STRANGELOOP 2013

A conference for the creators and users of the languages, libraries, and techniques at the forefront of the industry.

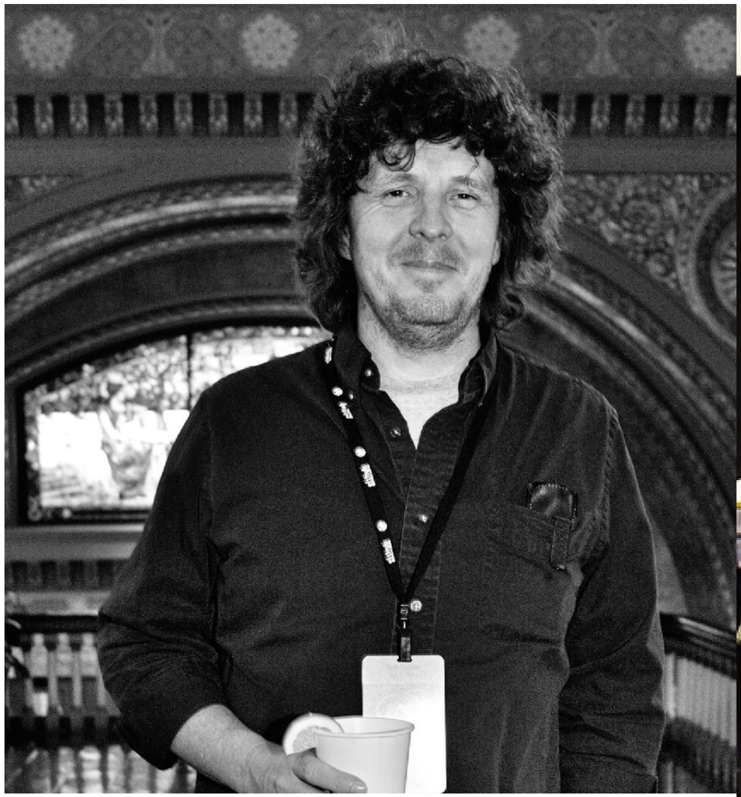
HN

LOTS OF COOL EMERGING TECHS!

BUT I'M NOT GOING TO TALK ABOUT ANY OF THEM TODAY.

