Designing a New Language for Safety: Fuzion

A minimal language for safety-critical systems

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 Compiler 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**

Backing Company

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

Safety-Critical Systems

Definition (Wikipedia)

- ➡ a system whose failure or malfunction may result in []:
  - death or serious injury to people
  - loss or severe damage to equipment/property
  - environmental harm
- ➡ often require certification (IEC61508, DO178C, etc.)

Safety-Critical Systems

Certification typically requires

- ➡ defined SW development process
- ➡ traceability
  - requirements ↔ code ↔ validation ↔ results
- ➡ rigorous verification and validation
  - static analysis can help

Fuzion Language Tutorial

Not part of this talk

➡ online at **flang.dev**

This talk will show how

➡ Java maps to Fuzion

Feature Examples

Features used as routines with code

Feature Examples

Features used as routines with code

```
HelloWorld is
  say "Hello World!"
```

Feature Examples

Nesting of Features

```
HelloWorld is
  hw is
    say "Hello World!"

  hw
```

Feature Examples

Features with arguments

```
HelloWorld is
  hw(name string) is
    say "Hello $name!"

  hw "World"
```

Feature Examples

Features with inner features

```
HelloWorld is
  hw(name string) is
    run is
      say "Hello $name!"

  x := hw "World"
  x.run
```

Feature Examples

Features with inner features

```
HelloWorld is
  hw(name string) is
    run is
      say "Hello $name!"

  x := hw "World"
  x.run
```

Fuzion code consists of feature declarations and feature calls.

Design by Contract

Design by Contract

Features define their behavior

- ➡ pre-condition: what has to hold before a call?
- ➡ post-condition: what guarantee is given after the call?
- ➡ concept presented by Betrand Meyer back in 1986

Design by Contract: Example

```
sqrt(a i32) i32
  pre
    a >= 0
  post
    result * result <= a,
    (result + 1) * (result + 1) > a
is
  ...
```

Controlling Contract Checks

Checking contracts dynamically

- ➡ will introduce run-time overhead
- ➡ may be prohibitively expensive
- ➡ may be required for safety

Solution

- ➡ qualified contracts

Qualified Contracts

```
sqrt(a i32) i32
  pre
    debug: a >= 0
  post
    debug 5 : result * result <= a,
    debug 5 : (result + 1) * (result + 1) > a
is
  ...
```

Contract Qualifiers

Fuzion contract qualifiers

- ➡ `safety`
- ➡ `debug`
- ➡ `debug n`
- ➡ `pedantic`
- ➡ `analysis`

Contracts for Static Analysis

```
max(a Sequence<i32>) i32
  pre
    debug: !a.isEmpty
  post
    debug: a ∀ x -> x <= result
    debug: a ∃ x -> x = result
    analysis : ∀<i32> x -> x ∈ a : x <= result
    analysis : ∃<i32> x -> x ∈ a && x = result
is
```

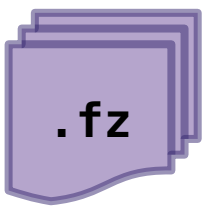
Design-by-Contract & Certification

Contracts provide

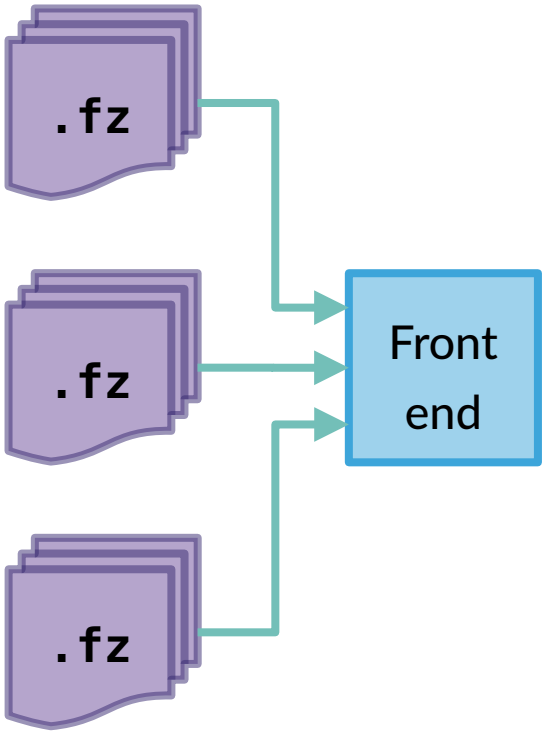
- ➡ direct way to add formal requirements to code
- ➡ means to verify these requirements at runtime
- ➡ means to define (or generate) tests
- ➡ formal analysis tools the required input

