A Survey on Abstract Data Types

Anthony Hunt

February 9, 2025

Contents

1 Introduction

From the categorization of species to the periodic table of elements, the organization and manipulation of data is integral to all aspects of science and

technology. Although the collection of data is not a new field of research by any means, the advent of computerized systems, with its ability to transfer and use massive amounts of data in mere seconds, necessitates understandable and sensible organization structures.

Over the past 60 years, as computational resource restrictions have eased dramatically, programming languages and software creation tools have evolved with increasingly better methods of dealing with data. By viewing pieces of data through specific lenses, programmers can then write algorithms that operate on any data matching the shape of those lenses, rather than the data itself. These lenses are more commonly referred to as data types, which assign a type to a piece of data. As we will see later in this paper, further abstractions over data types themselves enable extremely concise and readable code, both for the programmer and computer.

The rest of this article will explore data type concepts from ¡X¿ languages, structures to handle errors that might appear, and a small review of ergonomics and usability of those data types. A necessary discussion of various type systems will naturally evolve as a precursor to data types themselves.

2 Data Types in ¡X¿ Languages

Abstract Data Types (ADTs) can look quite different in many languages, but often share the same principles and rules that make it understandable for programmers. In its essential form, an ADT is a collection of data together with operations that can be carried out on that data [22]. The key differences between ADTs from different languages are the conventions used to collect data and the power operations have over collected data.

In popular object-oriented programming (OOP) languages, ADTs are approached with a "class" mindset, where programmers specify a blueprint for objects that will eventually be used in a program. For better code reuse, classes can be built from pre-existing classes through the use of inheritance, a term used to express that a new class inherits all the properties of the old class. Objects created by classes are referred to as instances of the class and often hold some amount of mutable data, with methods that have full access to the data. The methods can choose to make use of "side effects", referring to the modification of data not explicitly passed through regular function arguments or return values. Side effects in methods have the consequence that the output of a method may change independent of the values passed through its arguments. In other words, the object a method has access to may contain some state that influences the outcome of the method.

At the opposite end of the programming world lies purely functional languages. ADTs in these languages are more often referred to as data types rather than classes and are only used for collecting and operating on data. "Objects" created from these data types are immutable and methods are instead replaced with pure functions that explicitly take in the relevant data type as input. The operators attached to a functionally pure ADT are functionally pure themselves,

meaning they cannot make use of any hidden state and therefore cannot have side effects.

2.1 Python

As a dynamic interpreted scripting language with thousands of high quality libraries and a short learning curve, Python has become one of the most dominant modern programming languages across several subdomains of computing. The language itself is characterized mainly by its simple approach to writing readable, practical, and user-friendly code [14], used by many as a "glue" to easily join high performance computing libraries. The role of Python as a mediator between optimized C libraries and human understanding, especially in the machine learning space, is adjacent to the role of documentation in expressing the intention and meaning behind code, rather than focusing on the small details of the code itself.

2.1.1 Python and Types

The overarching data model behind Python is that everything is either an object or a relation between objects. Almost all objects in Python are used by reference, and each object contains an identifier (memory address), an immutable type, and then the actual value itself. In contrast to other scripting languages (namely JavaScript), an immutable type means that Python will never implicitly recast the type of an object to fit the expected types of functions or methods (ie. it is strongly typed).

Although Python is strongly typed, its lack of a static typing system serves as a double edged sword for readability and reliability. With small programs, some developers may prefer to just "get things done", making use of whatever combination of builtin types are available. A mixture of sets, dictionaries, and lists with unrestricted element types are all too common in these types of scripts. However, assigning proper data types to work with various pieces of information is essential to the longevity and maintenance of large scale projects. Since the introduction of optional static type hints via a library in Python 3.5 and increased support through Python 3.8, a 2022 survey [4] found a steady increase in the number of GitHub repositories making use of static type hints.

Aside from granting linters more information to statically analyze the program and warn developers of possible errors, types also provide programmers themselves with useful information regarding their program. As reinforced by the philosophy of TypeScript as a type-hinted version of JavaScript, data types serve as an important channel of communication between programmers. An additional finding of the 2022 survey [4] showed that typecheck errors in the linters did not prevent a large amount of code from being pushed, but were still found to be useful outside of the linter.

Fundamentally, Python places little restriction on the programmer's actions. For example, while data privacy within Python objects is theoretically possible through name mangled fields (beginning with double-underscores), there is no

3

real way to completely restrict access to an object's fields so long as that object exists in the code. Access to deeper implementation details, like a list of current variables, list of fields in any variable, and even interpreted bytecode is possible through the use of magic methods and variables (denoted by two underscores at the beginning and end of the identifier).

2.1.2 Example: A Linked List

Since Python is primarily an OOP language, the most common form of expressing data types is through the use of classes. A typical linked list data type in Python 3.11 would look something like:

```python
from __future__ import annotations
from typing import Generic, TypeVar

T = TypeVar("T")


class LinkedList(Generic[T]):
    def __init__(self, value: T, next: LinkedList[T] | None = None) -> None:
        self.value = value
        self.next = next

    def append(self, new_value: T) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = LinkedList(new_value)

    def concat(self, other: LinkedList[T]) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = other
```

Note that Python 3.12 and later make some major changes involving scoping and the syntax of generic types [15].

The above `LinkedList` class expresses the recursive linking nature of the data type while demonstrating commonly used side effects of OOP methods. Note that the return type for both `append` and `concat` is `None`. Instead, the data contained within the `self` object is mutated in place as needed to achieve the desired functionality.