PROGRAMMING LANGUAGE DESIGN

TYPE SYSTEMS

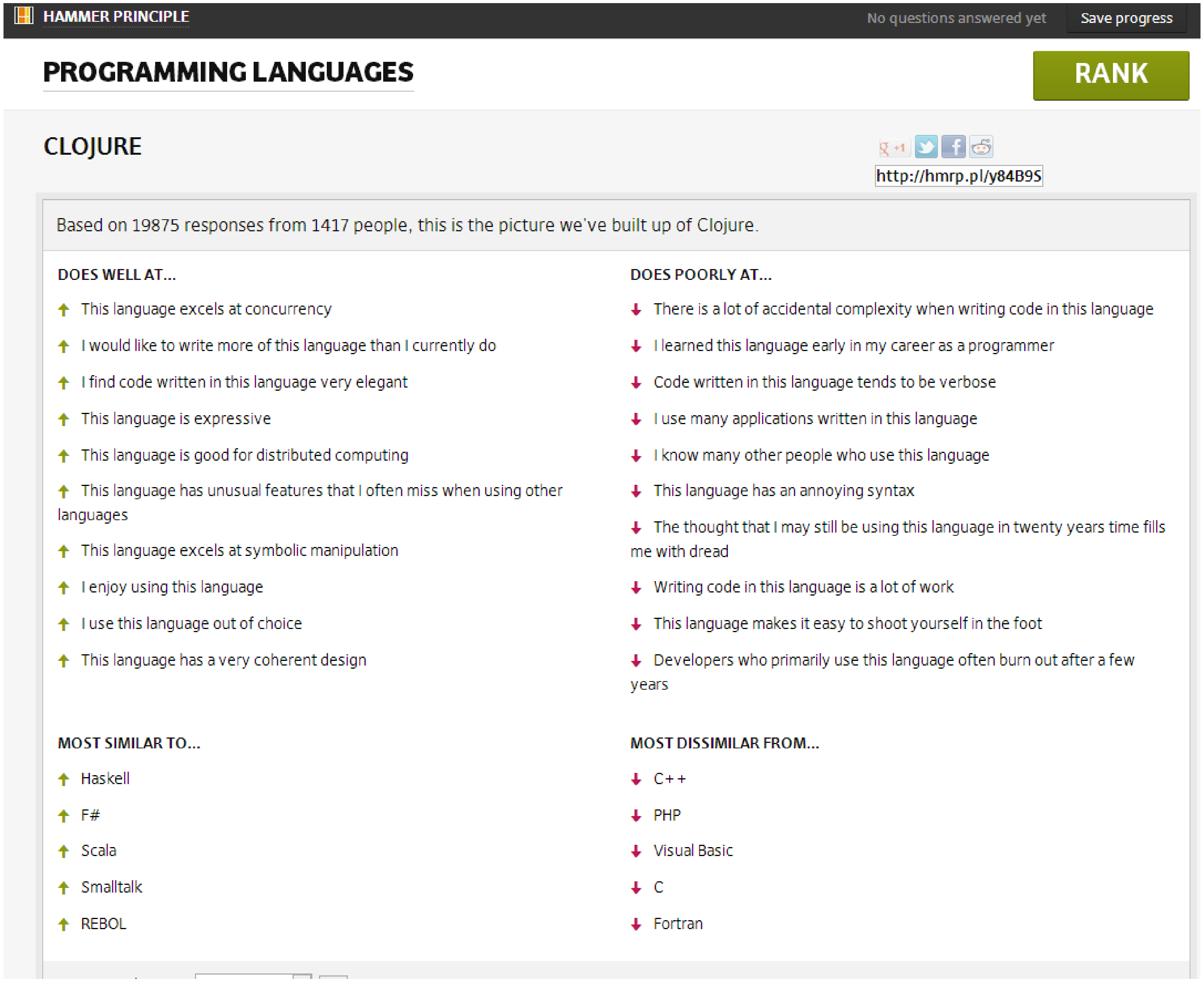


PROGRAMMING SUPERHEROES

These aren't the same at all...

```
(def hello (fn [] "Hello world"))
;;-> #'user/hello
(hello)
;;-> "Hello world"
```

```
object HelloWorld {
    def main(args: Array[String]) {
        println("Hello, world!")
    }
}
// > "Hello, world!"
```



HAMMER PRINCIPLE — No questions answered yet — Save progress

PROGRAMMING LANGUAGES — RANK

CLOJURE

http://hmrp.pl/y84B9S

Based on 19875 responses from 1417 people, this is the picture we've built up of Clojure.

DOES WELL AT...

- This language excels at concurrency
- I would like to write more of this language than I currently do
- I find code written in this language very elegant
- This language is expressive
- This language is good for distributed computing
- This language has unusual features that I often miss when using other languages
- This language excels at symbolic manipulation
- I enjoy using this language
- I use this language out of choice
- This language has a very coherent design

DOES POORLY AT...

- There is a lot of accidental complexity when writing code in this language
- I learned this language early in my career as a programmer
- Code written in this language tends to be verbose
- I use many applications written in this language
- I know many other people who use this language
- This language has an annoying syntax
- The thought that I may still be using this language in twenty years time fills me with dread
- Writing code in this language is a lot of work
- This language makes it easy to shoot yourself in the foot
- Developers who primarily use this language often burn out after a few years

MOST SIMILAR TO...

