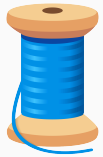
Zwirn

A small functional livecoding language



Martin Gius

2026-09-30

gius.martin@servus.at

01 The Uzu Family



- “Uzulangs” is a term coined recently in the paper “Uzulangs: a Community of Musical Pattern Languages” presented at the NIME earlier this year

01 The Uzu Family



- “Uzulangs” is a term coined recently in the paper “Uzulangs: a Community of Musical Pattern Languages” presented at the NIME earlier this year
- The term is used for describing a family of livecoding languages that share a similar representation of musical structures

01 The Uzu Family



- “Uzulangs” is a term coined recently in the paper “Uzulangs: a Community of Musical Pattern Languages” presented at the NIME earlier this year
- The term is used for describing a family of livecoding languages that share a similar representation of musical structures
- The most prominent examples are TidalCycles and Strudel

01 The Uzu Family



- “Uzulangs” is a term coined recently in the paper “Uzulangs: a Community of Musical Pattern Languages” presented at the NIME earlier this year
- The term is used for describing a family of livecoding languages that share a similar representation of musical structures
- The most prominent examples are TidalCycles and Strudel
- Patterns are represented as functions of time

01 The Uzu Family



- “Uzulangs” is a term coined recently in the paper “Uzulangs: a Community of Musical Pattern Languages” presented at the NIME earlier this year
- The term is used for describing a family of livecoding languages that share a similar representation of musical structures
- The most prominent examples are TidalCycles and Strudel
- Patterns are represented as functions of time
- Discrete events (like onsets) are determined by querying these functions

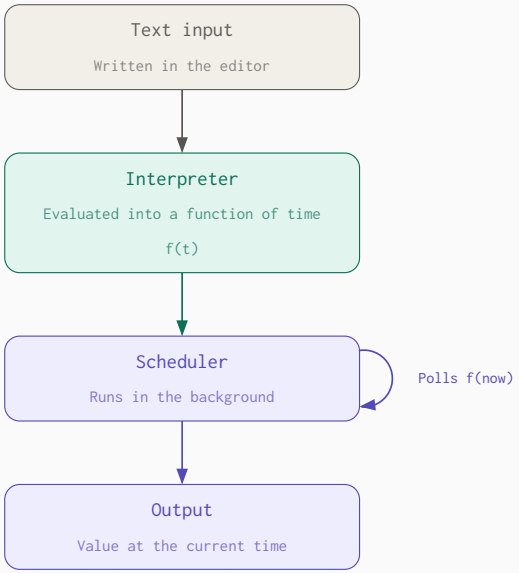
02 How to REPL in an Uzulang?



- REPL, short for read-eval-print-loop is an interactive environment in which a user can write and execute expressions of a programming language on the fly

02 How to REPL in an Uzulang?

- REPL, short for read-eval-print-loop is an interactive environment in which a user can write and execute expressions of a programming language on the fly
- for uzu languages it generally looks like this:



REPL for Uzulangs

03 Embedded Domain-Specific Languages



- An embedded domain-specific language (EDSL) is a set of functions and methods representing a specific domain (such as music) in a host language

03 Embedded Domain-Specific Languages



- An embedded domain-specific language (EDSL) is a set of functions and methods representing a specific domain (such as music) in a host language
- TidalCycles is embedded in Haskell and Strudel is embedded in JavaScript

03 Embedded Domain-Specific Languages



- An embedded domain-specific language (EDSL) is a set of functions and methods representing a specific domain (such as music) in a host language
- TidalCycles is embedded in Haskell and Strudel is embedded in JavaScript
- While this can be powerful for writing extensions, generally the tooling and syntax of the host languages is not laid out for livecoding

03 Embedded Domain-Specific Languages



- An embedded domain-specific language (EDSL) is a set of functions and methods representing a specific domain (such as music) in a host language
- TidalCycles is embedded in Haskell and Strudel is embedded in JavaScript
- While this can be powerful for writing extensions, generally the tooling and syntax of the host languages is not laid out for livecoding
- Both use string overloading to extend the syntax of the host language:

```
sometimes (fast 2) $ s "[bd sn ~ sn, hh*4]"
```

