Why be functional?

"I conclude that there are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies and
the other way is to make it so complicated that there are no obvious
deficiencies. The first method is far more difficult."

- C. A. R. Hoare

The goal of my talk is to convince you to do more functional programming.

- We start by looking at complexity and simplicity.
- Then we'll examine how our languages influence our design and even the way we think about programming.
- Finally, we'll look at functional programming and how it meets our goal for managing complexity.

What is software complexity?

“The purpose of software engineering is to control complexity, not to create it."

Dr Pamela Zave
AT&T Labs

- Morphology - the form and structure of a program
- Interrelationships and dependencies, coupling
- Any code that can be decomposed
- Any code that isn't beautiful
- The enemy of reliability

- Complexity is the morphology of software - that is, its form and structure. All form and structure is complexity - we just accept some complexity.
- Complexity is the interrelationships and dependencies of software components, how components are coupled. This is demonstrated by the impact one component may have on another.
- Complexity is any code that can be, but hasn't been, decomposed into simple components.
- Complexity is the part of our code that isn't beautiful. Beauty is the ultimate defense against complexity.
- Complexity is the enemy of reliability. Complexity and reliability are inversely proportional.

Why is complexity bad?

"[W]e have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making."

Edsger Djikstra

- We can't reason about our software.
- We can't test our software.
- We can't prove our software.
- Ergo, we can't trust it.

- There are limits to our ability to reason about a system, and there are limits to our ability to build complex systems. Unfortunately, our ability to build complex systems far outstrips our ability to reason about them.
- Testing is never 100% effective, but the effectiveness of testing decreases as the complexity of the software increases.
- There are ways we can use formal reasoning - that is, mathematics and logic - to prove the correctness of a system. But proving the correctness of a system becomes orders of magnitude more difficult - practically impossible - as the complexity increases. Not to mention that it's tremendously difficult to create a complex system that is correct.
- If we can't reason about software, can't test our software, and can't prove our software, then we can't trust our software.

Types of complexity

"Simplicity does not precede complexity, but follows it."

Alan J. Perlis

- Essential complexity
  - Inputs
  - Outputs
- Accidental complexity
  - Everything else

Essential complexity is the required inputs and outputs, and the business logic that defines the relationship between inputs and outputs.

Accidental complexity is everything else, including the program that implements the business logic.

Informal reasoning

"It is easier to write an incorrect program than understand a correct one."

Alan J. Perlis

- Needs to address the functional requirements
- Understanding a system from the inside out

Informal reasoning is understanding how a program works, and why. Being able to translate what the program needs to do into how the program will do it.

Good informal reasoning will lead to simpler software.

Complex software is difficult to reason about. Complexity breeds complexity. A program that is difficult to reason about becomes more complex, which becomes more difficult to reason about, and so on.

Testing

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

C. A. R. Hoare

- Looking from the outside in
- Detects errors
- Limited effectiveness

Testing leads to more bugs being found. Reasoning leads to fewer bugs being coded.
Testing is important, but it definitely has limitations too:

- Each test demonstrates correctness for only one set of inputs. (Property based testing is an exception
- Mutable state complicates testing drastically. A mere 24 bytes of state, 6 32-bit words, contains more possible states than there are atoms in the earth.
- Testing only demonstrates correctness in the code, not in the specification.
- And again, the more complex the software, the less effective our tests will be.

Formal reasoning

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

Edsger Djikstra

- Formal proof is proving the form, or structure, of a program
- Functional correctness: Proving that a function will return the correct value using MATH!

- Formal reasoning defines methods for proving, mathematically, the correctness of a program. How that's done is outside the scope of this talk.
- Formal proof of correctness can only be proven mathematically if both the algorithm and the specification are also given formally; i.e. mathematically.

What is simplicity?

"The price of reliability is the pursuit of the utmost simplicity."

C. A. R. Hoare

- We can reason about it
- We can test it
- We can prove it

If we can reason about software, can test our software, and can prove our software, then we can trust our software. Simple.

"Programming language design revolves around program design. A language's design reflects the opinions of its creators about the proper design of programs."

Reginald Braithwaite

The languages we use not only affect the shape of our code, they change the way we think.

You could say that programming in a programming language programs us.

Imperative versus declarative

"OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts."

Michael Feathers

- Declarative programming describes relationships and answers questions about them.
- Imperative programming changes the world.

Declarative programming describes relationships and then answers questions about them.
Imperative programming changes the state of something in the world, or at least in a program's model of the world.
 * Could be the state of a variable.
 * Could be the state of a user's screen.
 * Could be the state of a servo controlling the ailerons on an ICBM.