- Haskell
- F#
- Scala
- Smalltalk
- REBOL

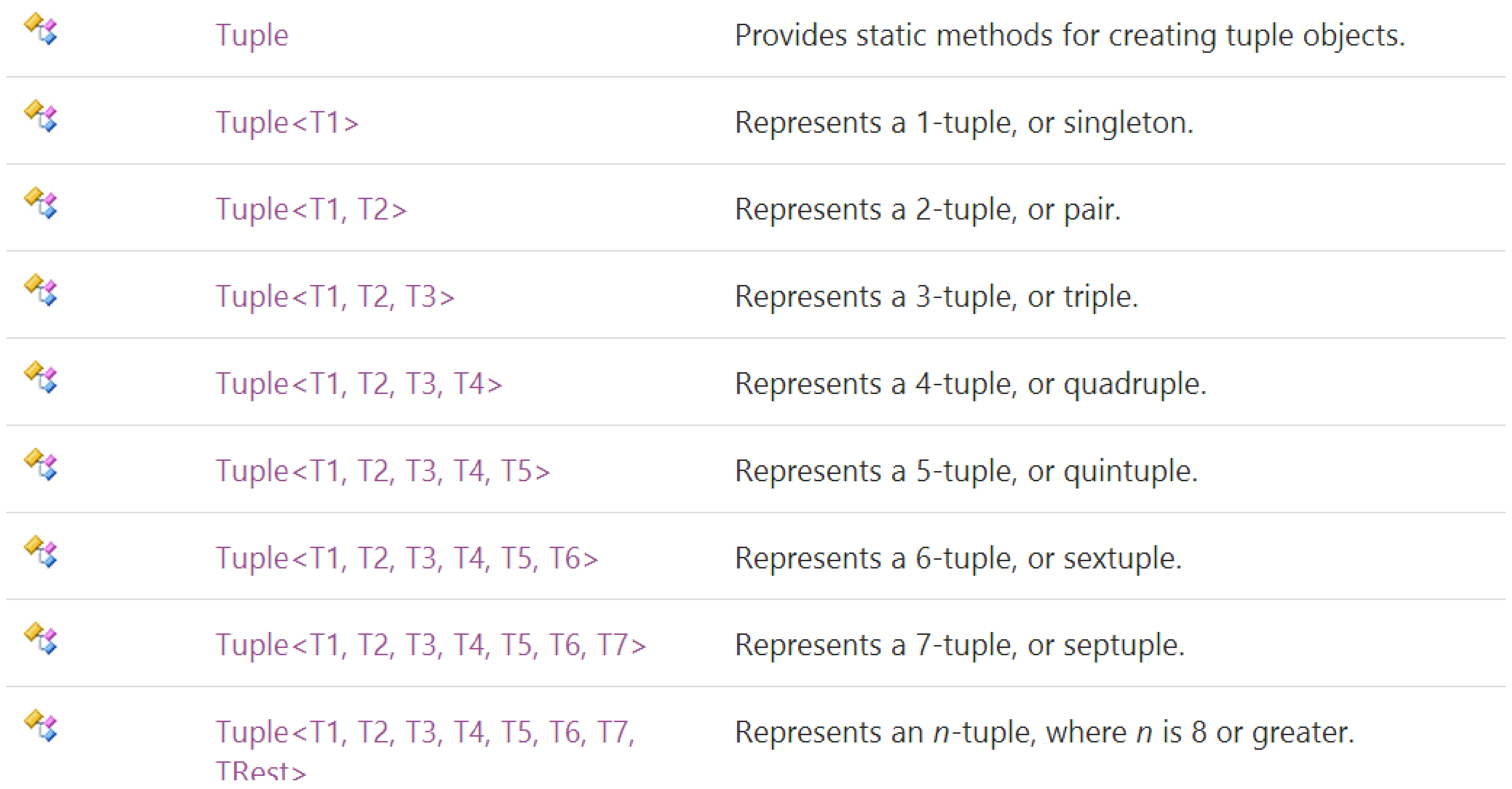
MOST DISSIMILAR FROM...

- C++
- PHP
- Visual Basic
- C
- Fortran

php

Types are a tradeoff between safety and expressiveness.

