

# Types as First-Class Values

---

in the Fuzion Language

Fridtjof Siebert  
Tokiwa Software GmbH

TyDe 2023, 4. Sep 2023, Seattle



# This Talk

---

## overview

- Fuzion quick intro
- Types as Values
- Type Features
- Types to Name Effects



# Motivation: Fuzion Language

---

principles

- One concept: a **feature**
- Tools make better decisions than developers
- Systems are safety-critical



# Fuzion Quick Intro

---

Fuzion is / has / supports

- statically typed
- algebraic types
- parametric types
- inheritance and redefinition
- dynamic binding
- pure using effects



# Product Type defined as feature

---

```
point (x, y f64).
```



# Function defined as feature dsq

---

```
point (x, y f64).
```

```
dsq(x, y f64) ⇒ x*x + y*y
```



# Feature nesting

---

`point (x, y f64) is`

`dsq ⇒ x*x + y*y`



# Immutable Fields

---

```
point (x, y f64) is
```

```
  dsq := x*x + y*y
```



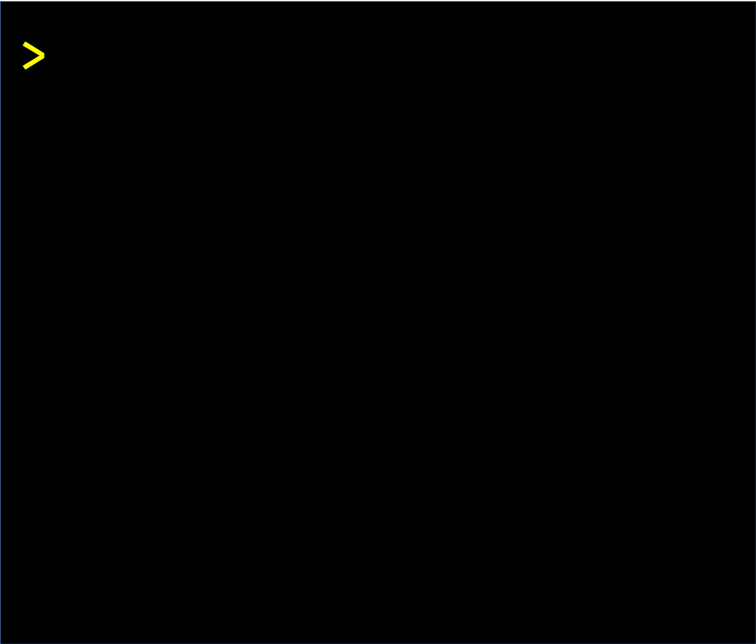


# Feature Calls

---

`point (x, y f64) is`

`dsq ⇒ x*x + y*y`



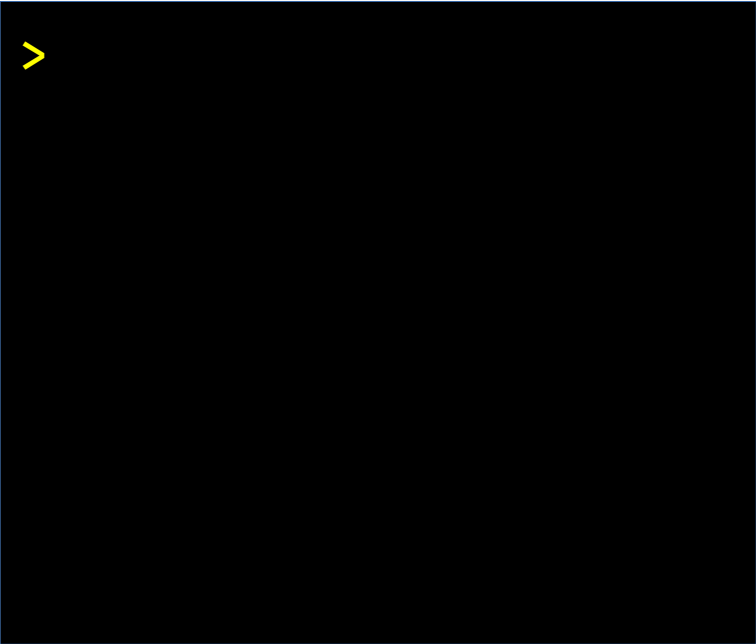
# Feature Calls

---

```
point (x, y f64) is
```

```
  dsq ⇒ x*x + y*y
```

```
p := point 3 4
```



