Implementation of Recursive Data Types in Modern Programming Languages

Josef Graf

September 30, 2025

1 Introduction

Recursive data types are a fundamental concept in computing, allowing for the representation of self-referential (recursive) types. By providing native support for them, programmers can easily represent complex structures such as trees without relying on fixed-size arrays or index-based representations, which are significantly more complicated. In this paper, we explore how various modern programming languages implement recursive data types, highlighting their syntax, semantics, and memory management techniques through the use of two examples.

Our first example is that of a tree where each node contains an integer value and optionally a left and right node of the same type.

```
type Node {
  value: Int;
  left: Node | nil;
  right: Node | nil;
}
```

We also define a recursive function `sum` that computes the sum of all values in the provided tree.

```
function sum(root: Node): Int {
  acc := root.value;
  if (root.left != nil) {
    acc := acc + sum(root.left);
  }
  if (root.right != nil) {
    acc := acc + sum(root.right);
  }
  return acc;
}
```

Finally, we illustrate how such trees are constructed, modified, and deconstructed through the following example.

```
tree := Node(1, Node(2, nil, nil), Node(3, nil, nil));
total := sum(tree); // 6
tree.right := nil; // tree is now Node(1, Node(2, nil, nil), nil)
tree := tree.left; // Node(2, nil, nil)
```

In our second example, we illustrate an abstract syntax tree (AST) for simple arithmetic expressions involving addition and multiplication.

```
type Ast = Add(Ast, Ast) | Mul(Ast, Ast) | Num(String);
```

We also define a recursive function `eval` that computes the integer result of the expression represented by the provided AST. To do so, we assume the existence of a function `parseInt` that takes a string as its sole parameter and returns the parsed integer value.

```
function eval(expr: Ast): Int {
  switch expr {
    Add(left, right) => eval(left) + eval(right),
    Mul(left, right) => eval(left) * eval(right),
    Num(num) => parseInt(num),
  }
}
```

Finally, we assume the existence an AST parsing function `parse` and use it to illustrate AST evaluation through the expression `2 + 3 * (4 + 5)`.

```
expr := parse("2 + 3 * (4 + 5)") // Add(Num("2"), Mul(Num("3"),
↪  Add(Num("4"), Num("5"))))
result := eval(expr); // 29
```

As a final note, all memory layout diagrams in this paper assume a 64-bit architecture with 8-byte pointers.

2 Languages

2.1 Rust

Rust is a statically typed, compiled language that relies on an ownership and borrowing system to provide memory safety without the need for a garbage collector.[1] Rust has two core forms of custom data types relevant to this paper: structs and enums, defined with the `struct` and `enum` keywords,

[1] *The Rust Programming Language.*

respectively. Struct types allow the definition of product types, and enums allow the definition of sum types; both can be used for recursive data types. To write our tree example in Rust, we would start with a struct `Node` and define our fields.[2]

```
struct Node {
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}
```

The `Node` type takes 24 bytes, 8 for each of our left and right fields, 4 for our value field, and 4 bytes to ensure the struct's fields are aligned to an 8-byte boundary according to Rust's layout rules. Note that since Rust manages all field accesses itself and does not expose raw pointers as a safe API, the compiler may order the fields in any order when using the default representation.[3]

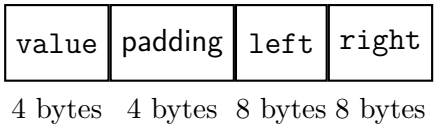

Figure 1: Layout of `Node`

Since all types in Rust are stack allocated by default,[4] Rust requires that types, except for dynamically sized types, which we will not touch on here, have a known size at compile time.[5] A byproduct of this rule is that recursive data types require some form of indirection, i.e., types cannot directly have fields of their same type; rather, they must be wrapped in another type with a known size.[6] When allocating on the heap, Rust provides the `Box<T>` type,

[2] *The Rust Programming Language*, chap. 5; *The Rust Programming Language*, chap. 6; *The Rust Reference*, § 10.1.
[3] *The Rust Reference*, § 10.3.
[4] *Rust by Example*, § 19.1.
[5] *The Rust Programming Language*, chap. 20 § 3.
[6] *The Rust Reference*, § 10.1.

which is a smart pointer to a heap-allocated instance of type `T`.[7] Thus, by using `Box<Node>` as the type of the left and right fields in `Node`, we are able to create recursive data types. However, if we just used `Box<Node>`, then our tree would always be required to have both left and right nodes. To address this, we introduce the `Option<T>` type, a built-in enum that can be either `Some(T)` or `None`. By doing so, Rust programmers can pattern match using constructs such as `match` and `if let` to safely unpack an optional type.[8] Conveniently, the Rust compiler also implements null-pointer optimization for `Option<Box<T>`, meaning that since `Box<T>` compiles down to a non-zero pointer, `Option<Box<T>` can be represented in the same space as a single pointer, using the null pointer to represent the `None` variant.[9]

```
// the following are provided as simplified definitions of the Option and
↪  Box types included in Rust's standard library
enum Option<T> {
    Some(T),
    None,
}

struct Box<T> {
    ptr: *const T, // non-zero pointer to a heap allocated instance of
    ↪  type T
}
```

