

Imperative Syntax for Dependent Types

Towards a user study...



Dependent Types

a.k.a. types that can depend on terms

```
data List : Type → Type where  
  Nil : List a  
  Cons : a → List a → List a
```



```
data Vect : Nat → Type → Type where  
  Nil : Vect 0 a  
  Cons : a → Vect n a → Vect (S n) a
```

“Dependent types are HARD”

“Dependent types are HARD”

But WHY?

“Dependent types are HARD”

But WHY?

And for WHOM?

“Dependent types are HARD”

But WHY?

And for WHOM?

It depends.

An Experienced Software Engineer

a.k.a our desired audience

- Likes side effects
- Is able to comprehend complex imperative programming features
- ???????

Before dependent types

a.k.a functional programming

- Most dependently typed languages use typically functional syntax
- Imperative and functional languages differ significantly in style
- While implementation details may differ, many core concepts exist within both paradigms

Before dependent types

a.k.a functional programming

- Most dependently typed languages use typically functional syntax
- Imperative and functional languages differ significantly in style
- While implementation details may differ, many core concepts exist within both paradigms
- An (important) example: Algebraic data types and pattern matching

Algebraic Data Types

In imperative languages

```
@dataclass
class Nil:
    pass
@dataclass
class Cons:
    head: int
    tail: List
List = Nil | Cons
```

Python: *dataclasses*
& *tagged unions*

```
struct Nil final {};
struct Cons final {
    int head;
    std::unique_ptr<std::variant<Nil, Cons>> tail;
};
using List = std::variant<Nil, Cons>;
```

C++: *structs & variants*

```
sealed interface List<T> {
    record Nil<T>() implements List<T> {}
    record Cons<T>(T head, List<T> tail)
    implements List<T> {}
}
```

Java: *sealed interfaces & records*

An Experienced Software Engineer

a.k.a our desired audience

- Likes side effects
- Is able to comprehend complex imperative programming features
- ???????

An Experienced Software Engineer

a.k.a our desired audience

- Likes side effects
- Is able to comprehend complex ~~imperative~~ **functional*** programming features
- ???????

Imperative syntax for an existing dependent type theory can enhance usability for an experienced programmer.

Syntax

Simple syntactic transformations

a.k.a. the low hanging fruit

```
type Vect(Nat n, Ty t) {  
  constructor Nil() of Vect(0, t);  
  constructor Cons(t head, Vect(n, t) tail) of Vect(n+1, t);  
}
```

Datatypes



```
data Vect : Nat -> Type -> Type where  
  Nil : Vect 0 t  
  Cons : (head : t) -> (tail : Vect n t) -> Vect (S n) t
```

Simple syntactic transformations

a.k.a. the low hanging fruit

```
func replicate<Ty t>(t x, Nat n) of Vect(t, n) {  
  switch(n) {  
    case 0: { return Nil; }  
    case S(n): { return Cons(x, replicate(x,n)); }  
  }}  
}
```

Pattern matching

```
replicate : {t : Type} -> t -> Nat -> Vect t n  
replicate x n = case n of  
  0 => Nil  
  S n => Cons x (replicate x n)
```


Simple syntactic transformations

a.k.a. the low hanging fruit

```
func varManip (Nat x, Nat y) of Nat {  
  let Nat z = x + y;  
  if (z < 10) {  
    z = z + 10;  
  } else {  
    z = z + 1;  
  }  
  z = z + x;  
  return z;  
}
```

*Block
statements /
variable
reassignment*



```
varManip : Nat -> Nat -> Nat  
varManip x y =  
  let z : Nat = (x + y) in  
    if (z < 10) then  
      let z : Nat = (z + 10) in  
        let z : Nat = (z + x) in  
          z  
    else  
      let z : Nat = (z + 1) in  
        let z : Nat = (z + x) in  
          z
```

Loops

a.k.a recursion

```
func f(t1 x, t2 y, ...) of t {  
  head  
  while(condition) {  
    body  
  }  
  tail  
}
```

Loops

a.k.a recursion

```
func f(t1 x, t2 y, ...) of t {  
  head  
  while(condition) {  
    body  
  }  
  tail  
}
```

```
func f(t1 x, t2 y, ...) of t {  
  head  
  f'(x, y, ..., xh, yh, ...)  
}  
where  
func f'(t1 x, t2 y, ..., th1 xh, th2 yh, ...) of t {  
  if(condition) {  
    body  
    f'(x, y, ..., xh, yh, ...)  
  } else {  
    tail  
  }  
}
```

Loops

a.k.a recursion

```
func f(t1 x, t2 y, ...) of t {  
  head  
  while(condition) {  
    body  
  }  
  tail  
}
```

```
func f(t1 x, t2 y, ...) of t {  
  head  
  f'(x, y, ..., xh, yh, ...)  
}  
where  
func f'(t1 x, t2 y, ..., th1 xh, th2 yh, ...) of t {  
  if(condition) {  
    body  
    f'(x, y, ..., xh, yh, ...)  
  } else {  
    tail  
  }  
}
```

From booleans to propositions

a.k.a enabling dependent pattern matching

```
if(x == Nil) {  
  return 0;  
} else {  
  return head(x);  
}
```



```
case (decEq x Nil) of  
  Yes prf => 0  
  No  prf => head x
```

