Table of Contents

unit

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

Understanding Unit Type

The Unit type has exactly one value: (). This might seem trivial, but it serves 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 exists
- **Functional Programming**: Represents "no useful return value" without using null or exceptions
- **Interface Consistency**: Maintains uniform function signatures in generic contexts

Unit Value Creation

The unit value can be created in multiple ways:

```

test "unit construction" {

  let u1 = ()


  let u2 = @unit default()
  fn println(_ : String) {

  }

  inspect(u1 == u2, content="true")


  fn log_message(msg : String) -> Unit {

    println(msg)
    ()
  }

  let result = log_message("Hello, world!")
  inspect(result, content="()")
}
```

Working with Side-Effect Functions

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

```

test "side effect patterns" {
  let numbers = [1, 2, 3, 4, 5]
  fn println(_ : Int) {

  }

  let processing_result = numbers fold(init=(), fn(_acc, n) {

    if n % 2 == 0 {

      println(n)
    }
    ()
  })
  inspect(processing_result, content="()")


  numbers each(fn(n) { if n % 2 == 0 { println(n) } })
}
```

String Representation and Debugging

Unit values have a standard string representation for debugging:

```

test "unit string conversion" {
  let u = ()
  inspect(u to_string(), content="()")


  fn perform_operation() -> Unit {

    ()
  }

  let result = perform_operation()
  let debug_msg = "Operation completed: \{result}"
  inspect(debug_msg, content="Operation completed: ()")
}
```

Generic Programming with Unit

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

```

test "generic unit usage" {

  let items = [1, 2, 3, 4, 5]


  items each(fn(x) {

    let processed = x * 2
    assert_true(processed > 0)
  })


  let completion_status = ()
  inspect(completion_status, content="()")


  let operation_result : Result[Unit, String] = Ok(())
  inspect(operation_result, content="Ok(())")
}
```

Built-in Trait Implementations

Unit implements essential traits for seamless integration with MoonBit’s type system:

```

test "unit trait implementations" {
  let u1 = ()
  let u2 = ()


  inspect(u1 == u2, content="true")


  inspect(u1 compare(u2), content="0")


  let h1 = u1 hash()
  let h2 = u2 hash()
  inspect(h1 == h2, content="true")


  let u3 = Unit::default()
  inspect(u3 == u1, content="true")
}
```

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 return value" in a type-safe way, enabling clean functional programming patterns and consistent interfaces across MoonBit code.