Fuzion Language Update

The marathon run from
a language prototype to
a full implementation and toolchain.

TOKIWA
software

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

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

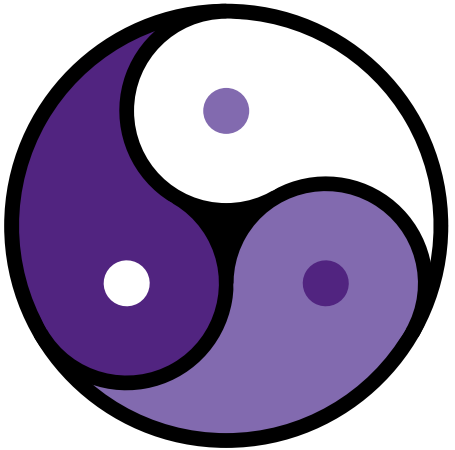
➡ we need to ensure correctness

Fuzion Summary

Fuzion

- ➡ uses the **feature** as its main concept
- ➡ is **statically typed**
- ➡ has **inheritance** and **redefinition**
- ➡ uses **value types** and **dynamic (ref) types**
- ➡ encourages **immutability**
- ➡ offloads tasks and decisions from developers to **tools**

Fuzion Logo



Backing Company

- ➡ supports development of Fuzion
- ➡ currently three employees
- ➡ hiring
- ➡ searching for funding

Using White Space

Using White Space

Avoiding bloat caused by

```
; { } ( ) ,
```

and

- ➡ use whitespace instead
- ➡ increase clarity and readability
- ➡ avoid errors

Using White Space: Semicolons

Flat line feeds: same indentation

```
stmnt1
stmnt2
stmnt3
```

Equivalent to

```
stmnt1;
stmnt2;
stmnt3;
```

Using White Space: Semicolons

Flat line feeds: same indentation

```
stmnt1
stmnt2
stmnt3
```

Equivalent to

```
stmnt1;
stmnt2;
stmnt3;
```

Statement separation in single line using `;`

```
stmnt1; stmnt2; stmnt3
```

Using White Space: Blocks

Indenting line feeds ⇔ {

```
if cond
  stmnt1
else
  stmnt2
```

Equivalent to

```
if cond {
  stmnt1
} else {
  stmnt2
}
```

Using White Space: Blocks

Indenting line feeds ⇔ {

```
if cond
  stmnt1
else
  stmnt2
```

Equivalent to

```
if cond {
  stmnt1
} else {
  stmnt2
}
```

Blocks in a single line:

```
if cond then stmnt1 else stmnt2
```

Using White Space: Calls

Calls do not require `( , )`

```
f a b c
```

Equivalent to

```
f(a, b, c)
```

Using White Space: Calls

Calls do not require `( , )`

```
f a b c
```

Equivalent to

```
f(a, b, c)
```

nesting

```
f (g x y) (h z) c
```

```
f(g(x, y), h(z), c)
```

Using White Space: Calls & Tuples

Calls with two arguments

```
f a b
```

Equivalent to

```
f(a, b)
```

Using White Space: Calls & Tuples

Calls with two arguments

```
f a b
```

Equivalent to

```
f(a, b)
```

Call with one tuple as argument

```
g (a, b)
```

```
g((a, b))
```

Using White Space: Calls & Tuples

Calls with two arguments

```
f a b
```

Equivalent to

```
f(a, b)
```

Call with one tuple as argument

```
g (a, b)
```

```
g((a, b))
```

Using White Space: Arrays

Accessing array `a` at indices `i, j`

```
a[i, j]
```

Using White Space: Arrays

Accessing array `a` at indices `i, j`

```
a[i, j]
```

Creating array with elements `x`, `y`, `z`

```
[x, y, z]
```

Using White Space: Arrays

Accessing array `a` at indices `i, j`

```
a[i, j]
```

Creating array with elements `x`, `y`, `z`

```
[x, y, z]
```

Passing this array as argument in call to `f`

```
f [x, y, z]
```

Using White Space: Arrays

Accessing array `a` at indices `i, j`

```
a[i, j]
```

Creating array with elements `x`, `y`, `z`

```
[x, y, z]
```

Passing this array as argument in call to `f`

```
f [x, y, z]
```

Using White Space: Operators

Using prefix operator on argument

```
f -x
```

Using White Space: Operators

Using prefix operator on argument

```
f -x
```

Equivalent to

```
f (-x)
```

Using White Space: Operators

Using prefix operator on argument

```
f -x
```

Equivalent to

```
f (-x)
```

while

```
f - x
```

```
f-x
```

Using White Space: Operators

Using prefix operator on argument	Equivalent to
`f -x`	`f (-x)`
while	
`f - x`	`f-x`
and	
`f- x`	`(f-) x`

Using White Space: Operators

Using infix operators in calls

```
f a-b
```

Equivalent to

```
f (a-b)
```

Using White Space: Operators

Using infix operators in calls

```
f a-b
```

Equivalent to

```
f (a-b)
```

while

```
f a -b
```

```
f (a) (-b)
```

Using White Space: Operators

Using infix operators in calls	Equivalent to
`f a-b`	`f (a-b)`
while	
`f a -b`	`f (a) (-b)`
and	
`f a - b`	`(f a)-b`

Using White Space: In Java

White space has important semantics in other languages as well, e.g., Java code

```
if (cc) x(); elsewhere();
```

is very different to

```
if (cc) x(); else where();
```

Semantic white space is basically a matter of practice and habit.

Type Inference

Avoid the need for explicit types.

Type Inference

Avoid the need for explicit types.