04 Breaking out of the Embedding



- In the 2021 paper “Alternate Timelines for TidalCycles” Alex McLean asks the question why the following expression shouldn’t be valid code:

```
jux <(rev) (iter 4)> $ sound [bd (every 3 (fast 4) [sn])]
```

04 Breaking out of the Embedding



- In the 2021 paper “Alternate Timelines for TidalCycles” Alex McLean asks the question why the following expression shouldn’t be valid code:

```
jux <(rev) (iter 4)> $ sound [bd (every 3 (fast 4) [sn])]
```

- This example makes it already clear, that in a truly uzu language that is not embedded everything should become a function of time, from simple number and text literals to (higher-order) functions

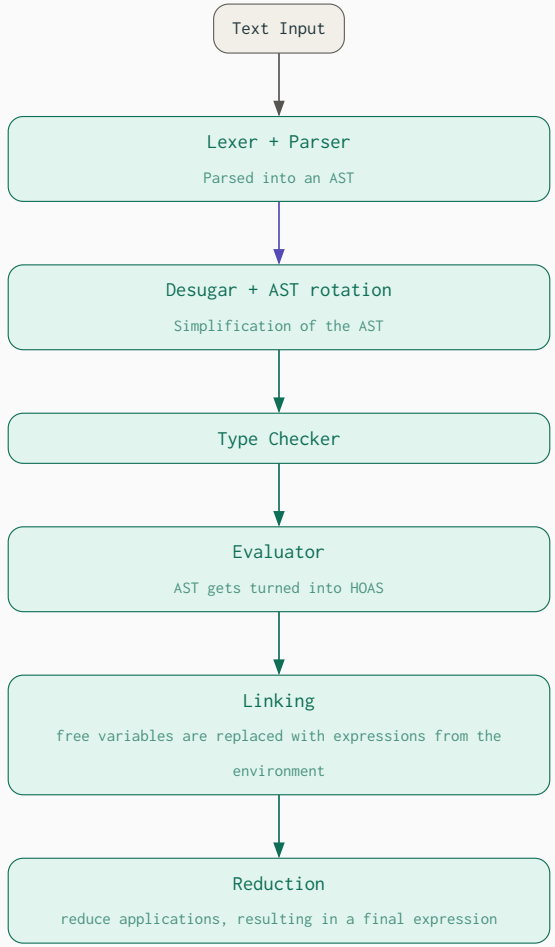
04 Breaking out of the Embedding



- In the 2021 paper “Alternate Timelines for TidalCycles” Alex McLean asks the question why the following expression shouldn’t be valid code:

```
jux <(rev) (iter 4)> $ sound [bd (every 3 (fast 4) [sn])]
```

- This example makes it already clear, that in a truly uzu language that is not embedded everything should become a function of time, from simple number and text literals to (higher-order) functions
- In the following, I will try to outline some of the implications of this



Architecture of the Interpreter,
roughly following Write You A Haskell by Stephen Diehl

06 Zwirn - The Idea (simplified)



- The core datatype of the Zwirn interpreter looks essentially like this:

```
type Zwirn a = Time -> (a, Time)
```

06 Zwirn - The Idea (simplified)



- The core datatype of the Zwirn interpreter looks essentially like this:

```
type Zwirn a = Time -> (a, Time)
```

- The inner Time value represents the structure (i.e. onsets)

06 Zwirn - The Idea (simplified)



- The core datatype of the Zwirn interpreter looks essentially like this:

```
type Zwirn a = Time -> (a, Time)
```

- The inner Time value represents the structure (i.e. onsets)
- An onset is exactly when the inner time crosses an integer value

06 Zwirn - The Idea (simplified)



- The core datatype of the Zwirn interpreter looks essentially like this:

```
type Zwirn a = Time -> (a, Time)
```

- The inner Time value represents the structure (i.e. onsets)
- An onset is exactly when the inner time crosses an integer value