Before we can implement our `sum` function, we first need a brief understanding of Rust's ownership and borrowing system, as well as deref coercion. This system enables the compiler to ensure memory safety by eliminating various classes of memory errors at compile time, without requiring a garbage collector or manual memory management. In Rust, each value has exactly one owner, and when that owner goes out of scope, the value is dropped, meaning Rust runs its destructor and deallocates any memory used by the value. Values can be borrowed through either mutable or immutable references; however, the compiler requires that each value have only a single

[7] *Rust by Example*, § 19.1.
[8] *The Rust Programming Language*, chap. 6.
[9] *std::boxed - Rust*; *std::option - Rust*.

mutable reference or any number of immutable references at any given time. Finally, references are also limited by the scope of the owner, meaning that a reference cannot outlive the value it is referencing.[10] Deref coercion is a key compiler feature that allows it to automatically convert between references to types that implement the `Deref` trait, allowing the programmer to use a `Box<T>` as if it were a reference `&T`, or treat a `&Option<T>` as a `Option<&T>`.[11]

With this in mind, we now move to the definition of `sum`. The function takes a reference to a `Node` and, utilizing Rust's `if let` pattern matching and deref coercion, unpacks the optional left and right nodes to recursively sum the values of all nodes in the tree.

```
fn sum(node: &Node) -> i32 {
    let mut acc = node.value;
    if let Some(left) = &node.left {
        acc += sum(left);
    }
    if let Some(right) = &node.right {
        acc += sum(right);
    }
    acc
}
```

In order to illustrate tree modification, we define our `tree` variable as mutable using the `mut` keyword. Then, we construct our left and right nodes, explicitly wrapping each in `Some` and a `Box::new` call to heap allocate them.

```
fn main() {
    let mut tree = Node {
        value: 1,
        left: Some(Box::new(Node {
            value: 2,
            left: None,
            right: None,
        })),
        right: Some(Box::new(Node {
            value: 3,
            left: None,
```

[10] *The Rust Programming Language*, chap. 4.
[11] *The Rust Programming Language*, chap. 15 § 2.

```
            right: None,
        })),
    };
```

When calling `sum`, we pass a reference to our tree using the `&` operator.

```
    let total = sum(&tree); // 6
```

To remove the right node, we assign `None` to the `right` field, which causes the compiler to implicitly drop the previous right node (since it is moved out of `tree`, but not into a variable).[12]

```
    tree.right = None; // tree is now Node { value: 1, left: Some(...),
    ↪  right: None }
```

Finally, to deconstruct the tree and take ownership of the left node, we assign `tree.left` to a new variable `left`, causing a partial move, disallowing further access to `tree.left` for the remainder of the program and leaving us with an `Option<Box<Node>>` (moved from the heap to the stack) in `left`. To extract the inner `Node`, we use the `Option::unwrap` method, which will panic (crash the program) if we have `None`, but safely return `T` if we have `Some(T)`.[13] Finally, to move ownership of the inner `Node` out of the `Box<Node>` and into `left`, we dereference the `Box<Node>` using the `*` operator.[14]

```
    let left = tree.left;
    let unwrapped = left.unwrap();
    tree = *unwrapped; // tree is now Node { value: 2, left: None, right:
    ↪  None }
}
```

Rust does not provide its own allocator; instead, it relies on the system allocator. For example, on Linux, this is often the `malloc` and `free` provided by glibc. glibc utilizes a series of arenas, which enable it to support multi-threading by creating new arenas as contention for existing arenas increases. Each arena then contains its own set of heaps, which are contiguous regions

[12] *Rust by Example*, §15.2.2.
[13] *std::option - Rust*.
[14] *std::boxed - Rust*.

of memory obtained from the operating system using `mmap` or `sbrk`. Heaps are further subdivided into chunks of various sizes and sorted into bins based on their size. Each bin contains a linked list of free chunks within that arena, allowing multiple threads to request memory from different arenas without contention and reuse memory without performing memory compaction, albeit at the risk of fragmentation. When a chunk is free, it contains metadata that includes pointers to the next and previous chunks in the bin, allowing `malloc` to easily find free chunks and `free` to easily add chunks to the appropriate bin without requiring any memory beyond the chunk itself.[15]

Below is a diagram illustrating a simplified memory layout of our stack and heap after deconstructing our tree, but before final cleanup.

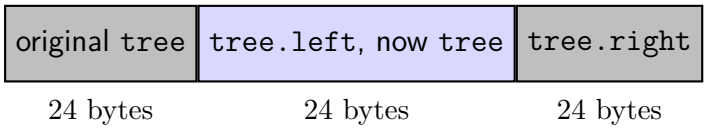

Figure 2: Heap (gray = deallocated)

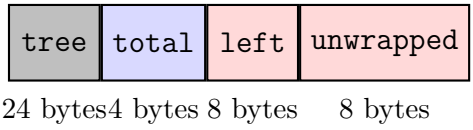

Figure 3: Stack (gray = deallocated, red = moved)

This memory layout results in gaps on the heap. However, since Rust uses the system allocator and performs no automatic memory management itself, these gaps will be handled by the underlying allocator and potentially reused in the future according to its implementation.[16]

Rust, however, is not limited to a single solution for type indirection; rather, it offers several options including typical heap allocation using (`Box<T>`,

