` (note that this is the same type `A` on which the receiving vector has be parametrised). This viewpoint adaption means, for example, that if the receiver has `val` capability then contents must of type how `val` sees `A`.

Method Return Type: `Vector[this->A!, #(_alloc + _size')]^`

The return value of the vector `add` method is a new vector which contains references to the elements of both the receiving `Vector`'s members and also the argument `Vector`'s members. Thus, we obtain a vector of type `Vector[this->A!, #(_alloc + _size')]^`. The return type denotes a vector which is of size `#(_alloc + _size')` (the sum of the sizes of the two vectors). The elements of the new vector are of type `this->A!`; the `!` represents that we have an alias to the elements, we justify that this is the type of the elements as we have created a new reference to the elements and we have also maintained the existing reference from the existing vectors. Finally, we return an ephemeral `Vector` (denoted by `^`), this is the case as the method will construct a new `Vector` to which no reference will exist once we leave the scope of the function.

We define the method in such a way using the `add` method as this suggests we could write the following:

```
1 actor Main
2   new create(env: Env) =>
3     let v1 = Vector[String, 4].init(["A", "B", "C", "D"])
4     let v2 = Vector[String, 2].init(["E", "F"])
5     let v3: Vector[String, 6] = v1 + v2
```

Here the `+` operator is syntactic sugar for calling the `add` method with `v1` as the receiver and `v2` as the argument; returning the joined vectors together to get us `v3`.

However, note we defined `add` to be parametrised on a `USize` argument. In the above example the `+` operator does not provide this argument; this means that we would have to define line 5 as follows:

```
5 let v3: Vector[String, 6] = v1.add[2](v2)
```

To be able to perform the addition operation for two vectors, the Pony compiler requires support for inference of value arguments. To demonstrate this, consider the following:

```
class C1[n: U32]
  fun apply(): U32 => n

  fun add[m:U32](c: C1[m]): U32 =>
    apply() + c()

actor Main
  new create(env: Env) =>
    C1[2].add[4](C1[4])
```

Noting again how we must provide the argument `4` to `add`, we can see however that from the argument provided to add (i.e. `C1[4]`) that we know the value of `m` at compile time. Thus it becomes possible to omit the argument to `add` and infer the value from the argument `c`. This is currently a desired feature of the Pony compiler (alongside inference of type arguments), which once complete will allow for more succinct use of value dependent types.

6.4.2 generate and string methods

We will briefly discuss two interesting methods in the `Vector` API; the `generate` method and the `string` method. Firstly, note the definition of the `generate` constructor:

```
new generate(f: {ref(USize): A^ ?} ref) ?
```

The `{ref(USize): A^ ?}` syntax is used to denote the type of a `ref` object that has a method called `apply` which accepts a `USize` argument and returns a value of type `A`, possibly raising an error. This allows us to pass lambda methods to the `generate()` constructor. This constructor generates a new `Vector` by repeatedly invoking the generator function, the `f` argument, supplied. This allows us to construct vectors in the following ways:

```
1 let v1 = Vector[U32, 6].generate(lambda ref()(rand=MT) => rand.u32() end)
2 let v2 = Vector[U32, 4].generate(v1)
```

Here `v1` is initialised to random values and `v2` is a vector which duplicates the first 4 elements of `v1`. Note here how we are able to pass either a function, as in line 1, or a previously constructed object, as in line 2. We can do this as the `Vector` adheres to type constraint of the `genereate()` constructor. This provides us with a fair amount of flexibility in how we construct `Vectors`.

Along a similar theme we define the `string` method:

```
fun string(f: {val(box->A!): String} val): String ref^
```

This allows us to construct a string representation of the vector by accepting a function which knows how to turn the element type of the vector into a string, this is then invoked on each element to build up the complete representation. Such a method gives us a lot of flexibility over how we represent the elements, we could even represent the same contents in multiple ways by passing different functions to the `string` method, furthermore it allows us to construct string representations for objects which are not a subtype of `Stringable`.

6.4.3 `filter` method

We have included a means by which to filter a vector to obtain only those values which satisfy the provided predicate function. This method is of interest as we cannot return a `Vector` as the result, we a forced to return some `Seq` object (here we have use `Array`) as we do not know the size of the resulting structure at compile time. Compare this to `Vector`'s in Idris where it is possible to lose the information of the dimensions of a vector by stating that it has an arbitrary size. We do not have this flexibility in Pony and so we must return an `Array`.

7 Matrix Class

To demonstrate further the flexibility in Ponyta using value-dependent types and compile-time expressions I extended the Pony standard library with another class, namely the `Matrix` class.

The `Matrix` class depends on two values. The first value is denoted by a `USize` named `n`, this details the number of dimensions of the matrix. The second value is a `Vector[USize, # n]` named `dims`, this stores information describing the size of each dimension. Example instantiations of a `Matrix` follow:

```
1 let m1 = Matrix[Student, 2, # {3, 4}] // 3 x 4 matrix
2 let m2 = Matrix[Student, 4, # {1, 2, 3, 4,}] // 1 x 2 x 3 x 4 matrix
3
4 let m3 = Matrix[Student, 5, # {1, 2}]
5 // will fail to compile as we haven't provided enough dimensions
```

The `{3, 4}` syntax denotes a literal `Vector`. The literal `Vector` on line 1 is inferred to be of type `Vector[USize, 2]`.

The signature for the `Matrix` was defined in section 1, reproduced here:

```
class Matrix[A, n: USize, dims: Vector[USize, # n] val]
  embed _data: Vector[A, # _alloc()]

  fun tag _alloc(): USize =>
    var i: USize = 0
    var acc: USize = 1
    while i < n do
      acc = acc * dims._apply(i = i + 1)
    end
    acc

  new undefined[B: (A & Real[B] val & Number) = A]() =>
    _data = Vector[A, # alloc()]._create()

  fun _calculate_address(indices: Vector[USize, # n]): USize ? =>
    if n == 0 then return 0 end
    var address: USize = 0
    var i: USize = 0
    while i < (n - 1) do
      if indices(i) > dims(i) then error end
      address = (address + indices(i)) * dims(i + 1)
      i = i + 1
    end
    if indices(i) > dims(i) then error end
    address + indices(i)

  fun apply(indices: Vector[USize, # n]): this->A ? =>
    _data._apply(_calculate_address(indices))

  fun ref update(indices: Vector[USize, # n], value: A): A^ ? =>
    _data._update(_calculate_address(indices), consume value)
```

We could have defined the `Matrix` using an `Array` whose size was determined dynamically, however we again aim to remove the indirection to elements and allocate an appropriately sized object.

The definition of the `Matrix` class we provide here, uses the richness provided by value-dependent types and compile-time expressions to define a fixed arbitrary sized data structure, which can be accessed in a non-trivial way to obtain a row-order layout of multi-dimensional data.

Note in this definition that the `Matrix` depends on a `Vector` object `dims`. This displays how we can parametrise type on values beyond just the primitive built-in values such as integers and booleans. Furthermore, note the type signature for the `apply()` method. We ensure that we have exactly the correct number of indices provided. We do this by expecting a `Vector` with as many elements as the number of dimensions. This is an interesting result as if we pass an argument with the incorrect number of arguments, the call will raise an error at compile-time. However, we expect a different number of indices based on the instantiation of the `Matrix`. This is something we could not have defined in Pony and is a very interesting achievement of Ponyta.

At line 2 we define the single field, `_data`, of the `Matrix` class which is the one-dimensional `Vector` used to store the elements of the `Matrix`. A point of interest is how the size of the `Vector` is determined using a compile-time expression, namely the evaluation of `alloc()`. I have defined the member `_data` to be embedded within the `Matrix`

The `alloc()` function iterates through `dims`, multiplying the dimensions together to calculate the number of elements in the `Matrix`. This is an example of compile-time iteration using `var` variables and statically known objects. Notice this method has `tag` capability; this capability is required because it allows us to call the method on an object which has not been initialised. Even though the `tag` capability forbids access to fields of the receiver, this does not impede the implementation of `_alloc()`, since all information required for this object is found in the type of the object.

To obtain a row-order layout of a `Matrix` we use a `_calcluate_address()` method which takes as an argument a `Vector[USize, # n]` where the i-th element in the `Vector` is the index into the i-th dimension. The calculation used to obtain the index into `_data` is similar to that used in section 9.2.1 in the one-dimensional vector case, here scaled to `n`-dimensions. The `_calcluate_address()` is used for both accessing and updating elements as can be seen at lines 28 and 31.

One of the most important results of this class is the fact that we have defined the entire class in Pony, thus demonstrating the flexibility in defining types provided by Ponyta.

7.1 Layout

Consider the following `Matrix` instantiation:

```
let matrix = Matrix[U32, # {1, 2, 3}]
```

The elements of the `matrix` are ordered as follows to obtain a row-order layout:

{0,0,0}	{0,0,1}	{0,0,2}	{0,1,0}	{0,1,1}	{0,1,2}

Note that the elements of each row appear contiguously. We select this layouts for two reasons. Firstly, it is the simplest representation to obtain using only high-level Pony and without having to incorporate extra compiler-support. Compare this with arbitrarily nested `Vectors`. We could not write this using only Pony as we would not know the depth of the nesting and therefore we would not know the type. Such a representation would require to be constructed when generating code. Secondly, and more importantly, this layout used to represent a two-dimensional `Matrix` using only a `Vector` elicited better runtime compared with a nested data structure. We will see this result in section 9.2.1.

8 Compiler Support

I have discussed so far how value-dependent types have been implemented and also how compile-time expressions are evaluated. Also, I have discussed how these have affected code generation. I now detail at what stages of compilation changes have been made to accommodate these new features. Detailing why these changes are required of their respective pass.

8.1 Parse

The changes for this pass can be found in section 3. This consists of extending the parser to support the syntax for compile-time expressions. Also, supporting the syntax for literal `Vectors`. Building the AST representation in both cases.

8.2 Syntax

This pass has been changed to ensure that the type parameters adhere to a legal syntax. This change in syntax now permits both upper and lower cases identifiers (previously only upper case was permitted for type parameters). Also, the constraints of these parameters are also check to ensure they are types and not values.

8.3 Name

We discussed earlier that references to value parameters are marked in the AST by a `valueformalparamref` node. It is in the "name" pass that this transformation is made. All references in the AST are initially marked by a `reference` node. Each `reference` node is inspected during this pass. A lookup is performed to find the original definition of the value referenced by the `reference`. If this definition is a value parameter then the `reference` is transformed to a `valueformalparamref`.

8.4 Traits

I have extended the reification of generic traits in this pass. This extension includes reifying traits with value arguments as described in section 5. Note that this replacement is entirely textual as discussed in section 2.5.1. The expressions are not evaluated before reification as they have not been typechecked and therefore may be unsafe to evaluate. This duplication of expressions is handled using the cached results to ensure we obtain the same result in all replacements. The cache is also used to avoid paying extra time during compilation to evaluate the same expression multiple times.