2.1.3 Pydantic: A data validation library

While native Python works well for simple data types like linked lists, type hints do not provide any guarantees that the type of value, for example, will be consistent across all elements in the list. So, while it does serve as a useful notation tool for documenting the structure of a data type to other programmers, there are no insurance that the runtime values for these objects are consistent with the type annotations. As with many topics in Python, a convenient library by the name of Pydantic will enable both type hints and data validation for

those types *at runtime.* The Pydantic version of our linked list will now look something like:

```python
import pydantic

from __future__ import annotations
from typing import Generic, TypeVar

T = TypeVar("T")


class LinkedList(pydantic.BaseModel, Generic[T]):
    value: T
    next: LinkedList[T] | None = None

    def append(self, new_value: T) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = LinkedList(value=new_value)

    def concat(self, other: LinkedList[T]) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = other
```

For this simple data type, only the init function changes. However, under the hood, Pydantic will now ensure that the type of `value` within this linked list remains consistent across all elements, throwing an exception otherwise. Pydantic's version of a class has the ability to transform an OOP construct into one more closely related to functional programming. If desired, the use of decorators above field names or methods (denoted by `@decorator`) can restrict mutability, modify constructors, and even change the data validation process to ensure objects are of the correct type.

2.1.4 Conclusion

Python's high-level nature, excellent documentation, library support, and exposure of underlying architectures when needed combine to create a powerful, flexible, and productive language. Although many aspects of the language are initially hidden or abstracted for the sake of convenience, Python allows programmers to dig deeper into the implementation when necessary and extend the language in a manner best suited for a specific task.

2.2 TypeScript

In JavaScript, types are very loosely bound, and many operations/functions will silently convert types at runtime. The silent recasting of JavaScript objects combined with unintuitive defaults can result in very confusing results for the uninitiated. For example, the expression `1 + "1"` returns a value of 11 and the

expression `[5, 11, 2].sort()` returns `[11, 2, 5]`. In both expressions, the numerical values are converted into strings before the operation is performed.

TypeScript, as an extension of JavaScript with types, is similar to Python in the sense that both languages started with dynamic typing and patched on static types after the fact, and both aim to resolve issues caused by dynamic typing. TypeScript (and Python) use types not only for more assurance in program correctness, but also for the sake of convenience. When making full use of the available typing systems, text editors and IDEs are able to assist in new code development through intellisense and autocompletion of methods and variables. Types of objects and documentation for those types additionally appear on mouse-hover, providing fast in-context knowledge.

The below list outlines key properties of types in TypeScript.

- Types are treated as sets of values, where types can be part of multiple sets, a union of sets, or rarely even an intersection of sets.

- Types are structural. Any object containing the required fields needed to be a member of a certain type will be considered an object of that type, even if a different type name has been assigned to it. Further, if an object has additional fields not specified by the type, the extra fields will be ignored in the typechecking process. Essentially, as long as an object meets the minimum fields required by the specified type, the object is assumed to be a member of that type.

- TypeScript allows any literal string to act as a subset of its respective type, with the only allowed value being the literal itself. For example, a field with the type `''name''` will only accept the value of "name". However, a field with the type `String` will accept any string object.

In comparison to algebraic data types often seen in functional languages, TypeScript replaces inductive data types with discriminated union sets. For example, our linked list type may look something like:

```
type LinkedList<T> =
        | { kind: "nil" }
        | { kind: "cons"; value: T; next: LinkedList<T> };

function append<T>(self: LinkedList<T>, new_value: T): LinkedList<T> {
        if (self.kind === "nil") {
                return { kind: "cons", value: new_value, next: { kind: "nil" } };
        }

        let l = self;
        while (l.kind === "cons" && l.next.kind !== "nil") {
                l = l.next;
        }
        l.next = { kind: "cons", value: new_value, next: { kind: "nil" } };
        return self;
}

function concat<T>(self: LinkedList<T>, other: LinkedList<T>): LinkedList<T> {
        if (self.kind === "nil") {
```

```
                return other;
        }

        let l = self;
        while (l.kind === "cons" && l.next.kind !== "nil") {
                l = l.next;
        }
        l.next = other;
        return self;
}
```

Because TypeScript allows literals to act as types, the only valid values in the "kind" field of an object represented by the `LinkedList` type are our constructor names: "nil" or "cons". Although TypeScript treats every object as a reference, similar to Python, we have not used side effects in the `append` or `concat` functions. Instead, we explicitly accept the linked list we want to transform as input, perform the required action, and return the head of the linked list in the output.

For a non-inductive approach to the LinkedList data type, we can use TypeScript's interfaces to create something that looks much more like the Pydantic version:

```
interface LinkedList2<T> {
        value: T;
        next: LinkedList2<T> | null;
}

function append2<T>(self: LinkedList2<T>, new_value: T): LinkedList2<T> {
        let l = self;
        while (l.next !== null) {
                l = l.next;
        }
        // Structural typing - No need to refer to LinkedList2 name explicitly
        l.next = { value: new_value, next: null };
        return self;
}

function concat2<T>(
        self: LinkedList2<T>,
        other: LinkedList2<T>,
): LinkedList2<T> {
        let l = self;
        while (l.next !== null) {
                l = l.next;
        }
        l.next = other;
        return self;
}
```