Fuzion Toolchain Design

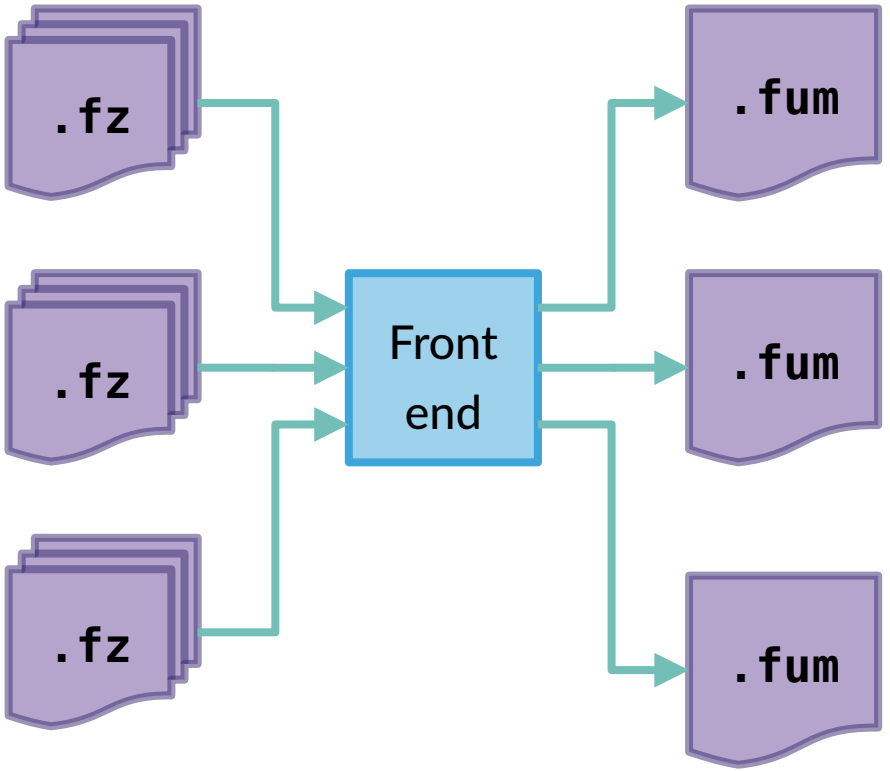
Fuzion Toolchain Design



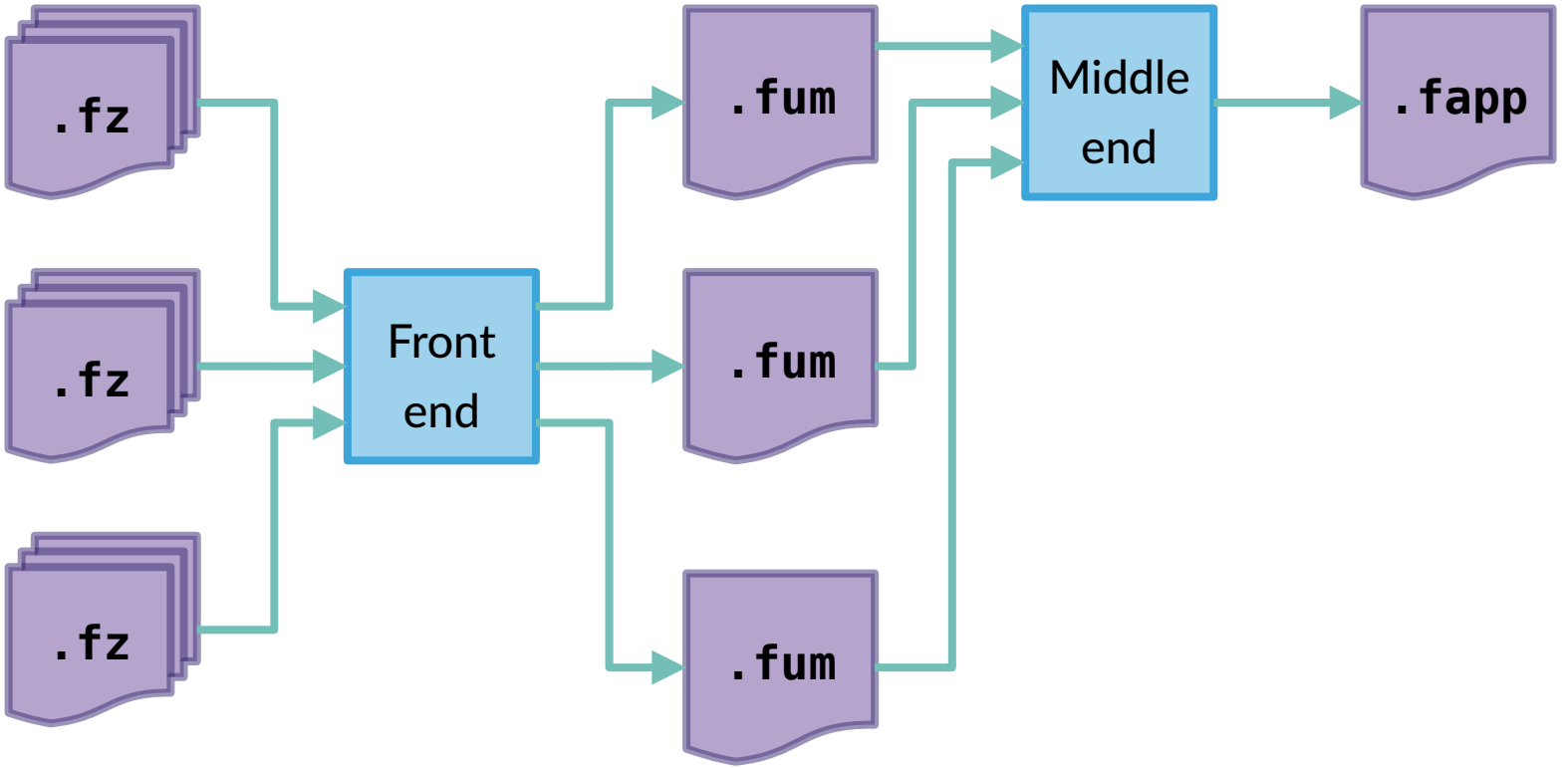
Fuzion Toolchain Design



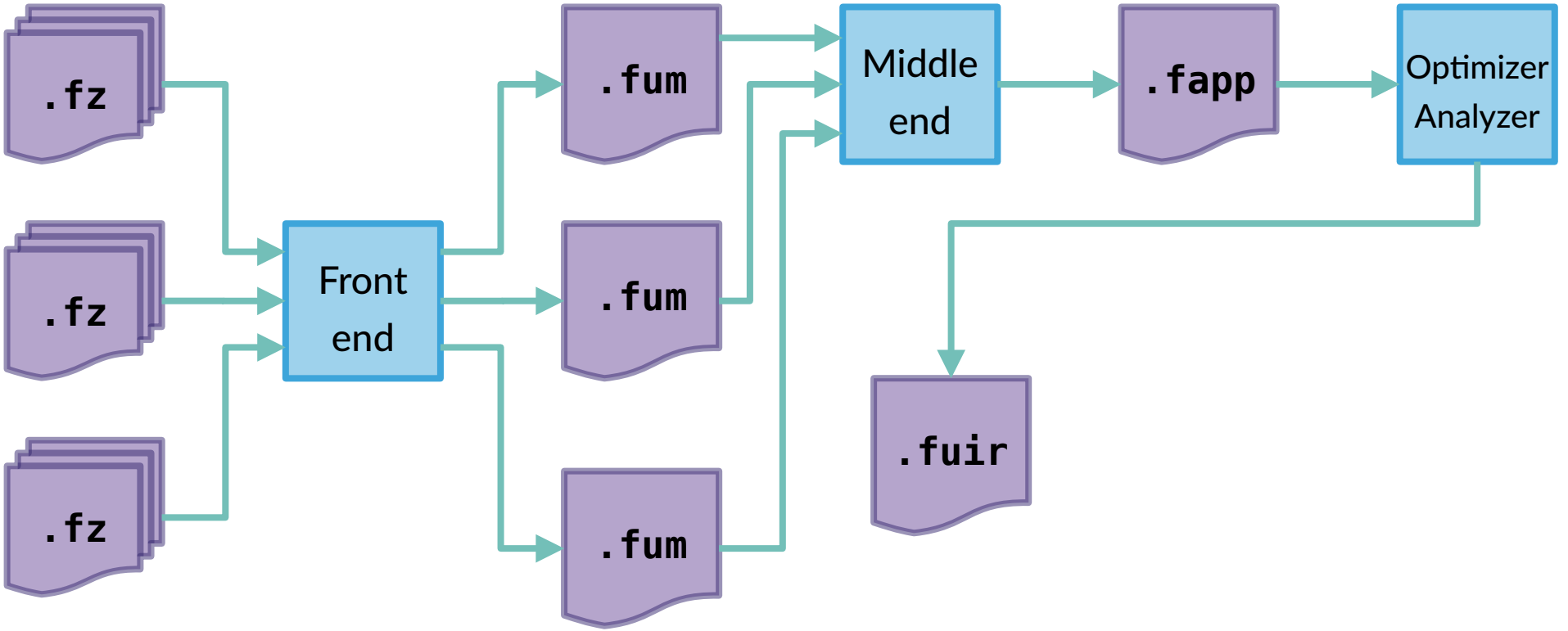
Fuzion Toolchain Design