[15] *MallocInternals - glibc wiki.*
[16] *System in std::alloc - Rust.*

as we saw above), reference-counted types (`Rc<T>` or `Arc<T>`), and, of course, references (`&T`). We have already covered `Box<T>`, so we now turn our attention to the remaining two.[17]

Reference-counted types function more like garbage collection. The actual `Rc<T>` is only a pointer to the inner `T` and the count of active strong and weak references (stored as `RcInner<T>`). When a `Rc<T>` is cloned (a new instance is created referring to the same inner `T`), the inner strong reference count is incremented. When one is dropped (deallocated), the inner strong reference count is decremented through the `Drop` trait, analogous to a destructor. Weak references on the other hand, are created manually to handle potentially cyclic use cases. Since these types are part of the standard library (not compiler internals), when the reference count reaches zero, the inner memory is deallocated just like any other heap-allocated chunk of memory (i.e., handled by the system allocator). Rust has two such types, `Rc<T>` and `Arc<T>`, the difference being that `Rc<T>` is restricted to a single thread and thus uses a plain location in memory to store its reference count, while `Arc<T>` can be sent safely between threads and uses an atomic internally for its reference count.[18]

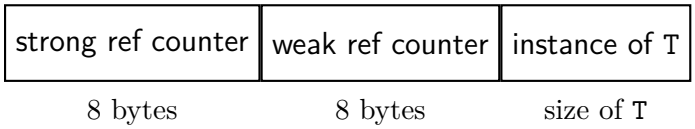

Figure 4: Layout of `RcInner<T>`

Finally, references are a form of indirection that do not involve heap allocation or reference counting; instead, they are pointers to a value owned by another variable. However, to use references, we need to introduce the concept of lifetimes. In Rust, all references have a lifetime, i.e., the portion of code during which the reference is valid before the value it is referencing

[17] *The Rust Programming Language*, chap. 15.
[18] *The Rust Programming Language*, chap. 15 § 4; *std::rc - Rust*; *Arc in std::sync - Rust*.

is dropped. The compiler can determine the lifetimes of many references on its own; however, defining types that contain references is not one of those cases. In order for a type to contain a reference, it must be constrained by that reference's lifetime.[19] To demonstrate this, we use a variation of our AST example.

In this example, we define our AST using an enum following a similar pattern to our tree example, but with one key difference: the enum definition has a lifetime parameter `<'a>` and the `Num` variant contains a reference `&'a str` rather than an owned `String`. These additions allow us to prevent unnecessary heap allocations and string copies, as we can tie the lifetime of our struct to that of the input string. Consequently, the Rust compiler ensures our AST cannot outlive the input string it references. This fact is exemplified by our `parse` function, which takes in a reference to a `str` (the primitive type Rust uses to refer to a length of string in memory) and returns an `Ast` tied to that same lifetime `'a`.[20]

```
enum Ast<'a> {
    Add(Box<Ast<'a>>, Box<Ast<'a>>),
    Mul(Box<Ast<'a>>, Box<Ast<'a>>),
    Num(&'a str),
}
```

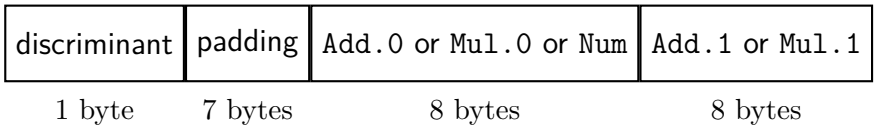

Figure 5: Layout of `Ast<'a>`

```
fn eval(expr: &Ast) -> i32 {
    match expr {
        Ast::Add(left, right) => eval(left) + eval(right),
        Ast::Mul(left, right) => eval(left) * eval(right),
```

[19] *The Rust Programming Language*, chap. 4 § 2; *The Rust Programming Language*, chap. 10 § 3.

[20] *The Rust Programming Language*, chap. 10 § 3.

```
        Ast::Num(num) => num.parse::<i32>().unwrap(),
    }
}

fn parse<'a>(input: &'a str) -> Ast<'a> {
    unimplemented!()
}

fn main() {
    let expr = Ast::Add(
        Box::new(Ast::Num("2")),
        Box::new(Ast::Mul(
            Box::new(Ast::Num("3")),
            Box::new(Ast::Add(
                Box::new(Ast::Num("4")),
                Box::new(Ast::Num("5"))
            ))
        ))
    );
    let result = eval(&expr); // 29
}

fn will_error() {
    let expr;
    // we can use {...} to define a new scope
    {
        let expr_str = String::from("2 + 3 * (4 + 5)");
        expr = parse(&expr_str); // error: expr_str does not live long
        ↪  enough
    }
}
```

When modifying an `Ast`, we can use similar patterns as before, but with the addition of pattern matching on enums. For example, to replace the left node of a `Add`, we would use the following:

```
let mut expr = Ast::Add(
    Box::new(Ast::Num("2")),
    Box::new(Ast::Num("3")),
);
if let Ast::Add(left, right) = &mut expr {
    *left = Box::new(Ast::Num("4"));
}
```