Finally, a pure OOP version of our LinkedList type, using classes, can be formed. Note that the append and concat functions are now methods with side effects but are otherwise similar to Python:

```
class LinkedList3<T> {
        value: T;
```

```
        next: LinkedList3<T> | null = null;

        append3(new_value: T) {
                let l: LinkedList3<T> = this;
                while (l.next !== null) {
                        l = l.next;
                }
                l.next = new LinkedList3();
                l.next.value = new_value;
        }

        concat3(other: LinkedList3<T>) {
                let l: LinkedList3<T> = this;
                while (l.next !== null) {
                        l = l.next;
                }
                l.next = other;
        }
}
```

2.3   Rust

Rust is a unique, high-performance language focused on efficiency and reliability. While most languages discussed in this paper attempt to either simplify the process of creating software or formally prove algorithmic correctness, Rust carries a unique blend of practicality and formality. Among its key features lies a new paradigm of memory management; rather than using inefficient garbage collectors prominent in almost every OOP language or dangerous pointers common to imperative C-like languages, Rust introduces the concept of memory ownership. If safe Rust code is able to pass successfully through the compiler, it is near-impossible to create memory leaks, dangling references, or other undesirable states of memory. As a result, instead of attempting to prove the correctness of the model, the language instead proves the correctness of implementation details related to the model.

Rust's semi-manual methodology for memory management means memory-related concepts seep into nearly every line of code. When working with data, constructs like object lifetimes, mutability, and even the passing of data between functions must be carefully considered. The guarantees of Rust in relation to memory enable "fearless concurrency", or in other words, the comforting knowledge that even with parallelized programs, the memory guarantees of Rust still ensure proper ownership and usage of available data. In addition, to offset the steep learning curve of these relatively complex topics, Rust's compiler is renowned for its helpful error messages and gentle guidance towards writing memory-correct code.

As such, data types in Rust, while somewhat similar to types in OOP, must be considerate to follow Rust's memory rules. `struct`s are the foundational record-like data type, and can be used to express functional-style inductive data types, enums, or a simple collection of named objects. On the other hand, data type operators are deliberately defined separate from the data type itself.

In Rust, type operators with common behaviours and signatures can be encapsulated into traits that act like OOP interfaces. Each type can define its own implementation of a trait, allowing the type to be used whenever a generic trait is allowed.

The Rust implementation of `LinkedList` is as follows:

```rust
struct LinkedList<T> {
    value: T,
    next: Option<Box<LinkedList<T>>>,
}

impl<T> LinkedList<T> {
    pub fn append(&mut self, value: T) {
        let new_node = Box::new(LinkedList { value, next: None });

        let mut l = self;
        while let Some(ref mut next_node) = l.next {
            l = next_node
        }
        l.next = Some(new_node);
    }

    pub fn concat(&mut self, other: LinkedList<T>) {
        let mut l = self;
        while let Some(ref mut next_node) = l.next {
            l = next_node
        }
        l.next = Some(Box::new(other));
    }
}
```

Note that the "recursive" `next` field cannot directly accept `Option<LinkedList<T>>`. Instead, the type must be wrapped in a dedicated pointer type `Box` to allow for separation of nodes in memory and therefore lazy evaluation of the data. Further, the use of `Box` forces the while let loop to make use of a mutable reference to the next node, rather than a mutable instance of the data itself.

2.4 Haskell

Haskell is a pure, lazy functional language with excellent support for concise function composition, elegant recursion, and powerful typing. Compared to the mostly-imperative languages described in previous sections, the syntax of Haskell may seem foreign or even unintuitive at a first glance. However Haskell, along with other functional languages, are not intended to be tools to control the exact step-by-step actions of a computer. Rather, the functional declarative style means programmers need only specify the results of some actions and trust the computer to execute it according to the specification.

Functional languages, by virtue of their stateless nature and discouragement of manual control flows, appear far more similar to formal specifications than imperative or OOP languages. Algebraic data types common to these languages truly embody the idea that types are just collections of objects viewed through a certain lens. In turn, these data types are far more suited to inductive reasoning,

enabling relatively simple proofs to guarantee useful properties of a program. As we will explore in section 2.6, attempting these proofs in a fully imperative style can lead to a lot of headaches.

In order to achieve higher abstraction over algebraic data types and allow for impure actions within a purely functional world, Haskell makes use of interfaces for types, called typeclasses. Intuitively, these interfaces can broadly fall into two categories:

1. The creation of type interfaces to group common behaviour across a selection of types. Typeclasses like Eq, Ord, Show, etc. can be derived for a specific type, allowing for equality comparisons, ordered comparisons, and type-to-string translations respectively. Types that derive these typeclasses can make use of general operators specified by the typeclass (like == for Eq).

2. Typeclasses that act as a broad blueprint for attaching context to an already-concrete type. These kinds of type classes indirectly allow for a form of recorded "state" in order to better deal with impure interactions. The `Maybe` typeclass is a classic example of this: Normally, the expected value of an `Int` instance would be a limited-size integer number. However, the addition of context via `Maybe Int` transforms this certainty into two possibilities: either the value is `Just` some number, or it holds `Nothing`. Functors, monoids, and monads are all part of this category of typeclass, and are essential to practical Haskell.

Although the distinction of these two categories may not actually exist (in fact, the second kind of typeclass is really just a subset of the first), keeping these ideas separate can help form an intuition for the application of typeclasses.