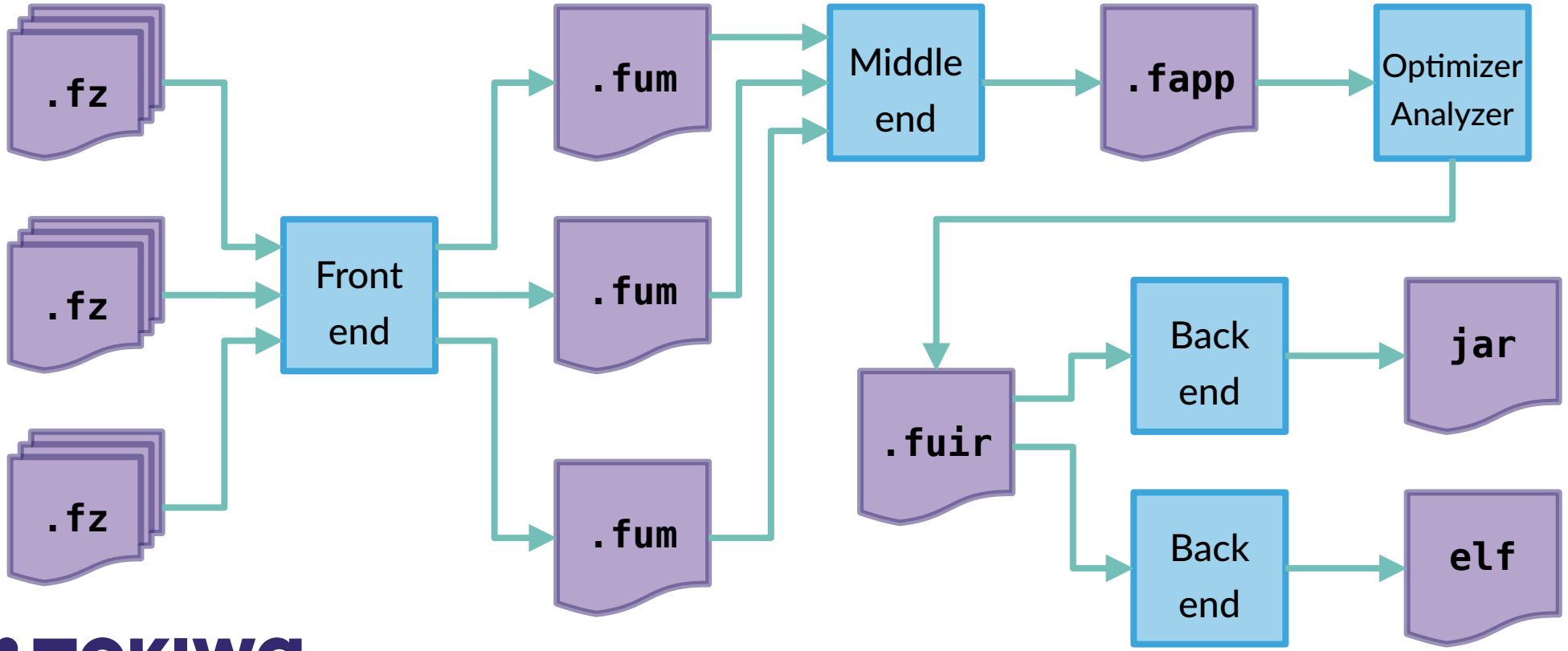
Fuzion Toolchain Design



Fuzion Toolchain Design



Fuzion Toolchain Design



Static Analysis In Fuzion Toolchain

Static analysis currently mostly non-existant.

Will be added to