State and time

"Lisp doesn't look any deader than usual to me."

David Thornley

- Imperative programming:
  The current state is the identity.
- Declarative programming:
  Identity is a collection of states over time.

In imperative languages, the current state typically defines identity. That state is changed by code: we say "Set X to 2." Identity is timeless; the only state we can get from it is "now". When we have multiple parts of the code changing our state on us, it will be increasingly difficult to guess what values we'll get.

In declarative languages, state doesn't change because values don't change. We don't say that the value of a variable changes any more than we'd say the value of 1 changes. Identity is a collection of states, each transformed into the next, passed to a function only when a transformation is needed. Time is inherent; each version of our state exists as separately.

The simplicity test for languages

"Lisp is still #1 for key algorithmic techniques such as recursion and condescension."

Verity Stob

- Does it help us to create code we can reason about?
- Does it help us to create code we can test?
- Does it help us to create code we can prove?

Maybe in evolution species tend to move from simple to complex.

That’s not our job. Our job is to control complexity, to model an inherently complex world in the simplest possible ways. We can choose a language based on how well it helps us do that.

Some corollaries

"Software is a gas; it expands to fill its container."

Nathan Myhrvold

- Power corrupts:
  Powerful tools ~= more complexity
- Slippery slope:
  Opinionated may be better than flexible

- The ability to allocate memory, manipulate pointers, or manage garbage collection is sometimes needed, but on the whole it leads to a more complex system, not a simpler one.
- An unopinionated programming language might let you choose to be declarative or imperative, to mutate or not, to use functions or classes. The result is that you have the flexibility to build more complex systems.
- There are exceptions, of course. We typically use lower level, flexible, powerful languages to build opinionated, more restrained languages. My favorite geeky tee-shirt is probably the one that reads "My compiler compiled your compiler."

Functions in functional programming

"To iterate is human, to recurse divine."

L. Peter Deutsch

- Referential transparency
- Composition
- Higher order functions
- Recursive

Most programming language have functions. In functional programming, functions are the stars, not the supporting players.

- Pure functions have exactly one result for any given input and do not generate side-effects. The result of a call to a pure function can replace the call itself for any given value. This is called referential transparency.
- Functions can be made of functions, and this is called composition.
- Functions that can have functions passed to them as arguments are called higher order functions.
- Because functional programming languages typically use immutable state, they don’t do loops (loop variables are state.) Recursion does not use mutable state, and is typically more efficient. In fact, many imperative programming languages implement iteration as recursion in the virtual machine.

Things to do while programming functionally

""In order to understand recursion, you must first understand recursion."

Unknown

- Testing
- Distributed computing
- Parallel computing
- Lazy evaluation

- Testing pure functions is super easy. Your inputs are known, your outputs don't change unless your inputs do, and you don't rely on mutable state to calculate a result. You probably won’t ever need to create a mock in a purely functional program.
- Distributing pure functions is super easy. Because a pure function will always return the same value when given the same inputs, it makes no difference where we run it.
- Parallelizing pure functions is super easy. Two functions that don't rely on the output from each other can always run in parallel.
- Even faster than distributing and parallelizing functions is to never run them. With lazy evaluation, we don't evaluate a function before it's needed. Sometimes it’s never needed.

Simplicity test for pure functions

"Take Lisp, you know it's the most beautiful language in the world - at least up until Haskell came along."

Larry Wall

- Can we reason about it? Yes.
- Can we test it? Yes.
- Can we prove it? Probably.

- Can we reason about it? Yes. You have an input, and you have an output, and the same input always generates the same output. If the essential complexity of a function is significant, we can build that function out of smaller, simpler functions.
- Can we test it? Yes. You have an input, and you have an output, and the same input always generates the same output.
- Can we prove it? Maybe. Functional programming is based on Lambda calculus and as such, follows a provable theorem. In fact, both Lambda calculus and Turing machines were created in large part as a way to determine whether a calculation is computable. Turns out it is possible to write functions that will never return, and it isn't always be possible to tell the difference between a function that will never return, and one that just hasn't returned *yet*. Then there's the complexity of the software itself still. A short function may be provable. A long program may also be provable, but not without impractical amounts of time and effort.

Simplicity test for immutable state and functional programming

"If you're willing to restrict the flexibility of your approach, you can almost always do something better."

John Carmack

- Can we reason about it? Yes.
- Can we test it? Yes.
- Can we prove it? Yes.

We've already discussed mutable and immutable state. Let's run immutable state through our simplicity test.