8.5 Expr

The "expr" pass has been further developed to incorporate the new value-dependent types and their subtyping rules. This pass also handles providing types to compile time expressions, recovering the capability to `val`. Also, this pass infers the type of literal vectors and coerces vector elements to their desired typed. For example, infering the type `Vector [String, 3]` from the expression:

```
let v = {"A", "B", "C"}
```

I have extended the type checking algorithm with an extra case which checks "subtyping" between values. This is in fact the check for equality that we discussed in section 5. Before checking this equality we attempt to evaluate the expressions. If the expression cannot be evaluated, possibly as we are type checking a template, we compare equality on the original expression.

8.6 IR

The first part of this pass involves the compiler determining which types and methods are reachable in the current program. Each of these types and methods are respectively added to a list of reachable types and reachable methods. It as at this stage that we evaluate all compile-time expressions (which haven't been evaluated as a result of type checking). We recursively search the AST representation of only the reachable methods and types, evaluating any compile-time expressions which we find along the way.

We defer evaluation of compile-time expression this late in compilation for the following reasons. We require that all expressions and any classes/methods/expressions transitively involved in the evaluation of the expression to have been typechecked. This is because the interpreter expects the AST of the program to have been appropriately transformed by the type-checking pass.

Deferring evaluation until this stage also means that we only evaluate expressions which will have a result on the produced program. We do not evaluate any expression which is unreachable. This has the effect that expressions which would result in failure of compilation but are unreachable will not raise any error to the developer. However, we argue that this is not a detrimental side-effect. A developer is only alerted to the errors which prevent the program from compiling.

One issue with this approach is that the compiler duplicates many AST nodes throughout the passes, including the compile-time expressions. This occurs for example when we infer the type of a variable based on the value which is assigned to the variable. Thus we invoke the interpreter on the same expression multiple times. This cost has been amortised by caching the result of evaluation as described in section 4.1.7.

It is also in this stage that all new code generation has been added to the compiler. This includes function generation for value-dependent types. Constructing `Vectors`, their compiler intrinsics methods and trace method. This generation also includes building a constant object in LLVM IR from constant objects in the AST.

8.7 Evaluation

I have developed the psuedo-interpreter as a component of the compiler that can be utilised by other passes of the compiler to evaluate expressions. This allows the interpreter to be used in both the "expr" pass for type checking and in the "ir" pass for evaluation of expressions.

The interpreter is passed an AST representation of an expression which is to be evaluated. The interpreter uses rules which are selected based on the identifier of an AST node. Evaluation proceeds recursively, evaluating the child nodes such as receivers and arguments, before evaluating the parent node.

Evaluation can involve looking up methods and values that are referenced in an expression. I have implemented the interpreter such that a new node is generated by evaluation. This is to avoid nodes which are being used elsewhere in the AST from being altered. After evaluation of an expression a final step is performed to replace the expression sub-tree with the result sub-tree, as show in section 5.

9 Evaluation

9.1 Comparison with the State of the Art

Throughout this report I have compared value-dependent types in Ponyta with those presented in Idris and also with the templated methods and compile-time expressions available in C++. I now present a qualitative analysis of Ponyta.

9.1.1 Comparison with C++ templates and `constexpr`

We first compare Ponyta with C++ in terms of the syntax and rules required of templates and `constexpr` functions. We can also compare Ponyta with C++ with respect to the flexibility that each language provides to a developer.

Compile-Time Expressions

In C++, `constexpr` functions can use any `literal` type, locals, iteration, conditionals and calls to other compile-time expressions. We make similar restrictions in Ponyta. Take the following C++ example:

```
int foo()
{
  return 1;
}

constexpr int bar(int x)
{
  return x * foo();
}

int main()
{
  constexpr int y = bar(12);
}
```

This program will fail to compile as the function `bar()` calls a `non-constexpr` function, namely `foo()`. However, note that the function adheres to all rules required of a `constexpr` function except the `constexpr` keyword.

We now show how Ponyta is more flexible in allowing a developer to evaluate expressions at compile-time. Consider the same example as above, however we will now write the program using Ponyta:

```
actor Main
  fun foo(): U32 => 1
  fun bar(x: U32): U32 => x * foo()

  new create(env: Env) =>
    # bar(12)
```

We impose the same restrictions as defined for C++. All variables must be known at compile-time and all functions called within a compile-time expressions must be compile-time functions. The above function will successfully compile and execute in Ponyta. In this respect Ponyta is more relaxed than C++. In Ponyta we trust the developer has adhered to the rules of compile-time expressions and attempt to evaluate their expressions. If an error is encountered during evaluation then an error is reported detailing what could not be evaluated.

By avoiding the `constexpr` keyword and using a `#` to denote compile-time expressions, we allow a developer more flexibility in using compile-time expressions. A developer can attempt to evaluate a function or expression at compile-time by placing a `#` in front of such an expression. This is a much less invasive and involved change than adding `constexpr` keywords to constructors and functions which may otherwise require no alteration.

We were able to avoid the `constexpr` keyword in some cases due to the capabilities of values and distinguishing between single-assignable and re-assignable names in Pony. The `val` capability gave us a read-only behaviour on values which was very similar to `const`. Ensuring value were `val` once they left a compile-time expression and together with the `let` constants provided a very good starting point to determining which values could be known at compile-time.

Furthermore, recall that Pony makes no distinction between objects and pointers as in C++. This allows us template functions and classes in a more straightforward manner. Consider the following C++ program:

```
class C1
{
public:
  int x;
  C1(int x): x(x) {}
};

template <C1* c>
class C2
{
public:
  int apply() { return c->x; }
};

static C1 c1(12);

int main()
{
  C2<&c1> c2;
  int x = c2.apply();
}
```

In the above, to instantiate `C2` we have to provide a `C1*` with static linkage. This is why we must define `c1` at line 15 as `static` and take the address of it at line 19. Once, again consider the same program written in Ponyta:

```
1 class C1
2   let x: USize
3   new create(x': USize) =>
4     x = x'
5
6 class C2[c: C1 val]
7   fun apply(): USize => c.x
8
9 actor Main
10   new create(env: Env) =>
11     let c2 = C2[# C1(12)]
12     let x: USize = c2()
```

Here we have no distinction between objects and pointers. We also do not require to declare type arguments as static global values as we did in C++. We construct a new `C1` object at line 11 as the result of a compile-time expression.

We can argue from the examples shown here that typically the Ponyta syntax is more succinct for defining and using value-dependent types than in C++.

Permissiveness of Templates

We can also compare the permissiveness of templates in C++ when compared with parametrised classes and methods in Ponyta. We discussed in section 5.2 that in Ponyta, parametrised classes and methods and type-checked using only the template definition. Under this approach we restricted equality of value-dependent types to syntactic equality of expressions.

C++ defers type checking a templated class or method until the template has been instantiate. This lazier approach to type check provides a developer with slightly more flexibility when defining templates. However, this lazier approach comes with the caveat of type checking every instantiation of the class or method. Ponyta's approach to this requires only a single type checking pass on the template.

Therefore we make a tradeoff between permissiveness and efficiency. The eager approach seems to have been the correct choice for Pony. This method matched the existing implementation for generics and thus provides a developer with a consistent behaviour. Moreover, it is not often that a developer will need to write two different expressions to represent the same value. Finally, consider type checking every different instantiation of a `Vector` or `Matrix`. This could lead to many type checking passes and, in turn, a significant overhead in compilation time.

Template Specialisation

C++ allows a developer to construct specialised instances of templated classes and methods. Consider the following, templated, definition of the factorial function in C++:

```
1  template<int n>
2  int fac() { return n * fac<n - 1>(); }
3
4  template<>
5  int fac<0>() { return 1; }
6
7  template<>
8  int fac<1>() { return 1; }
9
10 int main()
11 {
12   fac<10>();
13 }
```

Here we have provided specialised definitions for `fac<0>()` and `fac<1>()`. Notice the call `fac<10>()` at line 12. When the definition of `foo` is reified using the value 10, we will also have to instantiate `foo<9>`. Similarly, we will have to instantiate `foo<8>` and so until `foo<1>`. At this point we no longer have to reify any more instantiations as we have provided the method body for the instantiation `foo<1>`. Without this base case, we would keep instantiating methods until we exceeded the template instantiation depth.

In Pony we cannot specialise parametrise methods and classes as we can in C++. We can construct a program such as in the following:

```
actor Main
  fun fac[n: USize](): USize =>
    n * fac[#(n - 1)]()

  new create(env: Env) =>
    fac[10]()
```

Currently in Pony, we cannot prevent the call `fac[10]()` from instantiating `fac` for all values in `USize`. However, we discuss future support of template specialisation in section 11.5. This feature will build upon value-dependent types to give developers an even greater degree of flexibility.

9.1.2 Comparison with Idris value-dependent types

Let us now examine Ponyta in comparison with Idris. Recall the `Vect` data type:

```
data Vect : Nat -> Type -> Type where
  Nil   : Vect Z a
  (::)  : a -> Vect k a -> Vect (S k) a
```

Here the `Vect` data type has two constructors, `Nil` and `Cons` which build up a list which tracks its length in the type. We have the `Vector` which satisfies this property. For the purpose of this evaluation we will attempt to define something similar to the Idris style vector. We will define a linked list that knows its length in the type signature. Take the following definition of such a list:

```
trait List[A, n: USize]
class Nil[A] is List[A, 0]
class Cons[A, n: USize] is List[A, # n]
  let head: A
  let tail: List[A, # (n - 1)]
  new create(head': A, tail': List[A, # (n - 1)]) =>
    head = consume head'
    tail = consume tail'
```

In this definition we use a trait to define the `List` type which is composed of an element type and a length. We then use classes as our constructors. The `Nil[A]` class is used to represent an empty list (i.e. `List[A, 0]`. The `Cons` class is used to add another element to the front of the list. The `Cons` class uses a field, `head`, to represent the element which it is appending to the head of a list. The class has a second field, `tail`, used to track the remainder of the list. Notice that the `tail` must have type `List[A, # (n - 1)]` and that the `Cons` class is of type `List[A, # n]`. We can then go on to construct a list as follows:

```
actor Main
  new create(env: Env) =>
    let l3: List[String, 2] = Cons[String, 1]("B", Cons[String, 0]("A",
       Nil[String]))
```

Therefore, we have been able to construct the lists as in Idris. Notice that we couldn't pass the `Cons` constructor a list that would generate an incorrectly sized list. This is due to the constraint on the `tail` type, namely the `tail` must be of type `List[A, # (n - 1)]`. From this we get similar guarantees as we do about the length of the `Vect` in Idris.

