Table of Contents

1 unit
1.1 Understanding Unit Type
1.2 Unit Value Creation
1.3 Working with Side-Effect Functions
1.4 String Representation and Debugging
1.5 Generic Programming with Unit
1.6 Built-in Trait Implementations
1.7 Practical Use Cases
1.7.1 Result Accumulation
1.7.2 Builder Pattern Termination

unit

The unit package provides functionality for working with the singleton type Unit
, which represents computations that produce side effects but return no meaningf
ul value. This is a fundamental type in functional programming for operations li
ke I/O, logging, and state modifications.

Understanding Unit Type

The Unit type has exactly one value: (). This might seem trivial, but it serve
s important purposes in type systems:

- **Side Effect Indication**: Functions returning Unit signal they’re called for side effects
- **Placeholder Type**: Used when a type parameter is needed but no meaningful value e xists
- **Functional Programming**: Represents "no useful return value" without using null o r exceptions
- **Interface Consistency**: Maintains uniform function signatures in generic contexts

Unit Value Creation

The unit value can be created in multiple ways:

```
1
2   test "unit construction" {
3
4     let u1 = ()
5
6
7     let u2 = @unit.default()
8     fn println(_ : String) {
9
10    }
11
12    inspect(u1 == u2, content="true")
13
14
15    fn log_message(msg : String) -> Unit {
16
17      println(msg)
18      ()
19    }
20
21    let result = log_message("Hello, world!")
22    inspect(result, content="()")
23  }
```

Working with Side-Effect Functions

Functions that return Unit are typically called for their side effects:

```
1
2   test "side effect patterns" {
3     let numbers = [1, 2, 3, 4, 5]
4     fn println(_ : Int) {
5
6     }
7
8     let processing_result = numbers.fold(init=(), fn(_acc, n) {
9
10      if n % 2 == 0 {
11
12        println(n)
13      }
14      ()
15    })
16    inspect(processing_result, content="()")
17
18
19    numbers.each(fn(n) { if n % 2 == 0 { println(n) } })
20  }
```

String Representation and Debugging

Unit values have a standard string representation for debugging:

```
1
2   test "unit string conversion" {
3     let u = ()
4     inspect(u.to_string(), content="()")
5
6
7     fn perform_operation() -> Unit {
8
9       ()
10    }
11
12    let result = perform_operation()
13    let debug_msg = "Operation completed: \{result}"
14    inspect(debug_msg, content="Operation completed: ()")
15  }
```

Generic Programming with Unit

Unit is particularly useful in generic contexts where you need to represent "no meaningful value":

```
1
2   test "generic unit usage" {
3
4     let items = [1, 2, 3, 4, 5]
5
6
7     items.each(fn(x) {
8
9       let processed = x * 2
10      assert_true(processed > 0)
11    })
12
13
14    let completion_status = ()
15    inspect(completion_status, content="()")
16
17
18    let operation_result : Result[Unit, String] = Ok(())
19    inspect(operation_result, content="Ok(())")
20  }
```

Built-in Trait Implementations

Unit implements essential traits for seamless integration with MoonBit's type sy
stem:

```
1
2   test "unit trait implementations" {
3     let u1 = ()
4     let u2 = ()
5
6
7     inspect(u1 == u2, content="true")
8
9
10    inspect(u1.compare(u2), content="0")
11
12
13    let h1 = u1.hash()
14    let h2 = u2.hash()
15    inspect(h1 == h2, content="true")
16
17
18    let u3 = Unit::default()
19    inspect(u3 == u1, content="true")
20  }
```

Practical Use Cases

Result Accumulation

```

  test "result accumulation" {

    let operations = [
      fn() { () },
      fn() { () },
      fn() { () },
    ]
    let final_result = operations.fold(init=(), fn(acc, operation) {
      operation()
      acc
    })
    inspect(final_result, content="()")
  }
```

Builder Pattern Termination

```

  test "builder pattern" {

    let settings = ["debug=true", "timeout=30"]


    fn apply_config(config_list : Array[String]) -> Unit {

      let _has_settings = config_list.length() > 0
      ()
    }

    let result = apply_config(settings)
    inspect(result, content="()")
  }
```

The Unit type provides essential functionality for representing "no meaningful r
eturn value" in a type-safe way, enabling clean functional programming patterns
and consistent interfaces across MoonBit code.