Algebraic Effects and Types

as First-Class Features in the Fuzion Language

Fridtjof Siebert
Tokiwa Software GmbH

FOSDEM, 4. Feb 2023, Brussels

Who is this guy?

Fridtjof Siebert

Email: siebert@tokiwa.software
github: fridis
twitter: @fridi_s

‘90-‘94	AmigaOberon, AMOK PD
‘97	FEC Eiffel Sparc / Solaris
‘98-‘99	OSF: TurboJ Java Compiler
‘00-‘01	PhD on real-time GC
‘02-‘19	JamaicaVM real-time JVM based on CLASSSPATH / OpenJDK, VeriFlux static analysis tool
‘20-...	Fuzion
‘21-...	Tokiwa Software

Motivation: Fuzion Language

Many languages overloaded with concepts like classes, methods, interfaces, constructors, traits, records, structs, packages, values, …

➡ Fuzion has one concept: a feature

Today's compilers and tools are more powerful

➡ Tools make better decisions

Systems are safety-critical

➡ we need to ensure correctness

Fuzion Resources

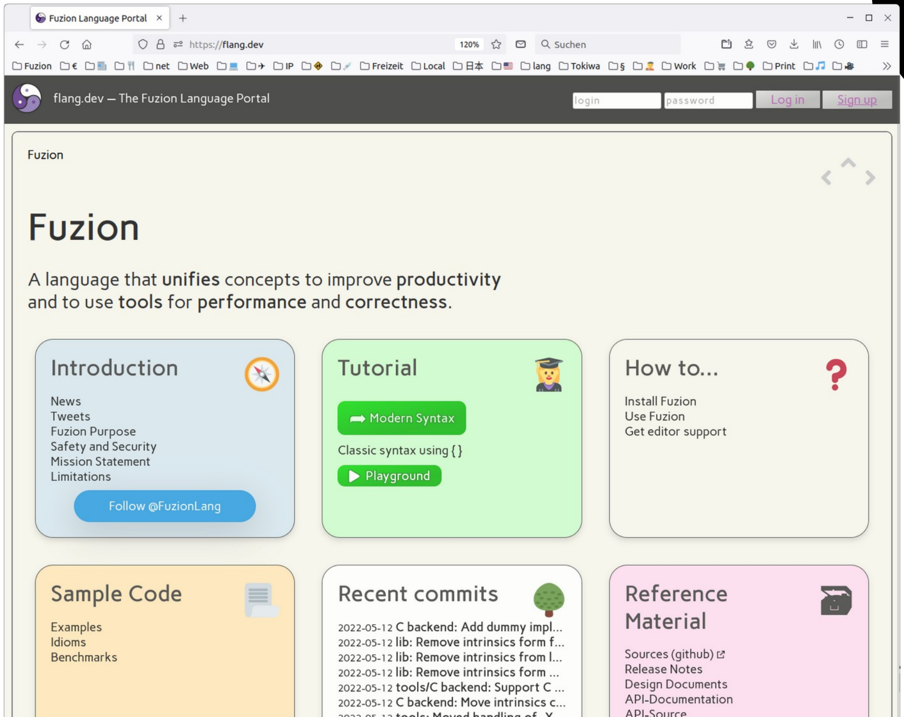
Fuzion available

- ➡ sources: github.com/tokiwa-software/fuzion

Fuzion Resources

Fuzion available

- ➡ sources: github.com/tokiwa-software/fuzion
- ➡ Website: flang.dev
  - tutorial
  - design
  - examples
  - …



TOKIWA software

Backing Company

TOKIWA software

- ➞ supports development of Fuzion
- ➞ currently four employees
- ➞ hiring
- ➞ searching for funding

This Talk

Complementarity of Effects and Types

- ➞ Algebraic Effects for Fuzion
- ➞ Types as first-class features
- ➞ Types used to name Effects

Fuzion Effects

Fuzion Features are pure functions

- ➡ no mutation of data, no side-effects

Effects are used to model non-functional aspects

- ➡ state changes
- ➡ I/O
- ➡ thread communication
- ➡ exceptions

Algebraic Effects

Definition

- an algebraic effect is a set of operations
  - `read`, `get_time`, `panic`, `log`, ...
  - operations often model a non-functional effect
- operations may resume or abort
- an effect's operations may be implemented by different handlers
- to execute code that uses an effect, a corresponding handler must be installed

TOKIWA software

Fuzion Effects

Static analysis verifies effects

- ➡ Static analysis determines all effects
- ➡ library code must list all effects
- ➡ unexpected effects are a compile-time error

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

hello_world
```

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

hello_world
```

```
> fz hw.fz
```

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

hello_world
```

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

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

hello_world
```

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

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

hello_world
```

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

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

my_handler : io.Can_Print is
  print(s Any) unit is
    io.err.print (($s).replace "!" "!!!11!")

io.out my_handler ()→hello_world
```