Let us now attempt to define, in Ponyta, the append operation we saw earlier in Idris. In the following example we use lists of only type `U32` for clarity.

```
1   fun append[n: USize, m: USize](xs: List[U32, # n], ys: List[U32, # m]):
        List[U32, # (m + n)] ? =>
2     match xs
3       | let nil: List[U32, 0] => ys
4       | let cons: Cons[U32, # n] => Cons[U32, # (m + n)](cons.head,
            append[# (n - 1), # m](cons.tail, ys))
5     else
6       error
7     end
```

One thing that we can see here is that the definition is much more verbose than in Idris. We require all sizes and types to be provided as parameters to the function. Furthermore, this definition will fail to compile. The compiler cannot prove that `# ((m + n)- 1`, the length of the `tail` at line 4, is equivalent to `# (m + (n - 1))`, the return type of the `append`. It is in this way that Ponyta only provides simple value-dependent types and not a fully dependently typed language. More research can be made in this area to help the compiler prove more equivalences between expressions using compile-time expressions and value-dependent types.

We will now see that there are cases where Ponyta provides clearer definitions and allows for easier construction of value-dependent types. In Idris to access a `Vect` we provide a finite set which must be smaller than or equal to the size of the `Vect`. The size of the set is used as the index into the `Vect`. An example of this access follows:

```
1 index (FS (FS FZ)) [1, 2, 3] -- compiles
2
3 index (FS (FS FZ)) [1] -- fails to type check
```

Here, `FZ` is set of size zero, `FS` is constructor which takes a set and creates a set which is one larger[4]. The example on line 1 is safe as the set has a size less than that of the `Vect`. The example on line 3, however, fails to compile as the set has size 2 whilst the `Vect` is of size 1. Whilst this ensures that the index is safe and guaranteed to be in bounds, the access is not particularly clear nor concise. In Ponyta we can define a similar access which statically guarantees an access will be in bounds and is more concise. For this example we reuse the `Assert` primitive:

```
class Vector[A, _size: USize]
  ... // API so far
  fun apply_static[i: USize]() =>
    # Assert(i < _size)
    _apply(i)
```

Here we use a static argument as an index into the `Vector`. We use `Assert` to ensure that the index is within bounds at compile-time. This definition allows us to safely access a `Vector` as follows:

```
let v = {1,2,3}
v.apply_static[2]() // compiles

v.apply_static[13]() // fails to compile
```

Here, we get a similar result but the syntax can be seen to be clearer and more succinct.

Moreover, recall that in Idris, in section 2.2.1, we constructed a different matrix type compared with the Ponyta `Matrix`. We constructed a list of lists in Idris instead of a matrix. We defined the Idris matrix in this way as constructing the multi-dimensional `Matrix` presented in Ponyta is non-trivial. This is due to us not being able to easily define and update an appropriately sized `Vector` as we can in Ponyta. This is a result of Idris being a functional programming language. The Idris data package represents a matrix as a two-dimensional data structure with a row and column size as part of the type[2]. Multi-dimensional matrices can be achieved by nesting matrices. This gives a nice result as we can see that there are types which become easier to construct with value-dependent types in Ponyta than in Idris. In Ponyta we have a single type which represents a matrix of any dimensionality.

From this we can conclude that Ponyta provides developers with a different set of features than Idris. This difference allows some types and functions to be more easily defined in one language compare to the other. Whilst this project hasn't developed a fully dependently-typed language like Idris, we have shown that we have created a fairly flexible extension to the Pony type system, successfully creating many of the types we explored earlier in Idris. Of particular importance is how we have been able to define the `Vector` and `Matrix` classes. We now go on to evaluate these classes.

9.2 Value-Dependent Data Structures

This project has presented two new data structures in Ponyta. Namely, the `Vector` class in section 6 and the `Matrix` class in section 7. I now evaluate the efficiency of these data structures when compared with the existing data structures in Pony.

9.2.1 Benchmarking Vectors

As detailed in section 6, the `Vector` adopts a different layout when compared with an `Array`. We remove a layer of indirection to data as we do not go through a pointer. This leads to considering whether there is any practical performance difference between using a `Vector` and using an `Array`.

Sorting Benchmark

To explore this, a sorting package was developed which included an interface as follows:

```
interface Sortable[A: Comparable[A] #read]
  fun size(): USize
  fun apply(i: USize): A ?
  fun ref update(i: USize, value: A): A^ ?
```

This interface represents a data structure which can be sorted; these structures require `apply` and `update` methods so that we can alter the contents of the structure. Note also the type constraint of `Sortable`, `[A: Comparable[A] #read]`. This ensures that we can compare elements of these structures. This allows us to treat `Vector`s, `Array`s and `List`s as `Sortable` and thus we can use the same sorting methods.

Included in the sorting package are some well known sorting algorithms such as insertion sort, quick sort, heap sort and bubble sort (among others). `Array`s and `Vector`s of increasing sizes were sorted using the various sorting algorithms to measure for observable differences in performance. Benchmarking was performed on a 3.40GHz 4 Core Intel Core i7-4770 with 16GB of memory. Each sorting was repeated 20 times and the average time taken. The graphs in fig. 15 detail the results of this benchmarking:

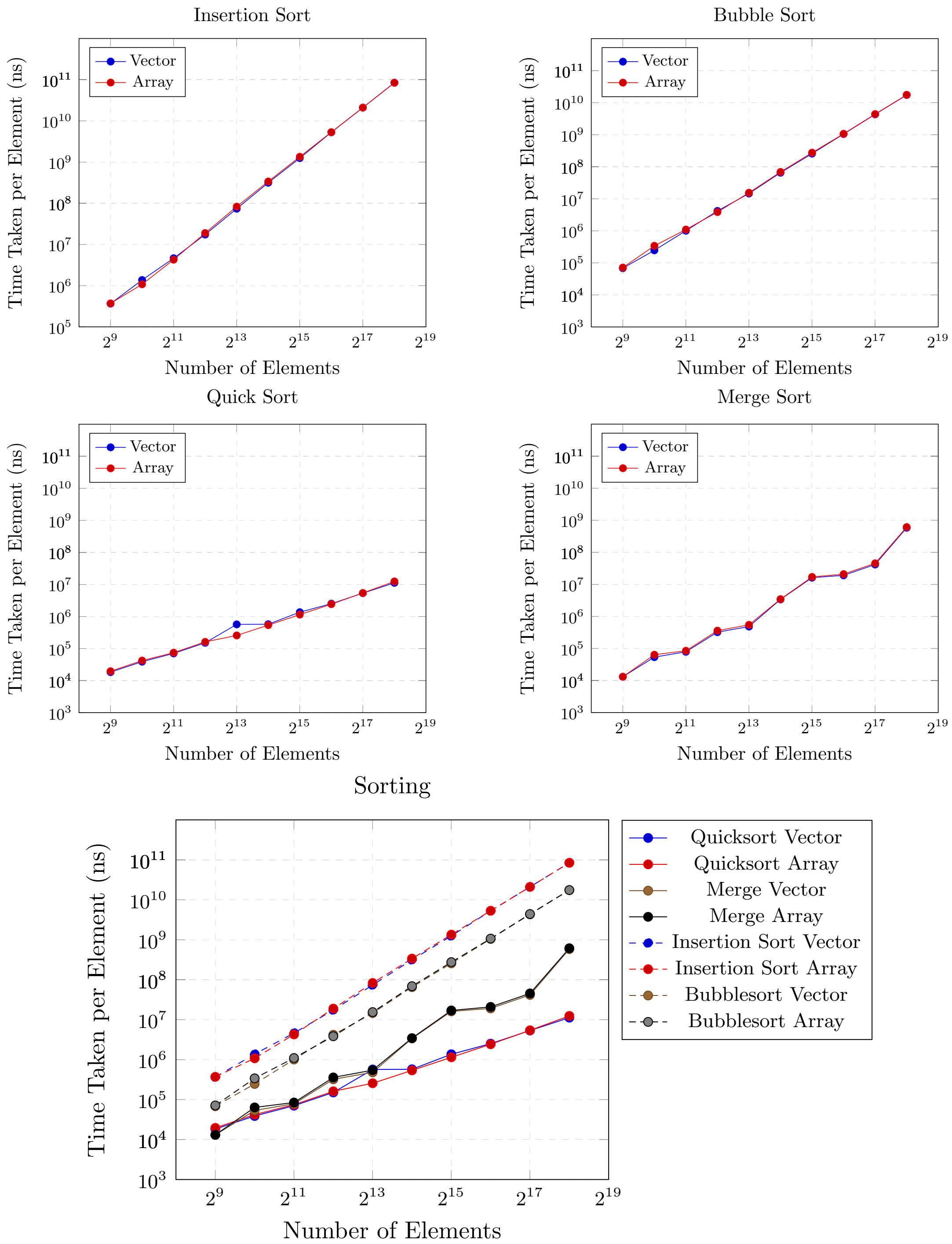

Figure 15: Comparison of sorting algorithms

Benchmark Results

The results for these benchmarks can be found in Appendix J. We can see from these results that there is a 1.03 times speed-up on average using `Vector`s instead of `Array`s. This speed-up is not as large as expected, however consider that the presented results are measured in nanoseconds. We measure these results in nanoseconds as the benchmarks were already fairly quick.

These results suggest that there is little or no runtime impact due to this change in object layout. Although this contradicts the initial hypothesis that such a change in layout would benefit execution time, we can conjecture the reasons for this result. The LLVM backend could heavily optimise the sorting algorithms such that all of the accesses to the `Array` do not first lookup the pointer in the `Array` but use the pointer directly. Furthermore, even if the compiler did not perform this optimisation, the pointer would likely be cached by the processor. These optimisations could amortise the cost of the indirection involved when using an `Array`.

Nested Data Benchmark

Further efforts to find differences between the runtime behaviour of `Array`s compared with `Vector`s lead to researching whether the equivalent of these structures in the C++ STL (Standard Template Library) had been benchmarked (noting that the Pony `Vector` corresponds to `std::array` and `Array` to `std::vector`).

Research for further benchmarking resulted in sources which compared the initialisation of a nested `std::vector` with a nested `std::array`, claiming to display a noticeable difference in performance. However, reviewing the provided code snippets revealed a use of `push` when initialising the `std::vector`, thus the the `std::vector` was being resized multiple times during the initialisation. Thus, this test wasn't an appropriate benchmark as it did not measure a difference as a result of object layout and it is possible to allocate enough memory for both the `std::vector` and `std::array` to avoid the use of `push`.

One result which is of interest, in C++, is that we can allocate an `std::array` entirely on the stack (members included) whereas when we construct a `std::vector`, even if the data structure is on the stack its elements will be on the heap. Allocating these different data structures in C++ can lead to a difference in performance due to the requirement of requesting heap memory when allocating the `std::vector` elements whereas the space for a `std::array` is reserved on the stack when entering a function [10]. However in Pony all objects live on the head, and as such a differentiation does not arise.