In such a case, we use `if let` to match the correct variant, deref coercion, and then finally assign to the position in memory that `left` (a mutable reference) points to. Since our old value of `left` is no longer valid, the compiler will automatically drop it, and since Rust string literals are `&'static str`, i.e., they are references to a sequence of bytes defined statically in the binary, we do not have to worry about lifetimes in this example.[21]

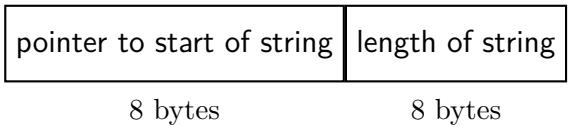

Figure 6: Layout of `&str`

In summary, `Box<T>` is the simplest solution, providing straightforward heap allocation with minimal overhead. It allows mutable and shared access, but is constrained by Rust's ownership and borrowing rules. It is best suited for situations where a single owner is required and heap allocation is an acceptable overhead (i.e., linked lists, trees, vectors, etc). `Rc<T>` and `Arc<T>` provide shared ownership through reference counting, allowing multiple parts of a program to have an owned reference to the same data. However, this comes with the overhead of managing the reference count. Since Rust requires sole ownership for mutable references, all data inside can only be accessed immutably unless wrapped in a type that provides interior mutability, such as a mutex. Reference-counted types are best suited for data that needs to be shared throughout the program or across threads (i.e., database connection pools), and the added overhead of managing the reference count is acceptable for the convenience compared to managing references and lifetimes manually. Finally, references (`&T`) provide a zero-copy form of indirection that avoids heap allocations and reference counting entirely by pointing to an existing value. However, they are constrained by Rust's restrictions on borrowing

[21] *str - Rust.*

and lifetimes, meaning they are best used as a temporary view into existing data when copying is too expensive (i.e., high-performance parsing or string manipulation).

2.2 C

C is a statically typed, compiled language that relies on manual memory management. C has three core forms of custom data types, structs, unions, and enums, defined with the `struct`, `union`, and `enum` keywords, respectively.

C requires that structs have a known size at compile time, but does not provide specialized smart pointer types for heap allocation or reference counting. Instead, C relies on the fixed size of raw pointers to achieve the required type indirection, requiring programmers to implement reference-counting themselves or through the use of external libraries.[22] C also does not provide a built-in optional type and relies on the use of null pointers and manual null checks to represent optional values.[23] To write our tree example in C using a struct `Node` and define our fields.[24]

```c
#include <stdlib.h>

struct Node {
  int value;
  struct Node *left;
  struct Node *right;
};
```

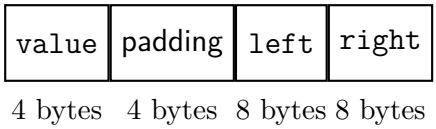

Figure 7: Layout of `Node`

[22] *Information technology - Programming Languages - C*, § 6.7.3.2.
[23] *Information technology - Programming Languages - C*, § 6.3.2.3.
[24] *Information technology - Programming Languages - C*, § 6. 7.

Our definition of `sum` takes a pointer to a `struct Node` and, utilizing basic null checks, recursively sums the values of all nodes in the provided tree.

```c
int sum(struct Node *node) {
  int acc = node->value;
  if (node->left != NULL) {
    acc += sum(node->left);
  }
  if (node->right != NULL) {
    acc += sum(node->right);
  }
  return acc;
}
```

To construct our tree, we use the `malloc` function from C's standard library to allocate memory on the heap for each of our nodes. The C standard provides only broad guidance on the semantics of this allocation (or subsequent deallocation with `free`), so implementation design falls on the implementer, such as glibc.[25] Once memory is allocated, we then use a compound literal to define our nodes and write them to the allocated memory.[26]

```c
int main() {
  struct Node *tree = malloc(sizeof(struct Node));
  *tree = (struct Node){
    .value = 1,
    .left = malloc(sizeof(struct Node)),
    .right = malloc(sizeof(struct Node))
  };
  *tree->left = (struct Node){.value = 2, .left = NULL, .right = NULL};
  *tree->right = (struct Node){.value = 3, .left = NULL, .right = NULL};
```

When calling `sum`, we pass our pointer to the root node and get out an integer result.

```c
  int total = sum(tree); // 6
```

[25] *Information technology - Programming Languages - C*, § 7.24.3.
[26] *Information technology - Programming Languages - C*, § 6.5.3.6.

To remove our right node, we first free the memory used by the right node to prevent a memory leak, then we assign the `right` field to `NULL` to represent the absence of a node.[27]

```
  free(tree->right);
  tree->right = NULL; // tree is now Node { value: 1, left: ..., right:
  ↪  NULL }
```

Finally, to deconstruct the tree and take ownership of the left node, we first assign `tree->left` to a new variable `tmp`, leaving us with a pointer to the left node in `left` and leaving our tree still intact. To fully deallocate the tree, we must manually call `free` on each of the nodes in the tree, which currently consists only of the root and its left child.

```
  struct Node *tmp = tree->left;
  free(tree);
  tree = tmp; // tree is now Node { value: 2, left: NULL, right: NULL }
}
```

In addition, since C uses null-terminated strings (a C string is a pointer to a sequence of bytes that ends with a null byte), C programmers cannot reference substrings without copying them into a new buffer. As such, we illustrate our AST example using owned strings rather than references.[28] Since C does not have built-in tagged unions, we define our AST using a struct, a union, and an enum to represent the various node types.