THE ELEGANT TUPLE API



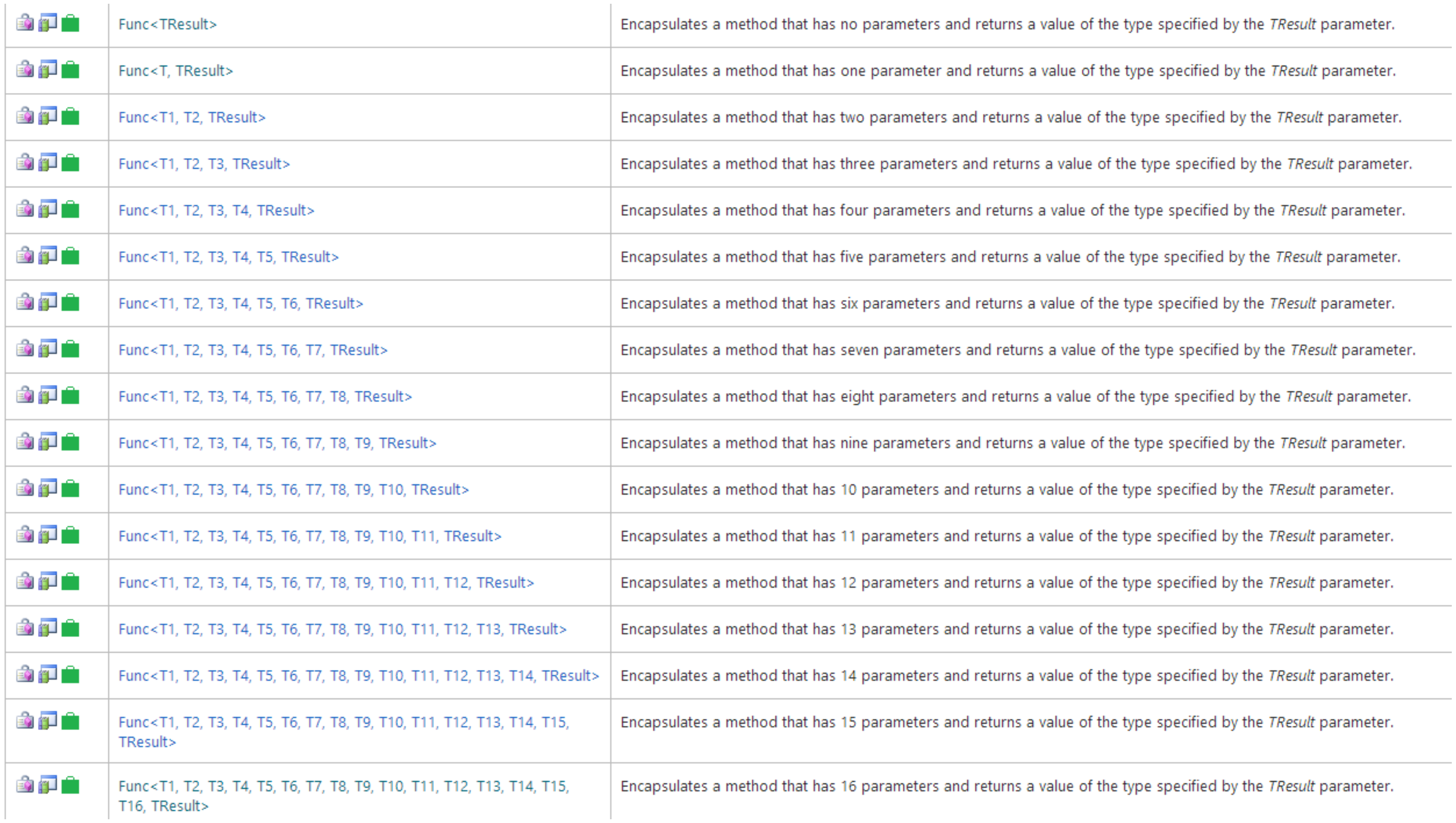
Tuple	Provides static methods for creating tuple objects.
Tuple<T1>	Represents a 1-tuple, or singleton.
Tuple<T1, T2>	Represents a 2-tuple, or pair.
Tuple<T1, T2, T3>	Represents a 3-tuple, or triple.
Tuple<T1, T2, T3, T4>	Represents a 4-tuple, or quadruple.
Tuple<T1, T2, T3, T4, T5>	Represents a 5-tuple, or quintuple.
Tuple<T1, T2, T3, T4, T5, T6>	Represents a 6-tuple, or sextuple.
Tuple<T1, T2, T3, T4, T5, T6, T7>	Represents a 7-tuple, or septuple.
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>	Represents an *n*-tuple, where *n* is 8 or greater.