This research lead to further consideration of initialising nested `Array`s and `Vector`s. A second benchmark can be found in Appendix C. We aimed to write code that is difficult for the LLVM backend to optimise; avoiding the performance benefits of caching by regularly

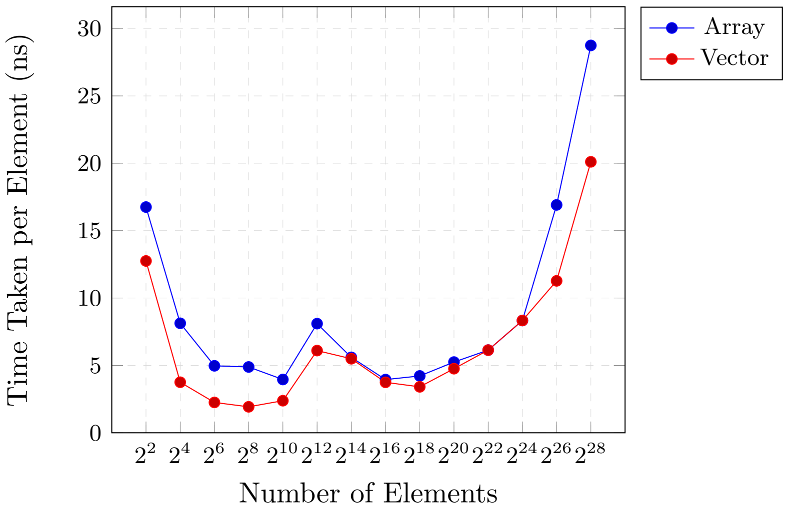

Figure 16: Updating Nested Data Structures

invalidating the cache and then requesting data that once existed in the cache. In Appendix C we construct two actors; one actor constructs a nested `Vector` and then initialises this structure such that every element is equal to its position in the structure. We then loop over the nested `Vectors`, however note that we update column by column instead of row by row. The second actor performs similarly, however uses `Arrays` as the data structure in place of `Vectors`

When we construct these nested `Vectors` and `Arrays` we get a two-dimensional data structure laid out in row-major order (i.e. the elements of rows are contiguous in memory). Therefore to access these structures as if they were laid out in column-major order would be inefficient as we would be accessing a different `Array`/`Vector` on each iteration which incurs a performance impact as we cannot (necessarily) cache the data structures. We use this method in an attempt to force the lookup of an element from the `Array` structure which has the double indirection when accessing elements.

The results of executing this on `Arrays` and `Vectors` can be found in Appendix K and these have been plotted in fig. 16. In fig. 16 we can observe more difference between the execution time when updating `Arrays` compared with updating `Vectors`. The speed-up obtained using `Vectors` ranged between 0 - 2.5 times speed-up with an average of 1.47 times speed-up when using `Vectors`. Inspection of the generated LLVM reveals that the actor interacting with the `Vectors` had been optimised much more than the actor which interacted with `Arrays`.

Floyd-Warshall Benchmark

I present a final benchmark which is of particular interest. I present this benchmark as it is used to assess the existing Pony compiler. This benchmark is of interest as altering the underlying representation of the graph had a dramatic impact on the performance of the Benchmark. The initial representation of the graph was a two-dimensional array of `U32` (i.e. `Array[Array[U32]]`), as described in section 9.2.1 such a representation involves finding the pointer field in the outer array, dereferencing this pointer to obtain the desired inner array and repeating on the obtained array. The representation of the graph was altered to use a one-dimensional array of `U32` (i.e `Array[U32]`) whose size was large enough to accommodate all nodes of the original graph, this required then changing index calculations to access the correct elements. Presented in the fig. 17 is a comparison of the Floyd-Warhsall benchmark which compares these two data-layouts. The one-dimensional `Array` showed a 1.41 times speed-up compared with two-dimensional `Arrays`.

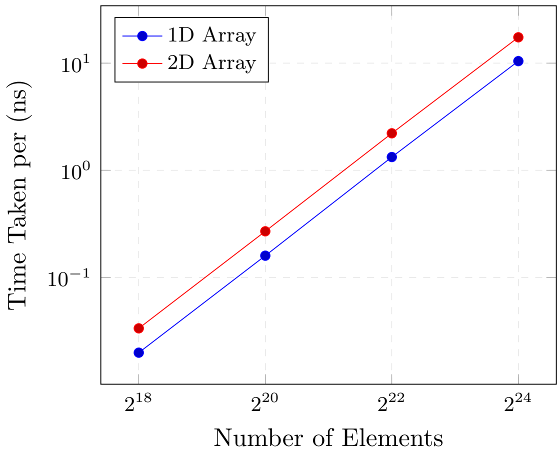

Figure 17: Floyd-Warshall Benchmark using one-dimensional and two-dimensional `Arrays`

I ran a similar experiment using the `Vector` class in place of the `Array` class to explore:

1. Whether a similar trend was observed between one-dimensional and two-dimensional `Vectors`.
2. Whether the `Vector` class performed any better than the `Array` class.

Figure 18 displays the results of using a `Vector` class instead of an `Array` class together with the original `Array` class.

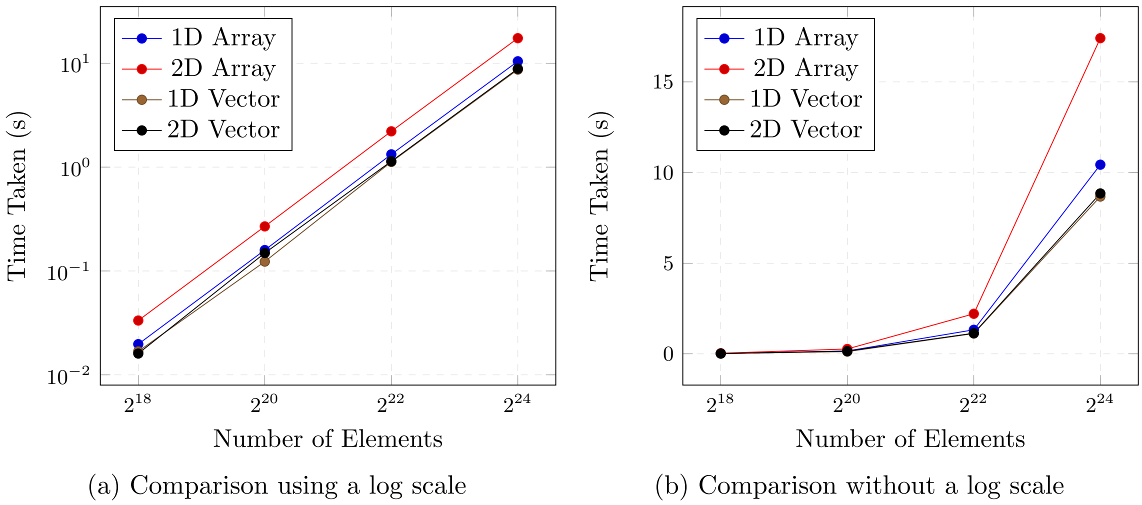

Figure 18: Floyd-Warshall Benchmark using one-dimensional and two-dimensional `Arrays` and `Vectors`

We can see from fig. 18 that the different `Vector` layouts do not display such a large difference in runtimes when compare with the `Array` layouts. The one-dimensional `Vector` performs very similarly to the two-dimensional `Vector`, with the one-dimensional layout providing a 1.10 times speed-up. Furthermore, both `Vector` layouts elicit better performance than the one-dimensional `Array`. In fig. 18a we can see that, for all number of elements, the `Vector` gives a reduced runtime when compared with the `Array`. On average the one-dimensional `Vector` gave a 1.24 times speed-up compared with the one-dimensional `Array`. We also plot the results without a log scale in fig. 18b, to display a more noticeable difference between the runtimes using `Array` and `Vector`. Notice again that in fig. 18b the nesting of `Vectors` did not noticeably impact the runtime.

The performance improvement displayed using `Vectors` is likely a result of the LLVM backend have greater capability to optimise the generated code. Using the `LLVMArrayType` layout provides the LLVM optimiser with extra information for analysis, namely the size of memory allocated. Furthermore, the LLVM optimiser has optimisation passes which are designed around LLVM aggregate types (such as the `LLVMArrayType` used for `Vectors`)[7].

The code for the implementation of each benchmark using the representations of `Array[Array[U32]]`, `Array[U32]`, `Vector[Vector[U32, #size], #size]` and `Vector[U32, #size]` can be found in Appendices E to H. The results for the Floyd-Warhsall benchmarks can be found in Appendix L.

9.2.2 Benchmarking Matrix

Having defined the `Matrix` class in section 7, an interesting experiment is to compare this with the one-dimensional `Vector` using the same Floyd-Warshall benchmark used to compare `Arrays` and `Vectors`. We expect that the `Matrix` behaves identically to the one-dimensional `Vector` case as we use the same representation. The benchmark used to evaluate the `Matrix` class can be found in Appendix I, the results for this benchmark can be found in Appendix L.

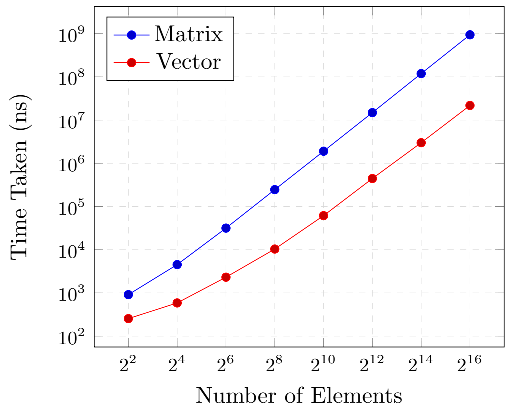

Figure 19: Floyd-Warshall Benchmark using a one-dimensional `Vector` and a `Matrix`

Benchmarking the `Vector` against the `Matrix` using the Floy-Warhsall algorithm produced the result in fig. 19. The results obtained indicate that, despite having a very similar underlying representation, the `Matrix` performed significantly worse than the `Vector`. We can first notice that we were not able to perform the Floyd-Warshall algorithm on `Matrix` objects as large as we were for `Vectors`. Using more that 2^{16} elements exceeded the amount of memory available. This is a particularly odd result as the size required should have been very similar. Secondly, the run-time for the `Matrix` class is significantly worse than the `Vector`. The `Vector` elicited a 25 times speed-up compared with the `Matrix`. Both of these results may be a consequence of LLVM being able to apply more powerful optimisations on the `Vector` and not on the `Matrix`, possibly making the `Vector` algorithm a constant-size operation.

Initial research into this suggested that an issue may be due to the fact that the calculation of the address within the `Matrix` benchmark occurs within the class instead of in the loop in the Floyd-Warhsall benchmark. This leads to LLVM not optimising the data accesses, possibly missing loop induction variables. More research is required to discover why this was the result of benchmarking. Improving the `Matrix` class to obtain a better runtime using this matrix is left as a future extension.

9.2.3 Conclusions on Results

The benchmarking presented in this section has shown that the new data structures for Ponyta are able to obtain better runtimes when executing certain benchmarks. We must consider the benchmarks that were used for researching these results. In all of these benchmarks, the size of the data structure was known a priori.