`head : (x : List a) -> { _ : Not (x = Nil) } -> a`

Decidable Equality...

```
data Vect : (n : Nat) -> (t : Type) -> Type where
  Nil : Vect 0 t
  Cons : (head : t) -> (tail : Vect n t) -> Vect (S n) t

{n : Nat} -> {t : Type} -> (DecEq t) => DecEq (Vect n t) where
  decEq (Nil) (Nil) = Yes Refl
  decEq (Cons head1 tail1) (Cons head2 tail2) with (decEq head1 head2)
    decEq (Cons head1 tail1) (Cons head1 tail2) | Yes Refl with (decEq tail1 tail2)
      decEq (Cons head1 tail1) (Cons head1 tail1) | Yes Refl | Yes Refl = Yes Refl
      decEq (Cons head1 tail1) (Cons head1 tail2) | Yes Refl | No prf = No (\h => (prf (case h of Refl =>
Refl))))
    decEq (Cons head1 tail1) (Cons head2 tail2) | No prf = No (\h => (prf (case h of Refl => Refl)))
```

Decidable Equality for FREE

```
data SoManyArgs : (t : Type) -> Type where
  C1 : (a : t) -> (b : t) -> (c : t) -> (d : t) -> SoManyArgs t
  C2 : (x : t) -> (y : t) -> SoManyArgs t

{t : Type} -> (DecEq t) => DecEq (SoManyArgs t) where
  decEq (C1 a1 b1 c1 d1) (C1 a2 b2 c2 d2) with (decEq a1 a2)
    decEq (C1 a1 b1 c1 d1) (C1 a1 b2 c2 d2) | Yes Refl with (decEq b1 b2)
      decEq (C1 a1 b1 c1 d1) (C1 a1 b1 c2 d2) | Yes Refl | Yes Refl with (decEq c1 c2)
        decEq (C1 a1 b1 c1 d1) (C1 a1 b1 c1 d2) | Yes Refl | Yes Refl | Yes Refl with (decEq d1 d2)
          decEq (C1 a1 b1 c1 d1) (C1 a1 b1 c1 d1) | Yes Refl | Yes Refl | Yes Refl | Yes Refl = Yes Refl
          decEq (C1 a1 b1 c1 d1) (C1 a1 b1 c1 d2) | Yes Refl | Yes Refl | Yes Refl | No prf = (No (\h => (prf (case h of Refl
=> Refl)))))
        decEq (C1 a1 b1 c1 d1) (C1 a1 b1 c2 d2) | Yes Refl | Yes Refl | No prf = (No (\h => (prf (case h of Refl => Refl)))))
        decEq (C1 a1 b1 c1 d1) (C1 a1 b2 c2 d2) | Yes Refl | No prf = (No (\h => (prf (case h of Refl => Refl)))))
        decEq (C1 a1 b1 c1 d1) (C1 a2 b2 c2 d2) | No prf = (No (\h => (prf (case h of Refl => Refl)))))
      decEq (C1 a1 b1 c1 d1) (C2 x2 y2) = (No (\h => (case h of Refl impossible)))
      decEq (C2 x1 y1) (C1 a2 b2 c2 d2) = (No (\h => (case h of Refl impossible)))
      decEq (C2 x1 y1) (C2 x2 y2) with (decEq x1 x2)
        decEq (C2 x1 y1) (C2 x1 y2) | Yes Refl with (decEq y1 y2)
          decEq (C2 x1 y1) (C2 x1 y1) | Yes Refl | Yes Refl = Yes Refl
          decEq (C2 x1 y1) (C2 x1 y2) | Yes Refl | No prf = (No (\h => (prf (case h of Refl => Refl)))))
          decEq (C2 x1 y1) (C2 x2 y2) | No prf = (No (\h => (prf (case h of Refl => Refl)))))
```

Putting it all together

```
func search (Nat n, Vect(n, Nat) ls, Nat x) of Maybe(Fin(n)) {  
  let Nat i = 0;  
  let Maybe(Fin(n)) ret = Nothing;  
  ewhile(i < n) {  
    eif (index(natToFinLT(i), ls) == x) {  
      ret = Just(natToFinLT(i));  
    }  
    else { ;; }  
    i = 1 + i;  
  }  
  return ret;  
}
```


Putting it all together

```
search : (n : Nat) -> (ls : Vect n Nat) -> (x : Nat) -> Maybe (Fin n)
```

```
search n ls x =
```

```
  let i : Nat = 0 in
```

```
    let ret : Maybe (Fin n) = Nothing in
```

```
      (search_rec0 n ls x i ret)
```

```
where
```

```
  search_rec0 : (n : Nat) -> (ls : Vect n Nat) -> (x : Nat) -> (i : Nat) -> (ret : Maybe (Fin n)) -> Maybe (Fin n)
```

```
  search_rec0 n ls x i ret =
```

```
    (case (isLT i n) of
```

```
      No noprf => ret
```

```
      Yes yesprf => (case (decEq (index (natToFinLT i) ls) x) of
```

```
        No noprf => let i : Nat = (S i) in
```

```
          (search_rec0 n ls x i ret)
```

```
        Yes yesprf => let ret : Maybe (Fin n) = Just (natToFinLT i) in
```

```
          let i : Nat = S i in
```

```
            search_rec0 n ls x i ret))
```

Syntax isn't everything

a.k.a. type theory matters

- Errors!!!!
- Semantics of effects (eg. mutability)
- Interactive type-checking (explicit proofs)
- And more ...

A study

What does it look like?

- Target participants: experienced software engineers
 - Does the choice of the (imperative) language they are experienced in matter?
- A purely syntactic comparison
 - Is it possible to evaluate usability of dependent types without interactivity?

Conclusion

- Designed an imperative syntax for dependently typed programming that can be elaborated to executable Idris code
- Developed an algorithm for automatic derivation of decidable equality
- Syntax isn't everything — dedicated semantics are necessary for true usability
 - Specifically, we need a better understanding of *how* imperative programmers would make use of dependent types
- This syntax is to be evaluated via a qualitative user study