# Feature Calls

---

```
point (x, y f64) is
```

```
  dsq ⇒ x*x + y*y
```

```
p := point 3 4
```

```
say p.dsq
```

```
> fz demo.fz
```



# Feature Calls

---

```
point (x, y f64) is
```

```
  dsq ⇒ x*x + y*y
```

```
p := point 3 4
```

```
say p.dsq
```

```
> fz demo.fz  
25.0  
>
```



# Polymorphism

---

## Three forms

- sum types
- parametric types
- dynamic binding



# Sum Types

---

```
point (x, y f64).
```



# Sum Types

---

```
point (x, y f64).
```

```
line (a, b point).
```



# Sum Types

---

```
point (x, y f64).
```

```
line (a, b point).
```

```
obj : choice point line is
```





# Sum Types

---

```
point (x, y f64).
```

```
line (a, b point).
```

```
obj : choice point line is
```

```
draw_obj(o obj) ⇒
```

```
  match o
```

```
    p point ⇒ drawPoint p.x p.y
```

```
    l line  ⇒ drawLine  l.a l.b
```



# Sum Types

---

```
point (x, y f64).
```

```
line (a, b point).
```

```
obj : choice point line is
```

```
draw_obj(o obj) ⇒
```

```
  match o
```

```
    p point ⇒ drawPoint p.x p.y
```

```
    l line  ⇒ drawLine  l.a l.b
```

```
draw_obj (point 3 4)
```

```
draw_obj (line p q)
```



# Type Parameters

---



# Abstract Features

---

obj is

draw unit is abstract



# Inheritance

---

```
obj is
```

```
draw unit is abstract
```

```
point (x, y f64) : obj is
```

```
draw ⇒ drawPoint x y
```



# Inheritance

---

```
obj is
```

```
draw unit is abstract
```

```
point (x, y f64) : obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : obj is
```

```
draw ⇒ drawLine a b
```



# Type Parameters

---

```
obj is
```

```
  draw unit is abstract
```

```
point (x, y f64) : obj is
```

```
  draw ⇒ drawPoint x y
```

```
line (a, b point) : obj is
```

```
  draw ⇒ drawLine a b
```

```
draw_obj (o T : obj) ⇒ o.draw
```



# Type Parameters

---

```
obj is
```

```
draw unit is abstract
```

```
point (x, y f64) : obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : obj is
```

```
draw ⇒ drawLine a b
```

```
draw_obj (o T : obj) ⇒ o.draw
```

```
draw_obj (point 3 4)
```

```
draw_obj (line p q)
```





# Dynamic Binding

---

```
obj is
```

```
draw unit is abstract
```

```
point (x, y f64) : obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : obj is
```

```
draw ⇒ drawLine a b
```

```
draw_obj (o T : obj) ⇒ o.draw
```

```
draw_obj (point 3 4)
```

```
draw_obj (line p q)
```



# Reference Types

---

```
Obj ref is
```

```
draw unit is abstract
```

```
point (x, y f64) : Obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : Obj is
```

```
draw ⇒ drawLine a b
```

```
draw_obj (o T : Obj) ⇒ o.draw
```

```
draw_obj (point 3 4)
```

```
draw_obj (line p q)
```



# Reference Types

---

```
Obj ref is
```

```
draw unit is abstract
```

```
point (x, y f64) : Obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : Obj is
```

```
draw ⇒ drawLine a b
```



# Reference Types

---

```
Obj ref is
```

```
draw unit is abstract
```

```
point (x, y f64) : Obj is
```

```
draw ⇒ drawPoint x y
```

```
line (a, b point) : Obj is
```

```
draw ⇒ drawLine a b
```

```
s Sequence Obj := [point 3 4, line p q, point2, line2, point3]
```

```
for o in s do
```

```
o.draw
```