The `Vector` and `Matrix` types presented in this report rely on the size being specified at compile-time. Therefore, to use these classes effectively requires knowledge of the dimensions before execution. The original Floyd-Warshall benchmark (provided to me by Juliana Franco) read the graph from a file. The benchmark constructed a graph whose size was specified by a command line argument. The command line argument is only available at runtime and not at compile time. To use the `Vector` class over the `Array` in such a situation may not be possible or at least not as precise in terms of memory usage. One could allocate a `Vector` large enough for any situation, leading to a possible over allocation of memory. This may not be possible at all if there are no known bounds on the size of a the store. Furthermore, these structures are not suitable if the stores are being resized frequently (although not a particularly effective use of `Array)`. Resizing is not an operation which is possible for `Vector`s. To increase the size of a `Vector` requires static knowledge on the previous size, the new size and a shallow copy from one data structure to the next.

From the results presented in this section I form the following conclusion. We have successfully developed more efficient containers to use when the size of the container is known in advance. Therefore these statically determined, fixed sized data structures should be used in conjunction with the data structures whose size is determined dynamically.

9.3 Compilation Time

Whilst value-dependent types and compile-time expression provide a developer with a greater degree of flexibility, these extensions come at the cost of compilation time. This is unavoidable as expressions are evaluated during compilation, however we can measure the extent to which this impacts compilation times. For this measurement I present the time required to compile the earlier presented Floyd-Warshall benchmarks. I use these benchmarks due to the increasing dependence on value-dependent types, the `Array` benchmark uses very few value-dependent types (only for defining a run-method), the `Vector` benchmark uses only compile-time integers and the `Matrix` uses compile-time integers, objects and methods. To obtain the results in this section, each benchmark was compiled 100 times and the lowest time taken.

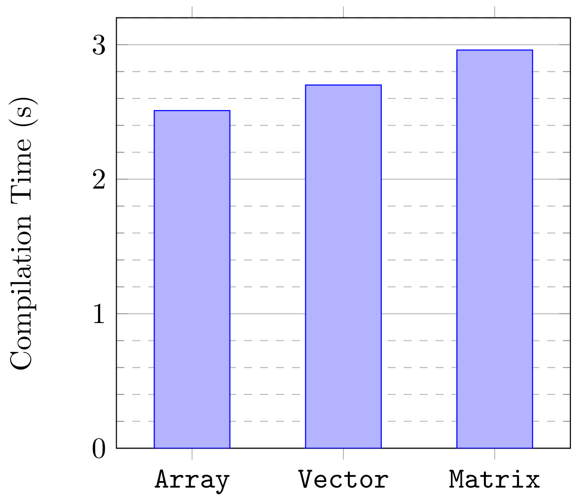

Figure 20: Compilation time for Floyd-Warshall benchmarks

In fig. 20, we can see the compilation times of each benchmark. Only the compilation times of the one-dimensional `Vector` and `Array` have been reproduced here to allow for easier comparison with the `Matrix` class. We can see that there is only a minor increase in compilation time between the `Array` and `Vector` benchmark.

We can observe from fig. 20 that we do not appear to suffer much overhead as a result of the extra evaluation at compile-time. Noting in particular that `Matrix` class, which heavily relies on compile-time expressions, suffers from only a 0.5 second increase in compilation time.

This is a very important result, we saw in section 9.2 that we can obtain better runtimes using `Vector`s instead of `Array`s. This result would be less useful if the compilation time required to achieve it was very high. Here we have shown this is not the case. In fact we only pay a small penalty in compilation for the increase in performance. Consider the one-time compilation cost against the multiple runtimes that could be improved using this feature. We can claim that we have developed the efficient containers using value-dependent types that we were aiming to create.

I have further compared the compilation times obtained using the unmodified Pony compiler and using the Ponyta compiler. For this comparison, I took a variation of the Floyd-Warshall `Array` benchmark and removed all value-dependent types. The results from this comparison can be seen in fig. 21.

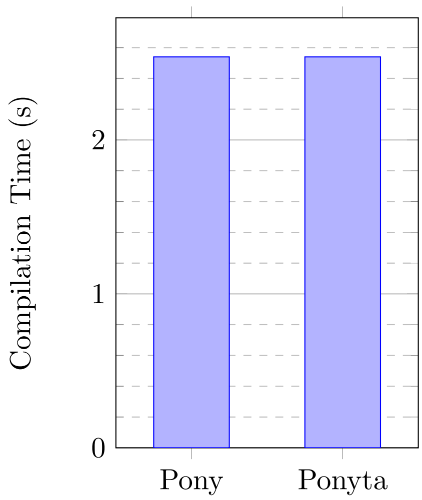

Figure 21: Compilation time for Pony and Ponyta compilers

In fig. 21, both compilers demonstrated a compilation time of 2.54 seconds. This result is due to compiling a fairly small program and so not much difference is visible. To account for this I replicated the body of the `getDistance()` method 560 times. The results from this comparison can be found in fig. 22.

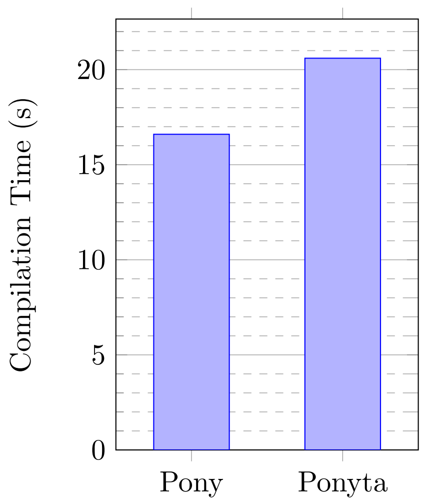

Figure 22: Compilation time for Pony and Ponyta compilers

A more noticeable distinction can be seen in fig. 22. The unmodified Pony compiler compiled the program 1.24 times faster than the Ponyta compiler. This result shows that the extra logic for value-dependent types has introduced an impact on compilation time. From this we can conclude that the implementation of value-dependent types should be reviewed to see where time can be saved during compilation. The most likely candidate for this is the following; during compilation all reachable methods are inspected for compile-time expressions so that they can be evaluated. We could instead mark a function in an earlier pass that has a compile-time expression and only inspect the definition of such functions.

9.4 Pony Developers Group

The work presented in this report has been shared with the Pony developer group who have had a positive reaction to the development. This has involved discussion with the group who responded with interest in using and incorporating the feature. Some response includes furthering the project by supporting value constraints which we will discuss further in section 11.4. The repository used for this project has also been made available to the Pony developer group, some of whom have marked their interest by starring the project.

Finally, there was some discussion around providing two access methods for the `Vector` API, one which allowed static verification of accesses to ensure the index was in bounds. This would be usable alongside the dynamic access verifications when we statically do not know the index being accessed.

9.5 Bug Fixes

During this project I was able to uncover and fix some bugs that existed in the Pony compiler. These bugs include inheriting from the intersection of two instances of the same trait, this would define ambiguous method inclusions. For example:

```
trait T1[A]
  fun foo() => Array[A]
class C2 is (T1[String] & T1[U32])
```

Other bugs were related constructing type constraints that resulted in infinite type checking. For example:

```
class C1[A: B, B: A]
class C2[A, B: A]
```

The above bugs were highlighted as a result of considering generics from a different perspective. The first bug regarding multiple trait inheritance is easier to see as a bug when one considers the instantiation to be able to change the definition of the body using a value. The second bug is easier to see when we first consider defining something as follows:

```
class C3[A, n: A]
```

Note, that this example successfully compiles. This experiment naturally progresses to testing with parametrising on types instead of values.

Another bug which was discovered through constructing a benchmark involved type aliases for lambda functions. Initially these crashed the compiler but were corrected. These type aliases however cannot use type parameters such as in the following:

```
type Sort[A] is {(Sortable[A]): Bool}
```

Finally, writing the API for `Vector` lead to realising some issues with alias types. These were discovered by Sylvan Clebsch upon reviewing my API. These bugs have either been reported to the Pony github repository[9, 13, 3] and waiting to be fixed, or I have fixed them as a part of this project. This is a positive effect of this project as it has helped improve the quality of the Pony compiler.

10 Conclusions

In this project I have extended the Pony compiler to support value-dependent types in Pony. The new version, Ponyta, allows developers to parametrise types on built-in as well as user-defined values, and offers a more flexible and more powerful type system to build programs.

This project involved understanding and altering many aspects of the Pony compiler. Almost all passes of the compiler required alteration and deep knowledge of their interactions to ensure features were implemented correctly. To give a sense of scale, this project resulted in adding nearly 4700 lines of new code to the Pony compiler, test suite and standard library. The size of the Pony compiler, test suite and standard library upon completion of this project was approximately 140,000 lines of code.

This project also involved a sophisticated design of the Vector and Matrix API (presented in sections 6 and 7). For these, I suggested new styles for using Pony's features to develop APIs; this style will be adopted in other, existing APIs.

Pony Interpreter

During this project I have developed an interpreter which is able to evaluate a substantial fragment of the Pony language as complex compile-time expressions. This fragment includes conditional expressions, while expressions, try expressions, errors, assignments, objects and method calls. Pony includes further expressions such as match expressions and for-loops over iterators which are currently not supported by the interpretor. Although Ponyta does not provide access to all Pony features, we have shown that Ponyta does provide a considerable degree of flexibility. The compile-time supported fragment of Pony has allowed us to construct some interesting classes which rely on compile-time expressions, namely the `Matrix` class.

Efficient Containers

Part of this project was to define efficient containers which used value-dependent types. We have been able to successfully define both the `Vector` and `Matrix` classes that we aimed to create. We have also shown that we managed to obtain improved performance from a `Vector` when compared with a `Array`. Therefore we have been able to construct an efficient, flexible data store during this project. Unfortunately, the `Matrix` data structure underperformed, this will be researched further to provide better behaviour.

Limitations

We have provided a more flexible type system for Pony developers. This type system still imposes some restrictions on developers, for example we cannot currently ensure equality of expressions in template definitions of classes and methods. This is a direction in which Ponyta could be explored further.

11 Future Work

Ponyta brings many new features to the Pony programming language. We will now discuss how these new features can be developed further.

11.1 Improving `Matrix`

As we discussed in section 9.2.2, the `Matrix` class did not produce the results that one would expect given the underlying representation. Further exploration of the `Matrix` class would be useful to understand and improve the performance of the `Matrix`. This may require more of an exploration into the representation of data together with an analysis of how to allow for better LLVM optimisations using this class.

11.2 Improving Compile-Time Expressions

The pseudo-interpreter produced in the project incorporates a reasonable about of the Pony language. This allows developer to execute much of their code at compile-time. The Pony programming language provides many features that have not been incorporated into the interpreter due to time constraints. This extension involves extending the interpreter to be able to evaluate more of the Pony language at compile-time.

Finally, as we discussed in section 9.3, the evaluation of compile-time expressions comes at the cost of an increased compilation time. Researching methods to reduce this cost would be worthwhile.