Occurs at assignments:

`x := expr`	explicit assignment
`f(a T) => g(a)`	implicit assignment to feature result
`h 3.14e3`	assignment of value to feature argument

Types propagates in both direction: forward and backward!

Type Inference: Feature Results

Expression type propagated forward:

`v := 123`	type **i32**, default for integer literal
`s := “hello”`	type **string**
`h(v u8) => v >> 4`	type **u8**
`origin => point 0 0`	type **point<i32>**

Type Inference: Type Arguments

Type of actual arguments in call propagated to call's type arguments:

```
square<T: integer<T>> (a T) => a * a

s² := square<i32> 123
```

using explicit type parameter

```
s² := square 123
```

using type inference

Type Inference: Lambdas

Backward propagation: Lambda type defined by what it is assigned to:

```
si(f (string, i32) -> string) => say (f “hello” 3)

ii(f (i32, i32) -> i32) => say (f 10 3)

si (s,i -> s * i)
```

lambda type **(string, i32) -> string**

```
ii (s,i -> s * i)
```

lambda type **(i32, i32) -> i32**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```

literal of type **u16**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```
literal of type **u16**

```
v u16 := 54.321E3
```
literal of type **u16**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```
literal of type **u16**

```
v u16 := 54.321E3
```
literal of type **u16**

```
w := u16 54321
```
literal of type **u16**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```
literal of type **u16**

```
v u16 := 54.321E3
```
literal of type **u16**

```
w := u16 54321
```
literal of type **u16**

```
x f64 := 54321
```
literal of type **f64**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```
literal of type **u16**

```
v u16 := 54.321E3
```
literal of type **u16**

```
w := u16 54321
```
literal of type **u16**

```
x f64 := 54321
```
literal of type **f64**

```
y f64 := 54.321E3
```
literal of type **f64**

Type Inference: Numeric Literals

Backward propagation: Target type defines type of literal

```
u u16 := 54321
```
literal of type **u16**

```
v u16 := 54.321E3
```
literal of type **u16**

```
w := u16 54321
```
literal of type **u16**

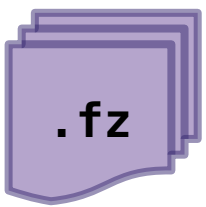
```
x f64 := 54321
```
literal of type **f64**

```
y f64 := 54.321E3
```
literal of type **f64**

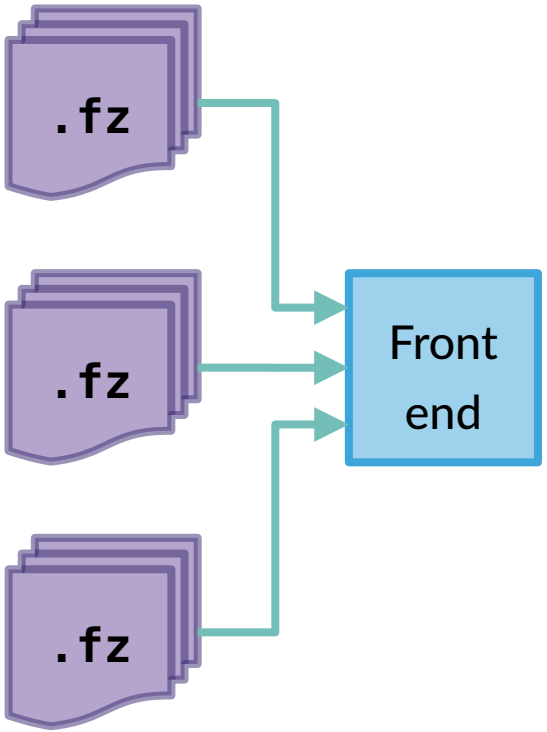
```
z := f64 54321
```
literal of type **f64**

