Functional Programming is Easy, and Good for You

Matthias Felleisen (PLT)
Northeastern University
I am not a salesman.
Functional Programming
Functional Programming  Functional Programming Languages
Functional Programming ≠ Functional Programming Languages
Theorem

Functional Programming $\neq$ Functional Programming Languages

Proof:
Theorem

Functional Programming $\neq$ Functional Programming Languages

Proof:
Functional Programming

Functional Programming Languages
Functional Programming

Functional Programming Languages

pure
Clean

mostly
OCaml
Functional Programming

Functional Programming Languages

pure
Clean

strict
all others

lazy
Haskell

mostly
OCaml
Functional Programming

- pure
- Clean

Functional Programming Languages

- lazy
- Haskell

- typed
- SML

- untyped
- Scheme

- strict
- all others

- mostly
- OCaml
Functional Programming

pure
Clean

higher-order
all others

untyped
Scheme

strict
all others

lazy
Haskell

typed
SML

first-order
ACL2

mostly
OCaml

Functional Programming Languages
Functional Programming

- Pure
- Clean
- Lazy
- Higher-order
- All others
- Typed
- First-order
- Standard VM
- F#, Scala
- Special VM
- Racket
- Untyped
- Scheme
- Mostly
- OCaml
- Strict
- All others
Functional Programming

- Functional Programming Languages
  - pure
    - Clean
  - higher-order
    - all others
  - lazy
    - Haskell
  - parallel
    - Clojure
  - standard VM
    - F#, Scala
  - special VM
    - Racket
  - distributed
    - Erlang
  - typed
    - SML
  - first-order
    - ACL2
  - mostly
    - OCaml
  - strict
    - Scheme
    - all others
If all of this is **functional programming (languages)**, isn’t it all **overwhelming and difficult**?
If all of this is functional programming (languages), isn’t it all overwhelming and difficult?

Not at all. And I am here to explain what, why, and +/-.
What is Functional Programming?
What is a Functional Programming Language?
Pop Quiz
Pop Quiz: Who said this?

Though [it] came from many motivations, ... one was to find a more flexible version of assignment, and then to try to eliminate it altogether.

Favor immutability.

Use value objects when possible.
Though [it] came from many motivations, ... one was to find a more flexible version of assignment, and then to try to eliminate it altogether.


Favor immutability.


Use value objects when possible.

So one definition of functional programming is

- no (few) assignment statements
- no (few) mutable objects.
So one definition of functional programming is

- no (few) assignment statements
- no (few) mutable objects.

One Definition of Functional Programming

some method
So one definition of functional programming is

no (few) assignment statements
no (few) mutable objects.
So one definition of functional programming is

no (few) assignment statements
no (few) mutable objects.
Another Definition

- Some method
- Another method
- Your program
Another Definition

some method

implicit communication

another method

your program
Another Definition

- Some method
- Another method

Your program

Implicit communication
Another Definition

some method

explicit composition

another method

your program
What does this mean concretely?
According to either definition, you can program functionally in any programming language.
According to either definition, you can program functionally in any programming language.

A functional language ensures that you don’t accidentally cheat.
traffic light simulator

initial states

intermediate states

final states
initial: setToRed
onTick:  setTime
onClick: nextColor
stopWhen: atMidnight, renderWarning
toDraw:  renderTrafficLight

type State = Color x Time
State current = ...

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
imperative OOPL

initial: setToRed
onTick:  setTime
onClick:  nextColor
stopWhen: atMidnight, renderWarning
toDraw:  renderTrafficLight

type State = Color x Time
State current = ...