11.3 Mutable Compile-Time Objects

Consider the following expression:

```
1 class C1
2   var x: USize
3   new create(x': USize) => x = x'
4   fun ref update(x': USize) => x = x'
5
6 actor Main
7   new create(env: Env) =>
8     let c = #(C1(2).update(32))
9     #(c.update(41)) // fails compilation as c is of type C1 val
```

In the compile-time expression at line 8 a `C1` object is created and then its field updated through the `update()` method. Notice that all the information regarding the creation and update of the `C1` object is contained within the compile-time expression and it is not until we leave the compile-time expression that we require the object to be of capability `val`. The `val` capability of `c` means that the expression at line 9 will still cause compilation to fail due to the incorrect capability to call update. Also note that we would not be able to use the `C1` object as a type parameter without first recovering the object to capability `val` due to the restriction that all type arguments must be compile-time expressions recoverable to `val`.

This extension, which allows the use of `ref` objects within compile-time expressions, requires work to be able to update the the state of internal objects. Allowing `ref` objects also means that caching results requires more care when handling the expressions such as the following:

```
#(let v = {1,2,3}.values(); v.next(); v.next())
```

In the above expression, it is necessary that `v.next()` returns a different result on each call. This would perhaps require caching results of method calls based on an objects current state.

11.4 Constraints for Value Parameters

As can be seen throughout this report, in Pony it is possible to provide an upper-bound (or constraint) to a type parameter. These type constraints can be seen in the following example:

```
class C1[A: Number]
```

Here the type parameter `A` is constrained to be a subtype of `Number`. In much the same way it would be useful and provide an even richer type system if we could also provide some predicate constraints for value parameters that must be met for an instantiation to be valid. Consider the following example:

```
class C2[n: {s: USize | s > 10}]
```

This example adopts the syntax for constraints on index types presented in [20]. Here we define a class `C2` which depends on a `USize` which is greater than `10`. Now consider the two following instantiations:

```
1 let x = C2[2]
2 let y = C2[37] // This instantiation will fail
```

The assignment at line 2 should fail as the type argument `37` does not satisfy the constraint `{s: USize | s > 10}`. This could be extended to arbitrary predicates which operate on any type provided the predicate could be evaluated statically. This notation, essentially, denotes a constraint of the union of all types which satisfy the predicate.

A suggestion made by a Pony developer was that we could go on to develop the `Vector` API using value constraints as follows:

```
class Vector[A, _size: USize]
  ... // API so far
  fun apply_static[i: {s: USize | s < _size}]() =>
    _apply(i)
```

Noting here that the `apply_static()` method is not partial as in the dynamic `apply()` method. This method is not partial as we can only instantiate the method with values which are guaranteed by the compiler to be within the bounds of the `Vector`.

11.5 Template Specialisation

We discussed template specialisation in section 9.1.1, we now go on to consider it as an extension to Ponyta. Consider the following example:

```
actor Main
  fun fac[n: USize](): USize =>
    if n < 2 then
      1
    else
      n * fac[#(n - 1)]
    end

  new create(env: Env) =>
    let x = fac[10]
```

The above definition of `fac()` will result in constructing many definitions of `fac()`. Notice that for any instantiated definition of `fac[# n]()`, we must also create the instantiated method `fac[# (n - 1)]()`. The compiler does not know that when `n < 2` then `fac[# (n - 1)]` will not be called. Therefore the compiler will create the reified definition of `fac[# (n - 1)]`. These reifications could go on until all values of `USize` have been used to instantiate `fac()`.

It would be useful to allow developers to define specialised definitions of a template for given values. Providing template specialisation would help developers avoid the infinite template instantiation problem described above. We could then perhaps define the `fac` function as follows:

```
actor Main
  fun fac[0](): USize => 1
  fun fac[1](): USize => 1
  fun fac[n: USize](): USize => n * fac[#(n - 1)]

  new create(env: Env) =>
    let x = fac[10]
```

Here we provide the definitions of `fac[0]()` and `fac[1]()` upfront. In such a case, the compiler would not have to create reified definitions of theses two cases. The compiler would use the definitions provided by the developer. We could perhaps go on to combine this with value-dependent types with constraints and write a definition of `fac()` as follows:

```
actor Main
  fun fac[n: {s: USize | s < 2}](): USize => n
  fun fac[n: USize](): USize => n * fac[#(n - 1)]
```

This can be compared to the case methods which are available in Pony. An example of these case methods is:

```
actor Main
  fun fac(n: USize): USize if n < 2 => n
  fun fac(n: USize): USize => n * fac(n - 1)
```

At runtime the patterns are exhaustively checked and the matching method selected (or `None` is returned if no pattern matches). An equivalent approach could be used for instantiating specialised definitions; raising a compilation error if no pattern matches.

11.6 Formal Model

An interesting piece of future work would be to extend the Pony formal model to incorporate value-dependent types. There is much literature on dependent types, it would be interesting to combine this with the novel type system presented in the Pony formal model.

12 References

[1] C++ concepts: Literaltype. http://en.cppreference.com/w/cpp/concept/LiteralType. Accessed January 25, 2016.

[2] Data.matrix - basic matrix operations with dimensionalities enforced at the type level. http://www.idris-lang.org/docs/0.10/contrib_doc/docs/Data.Matrix.html. Accesses June 11, 2016.

[3] Generic type causes compiler to not return. https://github.com/ponylang/ponyc/issues/918. Accessed June 08, 2016.

[4] Idris: A language with dependent types. http://www.idris-lang.org/. Accessed January 18, 2016.

[5] The LLVM compiler infrastructure. http://llvm.org/. Accessed January 18, 2016.

[6] Llvm language reference manual. http://llvm.org/docs/LangRef.html. Accessed May 16, 2016.

[7] Llvm's analysis and transform passes. http://llvm.org/docs/Passes.html. Accessed June 07, 2016.

[8] Pony tutorial. http://tutorial.ponylang.org/. Accessed January 17, 2016.

[9] Segfaulting type checker. https://github.com/ponylang/ponyc/issues/520. Accessed June 08, 2016.

[10] std::array vs std::vector subtle difference. http://stackoverflow.com/questions/16380740/stdarray-vs-stdvector-subtle-difference. Accessed May 11, 2016.

[11] Storage class specifiers. http://en.cppreference.com/w/cpp/language/storage_duration. Accessed May 13, 2016.

[12] Template parameters and template arguments. http://en.cppreference.com/w/cpp/language/template_parameters. Accessed January 18, 2016.

[13] Type parameters in type aliases aren't usable in lambdas. https://github.com/ponylang/ponyc/issues/919. Accessed June 08, 2016.

[14] Variances. http://docs.scala-lang.org/tutorials/tour/variances.html. Accessed January 23, 2016.

[15] Gul Agha and Carl Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, 1987.

[16] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. *Manuscript, available online*, page 235, 2005.

[17] Ana Bove and Peter Dybjer. Dependent types at work. In *Language engineering and rigorous software development*, pages 57–99. Springer, 2009.

[18] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[19] Joana Campos and Vasco T Vasconcelos. Indexed types in object-oriented programming.

[20] Joana Campos and Vasco T Vasconcelos. Imperative objects with dependent types. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, page 2. ACM, 2015.

[21] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM, 1989.

[22] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[23] Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. Ownership and reference counting based garbage collection in the actor world.

[24] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015.

[25] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[26] Andrew J Kennedy and Benjamin C Pierce. On decidability of nominal subtyping with variance. 2006.

[27] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. *ECOOP 2008–Object-Oriented Programming*, pages 260–284, 2008.

[28] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[29] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.

Appendices

A Linked List Vector

The following describes how to construct a `Vector` in C++ as linked list and where the elements of the `Vector` are accessible at compile time.

```
#include <type_traits>

template<typename T, int size>
struct Vector
{
  const T elem;
  const Vector<T, size-1> next;

  template<typename...TS>
  constexpr Vector(T elem, TS...ts) :
    elem(elem), next(ts...) {}

  constexpr Vector(T elem, Vector<T, size-1> next) :
    elem(elem), next(next) {}

  template<int n>
  constexpr T get() const;
};

template<typename T>
struct Vector<T, 0> {};

template<typename T, int size, int n, typename = void>
struct _get_helper;

template<typename T, int size, int n>
struct _get_helper<T, size, n, std::enable_if_t<(n < 0 || n >= size)>>
{
  static constexpr T get(Vector<T, size> v) {
    static_assert(0 <= n && n < size,
                  "index must be less than dimensions of vector");
    return 0;
  }
};

template<typename T, int size, int n>
struct _get_helper<T, size, n, std::enable_if_t<(n < size && n > 0)>>
{
  static constexpr T get(Vector<T, size> v) {
    return _get_helper<T, size-1, n-1>::get(v.next);
  }
};

template<typename T, int size>
```

```
struct _get_helper<T, size, 0, std::enable_if_t<(size > 0)>>
{
  static constexpr T get(Vector<T, size> v) {
    return v.elem;
  }
};

template<typename T, int size>
template<int n>
constexpr T Vector<T, size>::get() const
{
  return _get_helper<T, size, n>::get(*this);
}

int main() {
  constexpr Vector<int, 3> v1(1, 2, 3);
  constexpr Vector<int, 5> v2(7, 9, 8, 2, 1);

  constexpr int x = v2.get<2>();
  // Invalid, will fail at compile time
  v1.get<x>();
}
```

Each `Vector` has a value `elem` and a next node `next`. When we construct a `Vector` we pass in variable number of values of type `T`. The argument gets destructed to instantiate the `elem` with the first value and then constructing a `Vector` (whose size is one less than the current vector) with the remaining arguments. Compare this to the inductive definition we had in Idris.

The `get` method is implemented through templated `_get_helper`s which are enabled as appropriate through the meta-programming construct `std::enable_if_t`. We specify a base case, when `n` is `0` and `size` is greater than `0`. We also specify the inductive case when `n` is within the bounds of the vector dimensions; in this case we recurse and `get` with the `next` node, decrementing the `size` and `n`. We also define the error case for when `n` is not within the bounds of the vector.