TOKIWA software

Fuzion Effects Example

Hello World:

```
hello_world ! io.out ⇒
  say "hello world!"

my_handler : io.Can_Print is
  print(s Any) unit is
    io.err.print (($s).replace "!" "!!!11!")

io.out my_handler ()→hello_world
```

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

Fuzion Effects Example

Hello World:

```
hello_world ! io.out =>
  say "hello world!"

my_handler : io.Can_Print is
  print(s Any) unit is
    io.err.print (($s).replace "!" "!!!11!")

io.out my_handler ()->hello_world
```

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

Types as First-Class Features

Types as First-Class Features

Generics in Java

```
<T> void show_number(T a)
{
  System.out.println("a is " + a);
}
```

Types as First-Class Features

Type parameters in Fuzion

```
show_number(T type,
            a T) ⇒
  say "a is $a"
```

Generics in Java

```
<T> void show_number(T a)
{
  System.out.println("a is " + a);
}
```

Types as First-Class Features

Type parameters in Fuzion

```
show_number(T type,
            a T) ⇒
  say "a is $a"
```

```
>
```

Types as First-Class Features

Type parameters in Fuzion

```
show_number(T type,
            a T) ⇒
  say "a is $a"

show_number i32 1234
show_number f64 3.14
```

```
>
```

Types as First-Class Features

Type parameters in Fuzion

```
show_number(T type,
            a T) ⇒
  say "a is $a"

show_number i32 1234
show_number f64 3.14
```

```
> fz types.fz
```

Types as First-Class Features

Type parameters in Fuzion

```
show_number(T type,
            a T) ⇒
  say "a is $a"


show_number i32 1234
show_number f64 3.14
```

```
> fz types.fz
a is 1234
a is 3.14
>
```


TOKIWA software

Types as First-Class Features

Type inference

```
show_number(T type,
            a T) ⇒
  say "a is $a"

show_number i32 1234
show_number f64 3.14
```

```
> fz types.fz
a is 1234
a is 3.14
>
```


TOKIWA software

Types as First-Class Features

Type inference

```
show_number(T type,
            a T) ⇒
  say "a is $a"

show_number 1234
show_number 3.14
```

Types as First-Class Features

Type constraints

```
show_number(T type : numeric T,
            a T) ⇒
  say "a is $a"

show_number 1234
show_number 3.14
```

```
>
```

Types as First-Class Features

Type constraints

```
show_number(T type : numeric T,
            a T) ⇒
  say "a is $a, twice is {a+a}"

show_number 1234
show_number 3.14
```

```
>
```


TOKIWA software

Types as First-Class Features

Type constraints

```
show_number(T type : numeric T,
            a T) ⇒
  say "a is $a, twice is {a+a}"

show_number 1234
show_number 3.14
```

```
> fz types.fz
a is 1234, twice is 2468
a is 3.14, twice is 6.28
>
```


TOKIWA software

Types as First-Class Features

Type values

```
show_number(T type : numeric T,
            a T) ⇒
  say "a is $a of type {T.name}"

show_number 1234
show_number 3.14
```

```
>
```


TOKIWA software

Types as First-Class Features

Type values

```
show_number(T type : numeric T,
            a T) ⇒
  say "a is $a of type {T.name}"

show_number 1234
show_number 3.14
```

```
> fz types.fz
a is 1234 of type i32
a is 3.14 of type f64
>
```


TOKIWA software

Types as First-Class Features

Type with user defined features

```
sum_of(T type : numeric T,
       l list T              ) ⇒
```

Types as First-Class Features

Type with user defined features

```
sum_of(T type : numeric T,
          l list T            ) ⇒
   l ? nil    ⇒
     | c Cons ⇒
```

Types as First-Class Features

Type with user defined features

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

Types as First-Class Features

Type with user defined features

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

Types as First-Class Features

Type with user defined features

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

Types as First-Class Features

Type with user defined features

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

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

Types as First-Class Features

Type with user defined features

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

numeric is
  type.zero numeric.this.type is abstract
i32 : numeric is
  type.zero i32 is 0
```

Types as First-Class Features

Type with user defined features

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

Types as First-Class Features

Type with user defined features

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

say (sum_of [3.14159, 2.71828].as_list)
say (sum_of [1 ⁄ 3, 1 ⁄ 4].as_list)
say (sum_of f64 nil)
say (sum_of (fraction u8) nil)
```

Types as First-Class Features

Type with user defined features

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

say (sum_of [3.14159, 2.71828].as_list)
say (sum_of [1 ⁄  3, 1 ⁄  4].as_list)
say (sum_of f64 nil)
say (sum_of (fraction u8) nil)
```

```
> fz types.fz
```

Types as First-Class Features

Type with user defined features

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

say (sum_of [3.14159, 2.71828].as_list)
say (sum_of [1 ⁄ 3, 1 ⁄ 4].as_list)
say (sum_of f64 nil)
say (sum_of (fraction u8) nil)
```

```
> fz types.fz
5.85987
7⁄12
0.0
0⁄1
```