First, we define `AstKind` to represent the type of each AST node; this is the tag for our tagged union.

```
#include <stdlib.h>

enum AstKind { ADD, MUL, NUM };
```

Next, we define our `AstData` union to represent the data for each of our node types.

[27] *Information technology - Programming Languages - C*, § 7.24.3.3.
[28] *Information technology - Programming Languages - C*, § 7.1.1.

```c
union AstData {
  struct {
    struct Ast *left;
    struct Ast *right;
  } add;
  struct {
    struct Ast *left;
    struct Ast *right;
  } mul;
  char *num;
};
```

Finally, we define our `Ast` struct to contain both our tag and union.

```c
struct Ast {
  enum AstKind kind;
  union AstData data;
};
```

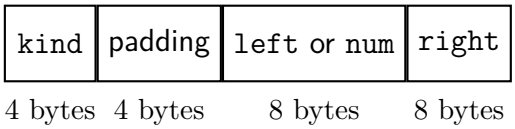

Figure 8: Layout of `Ast`

To evaluate, we then use the `switch` statement to branch based on the tag and utilize C's built-in `atoi` function to parse integer values as needed.[29]

```c
int eval(struct Ast *expr) {
  switch (expr->kind) {
  case ADD:
    return eval(expr->data.add.left) + eval(expr->data.add.right);
  case MUL:
    return eval(expr->data.mul.left) * eval(expr->data.mul.right);
  case NUM:
    return atoi(expr->data.num);
  default:
    return exit(1), 0; // unreachable)
  }
}
```

[29] *Information technology - Programming Languages - C*, § 7.24.1.2.

```c

int main() {
  struct Ast *expr = malloc(sizeof(struct Ast));
  *expr = (struct Ast){
    .kind = ADD,
    .data.add = {
      .left = &(struct Ast){.kind = NUM, .data.num = "2"},
      .right = &(struct Ast){
        .kind = MUL,
        .data.mul = {
        .left = &(struct Ast){.kind = NUM, .data.num = "3"},
          .right = &(struct Ast){
            .kind = ADD,
            .data.add = {
              .left = &(struct Ast){.kind = NUM, .data.num = "4"},
              .right = &(struct Ast){.kind = NUM, .data.num = "5"}
            }
          }
        }
      }
    }
  };
  int result = eval(expr); // 29
}
```

If we wish to modify an `Ast`, we can use similar patterns as before. For example, to replace the left node of an `ADD`, we would use the following:

```c
// ensure we have an ADD node, otherwise we could be writing to an
↪  incorrect field
if (expr->kind == ADD) {
  free(expr->data.add.left); // free the old left node to prevent a memory
  ↪  leak
  expr->data.add.left = malloc(sizeof(struct Ast));
  *expr->data.add.left = (struct Ast){.kind = NUM, .data.num = "4"};
}
```

2.3 Python

Python is a dynamically typed, garbage-collected, interpreted language. In CPython (the principal and reference implementation), it uses reference counting with optional detection of cyclically linked garbage through a garbage

collector.[30] Python allows for recursive data types through classes, which can be defined using the `class` keyword.

For our first example, we use a class `Node` and define our two fields, a constructor (the `__init__` method), and use the `None` object to represent the absence of a node.[31]

```
1 class Node:
2     def __init__(self, value, left = None, right = None):
3         self.value = value
4         self.left = left
5         self.right = right
```

Since Python is dynamically typed, all variables store the `id` (which is the memory address in CPython) of a reference-counted object on the heap. Thus, there is no need to define specific fields or provide any type information to Python since this is all handled at runtime. As a consequence of this, Python class instances, by default, use a hash map (`__dict__`) mapping field names to values (object IDs) rather than fixed slots in memory. While this provides complete flexibility, it comes with significant memory and performance overhead compared to storing fields directly.[32]

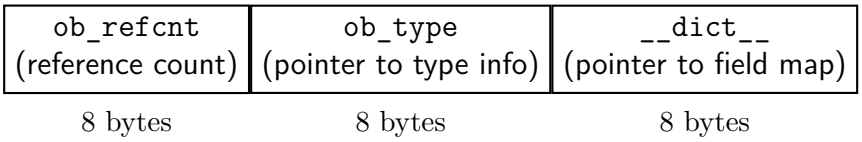

Figure 9: Simplified layout of a Python class instance

However, Python recognizes the performance implications of using a hash map for all field accesses and provides an alternative mechanism to define fixed fields using the `__slots__` attribute. By defining `__slots__`, Python will allocate space for each field inline in the object rather than using a hash

[30] *3. Data model — Python 3.13.7 documentation.*
[31] *9. Classes — Python 3.13.7 documentation.*
[32] *3. Data model — Python 3.13.7 documentation*; *Type Object Structures - Python 3.13.7 documentation*; *Common Object Structures - Python 3.13.7 documentation.*

map, improving both memory usage and performance, but at the cost of removing the ability to add new fields at runtime.[33]

```python
class NodeSlots:
    __slots__ = ['value', 'left', 'right']
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

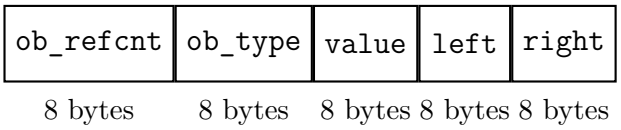

Figure 10: Simplified layout of a Python class instance with `__slots__`

Our definition of `sum` takes a `Node` and, utilizing the Python `is not` operator, checks for `None` to recursively sum the values of all nodes in the provided tree.