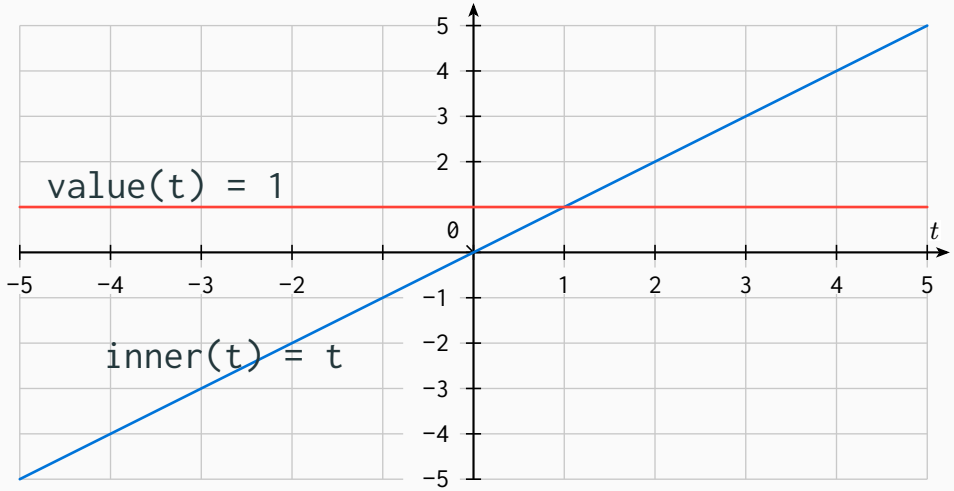
```
query :: Time -> Zwirn a -> Maybe a
query t z = if fractionalPart it == 0 then Just v else Nothing
        where (v,it) = z t
```

- In practice it is a bit more difficult to find the zeros of the fractional part of the inner time



- To lift values to signals we implement:

```
pure :: a -> Zwirn a
pure a = \t -> (a,t)
```



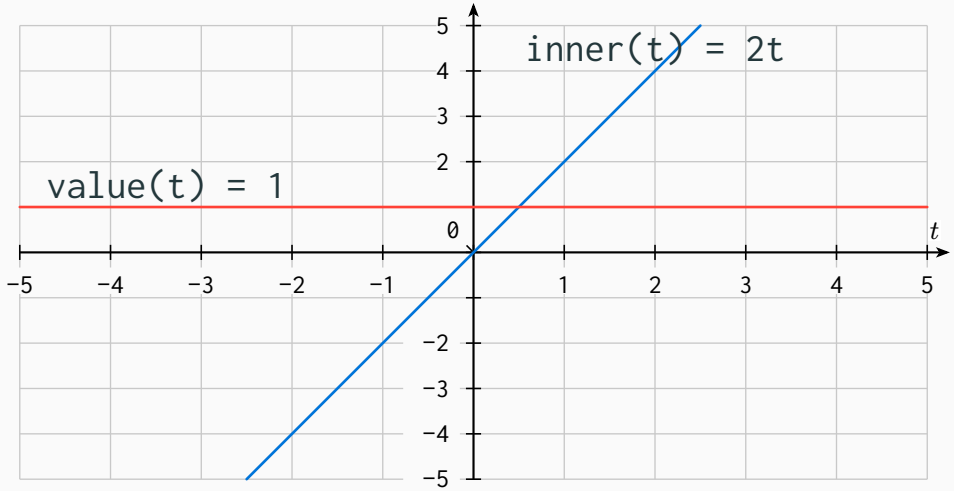
A visualisation of `pure 1`

06 Zwirn - The Idea (simplified)



- We can implement transformation functions, for example:

```
fast :: Time -> Zwirn a -> Zwirn a
fast factor z = \t -> z (factor * t)
```



A visualisation of `fast 2 (pure 1)`



- But the factor for fast should also be a function of time!



- But the factor for fast should also be a function of time!

```
fast :: Zwirn Time -> Zwirn a -> Zwirn a
fast fz z t = let (factor,ft) = fz t in z (factor * t)
```



- But the factor for `fast` should also be a function of time!

```
fast :: Zwirn Time -> Zwirn a -> Zwirn a
fast fz z t = let (factor,ft) = fz t in z (factor * t)
```

- Okay, but `fast` and it's partial application `fast (pure 2)` should also be functions of time!