The routine implementation of `LinkedList`, with additional instantiations for the functor, applicative, and monad typeclasses, is as follows:

```haskell
data LinkedList a = Nil | Cons a (LinkedList a)

append :: LinkedList a -> a -> LinkedList a
append Nil x = Cons x Nil
append (Cons y ys) x = Cons y (append ys x)

concatenate :: LinkedList a -> LinkedList a -> LinkedList a
concatenate Nil ys = ys
concatenate (Cons x xs) ys = Cons x (concatenate xs ys)

flatten :: LinkedList (LinkedList a) -> LinkedList a
flatten Nil = Nil
flatten (Cons xs xss) = concatenate xs (flatten xss)

instance Functor LinkedList where
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Applicative LinkedList where
    pure x = Cons x Nil
    Nil <*> _ = Nil
```

```
    _ <*> Nil = Nil
    (Cons f fs) <*> xs = fmap f xs `concatenate` (fs <*> xs)

instance Monad LinkedList where
    return x = Cons x Nil
    xs >>= f = flatten $ fmap f xs
```

Haskell's natural use of pattern matching and recursive iteration massively shortens the bodies of `append` and `concatenate` in addition to providing clarity on the precise structure of the function's output. The compiler further optimizes tail-recursive calls, as seen in the second pattern of both functions, so that these functions have comparable efficiency with imperative loops. Since everything in Haskell is a pure, stateless function, the distinction between data type operators and normal functions can be eliminated completely.

As a final note, applying `Monad` traits to `LinkedList` enables the use of Haskell's imperative-like `do` notation to map over elements of a list. The below example will first extract all elements from within the list `l`, add one to each element, and return the result:

```
do
    x <- xs
    return (x + 5)
```

2.5 Agda

Although Agda and Haskell may look somewhat similar at a first glance, Agda overhauls the functional programming paradigm with a completely new type system. Founded on the theory of dependent types, Agda's key features are its guarantees regarding the termination of well-formed programs and the types of function outputs. Most notably, the guarantees of error-free runtime execution and eventual termination come at the cost of Turing completeness, although the language is still plenty expressive in its own right.

Dependent types allow families of types to be indexed by values to create new, more specific types. In simpler terms, this inter-type relationship adds extra information to otherwise basic types, allowing Agda to reduce the number of producible errors at the cost of higher implementation complexity. For example, a user may create a `Matrix m n` type to represent matrices of size $m \times n$. In most languages, we would define a function for matrix multiplication with the type $mult : Mat \rightarrow Mat \rightarrow Mat$. However, there are no guarantees that this function will be successful since the first two Mat arguments may be of any size. By indexing the types via Agda, we can express an important property of matrix multiplication: `∀ i j k → (Mat i j) -> (Mat j k) -> (Mat i k)`. Now, it is immediately obvious from the type definition alone that matrix multiplication somehow converts matrices of size $i \times j$ and $j \times k$ into a new matrix of the shape $i \times k$. Powerfully precise expressions like this one virtually eliminate all possible forms of operation failure, since it makes a guarantee that the domain of the function arguments are always well-defined over their desired operations.

Because of its strong focus on types and logic as an integral part of writing programs, Agda is also well suited to perform as a proof assistant. Its language

extension plugin for emacs enable a proof mode to help programmers fill-in missing properties and complete theorems regarding the program.

An implementation of `LinkedList` in Agda is shown below, along with a proof for associativity of concatenation:

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong)
open Eq.≡-Reasoning
open import Data.Nat.Properties using
  (+-assoc; +-identityˡ; +-identityʳ; *-assoc; *-identityˡ; *-identityʳ; *-distribʳ-+)

data LinkedList (A : Set) : Set where
  []   : LinkedList A
  _::_ : A → LinkedList A → LinkedList A

infixr 5 _::_

append : ∀ {A : Set} → LinkedList A → A → LinkedList A
append [] x = x :: []
append (y :: ys) x = y :: (append ys x)

-- concat
_++_ : ∀ {A : Set} → LinkedList A → LinkedList A → LinkedList A
[]        ++ ys  =  ys
(x :: xs) ++ ys  =  x :: (xs ++ ys)