Fuzion Toolchain: Planned Design

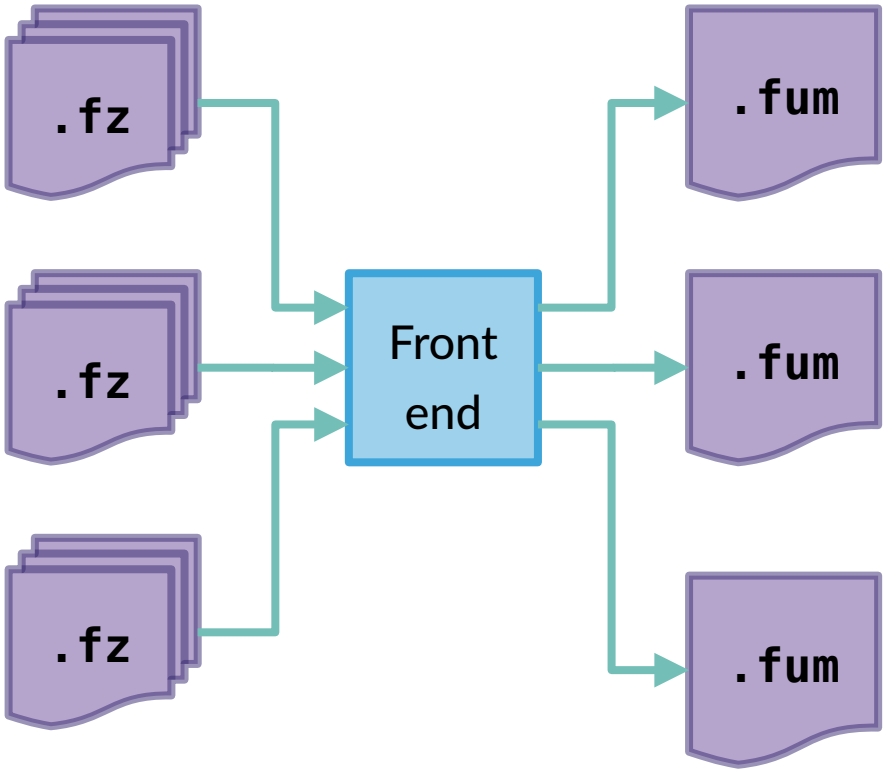
Fuzion Toolchain: Planned Design



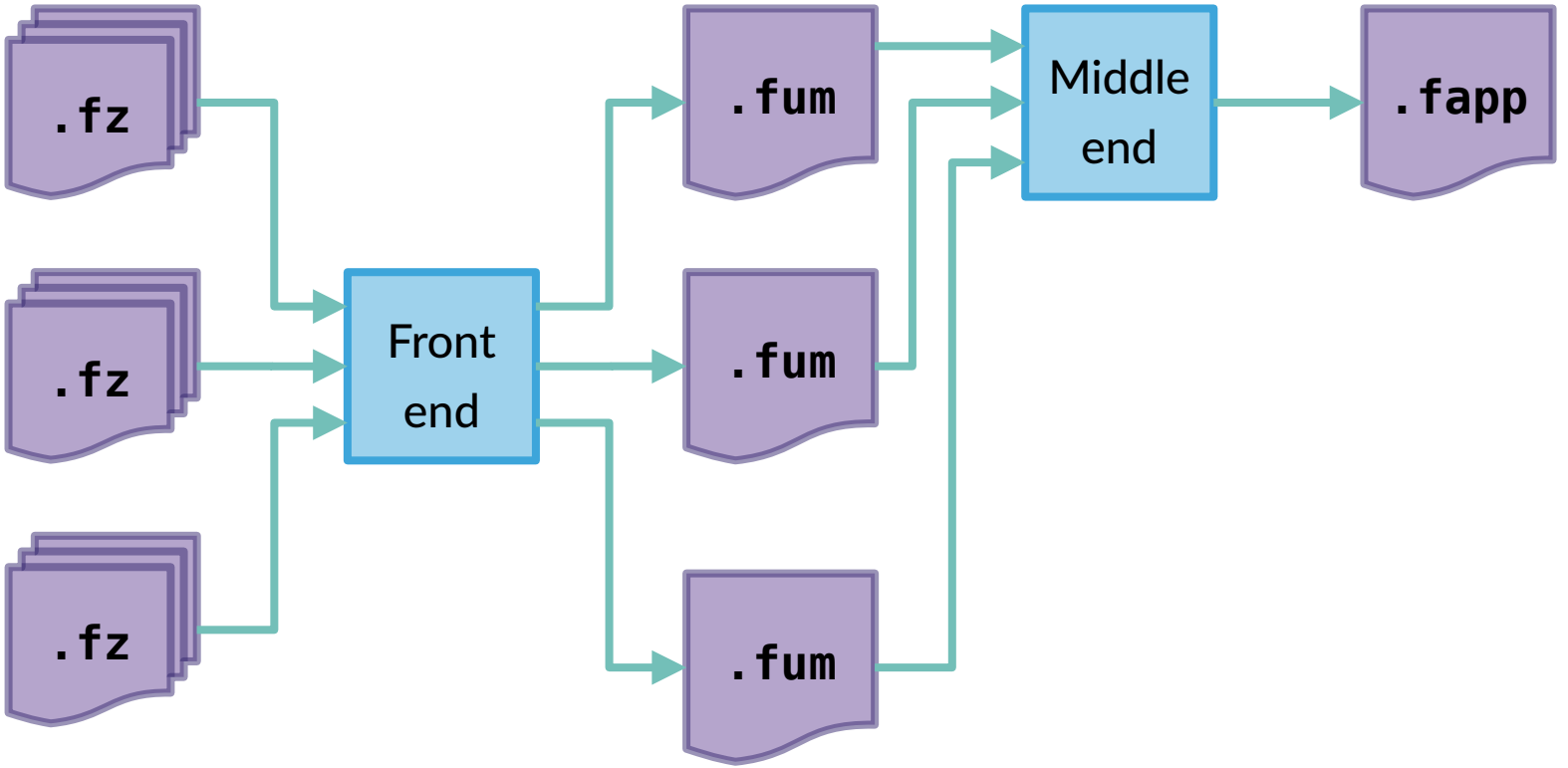
Fuzion Toolchain: Planned Design



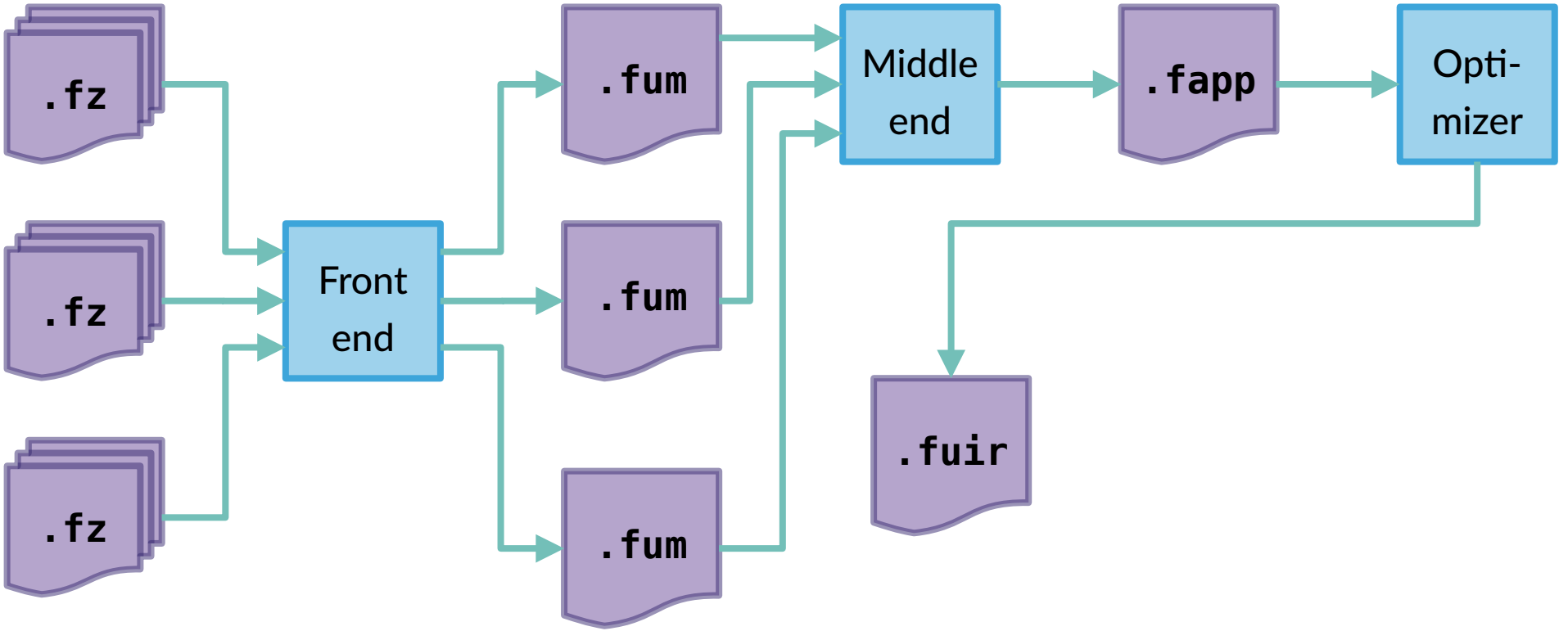