- ➡ Front End
- ➡ Middle End
- ➡ Optimizer/Analyzer

Analysis Facilitated by Simple IR

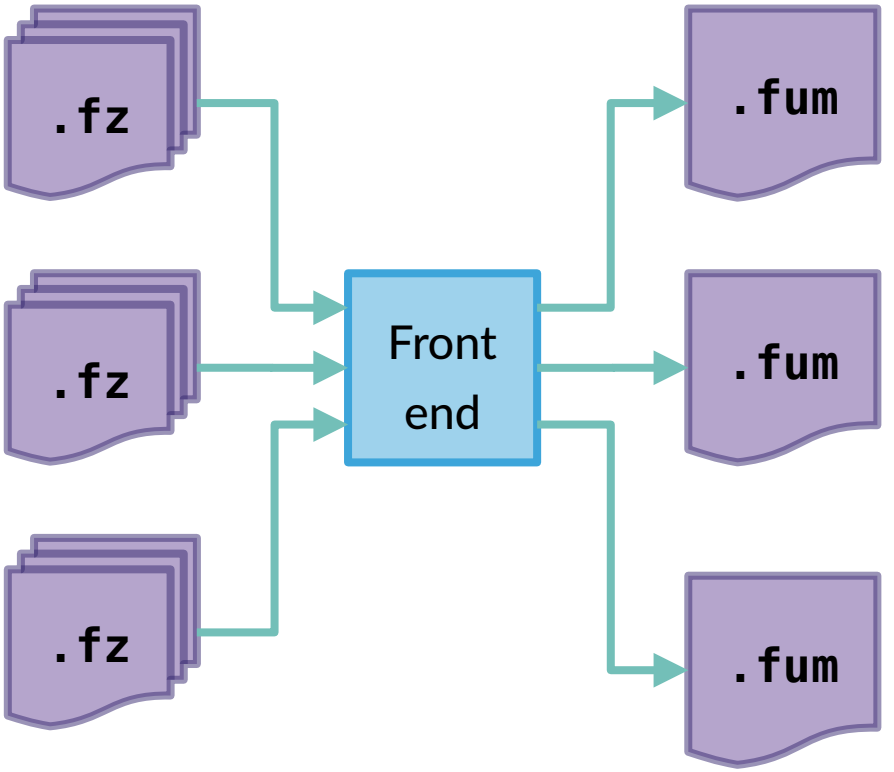
Fuzion Module files contain

- ➡ Features
  - five kinds: **routine**, **field**, **intrinsic**, **abstract** or **choice**
  - contain name, code, types, inner features
- ➡ Types are **feature types** or **type parameters**
- ➡ Code: 10 expressions: **call**, **match**, **const**, **assign**, **pop**, ...
  - no loops, no gotos