Tuple<T1, T2, T3, T4, T5, T6, T7, Tuple<T1, T2, T3, T4, T5, T6 > >

myTuple.Rest.Item5

TYPE CRAZINESS

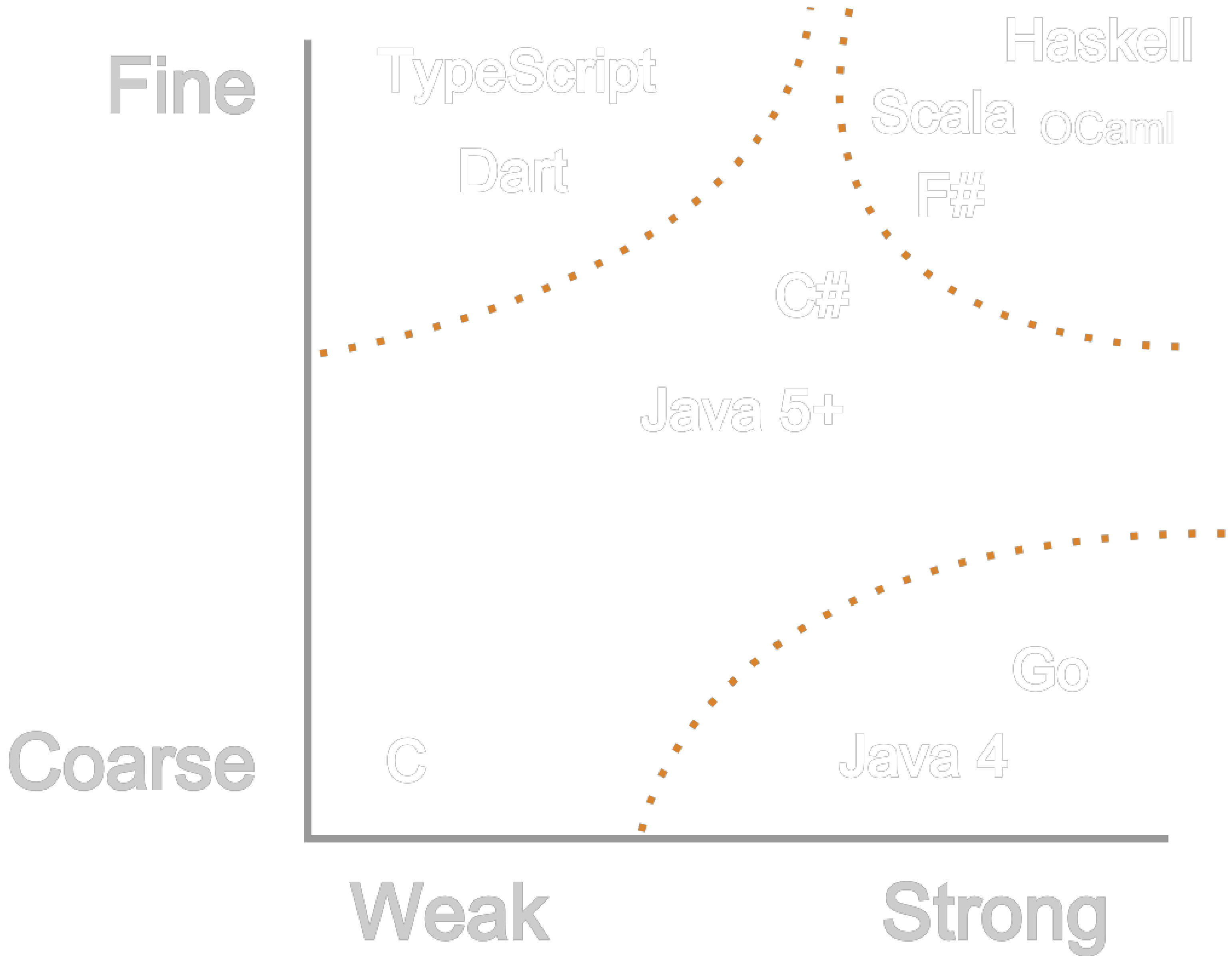
```
Tuple<n*T>(T1 item1, T2 item2, ....)
```