```python
def sum(node):
    acc = node.value
    if node.left is not None:
        acc += sum(node.left)
    if node.right is not None:
        acc += sum(node.right)
    return acc
```

To construct our tree, we call the `Node` constructor, passing in the value and optionally the left and right nodes (which default to `None` if not provided). When allocating memory for objects, Python's memory manager first turns to the `pymalloc` allocator, a specialized allocator optimized for objects of size 512 bytes or less, which functions very similarly to glibc's `malloc` and `free`, utilizing arenas and pools (bins) to efficiently allocate blocks of memory while minimizing fragmentation. If the request is too large for `pymalloc`, Python falls back to the system allocator (i.e., `malloc` and

[33] *3. Data model — Python 3.13.7 documentation*, § 3.3.2.4.

`free` from glibc on Linux). Since CPython uses the system allocator and memory addresses as object IDs, no memory compaction is performed, as this would be incompatible with system allocator usage.[34]

```
tree = Node(1, Node(2), Node(3))
```

When calling `sum`, we pass our tree and get out an integer result.

```
total = sum(tree) # 6
```

To remove our right node, we assign `None` to the `right` field of our tree, causing the reference count of the original right node to drop to zero, and thus causing it to be deallocated by Python's memory manager.

```
tree.right = None # tree is now Node(1, Node(2), None)
```

Finally, to deconstruct the tree into our left node, we assign the `left` field to our `tree` variable, causing the reference count of the original `tree` to drop to zero. However, the reference count of the left node stays at one since it is now being referred to by `tree`.

```
tree = tree.left # tree is now Node(2)
```

Similarly, we illustrate our AST example using classes. Since Python strings are immutable and do not allow storing zero-copy references to substrings, we use owned strings. Since Python does not have tagged unions, we will use Python classes to define our AST, using the built-in `isinstance` function to determine the type of each node.[35]

First, we define each class similarly to our `Node` class, with each containing the appropriate fields and constructor.

```
class Add:
    def __init__(self, left, right):
        self.left = left
        self.right = right
class Mul:
```

[34] *Memory Management in Python - Python 3.13.7 documentation.*
[35] *9. Classes — Python 3.13.7 documentation.*

```
    def __init__(self, left, right):
        self.left = left
        self.right = right
class Num:
    def __init__(self, num):
        self.num = num
```

When evaluating, we then use `isinstance` to determine the type of the node and recursively evaluate the appropriate fields, utilizing Python's built-in `int` function to parse integer values from strings stored in `Num`.[36]

```
def eval(expr):
    if isinstance(expr, Add):
        return eval(expr.left) + eval(expr.right)
    elif isinstance(expr, Mul):
        return eval(expr.left) * eval(expr.right)
    elif isinstance(expr, Num):
        return int(expr.num)

expr = Add(
    Num("2"),
    Mul(
        Num("3"),
        Add(
            Num("4"),
            Num("5")
        )
    )
)
result = eval(expr) # 29
```

2.4 Haskell

Haskell is a statically typed, compiled, functional language that relies on garbage collection when using the Glasgow Haskell Compiler (GHC), the principal implementation of Haskell.[37] Rather than using structs, unions, enums, or classes, Haskell uses algebraic data types to define custom types, defined using the `data` keyword, that allow the definition of both product

[36] *4. Defining Functions — Python 3.13.7 documentation.*
[37] *The Glasgow Haskell Compiler User's Guide*, § 5.7.

and sum types. To write our tree example in Haskell, we will use a data type `Node` and define our two fields.[38]

Since all values in Haskell are boxed by default (i.e., stored on the heap and accessed through a pointer), there is no need for any special pointer types to achieve indirection. Haskell also provides the `Maybe` type, a built-in type that can be either `Just a` or `Nothing`, to represent optional values.[39]

```haskell
data Node = Node {
  value :: Int,
  left :: Maybe Node,
  right :: Maybe Node
}
```

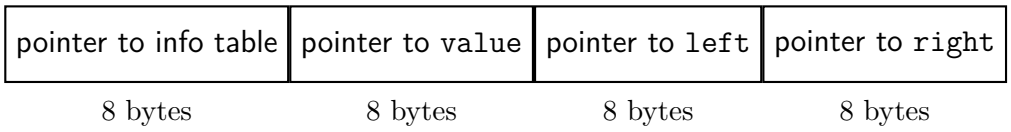

Figure 11: Layout of `Node`

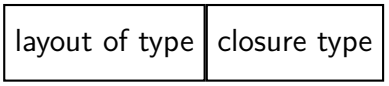

Figure 12: Simplified layout of info table

By using the closure type and layout fields of the info table, Haskell can determine at runtime that the closure (object) is a `Node` and that it is composed of three fields, all pointers to other closures.[40]

We define our `sum` function as `sumTree` to avoid overriding Haskell's built-in `sum` function. It then takes a `Node` and, utilizing Haskell's pattern match-

[38] Marlow, *Haskell 2010 Language Report*, chap. 4.

[39] *heap objects - Wiki - Glasgow Haskell Compiler / GHC - GitLab*; Marlow, *Haskell 2010 Language Report*, chap. 21.

[40] *heap objects - Wiki - Glasgow Haskell Compiler / GHC - GitLab*.

ing, we unpack the optional left and right nodes to recursively sum the values of the entire tree.

