Programming in Deduce

Jeremy G. Siek

July 26, 2024

Contents

1 Introduction **2**

2 Import **2**

3 Definitions **2**

4 Printing Values **3**

5 Functions (λ) **3**

6 Unions and Switch **4**

7 Natural Numbers **6**

8 Booleans, Conditional Expressions, and Assert **6**

9 Recursive Functions **7**

10 Generic Functions **8**

11 Higher-order Functions **9**

12 Pairs **9**

13 Exercises **9**

1 Introduction

The Deduce proof assistant includes two languages, the Deduce programming language and the Deduce proof language. This booklet introduces the Deduce programming language. This language is designed so that it is straightforward to prove the correctness of programs written in Deduce. Primarily this means that Deduce does not provide side effecting operations such as modifying the value stored at a location in memory.

2 Import

The `import` feature of Deduce makes available the contents of another Deduce file in the current file. For example, the following imports Deduce's library for natural numbers from the file `Nat.pf`.

```
import Nat
```

3 Definitions

The `define` feature of Deduce associates a name with a value. The following definitions associate the name `five` with the natural number 5, and the name `six` with the natural number 6.

```
    define five = 2 + 3
    define six : Nat = 1 + five
```

Optionally, the type can be specified after the name, following a colon. In the above, `six` holds a natural number, so its type is `Nat`.

4 Printing Values

You can ask Deduce to print a value to standard output using the `print` statement.

```
print five
```

The output is `5`.

5 Functions (λ)

Functions are created with a λ expression. Their syntax starts with λ, followed by parameter names, then the body of the function enclosed in braces. For example, the following defines a function for computing the area of a rectangle.

```
define area : fn Nat,Nat -> Nat = λ h, w { h * w }
```

The type of a function starts with `fn`, followed by the parameter types, then `->`, and finally the return type.

To call a function, apply it to the appropriate number and type of arguments.

```
print area(3, 4)
```

The output is `12`.

A λ expression may only appear in a context where Deduce knows what it's type should be. The following produces an error because the following `define` does not include a type annotation.

```
define area = λ h, w { h * w }
```

Deduce prints the following error message.

```
cannot synthesize a type for λh,w{h * w}
```

6 Unions and Switch

The `union` feature of Deduce defines a type whose values are created by one or more constructors. A union definition specifies a name for the union type and its body specifies the name of each constructor and its parameter types. For example, we define the following union to represent a linked-list of natural numbers.

```
union NatList {
  nil
  cons(Nat, NatList)
}
```

We construct values of type `NatList` using the constructors `nil` and `cons`. To create a linked-list whose elements are `1` and `2`, write:

```
define NL12 = cons(1, cons(2, nil))
```

Unions may be recursive: a constructor may include a parameter type that is the union type, e.g., the `NatList` parameter of `cons`. Unions may be generic: one can parameterize a union with one or more type parameters. For example, we generalize linked lists to any element types as follows.

```
union List<T> {
  empty
  node(T, List<T>)
}
```

Constructing values of a generic union looks the same as for a regular union. Deduce figures out the type for `T` from the types of the constructor arguments.

```
define L12 = node(1, node(2, empty))
```

You can branch on a value of union type using `switch`. For example, the following function returns the first element of a `NatList`.

```
import Option

define front : fn NatList -> Option<Nat> =
  λ ls {
    switch ls {
      case nil { none }
      case cons(x, ls') { just(x) }
    }
  }
```

The output of

```
print front(NL12)
```

is `just(1)`.

The `switch` compares the discriminated value with the constructor pattern of each `case` and when it finds a match, it initializes the pattern variables from the parts of the discriminated value and then evaluates the branch associated with the `case`.

If you forget a `case` in a `switch`, Deduce will tell you. For example, if you try the following:

```
define broken_front : fn NatList -> Option<Nat> =
  λ ls { switch ls { case nil { none } } }
```

Deduce responds with

```
this switch is missing a case for: cons(Nat,NatList)
```

7 Natural Numbers

Natural numbers are not a built-in type in Deduce but instead they are defined as a `union` type:

```
union Nat {
  zero
  suc(Nat)
}
```

The file `Nat.pf` includes the above definition together with some operations on natural numbers and theorems about them. The numerals `0`, `1`, `2`, etc. are shorthand for the natural numbers `zero`, `suc(zero)`, `suc(suc(zero))`, etc.