- Can we reason about it? Yes, because state is typically an input to, and an output from a function, and we can reason about the functions.
- Can we test it? Yes, see above.
- Can we prove it? Yes, at least to the extent we can prove our functions are correct.

What about side effects?

"In the end, any program must manipulate state. A program that has no side effects whatsoever is a kind of black box. All you can tell is that the box gets hotter."

Simon Peyton-Jones

- Can we reason about it? Probably.
- Can we test it? Maybe.
- Can we prove it? Nope.
- Does this matter? Maybe not…

Sooner or later we want to change the world. The thing is to do your side effects as late as possible, with an imperative shell wrapped around a functional core. In the core, do everything you can do functionally. In the shell, do only side effects.

- Can we reason about it? Maybe. Or maybe we won't even know what kind of client is using our interface.
- Can we test it? Not really, but we can mock up how we think the external world is going to work and see how our programs interact with that. By isolating our side effects and imperative programming, we limit the amount of work we need to do mocking the real world.
- Can we prove it? Nope. We can imperatively insist on stuff all day long, but in the end, we don't run the world.
- Does it matter? If our imperative shell is providing a minimal interface between our functional program and the rest of the world, chances are good we will be able to trust it based solely on informal reasoning.

Metaphor #1
Pipeline

“Object-oriented design is the roman numerals of computing.”

Rob Pike

- By composition: `c ( b ( a ) )`
- By concatenation: `a |> b |> c` (`a b c` in Forth)

Before we finish, let's talk about how to think about programming in a functional language.

- With object oriented languages, we have a consistent overarching metaphor. There's no similar master metaphor for functional programming, but here are some that get us started.
- Pipelines are really just function composition. This is the basis of functional programming; we build programs from larger functions built of smaller functions. It's functions all the way down.
- Many functional programming languages support concatenation as well as composition. The first example is in Elixir, and the second is Forth.

Metaphor #2: Concurrency-oriented programming

“Inside every well-written large program is a well-written small program.”

C. A. R. Hoare

- People are actors. We act upon the world.
- What’s past is past, and we can’t change it.
- We are isolated from each other, but connect by passing messages.
- We can't pick ourselves up when we crash, but we can watch out for each other.
- Processes should work like this too.

Concurrency-oriented programming is defined by Joe Armstrong, one of the creators of Erlang, and it’s based on looking at how the world works.

- COPLs must support processes, which can be thought of as a self-contained virtual machine.
- Several processes operating on the same machine must be strongly isolated, so that a fault in one process shouldn't adversely affect another process unless that's the behavior we want - one process to detect failure in another process, and know why.
- Each process must be identified by a unique unforgeable identifier. We will call this the Pid of the process.
- Processes don't share state, they send messages. If you know the Pid of a process then you can send a message to the process. BUT message passing is assumed to be unreliable with no guarantee of delivery.

Metaphor #3:
A river

"What I love most about rivers is you can't step in the same river twice. The water's always changing, always flowing."

Rich Hickey
(Probably.)

- (One of) Rich Hickey's definitions for Identity:
  "A putative entity we associate with a series of causally related values or states over time."
- My definition, inspired by Rich Hickey:
  "Identity is a bunch of states one after another that we associate with a thing."

Many programs model things in the real world, so they need to support identity.

Hickey's metaphor doesn't see the world as "becoming" in the way the caterpillar becomes a beautiful butterfly. I think he sees it as being a series of unchangeable states over time, and that our tendency to see things as discrete objects is a side effect of us being unable to move around on the time axis as easily as we move around on the space axes.

I suspect Hickey is not bound by time, and he can see even how each thing has its own timeline.

Metaphor #4: Mathematics

“Mathematicians stand on each others' shoulders. Computer scientists stand on each others' toes.”

Richard Hamming.

- Lambda calculus
- Combinatorial logic

Without getting into specifics, lambda calculus is the basis of most of our functional programming languages - except for a few languages, like Forth, Joy, Factor, and Cat, that are based on combinatorial logic.

- In fact, when McCarthy wrote about his ideas for Lisp in 1958, he was defining a computing machine that would implement Alonzo Church's lambda calculus.
- Shortly after that, Steve Russell - incidentally, the guy who created the world's first video game - went to McCarthy and said he thought he could implement McCarthy's ideas on an IBM 704 computer. McCarthy told him he was confusing theory with practice and that it couldn't be done. So Russell went and did it, and that is how Lisp was born.

"If debugging is the process of removing bugs,
then programming must be the process of putting them in."

- Edsger Djikstra

Questions?