Fuzion Toolchain: Planned Design



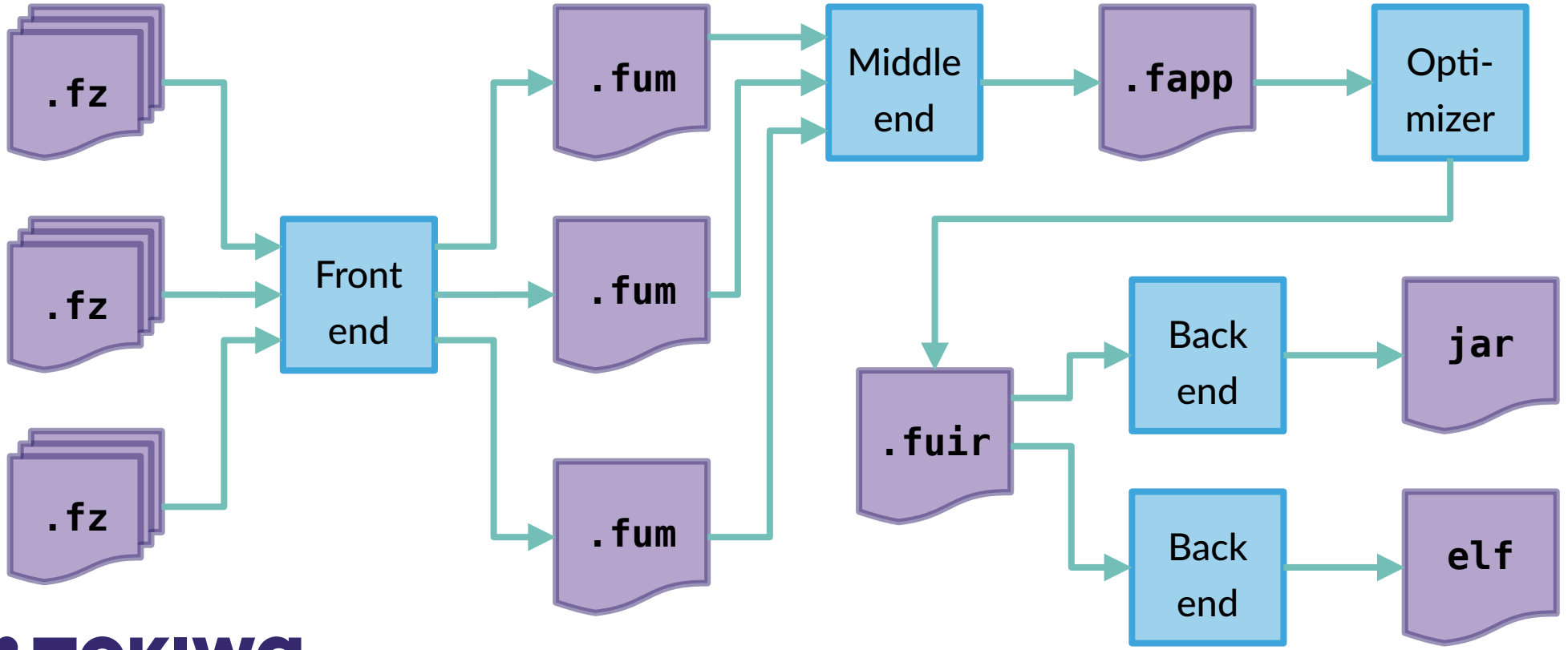
Fuzion Toolchain: Planned Design



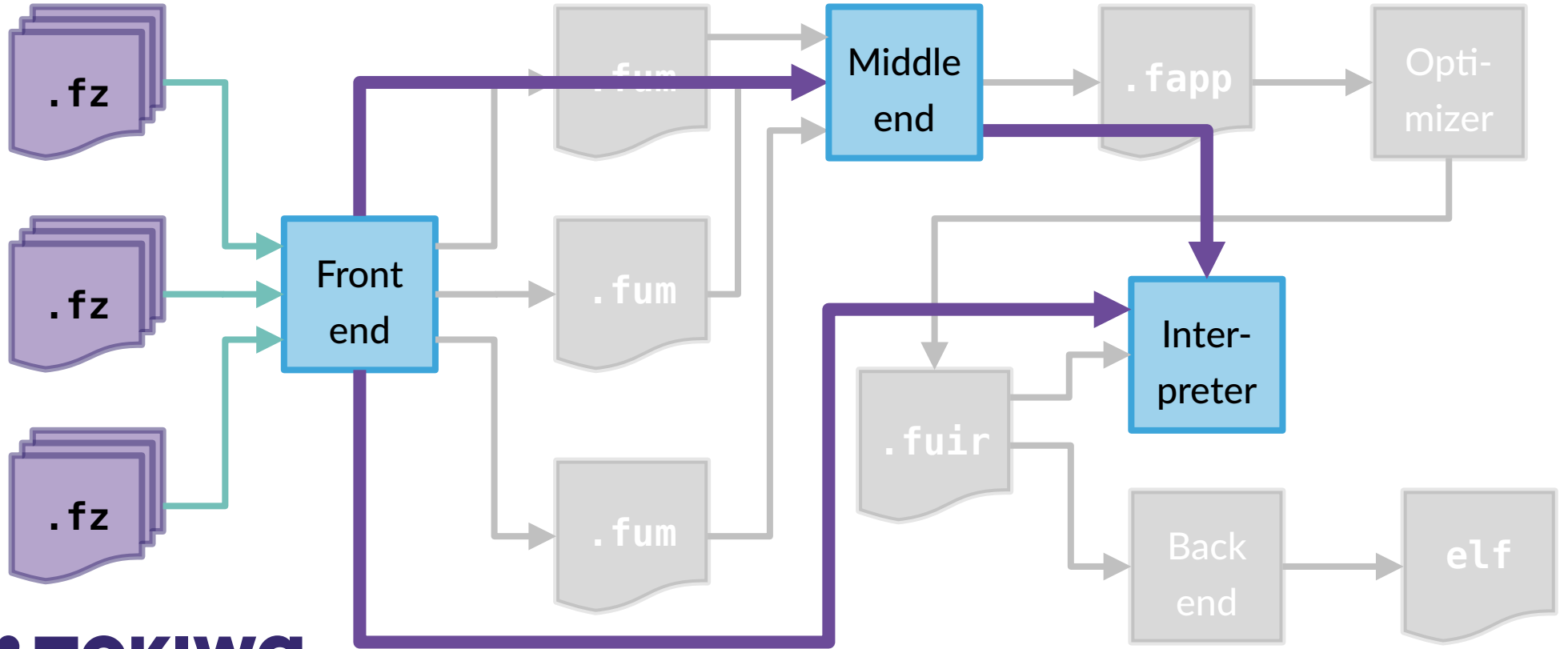
Fuzion Toolchain: Planned Design



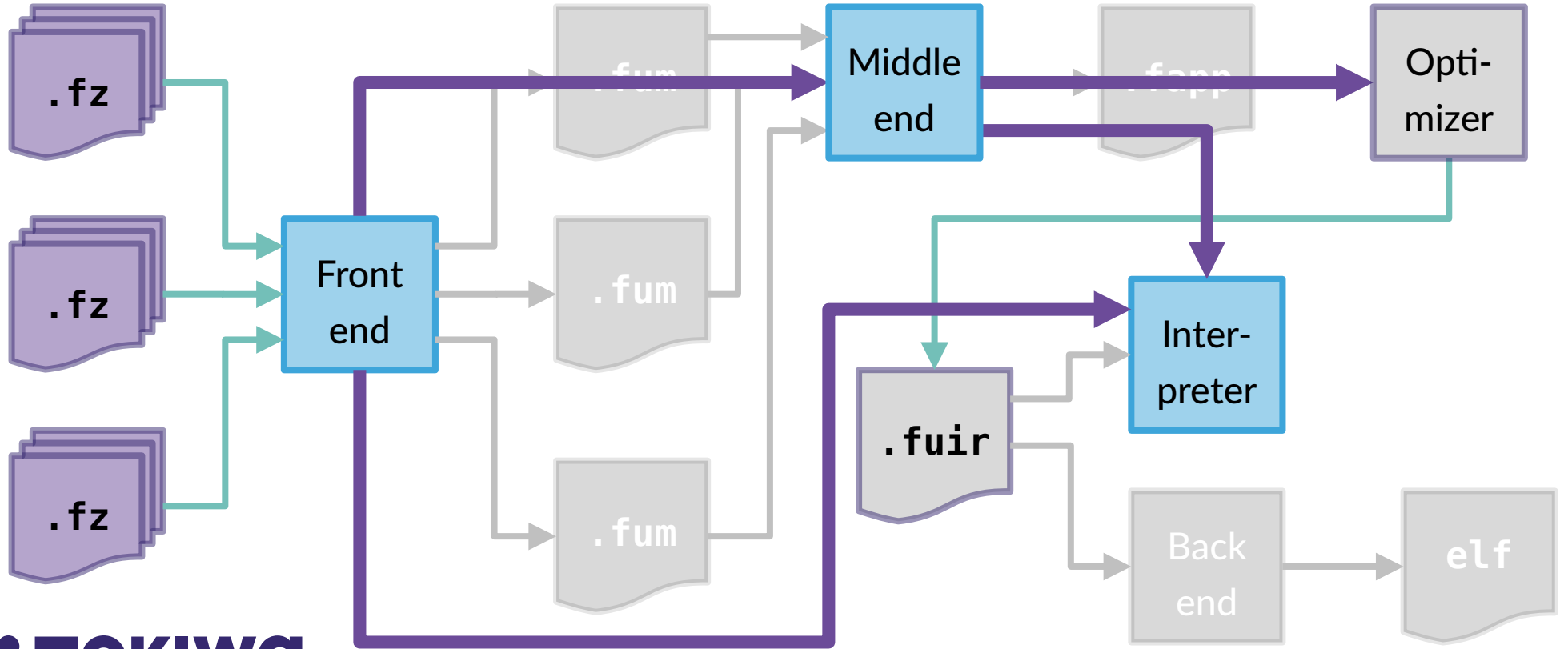
Fuzion Toolchain: Planned Design