Static Analysis in Front End

Analyze single module

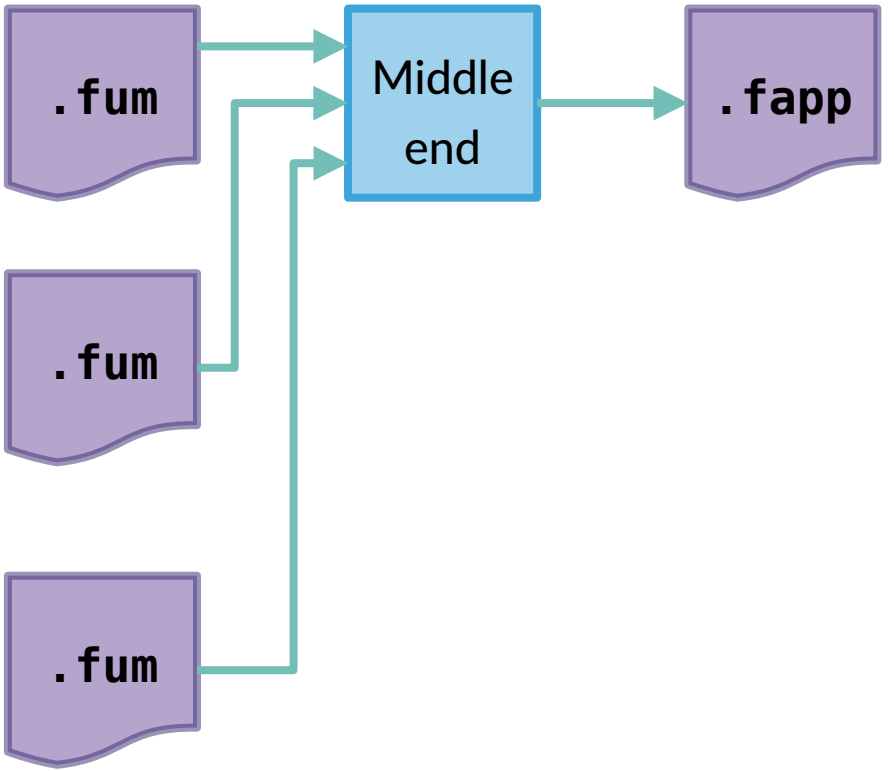
- ➡ Type Checking
- ➡ Init-before-use
- ➡ Immutability when escaped
- ➡ Thread safety



Static Analysis in Middle End

Analyze whole application

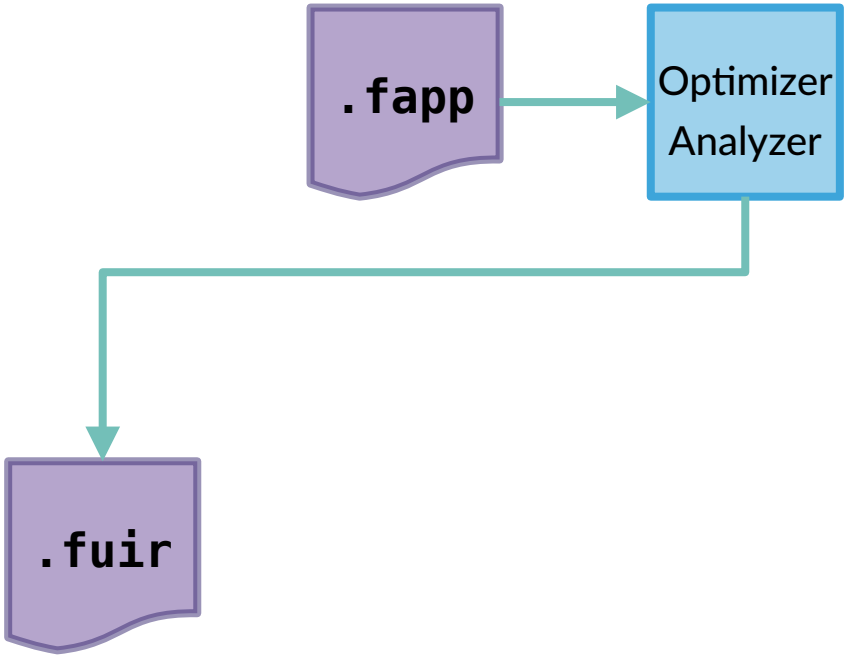
- ➡ Dead code removal
- ➡ Code Specialization
- ➡ Thread local data detection



Static Analysis in Optimizer/Analyzer

Analyze whole application

- ➡ Compile-time evaluation
- ➡ Code Specialization
- ➡ Call-graph analysis
- ➡ Lifespan analysis
  - stack vs. heap allocation
- ➡ Program-wide data flow



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

Fuzion is an exciting new language for safety

- ➡ simplicity
- ➡ design-by-contract
- ➡ prepared for static analysis
- ➡ 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

TOKIWA software