-- Proof by induction that list concatenation is associative
++-assoc : ∀ {A : Set} (xs ys zs : LinkedList A)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡⟨⟩
    ys ++ zs
  ≡⟨⟩
    [] ++ (ys ++ zs)
  ∎
++-assoc (x :: xs) ys zs =
  begin
    (x :: xs ++ ys) ++ zs
  ≡⟨⟩
    x :: (xs ++ ys) ++ zs
  ≡⟨⟩
    x :: ((xs ++ ys) ++ zs)
  ≡⟨ cong (x ::_) (++-assoc xs ys zs) ⟩
    x :: (xs ++ (ys ++ zs))
  ≡⟨⟩
    x :: xs ++ (ys ++ zs)
  ∎
```

The data type definition syntax and function definitions are strikingly similar between Haskell and Agda. As is customary for functional languages, pattern matching over all possible inputs enables ultra-concise expression of the intended function behaviour. Linked lists are a simple enough type to not warrant type indexing, but this implementation does make use of the builtin induction proof abilities to demonstrate the associativity of list concatenation.

2.6 Dafny

The Dafny language acts as a mediator between formal methods of programming expression and regular programming languages. Rather than separating the specification of a program from its implementation, Dafny implements several mechanisms to encourage inline verification statements alongside regular program definitions. The formal constructs used to verify programs often appear as a list of pre- and post-condition predicate statements that must be satisfied before or after running a function, conditional expression, or loop. In Dafny, the act of verifying a program is as natural and important as writing the program itself.

2.6.1 Specification as Part of the Implementation

In order to accomplish the ambitious task of combining specification and implementation into one code base, Dafny makes use of special programming structures labelled with "ghost". Put plainly, ghost structures are similar to normal programming constructs in terms of the ability to make variables and functions, but these constructs can only be used for the sake of verification. Often times, predicate statements with mixed ghost and non-ghost constructs are used to make guarantees about the result of a function, or restrict inputs as necessary. Once all listed conditions have been verified by Dafny, the compiler can then extract important program details and translate the results into another, more conventional language.

For a simple example, consider the specification of a banking system built on integer values. When a user makes a withdrawal request, the bank wants to ensure that the requested amount is not more than the amount available to the user. The specification for this requirement is very clear: if $balance - request < 0$, deny the request. However, there is no way to guarantee that this specification is met within the program implementation (aside from examining the source code). In Dafny, we can instead attach a pre-condition to inputs of the function, so that withdrawing money `requires balance > request`.

Both OOP-like classes and functional-like inductive data type constructs can be used when defining types in Dafny. The concept of Haskell Typeclasses also makes an appearance in its simplest form by allowing restrictions on generic types. For example, the Haskell signature `method: Eq a => a ...` can be modelled in Dafny by `method<T(==)> (argument1: T)...`. Like Agda, Dafny attempts to offer some assurance of program termination via the `decreases` clause. Through this clause, all recursive functions and loops are expected to specify the variant required to break out of iterative code. Of course, since the language is Turing complete, there is never a 100% guarantee for program termination, but most programs should be resolvable through this process.

On the note of finite vs. infinite, Dafny provides special tooling to separate infinitely large inductive data types from finite types through the co(-inductive) data type. Since these types may be theoretically infinite, they are lazily evaluated and use special operators for common data type tasks. For example,

checking for equality between two infinitely large data types is impractical at best, so Dafny allows checking for the first n components of the type while using the traditional == operator. Functions can additionally be labelled as co-recursive to specify that they may not terminate.

2.6.2 Example: A Linked List

In Dafny, we can define our `LinkedList` type in two ways: through inductive types and classes. For this specific type, the inductive format is extremely simple, albeit unoptimized:

```
datatype LinkedList<T> = Nil | Cons(T, LinkedList<T>)

function append<T>(lst: LinkedList<T>, x: T): LinkedList<T>
  decreases lst
{
  match lst {
    case Nil => Cons(x, Nil)
    case Cons(y, ys) => Cons(y, append(ys, x))
  }
}

function concatenate<T>(lst1: LinkedList<T>, lst2: LinkedList<T>): LinkedList<T>
  decreases lst1
{
  match lst1 {
    case Nil => lst2
    case Cons(x, xs) => Cons(x, concatenate(xs, lst2))
  }
}
```

Because this definition of `LinkedList` is inductive and close to its formal mathematical definition, we do not need to make many claims in the pre- and post-condition statements. We only claim that the recursive `append` and `concat` functions will eventually terminate as the first argument reaches Nil.

However, the OOP requires far more moving components, so we need to manually verify much more of the system. Because there are so many properties required to prove correctness of the OOP version, especially involving modification and mutability of data, we omit much of the verification code that would otherwise be placed in the body of the two method definitions:

```
class Node<T(==)> {
  var value: T
  var next: Node?<T>

  ghost var repr: set<object>
  ghost var contents: seq<T>

  constructor (new_value: T)
    ensures Valid() && contents == [new_value] && repr == {this}
  {
    value, next := new_value, null;
    repr, contents := {this}, [new_value];
```

```
}

ghost predicate Valid()
  reads this, repr
{
  && this in repr && |contents| > 0 && contents[0] == value
  && (next != null ==>
        && next in repr && next.repr <= repr && this !in next.repr
        && next.Valid() && next.contents == contents[1..])
  && (next == null ==> |contents| == 1)
}

method append(self: Node?<T>, new_value: T) returns (n: Node<T>)
  requires self == null || self.Valid()
  ensures n.Valid()
  ensures if self == null then n.contents == [new_value]
          else |n.contents| == |self.contents + [new_value]|
{
  var new_node := new Node(new_value);
  new_node.value := new_value;
  new_node.next := null;

  if (self == null) {
    n := new_node;
    return;
  }
  n := self;

  var iter_list: Node<T> := self;
  var i := 0;
  while (iter_list.next != null)
    invariant
      iter_list.Valid() &&
      i + |iter_list.contents| == |n.contents| &&
      n.contents[i..] == iter_list.contents
    decreases |n.contents| - i
  {
    iter_list := iter_list.next;
    i := i + 1;
  }

  iter_list.next := new_node;
  // Update the ghost var repr for all nodes in the list...
}

method concat(self: Node?<T>, other: Node?<T>) returns (n: Node?<T>)
  requires self == null || self.Valid()
  requires other == null || other.Valid()
  ensures n != null ==> n.Valid()
  ensures n == null ==> (self == null && other == null)
  ensures self != null ==> (n != null && |n.contents| >= |self.contents|)
  ensures other != null ==> (n != null && |n.contents| >= |other.contents|)
{
  if (self == null) {
    n := other;
    return;
  }
```