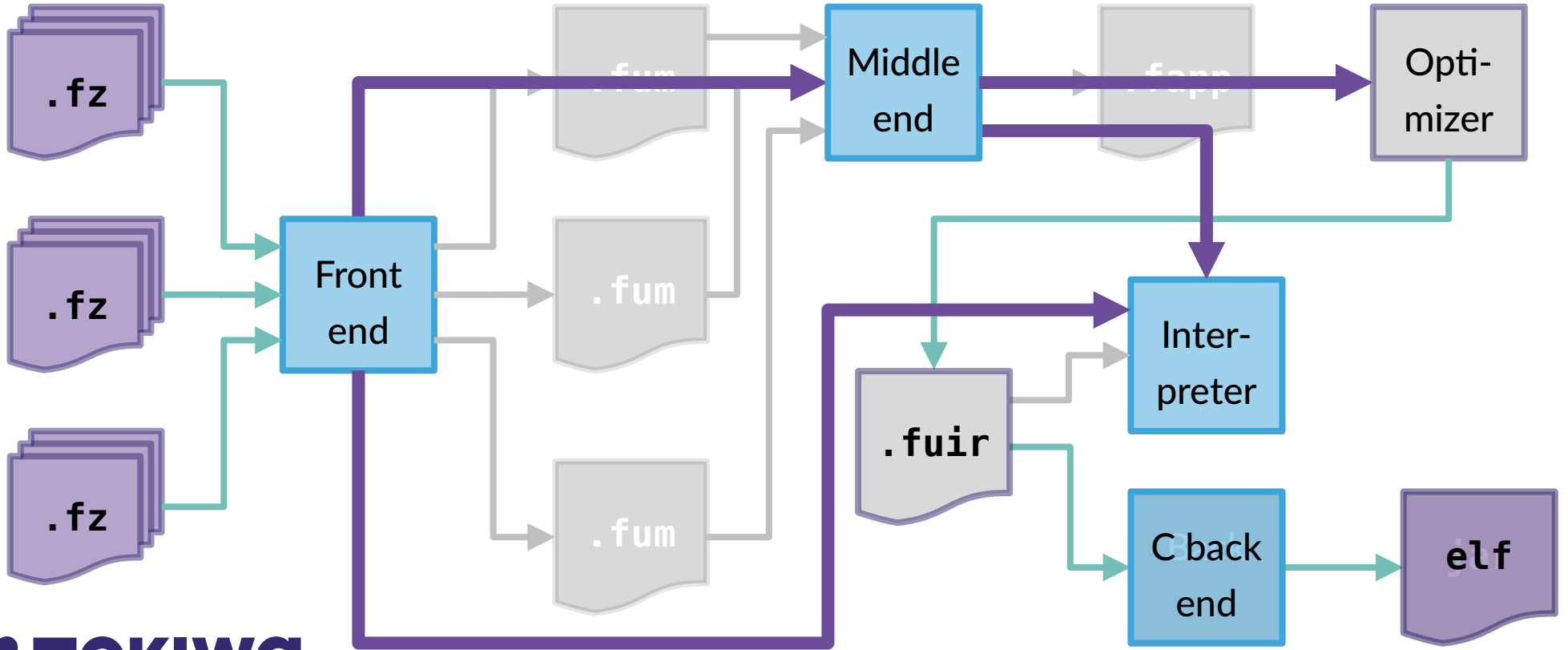
Fuzion Toolchain: State Jan'21



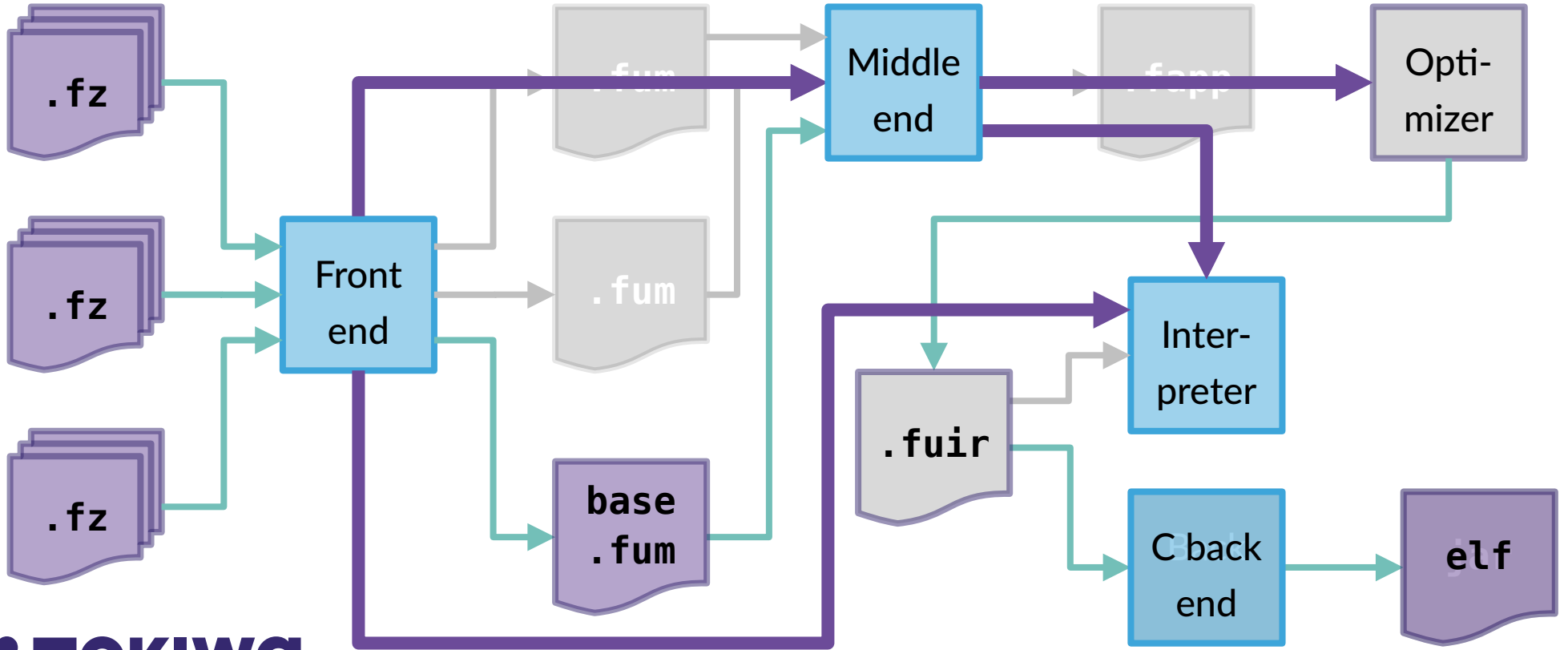
Fuzion Toolchain: Changes 2021



Fuzion Toolchain: Changes 2021



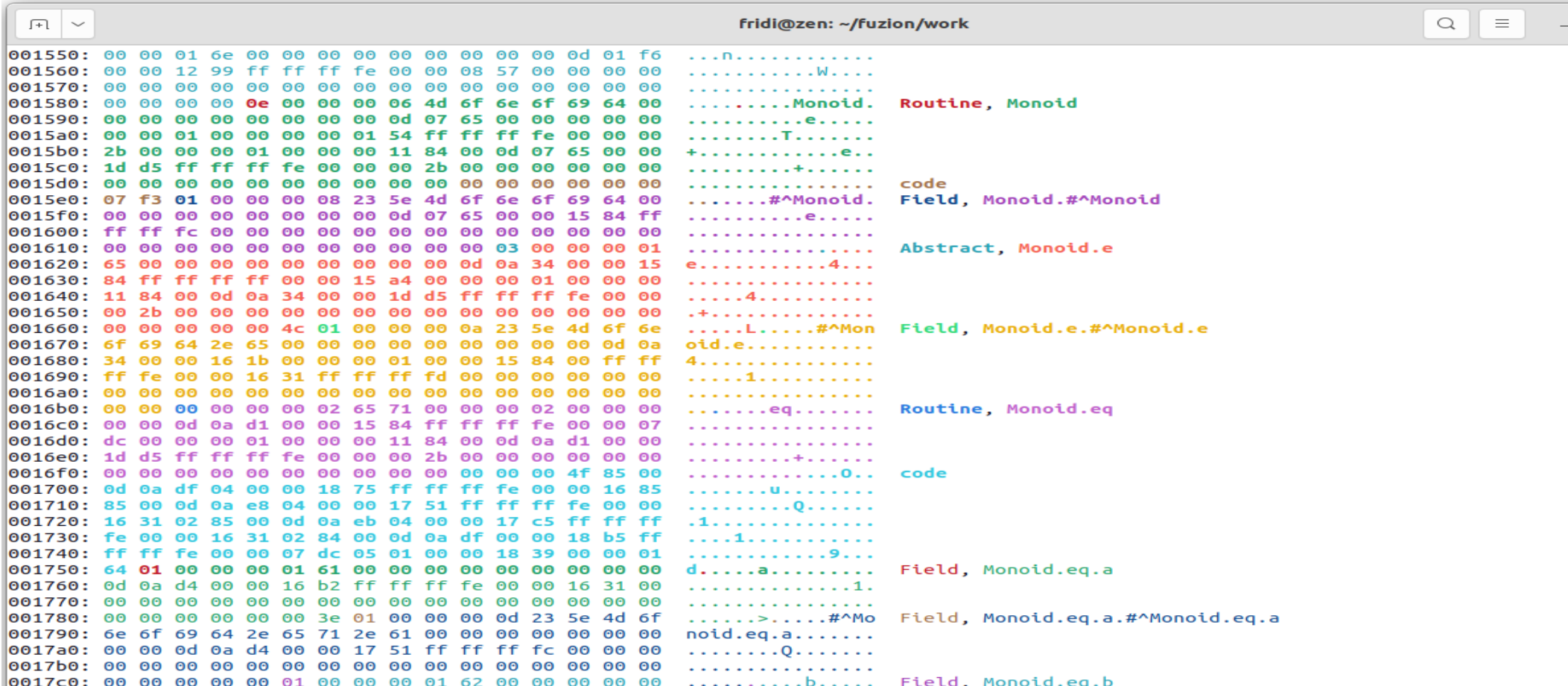
Fuzion Toolchain: State Jan'22



Fuzion Module Files

Binary file format

Fuzion Module Files



Fuzion Module Files

Binary file format

- ➡ `fmap()`ped to memory
- ➡ references using offsets, not names
- ➡ contains **features**, **types**, **expressions**, **source code**
- ➡ features are **routine**, **field**, **intrinsic**, **abstract** or **choice**
- ➡ types are **feature types** or **type parameters**
- ➡ 10 kinds of expressions: **call**, **match**, **const**, **assign**, **pop**, ...