	Func<TResult>	Encapsulates a method that has no parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T, TResult>	Encapsulates a method that has one parameter and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, TResult>	Encapsulates a method that has two parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, TResult>	Encapsulates a method that has three parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, TResult>	Encapsulates a method that has four parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, TResult>	Encapsulates a method that has five parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, TResult>	Encapsulates a method that has six parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, TResult>	Encapsulates a method that has seven parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, TResult>	Encapsulates a method that has eight parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, TResult>	Encapsulates a method that has nine parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, TResult>	Encapsulates a method that has 10 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, TResult>	Encapsulates a method that has 11 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, TResult>	Encapsulates a method that has 12 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, TResult>	Encapsulates a method that has 13 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, TResult>	Encapsulates a method that has 14 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, TResult>	Encapsulates a method that has 15 parameters and returns a value of the type specified by the *TResult* parameter.
	Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult>	Encapsulates a method that has 16 parameters and returns a value of the type specified by the *TResult* parameter.

`Action` and `Func` objects need some help too.

TYPE COMPROMISE



WARNING!

A monad is just a monoid in the category of endofunctors

... what's the problem?

NullPointerExceptions suck.
Let's get rid of them!

THE OPTION TYPE

```
val res1: Int = 42
val res2: Int = null
val res3: Int = 3

System.out.println(res1 + res2 + res3)
// Runtime error; type system fail!
```

THE OPTION TYPE

```
val res1: Option[Int] = Some(42)
val res2: Option[Int] = None
val res3: Option[Int] = Some(3)

System.out.println(res1 + res2 + res3)
// Compiler error, not runtime!  We can't add option types.
```

THE OPTION TYPE

```
val res1: Option[Int] = Some(42)
val res2: Option[Int] = None
val res3: Option[Int] = Some(3)

if(res1.isEmpty) return None
if(res2.isEmpty) return None
if(res3.isempty) return None

System.out.println(res1.get + res2.get + res3.get)
// No printing!
```

THE OPTION MONAD

```
val fails = for {
  res1 <- Some(42)
  res2 <- None
  res3 <- Some(3)
} yield (res1, res2, res3)
println(fails getOrElse "Nada")
// > "Nada"
```

THE OPTION MONAD

```
val fails = for {
  res1 <- Some(42)
  res2 <- Some(0)
  res3 <- Some(3)
} yield (res1, res2, res3)
println(fails getOrElse "Nada")
// > 45
```

ASYNC MONAD

```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }
```

MONADS ARE YESTERDAY'S NEWS.