```haskell
sumTree :: Node -> Int
sumTree (Node value Nothing Nothing) = value
sumTree (Node value (Just left) Nothing) = value + sumTree left
sumTree (Node value Nothing (Just right)) = value + sumTree right
sumTree (Node value (Just left) (Just right)) = value + sumTree left +
  ↪  sumTree right
```

To construct our tree, we call the `Node` constructor, passing in the value and the left and right nodes as either `Just Node` or `Nothing`. When allocating memory for objects, GHC utilizes a generational garbage collector that divides the heap into multiple generations, thereby optimizing for the common case of short-lived objects. Objects are first allocated in a nursery before being promoted to older generations when the nursery fills up. The garbage collector periodically scans the heap to reclaim memory used by objects that are no longer reachable, copying live objects to new areas of memory to reduce fragmentation and enable the block allocator to perform compaction on empty blocks of memory.[41]

```haskell
leftNode :: Node -> Maybe Node
leftNode (Node _ l _) = l
main :: IO ()
main = do
  let tree = Node 1 (Just (Node 2 Nothing Nothing)) (Just (Node 3 Nothing
  ↪  Nothing))
```

When summing, we call our `sum` function with our tree.

```haskell
  let total = sumTree tree -- 6
```

Since Haskell is a functional language, with immutable data types, to remove the right node, we create a new tree `tree2` using Haskell's record update syntax. Since Haskell does not allow variable reassignment, we must create a new variable rather than reassigning to `tree`.

[41] Marlow et al., *Parallel Generational-Copying Garbage Collection with a Block-Structured Heap*.

```
let tree2 = tree { right = Nothing } -- tree2 is now Node 1 (Just (Node
↪  2 Nothing Nothing)) Nothing
```

Finally, to deconstruct the tree, we define a helper function `leftNode` that extracts the left node from a `Node` and then use it to assign the left node to a new variable `tree3`

```
let tree3 = leftNode tree -- tree3 is now Just (Node 2 Nothing Nothing)
print total
```

Since Haskell uses garbage collection, there is no need to free any memory manually, and the garbage collector automatically reclaims all memory used by the tree once it is no longer reachable, which will be after the `main` function ends.

Similarly, we illustrate our AST example using algebraic data types and pattern matching. Since Haskell strings are immutable, we use owned strings rather than zero-copy references to substrings.[42]

First, we define our AST using a single algebraic data type with three variants, each containing the required fields.

```
data Ast = Add Ast Ast | Mul Ast Ast | Num String
```

Each of the three variants are assigned a unique type in memory, allowing Haskell to determine which variant is being used through the closure type stored in the info table of the object.[43]

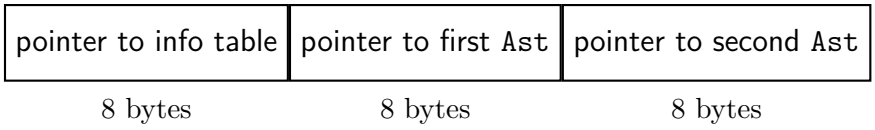

Figure 13: Layout of `Add`/`Mul`

[42] Marlow, *Haskell 2010 Language Report*, chap. 6 §1.2.
[43] *heap objects - Wiki - Glasgow Haskell Compiler / GHC - GitLab.*

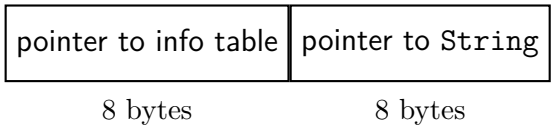

Figure 14: Layout of `Num`

To evaluate, we use Haskell's pattern matching syntax to branch over the variants and recursively evaluate the appropriate fields, utilizing Haskell's `read` function to parse integer values from the strings stored in `Num` variants.[44]