Types as Named Effects

Example: Simple linked ring

- ➡ Creation of a linked ring requires **mutation**
- ➡ Any calculation using ring therefore uses `mutate` effect
- ➡ But feature may still be **pure** if mutation affects only temporary **local state**

Types as Named Effects

Types as Named Effects

Ring using global mutate effect

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r.last    )
  next := mut  (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r         )
  last.next ← Ring.this
```

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r.last   )
  next := mut  (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r        )
  last.next ← Ring.this
```


TOKIWA software

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r.last    )
  next := mut  (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
> fz demo.fz
```


TOKIWA software

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r.last    )
  next := mut  (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
> fz demo.fz
A B C A B C A B C A
>
```


TOKIWA software

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r.last    )
  next := mut  (old ? nil     ⇒ Ring.this
                    | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
> fz demo.fz
A B C A B C A B C A
> fz -effects demo.fz
```

Types as Named Effects

Ring using global mutate effect

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil      ⇒ Ring.this
                    | r Ring ⇒ r.last    )
  next := mut  (old ? nil      ⇒ Ring.this
                    | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
> fz demo.fz
A B C A B C A B C A
> fz -effects demo.fz
exit
io.err
io.out
mutate
panic
>
```


TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(data String,
     old option Ring) ref is
  last Ring := (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r.last   )
  next := mut  (old ? nil    ⇒ Ring.this
                   | r Ring ⇒ r        )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```


TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(
     data String,
     old option  Ring    ) ref is
  last Ring    :=     (old ? nil     ⇒ Ring.this
                           | r Ring ⇒ r.last    )
  next  :=        mut (old ? nil     ⇒ Ring.this
                           | r Ring ⇒ r         )
  last.next ← Ring.this


demo  ⇒
  r  := Ring "A" (Ring "B" (Ring    "C" nil))
  for n  := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```

TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option  Ring    ) ref is
  last Ring    :=     (old ? nil     ⇒ Ring.this
                           | r Ring ⇒ r.last    )
  next  :=       mut (old ? nil     ⇒ Ring.this
                           | r Ring ⇒ r         )
  last.next ← Ring.this


demo  ⇒
  r  := Ring "A" (Ring "B" (Ring    "C" nil))
  for n  := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil    ⇒ Ring.this
                         | r Ring ⇒ r.last   )
  next :=        mut (old ? nil    ⇒ Ring.this
                         | r Ring ⇒ r        )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring    "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=     (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r.last    )
  next :=          mut (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring    "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```


TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r.last    )
  next  := M.env.new (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r         )
  last.next ← Ring.this


demo ⇒
  r := Ring "A" (Ring "B" (Ring    "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r.last   )
  next  := M.env.new (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r        )
  last.next ← Ring.this

mm : mutate is

demo  ⇒
  r  := Ring "A" (Ring "B" (Ring    "C" nil))
  for n  := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

>

TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil    ⇒ Ring.this
                         | r Ring ⇒ r.last    )
  next := M.env.new (old ? nil    ⇒ Ring.this
                         | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo ⇒
  r := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
demo
```

```
>
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r.last    )
  next := M.env.new (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo ⇒
  r := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
mm.use ()->demo
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=     (old ? nil    ⇒ Ring.this
                          | r Ring ⇒ r.last    )
  next := M.env.new (old ? nil    ⇒ Ring.this
                          | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo ⇒
  r := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
mm.use ()→demo
```

```
> fz demo.fz
```


TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=     (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r.last    )
  next  := M.env.new (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo  ⇒
  r  := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n  := r, n.next.get; i in 1..10 do
    yak "{n.data} "
mm.use ()→demo
```

```
> fz demo.fz
A B C A B C A B C A
```

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=     (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r.last    )
  next := M.env.new (old ? nil     ⇒ Ring.this
                          | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo ⇒
  r := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
mm.use ()→demo
```

```
> fz demo.fz
A B C A B C A B C A
> fz -effects demo.fz
```


TOKIWA software

Types as Named Effects

Ring using local mutability

```
Ring(M type : mutate,
     data String,
     old option (Ring M)) ref is
  last Ring M :=    (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r.last    )
  next  := M.env.new (old ? nil     ⇒ Ring.this
                         | r Ring ⇒ r         )
  last.next ← Ring.this

mm : mutate is

demo ⇒
  r := Ring "A" (Ring "B" (Ring mm "C" nil))
  for n := r, n.next.get; i in 1..10 do
    yak "{n.data} "
mm.use ()→demo
```

```
> fz demo.fz
A B C A B C A B C A
> fz -effects demo.fz
exit
io.err
io.out
panic
>
```

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

Conclusion

Algebraic effects and Types as 1st class features

➡ complement one another surprisingly well

➡ effects encapsulate non-functional aspects

- mutability
- i/o
- exceptions

➡ have a look, get involved!

@fuzion@types.pl
@FuzionLang
https://flang.dev
github.com/tokiwa-software/fuzion