CSC 347 - Concepts of Programming Languages

Metaprogramming

Instructor: Stefan Mitsch



Learning Objectives

- Combine parametric polymorphism and type bounds/subtyping relationships to let compiler execute algorithms
- Limits of type inference



Example: Peano Arithmetic

- Peano arithmetic: recursive definition of the natural numbers including 0
- Base case 0, every other number is the successor of a Peano number
- Ordinary recursive ADT

```
{withdialog}
enum Nat:
  case Zero
  case Succ(n: Nat)
end Nat

import Nat.*

val _0 = Zero
val _1 = Succ(_0)
val _2 = Succ(_1)

{exercise}
// Exercise 1: Create an infinite stream of Peano numbers
{solution}
val peano =
  def peanoList(current: Nat) : LazyList[Nat] = current #:: peanoList(Succ(current))
  peanoList(Zero)
end peano
{endexercise}
```



Peano Arithmetic at Compile Time with Types

- Base case is a class named `_0`
- Every other number is a class `Succ` parameterized with another Peano number

```
{withdialog}
trait Nat
class _0 extends Nat
class Succ[A <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]

// 1. Compile, what happens?
// 2. Try comparing _0 == _0, _1 == _1, _0 < _1, what happens?
// 3. Try instantiating _2, what happens?
// 4. Try comparing new _0 == new _0, what happens?
```



Comparison

Operation `<` compares 2 natural numbers

- 0 is smaller than a successor of any number, no number is smaller than 0
- The successor of `x` is smaller than the successor of `y` if `x` is smaller than `y`

- Ordinary code: function `def <`

```
{withdialog}
{dialogcode}
enum Nat:
  case Zero
  case Succ(n: Nat)

import Nat.*

val _0 = Zero
val _1 = Succ(_0)
val _2 = Succ(_1)
{enddialogcode}
def <(a: Nat, b: Nat) : Boolean = (a, b) match
  case (Zero, Succ(_))    => true
  case (_, Zero)          => false
  case (Succ(x), Succ(y)) => <(x, y)
end <
```

- `<(_0, _1)` evaluates to `true`

- As a type: interface `trait <`

```
{withdialog}
{dialogcode}
// turn off worksheet mode
trait Nat
class _0 extends Nat
class Succ[A <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]
{enddialogcode}
trait <[A <: Nat, B <: Nat]

object < {
  given [B <: Nat] => <[_0, Succ[B]]
  given [A <: Nat, B <: Nat] => <[A, B] => <[Succ[A], Succ[B]]
  // helper method to summon an instance (a proof)
  def apply[A <: Nat, B <: Nat](using lt: <[A, B]) = lt
}

{dialogcode}
object test:
  @main def main() =
```



Addition

Operation `+` takes 2 natural numbers and returns their sum, recursively

- 0+0 is 0, 0+x is x, x+0 is x
- Successor of `x` plus successor of `y` is the successor of the successor of `x+y`

- Recursive function `plus`

```
{withdialog}
{dialogcode}
enum Nat:
  case Zero
  case Succ(n: Nat)

import Nat.*

val _0 = Zero
val _1 = Succ(_0)
val _2 = Succ(_1)
{enddialogcode}
def plus(a: Nat, b: Nat) : Nat = (a, b) match
  case (Zero, Zero) => Zero
  case (Zero, x)    => x
  case (x, Zero)    => x
  case (Succ(x), Succ(y)) => Succ(Succ(plus(x, y)))
end plus
```

Type `plus` checks additions

```
trait plusP[A <: Nat, B <: Nat, R <: Nat]
```

`given` objects spawn checks

```
{withdialog}
{dialogcode}
// turn off worksheet mode
trait Nat
class _0 extends Nat
class Succ[A <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]

trait plusP[A <: Nat, B <: Nat, R <: Nat]
{enddialogcode}
object plusP {
  given plusP[_0, _0, _0]
  given [B <: Nat] => plusP[_0, B, B]
  given [A <: Nat] => plusP[A, _0, A]
  given [A <: Nat, B <: Nat, S <: Nat] =>
    plusP[A, B, S] =>
```