- But the factor for `fast` should also be a function of time!

```
fast :: Zwirn Time -> Zwirn a -> Zwirn a
fast fz z t = let (factor,ft) = fz t in z (factor * t)
```

- Okay, but `fast` and it's partial application `fast (pure 2)` should also be functions of time!

```
fastZwirn :: Zwirn (Zwirn Time -> Zwirn (Zwirn a -> Zwirn a))
fastZwirn = pure (\fz -> pure (\z -> fast fz z))
```

06 Zwirn - The Idea (simplified)



- But the factor for `fast` should also be a function of time!

```
fast :: Zwirn Time -> Zwirn a -> Zwirn a
fast fz z t = let (factor,ft) = fz t in z (factor * t)
```

- Okay, but `fast` and it's partial application `fast (pure 2)` should also be functions of time!

```
fastZwirn :: Zwirn (Zwirn Time -> Zwirn (Zwirn a -> Zwirn a))
fastZwirn = pure (\fz -> pure (\z -> fast fz z))
```

- Seems complicated - how to deal with it?



- But the factor for fast should also be a function of time!

```
fast :: Zwirn Time -> Zwirn a -> Zwirn a
fast fz z t = let (factor,ft) = fz t in z (factor * t)
```

- Okay, but fast and it's partial application fast (pure 2) should also be functions of time!

```
fastZwirn :: Zwirn (Zwirn Time -> Zwirn (Zwirn a -> Zwirn a))
fastZwirn = pure (\fz -> pure (\z -> fast fz z))
```

- Seems complicated - how to deal with it?

```
apply :: Zwirn (Zwirn a -> Zwirn b) -> Zwirn a -> Zwirn b
```

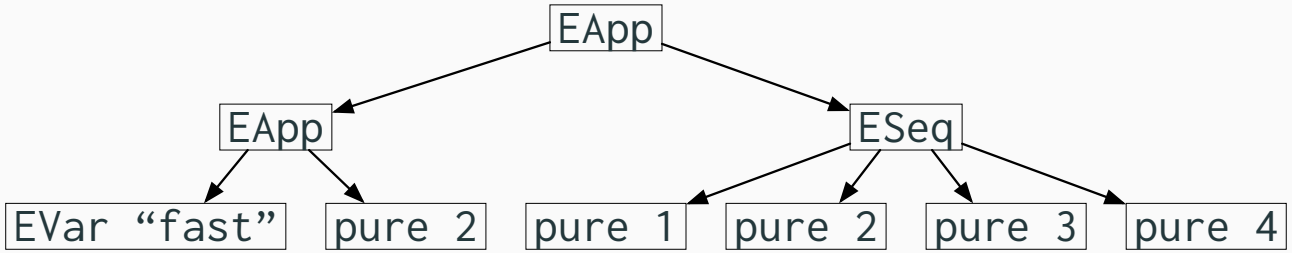
```
apply fastZwirn (pure 2) :: Zwirn (Zwirn a -> Zwirn a)
```

```
apply (apply fastZwirn (pure 2)) (pure 1) :: Zwirn Int
```

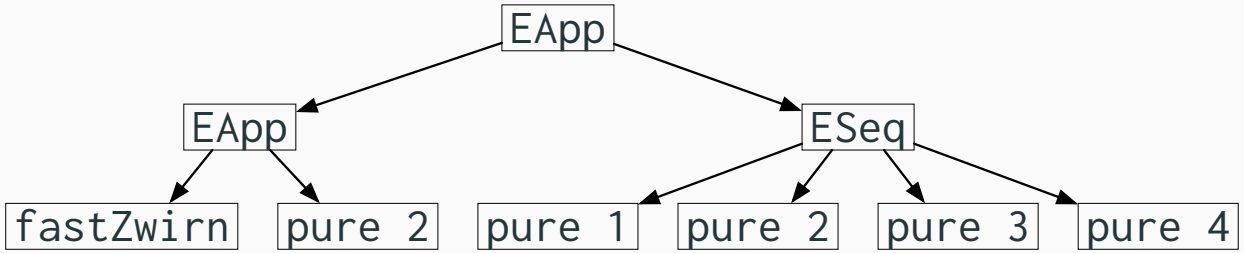
07 Example - fast 2 [1 2 3 4]