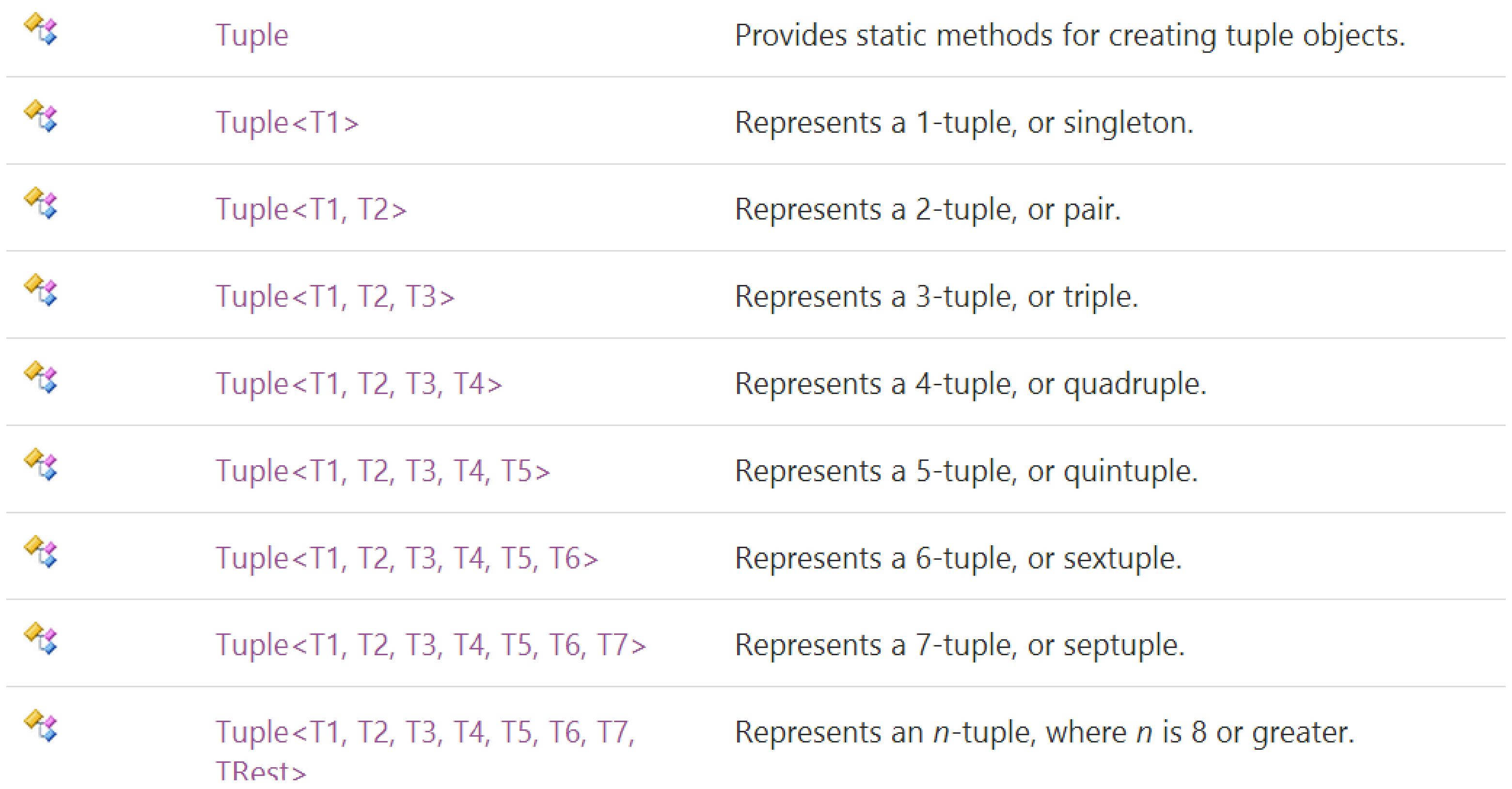
SO WHAT'S NEW?

DEPENDENT TYPES

BOUNDS CHECKING

```
class MyList<A> {
    public pairwiseAdd(MyList<A> other){
        var result = new MyList<A>();
        for (var i = 0; i<this.size(); i++){
            result.add(this.get(i) + other.get(i));
        }
        return result;
    }
}
var list1 = MyList(1,2,3);
var list2 = MyList(1,2,3,4);
list1.pairwiseAdd(list2);
// Runtime error, the lists are different sizes!
// We're making up syntax as we go along.
class MyList<A, listLength> {
    public pairwiseAdd(MyList<A, listLength> other){
        var result = new MyList<A, 2*listLength>();
        for (var i = 0; i<this.size(); i++){
            result.add(this.get(i) + other.get(i));
        }
        return result;
    }
}
var list1 = MyList(1,2,3);
var list2 = MyList(1,2,3,4);
list1.pairwiseAdd(list2);
// Compile time error; listLength doesn't match!
```

BACK TO THE TUPLE PROBLEM…



Tuple	Provides static methods for creating tuple objects.
Tuple<T1>	Represents a 1-tuple, or singleton.
Tuple<T1, T2>	Represents a 2-tuple, or pair.
Tuple<T1, T2, T3>	Represents a 3-tuple, or triple.
Tuple<T1, T2, T3, T4>	Represents a 4-tuple, or quadruple.
Tuple<T1, T2, T3, T4, T5>	Represents a 5-tuple, or quintuple.
Tuple<T1, T2, T3, T4, T5, T6>	Represents a 6-tuple, or sextuple.
Tuple<T1, T2, T3, T4, T5, T6, T7>	Represents a 7-tuple, or septuple.
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>	Represents an *n*-tuple, where *n* is 8 or greater.

unresolved.. for now

Types aren't worth the hassle!

-- Clojure

I’m not against types, but I don’t know of any type systems that aren’t a complete pain, so I still like dynamic typing.

-- Alan Kay

POINT

- New programming languages are still coming out.
- I'd argue that's a good thing.
- Pay attention, even if you still want to use Java.