8 Booleans, Conditional Expressions, and Assert

Deduce includes the values `true` and `false` of type `bool` and the usual boolean operations such as `and`, `or`, and `not`. Deduce also provides an if-then-else expression that branches on the value of a boolean. For example, the following if-then-else expression is evaluates to 7.

```
print (if true then 7 else 5+6)
```

The `assert` statement evaluates an expression and reports an error if the result is `false`. For example, the following `assert` does nothing because the expression evaluates to `true`.

```
assert (if true then 7 else 5+6) = 7
```

9 Recursive Functions

Recursive functions in Deduce are somewhat special to make sure they always terminate.

- The first parameter of the function must be a union.
- The function definition must include a clause for every constructor in the union.
- The first argument of every recursive call must be a sub-part of the current constructor of the union.

A recursive function begins with the `function` keyword, followed by the name of the function, then the parameters types and the return type. Finally, the function body includes one clause for every constructor of the union. Each clause is an equation whose left-hand side is the function applied to a constructor pattern and whose right-hand side is the value of the function for that case.

For example, here's the definition of a `len` function for lists of natural numbers.

```
function len(NatList) -> Nat {
  len(nil) = 0
  len(cons(n, next)) = 1 + len(next)
}
```

There are two clauses in the body. The clause for `nil` says that its length is `0`. The clause for `cons` says that its length is one more than the length of the rest of the linked list. Deduce approves of the recursive call `len(next)` because `next` is part of `cons(n, next)`.

Recursive functions may have more than one parameter but pattern matching is only supported for the first parameter. For example, here is a function `app` that combines two linked lists.

```
function app(NatList, NatList) -> NatList {
  app(nil, ys) = ys
  app(cons(n, xs), ys) = cons(n, app(xs, ys))
}
```

10 Generic Functions

Deduce supports generic functions, so we can generalize `len` to work on lists with any element type by defining the following `length` function.

```
function length<E>(List<E>) -> Nat {
  length(empty) = 0
  length(node(n, next)) = suc(length(next))
}
```

Generic functions that are not recursive can be defined using a combination of `define`, `generic`, and λ.

```
define head : < T > fn List<T> -> Option<T> =
  generic T { λ ls {
      switch ls {
        case empty { none }
        case node(x, ls') { just(x) }
      }
    }
  }
```

The type of a generic function, such as `head`, starts with its type parameters surrounded by < and >.

11 Higher-order Functions

Functions may be passed as parameters to a function and they may be returned from a function. For example, the following function checks whether every element of a list satisfies a predicate.

```
function all_elements<T>(List<T>, fn T->bool)->bool {
  all_elements(empty, P) = true
  all_elements(node(x, xs'), P) =
    P(x) and all_elements(xs', P)
}
```

12 Pairs

Pairs are defined as a `union` type:

```
union Pair<T,U> {
  pair(T,U)
}
```

The file `Pair.pf` includes the above definition and several operations on pairs, such as `first` and `second`.

13 Exercises

13.1 Sum the Elements in a List

Define a function named `sum` that adds up all the elements of a `List<Nat>`.

```
define L13 = node(1, node(2, node(3, empty)))
assert sum(L13) = 6
```

13.2 Inner Product

Define a function named `dot` that computes the inner product of two `List<Nat>`.

```
define L46 = node(4, node(5, node(6, empty)))
assert dot(L13,L46) = 32
```

13.3 Last Element in a List

Define a generic function named `last` that returns the last element of a `List<E>`, if there is one. The return type should be `Option<E>`. (`Option` is defined in the file `Option.pf`.)

```
assert last(L13) = just(3)
```

13.4 Remove Elements from a List

Define a generic function named `remove_if` that removes elements from a list if satisfy a predicate. So `remove_if` should have two parameters: (1) a `List<E>` and (2) a function whose parameter is `E` and whose return type is `bool`.

```
assert remove_if(L13, λx {x ≤ 1})
       = node(2, node(3, empty))
```

13.5 Non-empty Lists and Average

Define a `union` named `NEList` for non-empty list. Design the alternatives in the `union` to make it impossible to create an empty list.

Define a function named `average` that computes the mean of a non-empty list and check that it works on a few inputs. Note that the second parameter of the division operator `/` is of type `Pos`, which is defined in `Nat.pf`.