Compute result



Addition: Compute Result

- Prove/disprove conjectured result vs. compute result

Recursive function `plus`

Type `plusP` checks addition

```
1 trait plusP[A <: Nat, B <: Nat, R <: Nat]
```

`given` objects spawn checks

```
 1 object plusP {
 2   given plusP[_0, _0, _0]
 3   given [B <: Nat] => plusP[_0, B, B]
 4   given [A <: Nat] => plusP[A, _0, A]
 5   given [A <: Nat, B <: Nat, S <: Nat]
 6     => plusP[A, B, S]
 7     => plusP[Succ[A], Succ[B], Succ[Succ[S]]]
 8
 9   // public interface to check addition
10   def apply[A <: Nat, B <: Nat, S <: Nat]
11     (using p: plusP[A, B, S]) = p
12 }
```

Type `plus` computes result

```
1 trait plus[A <: Nat, B <: Nat] extends Nat:
2   type Result <: Nat // deferred given
3 end plus
```

`given` objects spawn computation

```
 1 {withdialog}
 2 {dialogcode}
 3 // turn off worksheet mode
 4 trait Nat
 5 class _0 extends Nat
 6 class Succ[A <: Nat] extends Nat
 7
 8 type _1 = Succ[_0]
 9 type _2 = Succ[_1]
10
11 trait plus[A <: Nat, B <: Nat] extends Nat:
12   type Result <: Nat // deferred given
13 end plus
14 {enddialogcode}
15 object plus {
16   // base cases: 0+0=0, 0+B=0, A+0=A
17   given plus[_0, _0] { type Result = _0 }
18   given [B <: Nat] => plus[_0, B] { type Result = B }
19   given [A <: Nat] => plus[A, _0] { type Result = A }
20   // inductive case:
21   //  Succ[A] + Succ[B] = Succ[Succ[R]] if A + B = R
22   given [A <: Nat, B <: Nat, R <: Nat]
23     => (plus[A,B] { type Result = R })
24     => plus[Succ[A], Succ[B]] { type Result = Succ[Succ[R]] }
25
26   // public interface to compute addition
27   def apply[A <: Nat, B <: Nat]
28     (using p: plus[A, B]):
29       (plus[A, B] { type Result = p.Result } ) = p
30 }
31
32 {dialogcode}
```



Evaluate to BigInt

Traversal at runtime

```
{withdialog}
enum Nat:
  case Zero
  case Succ(n: Nat)

import Nat.*

val _0 = Zero
val _1 = Succ(_0)
val _2 = Succ(_1)

def eval(n: Nat) : BigInt = n match
  case Zero    => 0
  case Succ(i) => 1 + eval(i)
end eval
```

- Value computed at runtime

Type traversal at compile time

```
{withdialog}
trait Nat
class _0 extends Nat
class Succ[A <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]

{dialogcode}
trait plus[A <: Nat, B <: Nat] extends Nat:
  type Result <: Nat // deferred given
end plus

object plus {
  // base cases: 0+0=0, 0+B=0, A+0=A
  given plus[_0, _0] { type Result = _0 }
  given [B <: Nat] => plus[_0, B] { type Result = B }
  given [A <: Nat] => plus[A, _0] { type Result = A }
  // inductive case:
  //  Succ[A] + Succ[B] = Succ[Succ[R]] if A + B = R
  given [A <: Nat, B <: Nat, R <: Nat]
    => (plus[A,B] { type Result = R })
    => plus[Succ[A], Succ[B]] { type Result = Succ[Succ[R]] }

  // public interface to compute addition
  def apply[A <: Nat, B <: Nat]
    (using p: plus[A, B]):
      (plus[A, B] { type Result = p.Result } ) = p
}
{enddialogcode}
transparent inline def eval[N <: Nat]: BigInt =
  inline scala.compiletime.erasedValue[N] match
    case _: _0       => 0
    case _: Succ[i]  => 1 + eval[i]
end eval

{dialogcode}
object test:
  @main def main() =
    // 1. Try to compile val two = eval[_2], what happens (hover over two to inspect type)?
    // 2. Try to compile val _3 = plus[_1, _2]; val three = eval[_3.Result], what happens (hover over _3 and three to inspect types)?
    ???
  end main
end test
{enddialogcode}
```