B Benchmarking Sorting Sequences

```
use "random"
use "collections"
use "time"

actor Sorter[dim: USize]
  fun _sort(s: Sortable[U32], f: {(Sortable[U32]) ?} val,
            total_itrs: U64): U64 ? =>
    var j: U64 = 0
    var total: U64 = 0
    while (j = j + 1) < total_itrs do
```

```
      var time = Time.nanos()
      f(s)
      time = Time.nanos() - time
      total = total + time
    end
    total / total_itrs

  be sort(ret: Main, f: {(Sortable[U32]) ?} val, name: String,
          total_itrs: U64) =>
    try
      ret.complete(
        var g = lambda ref(i: USize)(rand=MT): U32 => rand.u32() end
        _sort(Vector[U32, #dim].generate(g), f, total_itrs),
        g = lambda ref(x: USize)(rand=MT): U32 => rand.u32() end
        _sort(Array[U32].generate(g, dim), f, total_itrs),
        name, dim
      )
    else
      ret.failed()
    end

actor Main
  let env: Env
  let results_map: Map[String, Map[USize, (U64, U64)]]
    = Map[String, Map[USize, (U64, U64)]]
  var runners: USize = 18 * 6 // number of dims * number of sorts
  var fail: Bool = false

  be complete(vec_time: U64, array_time: U64, name: String, size: USize)
     =>
    try
      results_map(name).update(size, (vec_time, array_time))
    else
      failed()
    end
    runners = runners - 1
    if runners == 0 then
      display_results()
    end

  be failed() =>
    fail = true

  be display_results() =>
    if fail then
      env.err.print("Failed")
      return
    end
    var line = String.append("Sort | Size | Vector Time | Array Time")
    env.out.print(line.string())
    line = String.append("---------------------------------------")
    env.out.print(line.string())
```

```
    for (name, map) in results_map.pairs() do
      var i: USize = 0
      while i < map.size() do
        let dim =  2 << i
        try
          (let v_time, let a_time) = map(dim)
          env.out.print(String.append(name).append(" | ")
                                    .append(dim.string()).append(" | ")
                                    .append(v_time.string()).append(" | ")
                                    .append(a_time.string()).string())
        else
          failed()
        end
        i = i + 1
      end
    end

  fun ref init() =>
    let sorts
      = ["quick", "heap", "merge", "insertion", "selection", "bubble"]
    for name in sorts.values() do
      results_map.update(name, Map[USize, (U64, U64)])
    end

  new create(env': Env) =>
    env = env'

    init()

    let sorts = Array[({(Sortable[U32]) ?} val, String)]
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].quick_sort(s) end,
       "quick"))
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].heap_sort(s) end,
       "heap"))
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].merge_sort(s) end,
       "merge"))
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].insertion_sort(s) end,
       "insertion"))
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].selection_sort(s) end,
       "selection"))
    sorts.push(
      (lambda (s: Sortable[U32]) ? => Sort[U32].bubble_sort(s) end,
       "bubble"))

    let total_itrs: U64 = 20
    for (sort, name) in sorts.values() do
      Sorter[2].sort(this, sort, name, total_itrs)
```

```
    Sorter[4].sort(this, sort, name, total_itrs)
    Sorter[8].sort(this, sort, name, total_itrs)
    Sorter[16].sort(this, sort, name, total_itrs)
    Sorter[32].sort(this, sort, name, total_itrs)
    Sorter[64].sort(this, sort, name, total_itrs)
    Sorter[128].sort(this, sort, name, total_itrs)
    Sorter[256].sort(this, sort, name, total_itrs)
    Sorter[512].sort(this, sort, name, total_itrs)
    Sorter[1024].sort(this, sort, name, total_itrs)
    Sorter[2048].sort(this, sort, name, total_itrs)
    Sorter[4096].sort(this, sort, name, total_itrs)
    Sorter[8192].sort(this, sort, name, total_itrs)
    Sorter[16384].sort(this, sort, name, total_itrs)
    Sorter[32768].sort(this, sort, name, total_itrs)
    Sorter[65536].sort(this, sort, name, total_itrs)
    Sorter[131072].sort(this, sort, name, total_itrs)
    Sorter[262144].sort(this, sort, name, total_itrs)
  end
```

C Benchmarking Nested Data Structures

```
use "collections"
use "time"

actor ArrayActor[dim: USize]
  fun create_array(): Array[Array[USize]] ? =>
    let x: USize = 2 // FIXME: just for the capture
    Array[Array[USize]].generate(
      lambda (i: USize)(x): Array[USize]^ ? =>
        Array[USize].generate(
        lambda (j: USize)(i): USize^ =>
          (i * dim) + j
        end, dim)
      end, dim)

  fun update_array(a: Array[Array[USize]]) ? =>
    var i: USize = 0
    while i < a(0).size() do
      var j: USize = 0
      while j < a.size() do
        a(j).update(i, a(j)(i) + 1)
        j = j + 1
      end
      i = i + 1
    end

  be run(ret: Main tag, total_itrs: U64, name: String) =>
    try
      var array_time: U64 = 0
      var itr: U64 = 0
      let a: Array[Array[USize]] = create_array()
```

```
      while (itr = itr + 1) < total_itrs do
        var time = Time.nanos()
        update_array(a)
        time = Time.nanos() - time
        array_time = array_time + time
      end
      array_time = array_time / total_itrs
      ret.complete(array_time, name)
    else
      ret.failed()
    end

actor VectorActor[dim: USize]
  fun create_vector(): Vector[Vector[USize, # dim], # dim] ? =>
    let x: USize = 2 // FIXME: just for the capture
    Vector[Vector[USize, # dim], # dim].generate(
      lambda (i: USize)(x): Vector[USize, # dim]^ ? =>
        Vector[USize, # dim].generate(
        lambda (j: USize)(i): USize^ =>
          USize((i * dim) + j)
        end)
      end)

  fun update_vector(v: Vector[Vector[USize, # dim], # dim]) ? =>
    var i: USize = 0
    while i < v(0).size() do
      var j: USize = 0
      while j < v.size() do
        v(j).update(i, v(j)(i) + 1)
        j = j + 1
      end
      i = i + 1
    end

  be run(ret: Main tag, total_itrs: U64, name: String) =>
    try
      var vector_time: U64 = 0
      var itr: U64 = 0
      //FIXME: type inference is messed up on value dependent types
      let v: Vector[Vector[USize, #dim], #dim] = create_vector()
      while (itr = itr + 1) < total_itrs do
        var time = Time.nanos()
        update_vector(v)
        time = Time.nanos() - time
        vector_time = vector_time + time
      end
      vector_time = vector_time / total_itrs
      ret.complete(vector_time, name)
    else
      ret.failed()
    end
```

```
actor Main
  var runners: U64 = 2
  let runs: Map[String, U64] = Map[String, U64]
  let env: Env

  be failed() =>
    env.err.print("Error!")

  be complete(time: U64, name: String) =>
    runs.update(name, time)
    runners = runners - 1
    if runners == 0 then
      print_results()
    end

  be print_results() =>
    for (key, value) in runs.pairs() do
      let s = String.append(key).append(": ").append(value.string()).
          string()
      env.out.print(s)
    end

  new create(env': Env) =>
    env = env'
    let size: USize = # 2048
    let v_out = String.append("Vector | ").append(size.string()).string()
    let a_out = String.append("Array | ").append(size.string()).string()
    VectorActor[#size].run(this, 100, v_out)
    ArrayActor[#size].run(this, 100,  a_out)
```

D Vector Trace Test

```
use "ponytest"

class iso _TestVectorTrace is UnitTest
  """
  Test val trace optimisation
  """
  fun name(): String => "builtin/Vectortrace"

  fun apply(h: TestHelper) =>
    _VectorTrace.one(h)
    h.long_test(2_000_000_000) // 2 second timeout


actor _VectorTrace
  be one(h: TestHelper) =>
    @pony_triggergc[None](this)
    let s1 = recover String.append("wombat") end
    let s2 = recover String.append("aardvark") end
    let s3 = recover String.append("meerkat") end
```

```
    _VectorTrace.two(h, consume s1, consume s2, consume s3)

  be two(h: TestHelper, s1: String, s2: String, s3: String) =>
    @pony_triggergc[None](this)
    try
      let v = recover Vector[String, 3].init([s1, s2, s3].values()) end
      _VectorTrace.three(h, consume v)
    else
      h.fail("constructing vector failed")
    end

  be three(h: TestHelper, v: Vector[String, 3] iso) =>
    @pony_triggergc[None](this)
    try
      h.assert_eq[String]("wombat", v(0))
      h.assert_eq[String]("aardvark", v(1))
      h.assert_eq[String]("meerkat", v(2))
    else
      h.fail("access to vector failed")
    end
    h.complete(true)
```

E Floyd-Warshall One-Dimensional Array

```
use "collections"
use "random"
use "time"

actor GraphActor
  fun getDistance[n: USize](dp: Array[USize]): Array[USize] ? =>
    for k in Range(0, n) do
      for i in Range(0, n) do
        for j in Range(0, n) do
          let a = (i * n) + j
          let b = (i * n) + k
          let c = (k * n) + j
          dp.update(a, (dp(b) + dp(c)).min(dp(a)))
        end
      end
    end
    dp

  be run[n: USize](ret: Main tag) =>
  try
    let dp = Array[USize].generate(
      lambda ref(i:USize)(rand=MT): USize =>
        rand.next().usize()
      end, n * n)

    var time = Time.nanos()
    getDistance[#n](dp)
```

```
    time = Time.nanos() - time
    ret.complete[#n](time)
  else
    ret.failed()
  end

actor Main
  let env: Env
  let ga: GraphActor = GraphActor
  let runs: U64 = 100
  let results: Map[USize, (U64, U64)] = Map[USize, (U64, U64)]

  be complete[n: USize](time: U64) =>
    try
      (var finished, var total_time) = results(n)
      finished = finished + 1
      total_time = total_time + time
      if finished == runs then
        total_time = total_time / runs
        env.out.print(total_time.string())
        return
      end
      results.update(n, (finished, total_time))
      ga.run[#n](this)
    else
      failed()
    end

  be failed() =>
    env.err.print("Failed")

  new create(env': Env) =>
    env = env'
    try
      for i in Range(0, 12) do
        results.insert(2 << i, (U64(0), U64(0)))
      end
      ga.run[#(2 << 0)](this)
      ga.run[#(2 << 1)](this)
      ga.run[#(2 << 2)](this)
      ga.run[#(2 << 3)](this)
      ga.run[#(2 << 4)](this)
      ga.run[#(2 << 5)](this)
      ga.run[#(2 << 6)](this)
      ga.run[#(2 << 7)](this)
      ga.run[#(2 << 8)](this)
      ga.run[#(2 << 9)](this)
      ga.run[#(2 << 10)](this)
      ga.run[#(2 << 11)](this)
    else
      failed()
    end
```

F Floyd-Warshall Two-Dimensional Array

```
use "collections"
use "random"
use "time"

actor GraphActor
  fun getDistance[n: USize](dp: Array[Array[USize]]) ? =>
    for k in Range(0, n) do
      for i in Range(0, n) do
        for j in Range(0, n) do
          dp(i).update(j, (dp(i)(k) + dp(k)(j)).min(dp(i)(j)))
        end
      end
    end
    dp

  be run[n: USize](ret: Main tag) =>
  try
    let dp = Array[Array[USize]].generate(
      lambda ref(i:USize)(rand=MT): Array[USize] ? =>
        Array[USize].generate(
          lambda ref(j:USize)(rand): USize =>
            rand.next().usize()
          end, n)
      end, n)

    var time = Time.nanos()
    getDistance[#n](dp)
    time = Time.nanos() - time
    ret.complete[#n](time)
  else
    ret.failed()
  end

actor Main
  let env: Env
  let ga: GraphActor = GraphActor
  let runs: U64 = 100
  let results: Map[USize, (U64, U64)] = Map[USize, (U64, U64)]

  be complete[n: USize](time: U64) =>
    try
      (var finished, var total_time) = results(n)
      finished = finished + 1
      total_time = total_time + time
      if finished == runs then
        total_time = total_time / runs
        env.out.print(total_time.string())
        return
      end
```

```
      results.update(n, (finished, total_time))
      ga.run[#n](this)
    else
      failed()
    end

  be failed() =>
    env.err.print("Failed")

  new create(env': Env) =>
    env = env'
    try
      for i in Range(0, 12) do
        results.insert(2 << i, (U64(0), U64(0)))
      end
      ga.run[#(2 << 0)](this)
      ga.run[#(2 << 1)](this)
      ga.run[#(2 << 2)](this)
      ga.run[#(2 << 3)](this)
      ga.run[#(2 << 4)](this)
      ga.run[#(2 << 5)](this)
      ga.run[#(2 << 6)](this)
      ga.run[#(2 << 7)](this)
      ga.run[#(2 << 8)](this)
      ga.run[#(2 << 9)](this)
      ga.run[#(2 << 10)](this)
      ga.run[#(2 << 11)](this)
    else
      failed()
    end
```