# This Talk

---

## overview

- Fuzion quick intro ✓
- **Types as Values**
- Type Features
- Types to Name Effects



# Type and Value Arguments

---

Example from above

```
draw_obj (o T : obj) ⇒ o.draw
```

is syntactic sugar for

```
draw_obj (T type : obj,  
         o T      ) ⇒ o.draw
```

- a feature has type arguments and value arguments
- type arguments first, then value arguments



# Types in Fuzion

---

Defined by

→ constructor features

```
point (x, y f64).
```

→ choice features

```
obj : choice point line is
```

→ may have type parameters

```
point(T type : numeric, x, y T).
```

```
obj(T type : numeric) : choice (point T) line is
```



# Calls vs. Types

---

## Constructor pair

```
pair (T type,  
      a, b T).
```

## calls

```
p1 := pair i32 47 11  
p2 := pair String "Hello" "World!"  
p3 := pair (option f64) nil 3.14
```

## type inference

```
p1 := pair 47 11  
p2 := pair "Hello" "World!"
```





# Calls vs. Types

---

## Constructor pair

```
pair (T type,  
      a, b T).
```

types need type parameters

```
add (p pair i32) ⇒ p.a + p.b
```



# This Talk

---

## overview

- Fuzion quick intro ✓
- Types as Values ✓
- **Type Features**
- Types to Name Effects



# Type Features

---

Example sum of numeric values

```
sum_of(T type : numeric, l list T) =>
  match l
    nil      => ?
    c Cons => c.head + sum_of c.tail
```

what do we return for an empty list?



# Type Features

---

Solution: type features:

```
numeric is
...
type.zero numeric.this is abstract
type.one  numeric.this is abstract
```

implemented by heirs, e.g.,

```
i32 : numeric is
...
fixed type.zero ⇒ 0
fixed type.one  ⇒ 1
```



# Type Features

---

Use `T.zero` in `sum_of`:

```
sum_of(T type : numeric, l list T) ⇒  
  match l  
    nil      ⇒ T.zero  
    c Cons  ⇒ c.head + sum_of c.tail
```



# Type Features

---

Used directly in `numeric` for monoids `sum` and `product`:

```
numeric is
...
# monoid of numeric with infix + operation.
type.sum : Monoid numeric.this is
  infix • (a, b numeric.this) ⇒ a + b
  e ⇒ zero

type.product : Monoid numeric.this is
  infix • (a, b numeric.this) ⇒ a * b
  e ⇒ one
```



# Type Feature Inheritance

Used directly in `numeric` for monoids `sum` and `product`:

`numeric`

`prefix + numeric.this is abstract`  
`prefix - numeric.this is abstract`



`i32`

`prefix + i32 is i32.this`  
`prefix - i32 is intrinsic`

`numeric.type`

`zero numeric.this is abstract`  
`one numeric.this is abstract`  
`sum : Monoid numeric.this is`  
`...`  
`product : Monoid numeric.this is`  
`...`



`i32.type`

`zero i32 is 0`  
`one i32 is 1`



# This Talk

---

## overview

- Fuzion quick intro ✓
- Types as Values ✓
- Type Features ✓
- Types to Name Effects





# Fuzion Effects: `<type>.env`

---

Hello World:

```
hello_world =>  
  io.out.env.println "hello world!"
```

```
hello_world
```



# Required Effects in Signature

---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```



# Fuzion Effects Example

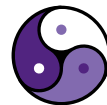
---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```

```
> fz hw.fz
```



# Fuzion Effects Example

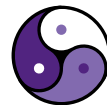
---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```

```
> fz hw.fz  
hello world!  
>
```



# Fuzion Effects Example

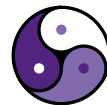
---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```

```
> fz hw.fz  
hello world!  
> fz -effects hw.fz
```



# Fuzion Effects Example

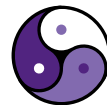
---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```

```
> fz hw.fz  
hello world!  
> fz -effects hw.fz  
io.out  
>
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
hello_world
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```





# Fuzion Effect Handlers

---

Hello World:

```
hello_world ! io.out =>  
  io.out.env.println "hello world!"
```

```
my_handler : io.Print_Handler is  
  print(s Any) =>  
    io.err.print (($s).replace "!" "!!!11!")
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out ⇒  
  io.out.env.println "hello world!"
```

```
my_handler : io.Print_Handler is  
  print(s Any) ⇒  
    io.err.print (($s).replace "!" "!!!11!")
```

```
(io.out my_handler)
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out ⇒  
  io.out.env.println "hello world!"
```

```
my_handler : io.Print_Handler is  
  print(s Any) ⇒  
    io.err.print (($s).replace "!" "!!!11!")
```

```
(io.out my_handler).use ()→hello_world
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out ⇒  
  io.out.env.println "hello world!"
```

```
my_handler : io.Print_Handler is  
  print(s Any) ⇒  
    io.err.print (($s).replace "!" "!!!11!")
```

```
(io.out my_handler).use ()→hello_world
```

```
> fz hw.fz  
hello world!!!11!  
>
```



# Fuzion Effects Example

---

Hello World:

```
hello_world ! io.out ⇒  
  io.out.env.println "hello world!"
```

```
my_handler : io.Print_Handler is  
  print(s Any) ⇒  
    io.err.print (($s).replace "!" "!!!11!")
```

```
(io.out my_handler).use ()→hello_world
```

```
> fz hw.fz  
hello world!!!11!  
> fz -effects hw.fz  
io.err  
>
```



# Fuzion Effects: mutate

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  n := mutate.env.new 0
  l.for_each x->
    n ← n.get + 1
  n.get
```



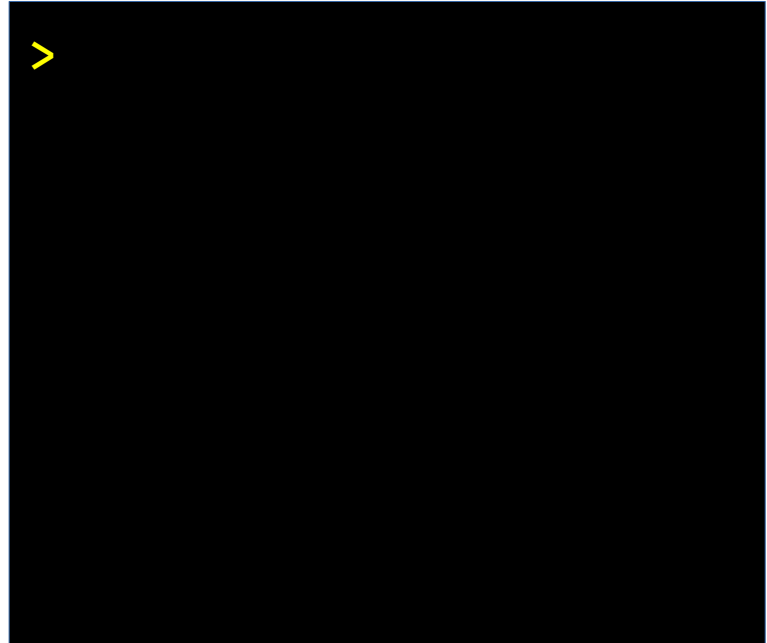
# Fuzion Effects: mutate

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  n := mutate.env.new 0
  l.for_each x->
    n ← n.get + 1
  n.get

mutate.use ()->
  say (count ((1..10).filter x->x%%2))
```



# Fuzion Effects: mutate

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  n := mutate.env.new 0
  l.for_each x->
    n ← n.get + 1
  n.get

mutate.use ()->
  say (count ((1..10).filter x->x%%2))
```

```
> fz count.fz
5
>
```





# Fuzion Effects: mutate

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  n := mutate.env.new 0
  l.for_each x->
    n ← n.get + 1
  n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
```