After being parsed, transformed and typechecked, the Zwirn code `fast 2 [1 2 3 4]` results in the following HOAS:

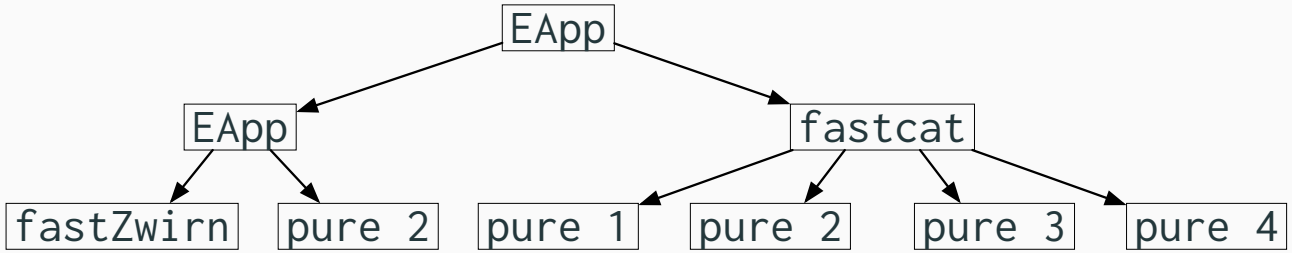


07 Example - fast 2 [1 2 3 4]



- By linking with the environment, the variables get replaced by expressions:

07 Example - fast 2 [1 2 3 4]

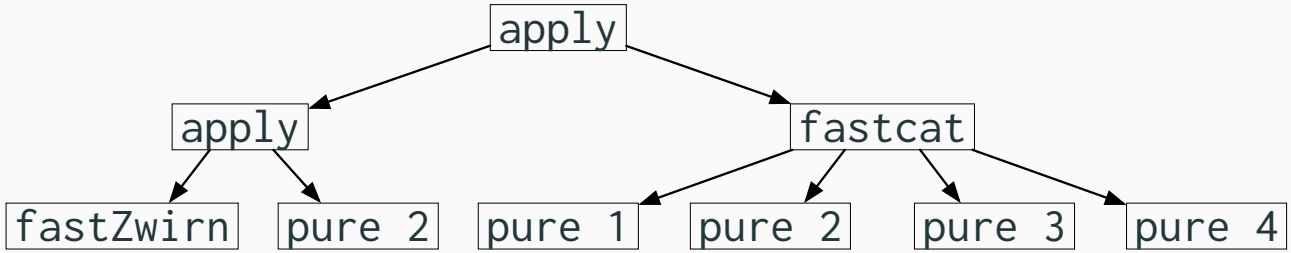


- `ESeq` nodes get evaluated with corresponding function `fastcat :: [Zwirn a] -> Zwirn a`:

07 Example - fast 2 [1 2 3 4]



- `EApp` nodes get evaluated with corresponding function `apply :: Zwirn (Zwirn a -> Zwirn b) -> Zwirn a -> Zwirn b`:



- The resulting final expression can then be unpacked as `Zwirn Int`
- This expression is then placed in the scheduler



- This could be seen as a general method for implementing small esoteric functional languages