G Floyd-Warshall One-Dimensional Vector

```
use "collections"
use "random"
use "time"

actor GraphActor
  fun getDistance[m: USize](dp: Vector[USize, #(m * m)]) ? =>
    for k in Range(0, m) do
      for i in Range(0, m) do
        for j in Range(0, m) do
          let a = (i * m) + j
          let b = (i * m) + k
          let c = (k * m) + j
          dp.update(a, (dp(b) + dp(c)).min(dp(a)))
        end
      end
    end

  be run[n: USize](ret: Main tag) =>
```

```
  try
    let dp = Vector[USize, #(n * n)].generate(
      lambda ref(i:USize)(rand=MT): USize =>
        rand.next().usize()
      end)

    var time = Time.nanos()
    getDistance[#n](dp)
    time = Time.nanos() - time
    ret.complete[#n](time)
  else
    ret.failed()
  end

actor Main
  let env: Env
  let ga: GraphActor = GraphActor
  let runs: U64 = 100
  let results: Map[USize, (U64, U64)] = Map[USize, (U64, U64)]

  be complete[n: USize](time: U64) =>
    try
      (var finished, var total_time) = results(n)
      finished = finished + 1
      total_time = total_time + time
      if finished == runs then
        total_time = total_time / runs
        env.out.print(total_time.string())
        return
      end
      results.update(n, (finished, total_time))
      ga.run[#n](this)
    else
      failed()
    end

  be failed() =>
    env.err.print("Failed")

  new create(env': Env) =>
    env = env'
    try
      for i in Range(0, 12) do
        results.insert(2 << i, (U64(0), U64(0)))
      end
      ga.run[#(2 << 0)](this)
      ga.run[#(2 << 1)](this)
      ga.run[#(2 << 2)](this)
      ga.run[#(2 << 3)](this)
      ga.run[#(2 << 4)](this)
      ga.run[#(2 << 5)](this)
      ga.run[#(2 << 6)](this)
```

```
      ga.run[#(2 << 7)](this)
      ga.run[#(2 << 8)](this)
      ga.run[#(2 << 9)](this)
      ga.run[#(2 << 10)](this)
      ga.run[#(2 << 11)](this)
    else
      failed()
    end
```

H Floyd-Warshall Two-Dimensional Vector

```
use "collections"
use "random"
use "time"

actor GraphActor
  fun getDistance[n: USize](dp: Vector[Vector[USize, #n], #n]) ? =>
    for k in Range(0, n) do
      for i in Range(0, n) do
        for j in Range(0, n) do
          dp(i).update(j, (dp(i)(k) + dp(k)(j)).min(dp(i)(j)))
        end
      end
    end

  be run[n: USize](ret: Main tag) =>
  try
    let dp = Vector[Vector[USize, #n], #n].generate(
      lambda ref(i:USize)(rand=MT): Vector[USize, #n] ? =>
        Vector[USize, #n].generate(
          lambda ref(j:USize)(rand): USize =>
            rand.next().usize()
          end)
      end)

    var time = Time.nanos()
    getDistance[#n](dp)
    time = Time.nanos() - time
    ret.complete[#n](time)
  else
    ret.failed()
  end

actor Main
  let env: Env
  let ga: GraphActor = GraphActor
  let runs: U64 = 100
  let results: Map[USize, (U64, U64)] = Map[USize, (U64, U64)]

  be print(s: String) =>
    env.out.print(s)
```

```

  be complete[n: USize](time: U64) =>
    try
      (var finished, var total_time) = results(n)
      finished = finished + 1
      total_time = total_time + time
      if finished == runs then
        total_time = total_time / runs
        env.out.print(total_time.string())
        return
      end
      results.update(n, (finished, total_time))
      ga.run[#n](this)
    else
      failed()
    end

  be failed() =>
    env.err.print("Failed")

  new create(env': Env) =>
    env = env'
    try
      for i in Range(0, 12) do
        results.insert(2 << i, (U64(0), U64(0)))
      end
      ga.run[#(2 << 0)](this)
      ga.run[#(2 << 1)](this)
      ga.run[#(2 << 2)](this)
      ga.run[#(2 << 3)](this)
      ga.run[#(2 << 4)](this)
      ga.run[#(2 << 5)](this)
      ga.run[#(2 << 6)](this)
      ga.run[#(2 << 7)](this)
      ga.run[#(2 << 8)](this)
      ga.run[#(2 << 9)](this)
      ga.run[#(2 << 10)](this)
      ga.run[#(2 << 11)](this)
    else
      failed()
    end
```

I Floyd-Warshall Matrix

```
use "random"
use "collections"
use "time"

actor GraphActor
  fun getDistance[n: USize](dp: Matrix[USize, 2, # {n, n}]) ? =>
    for k in Range(0, n) do
```

```
      for i in Range(0, n) do
        for j in Range(0, n) do
          dp({i, j}) = (dp({i, k}) + dp({k, j})).min(dp({i, j}))
        end
      end
    end

  be run[n: USize](ret: Main tag) =>
  try
    let dp = Matrix[USize, 2, # {n, n}].generate(
      lambda ref(i: USize)(rand=MT): USize =>
        rand.next().usize()
      end
    )

    var time = Time.nanos()
    getDistance[#n](dp)
    time = Time.nanos() - time
    ret.complete[#n](time)
  else
    ret.failed()
  end

actor Main
  let env: Env
  let ga: GraphActor = GraphActor
  let runs: U64 = 100
  let results: Map[USize, (U64, U64)] = Map[USize, (U64, U64)]

  be complete[n: USize](time: U64) =>
    try
      (var finished, var total_time) = results(n)
      finished = finished + 1
      total_time = total_time + time
      if finished == runs then
        total_time = total_time / runs
        env.out.print((n * n).string() + ": " + total_time.string())
        return
      end
      results.update(n, (finished, total_time))
      ga.run[# n](this)
    else
      failed()
    end

  be failed() =>
    env.err.print("Failed")

  new create(env': Env) =>
    env = env'
    try
      for i in Range(0, 12) do
```

```
        results.insert(2 << i, (U64(0), U64(0)))
      end
      ga.run[#(2 << 0)](this)
      ga.run[#(2 << 1)](this)
      ga.run[#(2 << 2)](this)
      ga.run[#(2 << 3)](this)
      ga.run[#(2 << 4)](this)
      ga.run[#(2 << 5)](this)
      ga.run[#(2 << 6)](this)
      ga.run[#(2 << 7)](this)
      ga.run[#(2 << 8)](this)
      ga.run[#(2 << 9)](this)
      ga.run[#(2 << 10)](this)
      ga.run[#(2 << 11)](this)
    else
      failed()
    end
```

J Sorting Benchmark Results

Number of Elements	Array Time (ns)	Vector Time (ns)	Speed-up
512	19598	18368	1.07
1024	42101	38897	1.08
2048	74433	70433	1.06
4096	161410	151172	1.07
8192	256574	567521	0.45
16384	538240	574021	0.94
32768	1145006	1363374	0.84
65536	2426292	2513981	0.97
131072	5421476	5349500	1.01
262144	12438516	11235882	1.11
		Average Speed-up	0.96

Table 2: Quicksort Results

Number of Elements	Array Time (ns)	Vector Time (ns)	Speed-up
512	13086	13097	1.00
1024	63732	53517	1.19
2048	84533	78581	1.08
4096	358713	322019	1.11
8192	546127	481698	1.13
16384	3459813	3383352	1.02
32768	17035071	16153593	1.05
65536	20871251	19122859	1.09
131072	45455351	41612666	1.09
262144	613874697	583460529	1.05
		Average Speed-up	1.08

Table 3: Mergesort Results

Number of Elements	Array Time (ns)	Vector Time (ns)	Speed-up
512	371906	366447	1.01
1024	1079171	1370761	0.79
2048	4271018	4630770	0.92
4096	18939902	17416206	1.09
8192	82978200	73742683	1.13
16384	337927538	314694614	1.07
32768	1345116348	1246506808	1.08
65536	5319004868	5273600274	1.01
131072	21021372053	21012888861	1.00
262144	84161782776	84505306957	1.00
		Average Speed-up	1.01

Table 4: Insertion Sort Results

Number of Elements	Array Time (ns)	Vector Time (ns)	Speed-up
512	71506	68359	1.05
1024	340598	246364	1.38
2048	1099031	1006907	1.09
4096	3848380	4184625	0.92
8192	15478134	14618474	1.06
16384	68931104	65160613	1.06
32768	275607046	256874318	1.07
65536	1068993529	1049785698	1.02
131072	4403358398	4318378983	1.02
262144	17564798129	17399321766	1.01
		Average Speed-up	1.07

Table 5: Bubble Sort Results

K Nested Benchmark Results

Number of Elements	Time per Array Element (ns)	Time per Vector Element (ns)	Speed-up
4	16.75	12.75	1.31
16	8.13	3.75	2.17
64	4.97	2.25	2.21
256	4.88	1.93	2.54
1024	3.95	2.38	1.66
4096	8.10	6.10	1.33
16384	5.60	5.49	1.02
65536	3.95	3.74	1.05
262144	4.22	3.41	1.24
1048576	5.25	4.75	1.10
4194304	6.13	6.13	1.00
16777216	8.33	8.33	1.00
67108864	16.91	11.27	1.50
268435456	28.75	20.11	1.43
		Average Speed-up	1.47

Table 6: Nested Benchmark Results

L Floyd-Warshall Benchmark

Number of Elements	1D Array Time (ns)	2D Array Time (ns)	1D Vector Time (ns)	2D Vector Time (ns)	Matrix Time (ns)
4	485	383	254	474	911
16	660	668	586	693	4519
64	2633	2594	2308	2136	31588
256	11630	14785	10354	10789	245435
1024	70119	99306	61353	64138	1902437
4096	520182	781953	443619	464513	14876344
16384	3566970	5581209	2994001	3033874	119047260
65536	25562820	42503486	21788826	20994411	941635128
262144	197582482	333438148	168388445	160383423	-
1048576	1590107000	2686352289	1230765172	1488617687	-
4194304	13264552889	22098325496	11207657062	11371675433	-
16777216	104326614037	174120470468	86764696659	88472051624	-

Table 7: Floyd-Warshall Benchmark Results