```
    n := self;

    var iter_list: Node<T> := self;
    var i := 0;
    while (iter_list.next != null)
      invariant
        iter_list.Valid() &&
        i + |iter_list.contents| == |n.contents| &&
        n.contents[i..] == iter_list.contents
      decreases |n.contents| - i
    {
      iter_list := iter_list.next;
      i := i + 1;
    }

    iter_list.next := other;
    // Update the ghost var repr for all nodes in the list...
  }
}
```

While the essential imperative aspects of the language are similar to convention, the burden of specification gives the assurance of correctness at the cost of verbosity. Note the addition of ghost variables `repr` and `contents`, which hold all nodes and the contents of those nodes only for verification purposes. These variables are used to formally express the behaviour of loops (ex. the loop will run at most through all nodes of the list (a maximum of $|contents|$ times)), the "growing" nature of both `append` and `concat` (ex. $|n.contents|$ must be bigger than $|self.contents|$), and membership of a node inside the current linked list (ex. the `Validate` predicate requires that `this` and `next` are in `repr`).

2.7   APL

APL is an extremely unique language that prides itself on ultra-concise expressions and data-based SIMD parallelism. Among the most defining characteristics of APL is the tendency to make use of non-ascii symbols for operations, right-to-left operator associativity, and the fact that everything in APL is array-based. APL statements are often branchless in nature, and mapping functions over data is strongly preferred over regular loops.

One APL reference [9] compares the types within APL to the numpy library in Python. Thus, instead of providing syntax to create data types as seen in other languages, APL strongly encourages the use of arrays and matrices to solve problems. The shape of the array (rank and depth) determines the type of an object and the effects of applied operators.

Because APL only has one major data type (the array), operators are often composed without explicitly calculating intermediary values. APL categorizes all operators into two groups: monadic operators that take in one argument, and dyadic operators for two arguments. More explicitly, all operators must accept either one or two arguments, which may drastically change the operator's behaviour. As a simple example, the monadic tilde operator ($\tilde{\ }$) is used for boolean *not*, while the dyadic version is for set difference.

Given the knowledge that everything is an array, we cannot reasonably recreate our linked list example. Nor should we, since lists and vectors are first-class citizens in APL. Instead, we take a look at a more interesting aspect of APL's feature set - the avoidance of branching through bit masks. Consider the following "loop and branch" Python code, where we loop over an array of objects and return every object that satisfies some function:

```python
def select(f, array):
    """Loop & branch"""
    result = []
    for element in array:
        if f(element):
            result.append(element)
    return result
```

The concise APL equivalent would instead be the following, notably branchless, code:

```
elems ← f array
elems/array
```

Note that the first line maps a function $f : T \rightarrow bool$ to create a bit mask (an array of zeroes and ones) $elems$, which is then passed to the selection operator $/$. Since both implementations must apply f to all elements of the input array, the branchless version can be far more efficient due to the use of SIMD instructions.

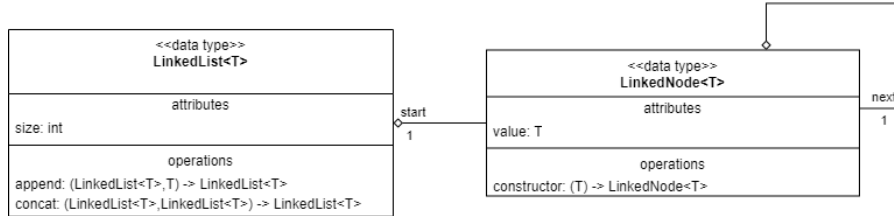## 2.8 Unified Modelling Language (UML)

Up until this point, all the languages explored in this document have been targeted at making computers execute a desired set of instructions over some type of data. Of course, the specific methodology and concepts used to achieve this human-computer interaction differ greatly between languages, with each maintaining their own idioms and customs. Conversely, UML is not a programming language in the normal sense, but a standardized method of visualizing and clearly communicating the intentions behind a system.

According to the UML documentation, UML is intentionally independent of any and all processes that could be used to actually implement a system. UML diagrams are not intended to comprehensively cover every aspect of a system's implementation, including different quirks and features of different languages, but rather should be used as a powerful aid in the creation of program documentation. Generally, different types of diagrams are used for recording types, state, information flow, and even the execution of the program itself.

In the context of UML, a data type is a specialized form of classifier with no internal structure. Each data type may contain several named fields to contain data, along with operations on that data. The focus of data type based diagrams is to convey the following to programmers:

1. All data types available to the user.

2. All relationships between data types, including inheritance, extensions, use of types within the fields of other types, and even type usage in operators.

Below is an example of our famous LinkedList type, modelled by UML:

```
        <<data type>>                              <<data type>>
        LinkedList<T>                              LinkedNode<T>
                                    start                                    next
        attributes                   1          attributes                   1
size: int                                  value: T