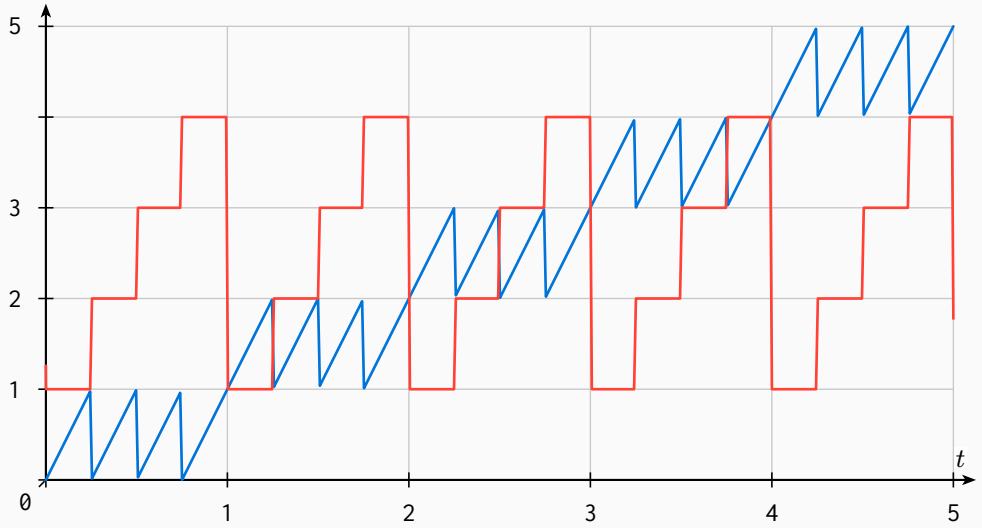
- This could be seen as a general method for implementing small esoteric functional languages
- All one needs is:
  - `pure :: a -> m a`
  - `apply :: m (m a -> m b) -> m a -> m b`



- This could be seen as a general method for implementing small esoteric functional languages
- All one needs is:
  - `pure :: a -> m a`
  - `apply :: m (m a -> m b) -> m a -> m b`
- In fact, if we choose `m = Identity`, with `pure ≡ id` and `apply ≡ ($)`, we get the polymorphic lambda calculus



Thank you!



A visualisation of the sequence [1 2 3 4]

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using $\rightarrow$

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using →
- For example:
  - `fast` has the Zwirn type `Number → a → a`

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using →
- For example:
  - `fast` has the Zwirn type `Number → a → a`
  - `[1 2 3 4]` has the type `Number`

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using →
- For example:
  - `fast` has the Zwirn type `Number → a → a`
  - `[1 2 3 4]` has the type `Number`
  - `[fast 2]` results in a type error

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using →
- For example:
  - `fast` has the Zwirn type `Number → a → a`
  - `[1 2 3 4]` has the type `Number`
  - `[fast 2]` results in a type error
  - `fast 2` has the type `a → a`

011 Addendum (Types)



- Zwirn has a Hindley-Milner Typesystem with the following types:
  - Basic types: Number, Text, Map, Action
  - Type variables: a,b,c..
  - Function types are built recursively from these basic types using →
- For example:
  - `fast` has the Zwirn type `Number → a → a`
  - `[1 2 3 4]` has the type `Number`
  - `[fast 2]` results in a type error
  - `fast 2` has the type `a → a`
- If every expression is a function of time, you can only distinguish them by their values!

012 Addendum (Types)



- Zwirn types can be mapped back to Haskell types:
  - $\tau(\texttt{Number}) = \texttt{Zwirn Time}$
  - $\tau(\texttt{Text}) = \texttt{Zwirn Text}$
  - $\tau(\texttt{Map}) = \texttt{Zwirn ControlMap}$
  - $\tau(\texttt{Action}) = \texttt{Zwirn (IO ())}$
  - $\tau(\texttt{a}) = \texttt{Zwirn a}$, for a type variable a
  - $\tau(\alpha \to \beta) = \texttt{Zwirn } (\tau(\alpha) \to \tau(\beta))$, with α, β any Zwirn types

012 Addendum (Types)



- Zwirn types can be mapped back to Haskell types:
  - $\tau(\texttt{Number}) = \texttt{Zwirn Time}$
  - $\tau(\texttt{Text}) = \texttt{Zwirn Text}$
  - $\tau(\texttt{Map}) = \texttt{Zwirn ControlMap}$
  - $\tau(\texttt{Action}) = \texttt{Zwirn (IO ())}$
  - $\tau(\texttt{a}) = \texttt{Zwirn a}$, for a type variable a
  - $\tau(\alpha \to \beta) = \texttt{Zwirn } (\tau(\alpha) \to \tau(\beta))$, with α, β any Zwirn types
- For example:
  - `fast` has the Zwirn type `Number → a → a`
  - this translates to the following Haskell type:
    `Zwirn (Zwirn Time -> Zwirn (Zwirn a -> Zwirn a))`