```haskell
eval :: Ast -> Int
eval (Add left right) = eval left + eval right
eval (Mul left right) = eval left * eval right
eval (Num num) = read num :: Int
main :: IO ()
main = do
    let expr = Add (Num "2") (Mul (Num "3") (Add (Num "4") (Num "5")))
    let result = eval expr -- 29
    print result
```

3 Conclusion

Every language we have examined has offered a different set of features for memory management and recursive data types. C relies directly on raw pointers and manual memory management, providing maximum control but providing no protection against double-free, use-after-free, or memory leaks. On the opposite end of the spectrum, Haskell provides automatic garbage collection with built-in protection against reference cycles through its generational design. In the middle, we have languages like Rust, which guarantee memory safety through its ownership and borrowing system, preventing use-after-free and double-free vulnerabilities at compile time. Finally, Python provides automatic memory management through reference counting but does not have cycle detection by default.

[44] Marlow, *Haskell 2010 Language Report*, chap.6 §3.3.

Language	Memory Management	Fragmentation	Allocator
Rust	Ownership & borrowing, reference counting	Yes, handled by system allocator	System (i.e., glibc)
C	Manual	Yes, handled by system allocator	System (i.e., glibc)
Python	Reference counting	Yes, handled by system allocator	`pymalloc`, system (i.e., glibc)
Haskell	Garbage collection	No, handled by garbage collector	Generational garbage collector

Table 1: Summary of memory management

Language	Data Types	Type Indirection	Optional Values
Rust	Structs, enums	`Box<T>`, `Rc<T>`, `Arc<T>`, `&T`	`Option<T>`
C	Structs, unions, enums	Raw pointers	Null Pointers
Python	Classes	Object references	`None`
Haskell	Algebraic data types	Boxed values	`Maybe a`

Table 2: Summary of data types

However, each memory management strategy comes with varying degrees of runtime overhead. Manual memory management, as in C, has no overhead beyond the cost of allocation and deallocation. Rust's ownership and borrowing system also has no runtime overhead since all code compiles down to manual memory management. However, reference counting through `Rc` and `Arc` incurs the overhead of incrementing and decrementing the reference count on clones and drops, with `Arc` also incurring the additional overhead of atomic operations to enable multi-threaded use. Python's reference counting also has the overhead of incrementing and decrementing the reference

count, but unlike Rust, it has an optional garbage collector to detect cyclic references. Finally, Haskell's garbage collector has the most overhead since it must periodically stop the program to scan the heap for unreachable objects. However, Haskell's generational design allows it to optimize for the most common case of short-lived objects, enabling it to scan only certain generations rather than the entire heap on every run.

Language	Runtime Overhead	Safety Guarantees	Memory Footprint
Rust (Ownership & Borrowing)	None	Prevents use-after-free and double-free at compile time, cycles cannot exist	Data
Rust (Rc/Arc)	Minor	Prevents use-after-free and double-free at compile time, but no cycle detection	Pointer to ref count + data
C	None	No protection against use-after-free, double-free, or memory leaks, no protection against cycles	Data
Python	Minor	Prevents use-after-free and double-free, but no cycle detection by default	Pointer to ref count + object + hash map + pointers to fields
Python (with GC)	Significant	Prevents use-after-free and double-free, prevents cycles	Pointer to ref count + object + hash map + pointers to fields
Haskell	Significant	Prevents use-after-free and double-free, prevents cycle	Pointer to info table + pointers to fields

Table 3: Summary of memory management and runtime overhead

Overall, the choice of language and memory management strategy de-

pends on the specific requirements of the application. For maximum control and performance, manual memory management in a language like C is the best choice; however, it comes with the potential for memory safety vulnerabilities. For safety and performance, but lacking the flexibility of raw pointers, Rust's ownership and borrowing system is ideal. For ease of use, the dynamic typing and automatic memory management of a language like Python are a great choice. However, they come with the overhead of an interpreter and reference counting, alongside the lack of type safety. Finally, for maximum safety and ease of use, a garbage-collected language like Haskell is an excellent choice. However, it comes with significant runtime overhead and lacks the control of manual memory management.

References

3. Data model — Python 3.13.7 documentation. URL: https://docs.python.org/3/reference/datamodel.html.

4. Defining Functions — Python 3.13.7 documentation. URL: https://docs.python.org/3/library/functions.html.

9. Classes — Python 3.13.7 documentation. URL: https://docs.python.org/3/tutorial/classes.html.

Arc in std::sync - Rust. URL: https://doc.rust-lang.org/stable/std/sync/struct.Arc.html.

Common Object Structures - Python 3.13.7 documentation. URL: https://docs.python.org/3/c-api/structures.html.

heap objects - Wiki - Glasgow Haskell Compiler / GHC - GitLab. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects.

Information technology - Programming Languages - C. ISO/IEC 9899:2024. International Organization for Standardization. 2024.

MallocInternals - glibc wiki. URL: https://sourceware.org/glibc/wiki/MallocInternals.

Marlow, Simon. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

Marlow, Simon et al. *Parallel Generational-Copying Garbage Collection with a Block-Structured Heap*. URL: https://simonmar.github.io/bib/papers/parallel-gc.pdf.

Memory Management in Python - Python 3.13.7 documentation. URL: https://docs.python.org/3/c-api/memory.html.

Rust by Example. URL: https://doc.rust-lang.org/stable/rust-by-example.

std::boxed - Rust. URL: https://doc.rust-lang.org/stable/std/boxed.

std::option - Rust. URL: https://doc.rust-lang.org/stable/std/option.

std::rc - Rust. URL: https://doc.rust-lang.org/stable/std/rc.

str - Rust. URL: https://doc.rust-lang.org/stable/std/primitive.str.html.

System in std::alloc - Rust. URL: https://doc.rust-lang.org/stable/std/alloc/struct.System.html.

The Glasgow Haskell Compiler User's Guide. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/.

The Rust Programming Language. URL: https://doc.rust-lang.org/stable/book.

The Rust Reference. URL: https://doc.rust-lang.org/stable/reference.

Type Object Structures - Python 3.13.7 documentation. URL: https://docs.python.org/3/c-api/typeobj.html.