```
  n := mutate.env.new 0
  l.for_each x →
    n ← n.get + 1
  n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>  
  mm : mutate.  
  n := mutate.env.new 0  
  l.for_each x->  
    n ← n.get + 1  
  n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  mm : mutate.
  mm.go ()->
    n := mutate.env.new 0
    l.for_each x->
      n ← n.get + 1
    n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  mm : mutate.
  mm.go ()->
    n := mm.env.new 0
    l.for_each x->
      n ← n.get + 1
    n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ! mutate =>
  mm : mutate.
  mm.go ()→
    n := mm.env.new 0
    l.for_each x→
      n ← n.get + 1
    n.get
```



# Local Mutability

---

## Counting using a mutable field

```
count(l Sequence T) ⇒  
  mm : mutate.  
  mm.go () →  
    n := mm.env.new 0  
    l.for_each x →  
      n ← n.get + 1  
    n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(  
  l Sequence T) ⇒  
  mm : mutate.  
  mm.go ()→  
    n := mm.env.new 0  
    l.for_each x→  
      n ← n.get + 1  
  n.get
```





# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ⇒  
  mm : mutate.  
  mm.go ()→  
    n := mm.env.new 0  
    l.for_each x→  
      n ← n.get + 1  
  n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ⇒  
  mm : mutate.  
  mm.go ()→  
    n := M.env.new 0  
    l.for_each x→  
      n ← n.get + 1  
  n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ⇒
```

```
  n := M.env.new 0  
  l.for_each x →  
    n ← n.get + 1  
  n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ⇒  
  n := M.env.new 0  
  l.for_each x →  
    n ← n.get + 1  
  n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ! M ⇒  
  n := M.env.new 0  
  l.for_each x→  
    n ← n.get + 1  
  n.get
```



# Effect Parameters

---

## Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ! M ⇒  
  n := M.env.new 0  
  l.for_each x→  
    n ← n.get + 1  
  n
```



# Mutable Value w/ Parametric Type

---

Counting using a mutable field

```
count(M type : mutate,  
      l Sequence T) ! M ⇒  
  n := M.env.new 0  
  l.for_each x→  
    n ← n.get + 1  
  n
```



# Mutable Value w/ Parametric Type

---

Counting using a mutable field

```
count(n (M : mutate).new i32,  
      l Sequence T) ! M ⇒  
  n := M.env.new 0  
  l.for_each x→  
    n ← n.get + 1  
  n
```





# Mutable Value w/ Parametric Type

---

Counting using a mutable field

```
count(n (M : mutate).new i32,  
      l Sequence T) ! M ⇒  
  n := M.env.new 0  
  l.for_each x→  
    n ← n.get + 1  
  n
```



# Mutable Value w/ Parametric Type

---

Counting using a mutable field

```
count(n (M : mutate).new i32,  
      l Sequence T) ! M =>  
  l.for_each x->  
    n ← n.get + 1  
  n
```



# Mutable Value w/ Parametric Type

---

Counting using a mutable field

```
count(n (M : mutate).new i32,  
      l Sequence T) ! M =>  
  l.for_each x->  
    n ← n.get + 1  
  n
```

```
mm : mutate.  
mm.use ()->  
  cnt := mm.env.new 100  
  cnt := count mm i32 cnt [1,2,3]  
  say cnt
```



# This Talk

---

## overview

- Fuzion quick intro ✓
- Types as Values ✓
- Type Features ✓
- Types to Name Effects ✓



# Conclusion

---

Fuzion aims at unifying concepts

- types play an integral part
- parametric types and value arguments treated similarly
- types used to distinguish effects

[@fuzion@types.pl](mailto:@fuzion@types.pl)

[@FuzionLang](https://twitter.com/FuzionLang)

<https://flang.dev>

[github.com/tokiwa-software/fuzion](https://github.com/tokiwa-software/fuzion)



# Fuzion: Status

---

Fuzion still under development

- language definition slowly getting more stable
- base library work in progress
- current implementation providing JVM and C backends
- Basic analysis tools available



# Fuzion: Status

---

Fuzion still under development

- language definition slowly getting more stable
- base library work in progress
- current implementation providing JVM and C backends
- Basic analysis tools available
- Felix