        operations                              operations
append: (LinkedList<T>,T) -> LinkedList<T>   constructor: (T) -> LinkedNode<T>
concat: (LinkedList<T>,LinkedList<T>) -> LinkedList<T>
```

A simple standalone type like LinkedList may not be complex enough to showcase the full advantages of UML, but it is enough to demonstrates the effectiveness of visualization in understanding code. From the arrows in the image, it is immediately obvious that the start attribute of LinkedList requires a LinkedNode and LinkedNodes may build off of themselves indefinitely.

UML diagrams are a simple and concise method for conveying the meaning and intention behind a system. While the syntax is largely standardized, because this language is meant for communication with humans rather than machines, slight modifications of syntax for the sake of clarity can be made as necessary. For example, the types of operators may be omitted for an ultra-compact diagram, or kept to ensure an agreement on the interface implementation is met.

2.9 Event-B

Although UML is a powerful tool for communicating the relationship between data types, the visual methodology falls short when attempting to convey the actionable properties of a system. In continuing the exploration of modelling languages, we find that the Event-B model not only encompasses actionable system traits, but does so with consideration of the system's environment. Therefore, Event-B is suitable for formal system analysis and proving the correctness of software following the model specification.

Data types for Event-B may not exist in the normal sense of the word, but a different approach to data abstraction through the concepts of machines, contexts, and events is more than suitable for the same purpose:

- Machines are somewhat adjacent to classes in OOP, acting like miniature programs with some provable traits consisting of variables, invariants, theorems, a single variant, and a list of events that may cause the machine to "run" (analogous to an object's methods). Like class inheritance, new machines can "refine" old ones by carrying over and adding new entries to the aforementioned traits.

- Contexts, on the other hand, are much closer the formal definition of abstract data types, defining a set of axioms, theorems, constants and data in the form of carrier sets. Contexts can be extended in a similar fashion to class inheritance, with the child context carrying over all properties of the parent context.

- Finally, events serve as activation conditions for the machine they are assigned to. They introduce possibly nondeterministic actions into the system in a formal manner, and must clearly state how they affect a machine's variant. A key aspect of events is that all actions within an event are performed simultaneously. This means that, unlike in sequential programming languages, the order in which actions are defined have no relevance on the outcome of the event. Of course, the restriction of simultaneous execution means that users of the language cannot reassign variables in more than one action, nor should they rely on the deterministic nature of normal program execution.

All expressions in the above abstractions use formal discrete math notation.

Although a modelling language focused on provable aspects of a system is not very suitable for a simple low-level data type, we nonetheless attempt a simplified implementation of `LinkedList` with only the append method.

LinkedListContext
    sets
        $Nodes$ // Set of nodes in the list
        $References$ // Set of references
    constants
        $null$ // End of list
    axioms
        axm1: $References = Nodes \cup \{null\}$
        axm2: $\forall n : n \in Nodes \implies n.next \in References$ // All nodes must have a reference

---

AppendEvent
    any
        $last\_node$
        new_node
    where
        $last\_node \in Nodes$
        $\{last\_node\} = \{node \in Nodes : node.next = null\}$ // last_node is the end of the list
        $new\_node.value : T$ // New value must have the same type as existing nodes
        $new\_node.next = null$ // New node is end of the list
    then
        $last\_node.next := new\_node$ // Last node is no longer the last node
        $Nodes := Nodes \cup new\_node$ // Add node to list