Fuzion Module Files

Impact:

- ➡ faster compilation
- ➡ module-based correctness analysis
- ➡ define feature visibility
- ➡ container for exchange of library code

Other Tools: FZJava

Create Fuzion interface to Java module


```
java.lang.System.out.println "Hello Java   !"
```

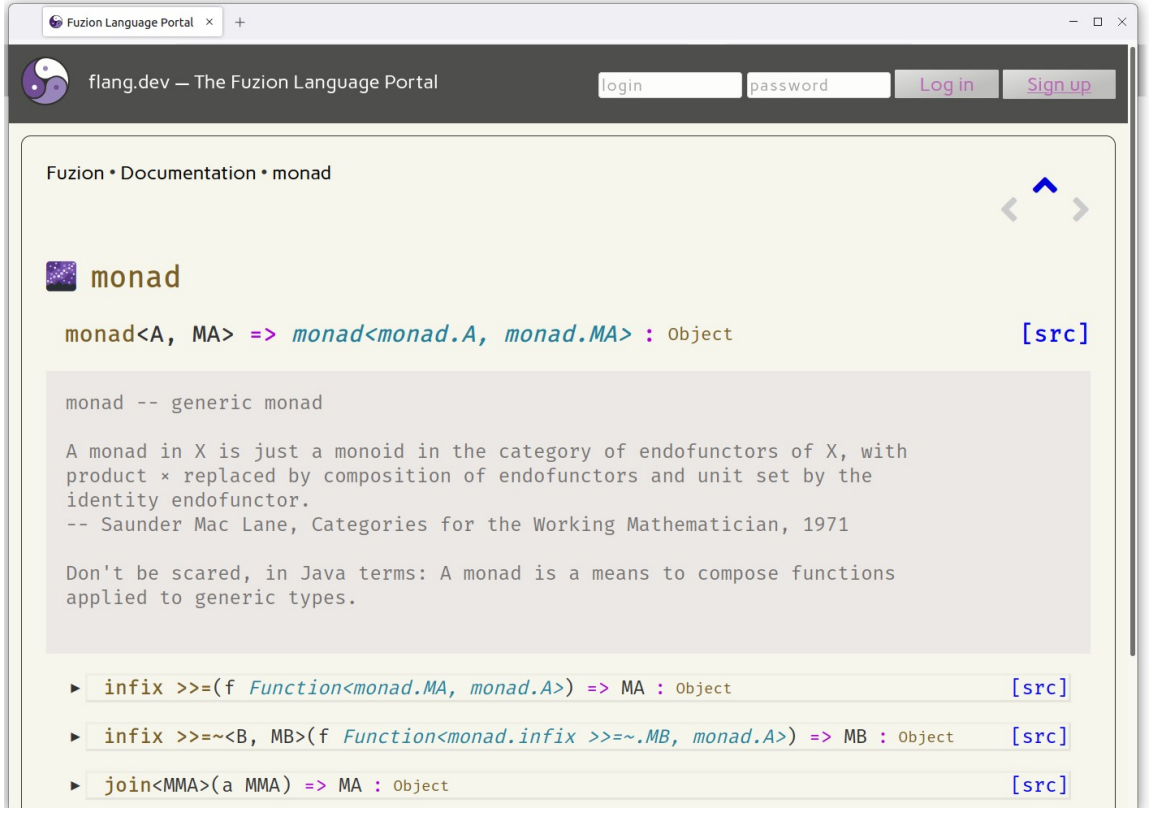
Other Tools: Language Server

Support for IDEs and editors (vim, emacs)

- ➡ completion
- ➡ signature help
- ➡ documentation
- ➡ ...

Other Tools: FuzionDoc tool

Extract documentation from Fuzion source code



TOKIWA software

Fuzion: Next Steps

Development Plan

- ➡ intermediate files: .fum, .fapp, .fuir
- ➡ simple analysis tools: field init, immutability
- ➡ C back-end: GC, floats, etc.
  - interfacing C library code
- ➡ Standard Library
- ➡ Modeling I/O, thread communication and immutability
  - using automatic monadic lifting?

Conclusion

We are running a Marathon here

- ➡ we have not made the first 10km yet
- ➡ we need
  - to grow our team
  - get developer feedback
  - secure long-term funding
- ➡ please get involved!

http://flang.dev

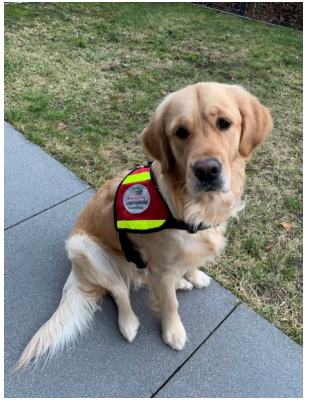
siebert@tokiwa.software

github.com/tokiwa-software/fuzion

Conclusion

We are running a Marathon here

- ➡ we have not made the first 10km yet
- ➡ we need
  - to grow our team
  - get developer feedback
  - secure long-term funding
- ➡ please get involved!



http://flang.dev

siebert@tokiwa.software

github.com/tokiwa-software/fuzion