- Values computed in compiler
- At runtime: lookup



Fibonacci Numbers

```
{withdialog}
{dialogcode}
// deactivate worksheet
trait Nat
class _0 extends Nat
class Succ[A <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]
type _3 = Succ[_2]
type _4 = Succ[_3]
type _5 = Succ[_4]
type _6 = Succ[_5]

trait plus[A <: Nat, B <: Nat] extends Nat:
  type Result <: Nat // deferred given
end plus

object plus {
  // base cases: 0+0=0, 0+B=0, A+0=A
  given plus[_0, _0] { type Result = _0 }
  given [B <: Nat] => plus[_0, B] { type Result = B }
  given [A <: Nat] => plus[A, _0] { type Result = A }
  // inductive case:
  //  Succ[A] + Succ[B] = Succ[Succ[R]] if A + B = R
  given [A <: Nat, B <: Nat, R <: Nat]
    => (plus[A,B] { type Result = R })
    => plus[Succ[A], Succ[B]] { type Result = Succ[Succ[R]] }

  // public interface to compute addition
  def apply[A <: Nat, B <: Nat]
    (using p: plus[A, B]):
      (plus[A, B] { type Result = p.Result } ) = p
}

transparent inline def eval[N <: Nat]: BigInt =
  inline scala.compiletime.erasedValue[N] match
    case _: _0       => 0
    case _: Succ[i]  => 1 + eval[i]
end eval
{enddialogcode}
trait fib[N <: Nat] extends Nat:
  type Result <: Nat
end fib

object fib {
  {exercise}
  // Exercise: implement base cases fib(0) = 0, fib(1) = 1
  {solution}
  given fib[_0] { type Result = _0 }
  given fib[_1] { type Result = _1 }
  {endexercise}
  {exercise}
  // Exercise: implement inductive case fib(Succ(Succ(N))) = fib(N) + fib(Succ(N))
  {solution}
  given [N <: Nat, R <: Nat]
    => (n: fib[N], sn: fib[Succ[N]], p: plus[n.Result, sn.Result] { type Result = R })
    => fib[Succ[Succ[N]]] { type Result = p.Result }
  {endexercise}

  // interface to get Fibonacci number
  def apply[N <: Nat](using p: fib[N]): (fib[N] { type Result = p.Result }) = p
}

{dialogcode}
object test:
  @main def main() =
    val fib3 = fib[_3] // fib(3) = 2
    val fib4 = fib[_4] // fib(4) = 3
    val fib5 = fib[_5] // fib(5) = 5
    val fib6 = fib[_6] // fib(6) = 8

    val fib6Result = eval[fib6.Result]
  end main
end test
{enddialogcode}
```



Template Metaprogramming in C++

- Uses compiler as interpreter with pure functional features
- Evaluates templates to generate values and code
- C++ templates can take values, not only types
- Example: compile-time factorial

```cpp
template <int N>
struct Factorial {
  static constexpr int value = N * Factorial<N-1>::value; // :: is static field access
}

// base case: template specialization
template <>
struct Factorial<0> {
  static constexpr int value = 0;
}

constexpr int result = Factorial<5>::value; // computes result at compile time
```



Type Constraints in C++

- Modern (C++ 20) replaces some template metaprogramming with "concepts"
- Concepts constrain type parameters, similar to type bounds in Scala
- Example: a function that takes only numeric types

```cpp
#include <concepts>

template <typename T>
requires std::integral<T> || std::floating_point<T> // constrains type T
T add(T a, T b) {
    return a + b;
}
```



Summary

- Metaprogramming uses compiler type inference to execute algorithms at compile time
- Type inference performs proofs
- Template expansion evaluates recursive template expressions
- Shifts computation from runtime to compile-time (if no user input necessary)
- Bloats compile time and size of binary