```
LinkedListMachine
    variables
        head
        value
        next
    invariant
        inv1: head ∈ Nodes
        inv2: value : T // Generic type T
        inv3: next ∈ References
        inv4: ∀n : n.next ≠ null  ⟹  n.next.next ∈ References // Nodes
can either end or continue the list
    events
        AppendEvent
```

The latest Event-B wiki [5] notes future plans to support recursive structured types like inductive LinkedLists in a more concise format.

2.10 Exploration of Conceptual Data Types

2.10.1 The Axiomatic Approach to Data Types

As stated in the introductory textbooks [12] and [22], ADTs at their core are just collections of data coupled with operations. Both textbooks introduce data types in a manner not unlike the ideas of Event-B, using formal notation to guarantee precise properties of the given data type. Interestingly, both textbooks opt to separate data type operations into syntactic and semantic components, where the semantics contain a list of axioms to be fulfilled by each operation. Note that these axioms are almost entirely removed from the implementation of the type, acting more as a specification of requirements rather than a how-to guide.

Guaranteed type completeness and correctness are two key features of the axiomatic approach. That is, axioms encourage operations to consider all possible values that the type may take, especially regarding edge cases and errors. For example, in our linked list, we need to ensure operations over an empty list are well-defined and that we can create the empty list itself in the first place. Of course, in the case of linked lists, this is near-trivial, but focusing on the edge cases in this manner can guarantee correctness of far more complex types.

Consider the axiomatic definition of a linked list, according to the specification of [22]:

NAME
    LinkedList(item)
SETS
    N: Set of nodes
    I: Set of items
SYNTAX
    $new :\to N$
    $append : N \times I \to N$
    $concat : N \times N \to N$
    $deleteLast : N \to N$
    $tail : N \to I$
    $isEmpty : N \to bool$
SEMANTICS
$\forall i \in I, \forall n, n1 \in N :$
    $deleteLast(append(n, i)) = n$
    $tail(append(n, i)) = i$
    $tail(concat(n, n1)) = tail(n1)$
    $isEmpty(new()) = True$
    $isEmpty(append(n, i)) = False$
    $tail(new()) = error$ // New initializes a list but does not hold any
value
    $deleteLast(new()) = error$ // Cannot delete from an empty list

The above axioms do not express the behaviour of each function independently, but instead focus on relationship between one another. For instance, the axiom $tail(new()) = error$ outlines two properties with one axiom:

1. Finding `tail` on an empty list produces an error.

2. Since tail produces an empty list error, we must know that `new` creates an empty list.

Similarly, the axiom $deleteLast(append(n, i)) = n$ shows that append must add the item i to the end of the list n, in order for deleteLast to perform correctly.

Unfortunately, this approach may present drawbacks for implementing unknown data types in popular programming languages *because* of its coupled operation behaviour. In many programming languages, especially non-functional ones, developers need to implement each data type operation mostly independent of other operations. Therefore, ensuring congruency between written code and the given specification presents a new challenge for programmers; data types becoming lost in the translation from dependent specification to independent implementation serve as an undesirable source of bugs and mistakes. A compiler for such a language would greatly reduce friction between theory and application, but the semantic expressivity available from the axiomatic approach may be worth the costs.

2.10.2 A More Constructive Approach

The textbook [22] specifies an additional method of data type specification focused on construction of new objects from old ones. Instead of axioms, each operation can specify relevant pre-conditions and post-conditions. This method is far closer to the implementation of data types in programming languages, with an uncanny resemblance to the optional input restrictions seen in Dafny functions. For conciseness, the example given in Section 2.4 serves as a good approximate illustration of the constructive approach outlined in the textbook.

3 Error Handling and Undefinedness

4 A Word on Usability

5 Conclusion

References

[1] Jean-Raymond Abrial. *Modeling in Event-B : system and software engineering*. eng. Cambridge: Cambridge University Press, 2010. ISBN: 9781139195881.

[2] *C++ reference*. 2024. URL: https://en.cppreference.com/w/ (visited on 02/07/2025).

[3] Wikipedia contributors. *Type class — Wikipedia*. 2025. URL: https://en.wikipedia.org/wiki/Type_class (visited on 02/07/2025).

[4] Luca Di Grazia and Michael Pradel. "The evolution of type annotations in python: an empirical study". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220. ISBN: 9781450394130. DOI: 10.1145/3540250.3549114. URL: https://doi.org/10.1145/3540250.3549114.

[5] *Event-B and Rodin Documentation Wiki*. 2024. URL: https://wiki.event-b.org/index.php/Main_Page (visited on 02/07/2025).

[6] A. Hsu. *Is APL Dead?* 2020. URL: https://www.sacrideo.us/is-apl-dead/ (visited on 02/07/2025).

[7] Kenneth E. Iverson. "Notation as a tool of thought". In: *Commun. ACM* 23.8 (Aug. 1980), pp. 444–465. ISSN: 0001-0782. DOI: 10.1145/358896.358899. URL: https://doi.org/10.1145/358896.358899.

[8] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. ISBN: 9781718503106. URL: https://books.google.ca/books?id=SE2GEAAAQBAJ.

[9]   *LearningAPL*. 2024. URL: `https://xpqz.github.io/learnapl/intro.html` (visited on 02/07/2025).

[10]   K Rustan M Leino. "Specification and verification of object-oriented software". In: *Engineering Methods and Tools for Software Safety and Security*. IOS Press, 2009, pp. 231–266.

[11]   M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press Series. No Starch Press, 2011. ISBN: 9781593272838. URL: `https://books.google.ca/books?id=R2RbBAAAQBAJ`.

[12]   J.J. Martin. *Data Types and Data Structures*. Prentice-Hall International series in personal computing. Prentice-Hall International, 1986. ISBN: 9780131959835. URL: `https://books.google.ca/books?id=554ZAQAAIAAJ`.

[13]   *OMG Unified Modeling Language*. Version 2.5.1. Object Management Group. 2017. URL: `https://www.omg.org/spec/UML/2.5.1/PDF` (visited on 02/07/2025).

[14]   *PEP 20 - The Zen of Python*. Python Software Foundation. 2004. URL: `https://peps.python.org/pep-0020/` (visited on 02/07/2025).

[15]   *PEP 695 - Type Parameter Syntax*. Python Software Foundation. 2022. URL: `https://peps.python.org/pep-0695/` (visited on 02/07/2025).

[16]   *Pydantic*. Python Software Foundation. 2025. URL: `https://docs.pydantic.dev/latest/` (visited on 02/07/2025).

[17]   A. Rauschmayer. *JavaScript metaprogramming with the 2022-03 decorators API*. 2024. URL: `https://2ality.com/2022/10/javascript-decorators.html` (visited on 02/07/2025).

[18]   D. Rosenwasser. *Announcing Typescript 5.0*. 2024. URL: `https://devblogs.microsoft.com/typescript/announcing-typescript-5-0/#decorators` (visited on 02/07/2025).

[19]   *The Dafny Programming and Verification Language*. 2024. URL: `https://dafny.org/` (visited on 02/07/2025).

[20]   *The Python Language Reference (3.13.2)*. Python Software Foundation. 2025. URL: `https://docs.python.org/3/reference/` (visited on 02/07/2025).

[21]   *The Rust Reference*. 2024. URL: `https://doc.rust-lang.org/reference/` (visited on 02/07/2025).

[22]   P.G. Thomas, H. Robinson, and J. Emms. *Abstract Data Types: Their Specification, Representation, and Use*. Oxford applied mathematics and computing science series. Clarendon Press, 1988. ISBN: 9780198596639. URL: `https://books.google.ca/books?id=u61QAAAAMAAJ`.

[23]   *Type — HaskellWiki*. HaskellWiki. 2024. URL: `https://wiki.haskell.org/index.php?title=Type` (visited on 02/07/2025).

[24]   *Typeclassopedia — HaskellWiki*. HaskellWiki. 2022. URL: `https://wiki.haskell.org/index.php?title=Typeclassopedia` (visited on 02/07/2025).

[25]   *Typescript Documentation.* 2024. URL: `https://www.typescriptlang.org/docs/` (visited on 02/07/2025).

[26]   *Welcome to Agda's documentation!* Version 2.7.0.1. 2025. URL: `https://agda.readthedocs.io/en/v2.7.0.1/#` (visited on 02/07/2025).