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
initial: setToRed
onTick:  setTime
onClick:  nextColor
stopWhen: atMidnight, renderWarning
toDraw:  renderTrafficLight

type State =
  Initial U Intermediate U Final

Initial  setToRed()
State    nextColor(State current)
Image    renderLight(State current)
State    setTime(State current)
boolean  atMidnight(State current) : Final
Image    renderWarning(Final current)
initial: setToRed
onTick: setTime
onClick: nextColor
stopWhen: atMidnight, renderWarning
toDraw: renderTrafficLight

type State =
  Initial U Intermediate U Final

Initial setToRed()
State nextColor(State current)
Image renderLight(State current)
State setTime(State current)
boolean atMidnight(State current) : Final
Image renderWarning(Final current)

explicit state transformations allow local reasoning
```javascript
initial: setToRed
onTick:  setTime
onClick: nextColor
stopWhen: atMidnight,renderWarning
toDraw:   renderTrafficLight

type State =
    Initial U Intermediate U Final

Initial setToRed()
State nextColor(State current)
Image renderLight(State current)
State setTime(State current)
boolean atMidnight(State current) : Final
Image renderWarning(Final current)
```

**functional**

**explicit state transformations allow local reasoning**
Imperative Programming

setToRed();
renderLight();
nextColor();
nextColor();
setTime();
renderLight();
nextColor();

if atMidnight()
  renderWarning();
else
  renderLight();
Imperative Programming

```java
setToRed();
renderLight();
nextColor();
nextColor();
setTime();
renderLight();
nextColor();
if atMidnight()
    renderWarning()
else
    renderLight();
```

Functional Programming

```java
State s1 = setToRed()
Image i1 = renderLight(s1)
State s2 = nextColor(s1)
State s3 = nextColor(s2)
State s4 = setTime(s3)
Image i4 = renderLight(s4)
State s5 = nextColor(s4)
Image i5 =
    atMidnight(s5) ?
        renderWarning(s5),
    renderLight(s5)
```
It all looks easy.
It all looks easy.

So what’s the catch?
Imagine a state that uses a record, which contains vector in each slot, and each record contains maps that map names to lists of immutable data. And imagine that you want to equip the monster with a dagger.

```plaintext
type State =
    { monsters : Vector<Monster>,
      fighter  : Status,
      turns    : Natural }

type Monster =
    Map<String,List<Weapon>>

type Weapon = ...
```
Imagine a state that uses a record, which contains vector in each slot, and each record contains maps that map names to lists of immutable data. And imagine that you want to equip the monster with a dagger.

type State =
{ monsters : Vector<Monster>,
  fighter  : Status,  
n  turns    : Natural } 

type Monster =
  Map<String,List<Weapon>>

type Weapon  = ...
Imagine a state that uses a record, which contains vector in each slot, and each record contains maps that map names to lists of immutable data. And imagine that you want to equip the monster with a dagger.

```haskell
type State =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }

type Monster =
  Map<String, List<Weapon>>

type Weapon = ...
```
Imagine a state that uses a record, which contains vector in each slot, and each record contains maps that map names to lists of immutable data. And imagine that you want to equip the monster with a dagger.

```haskell
type State =
  { monsters : Vector<Monster>,
    fighter : Status,
    turns : Natural }

type Monster =
  Map<String,List<Weapon>>

type Weapon = ...
```
Imagine a state that uses a record, which contains vector in each slot, and each record contains maps that map names to lists of immutable data. And imagine that you want to equip the monster with a dagger.

```
type State =
  { monsters : Vector<Monster>,
    fighter : Status,
    turns : Natural }

type Monster =
  Map<String, List<Weapon>>

type Weapon = ...
```
type State =
{
  monsters : Vector<Monster>,
  fighter : Status,
  turns : Natural
}
type Monster =
  Map<String, List<Weapon>>
type Weapon = ...

Imperative Programming

state.monsters[i][“orc”].addList(“dagger”);
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList("dagger");
zip(c,new_list);

type State =
  { monsters : Vector<Monster>,
    fighter : Status,
    turns : Natural }

type Monster =
  Map<String,List<Weapon>>

type Weapon = ...
Context $x$ List$<\text{Weapon}>$ $<c, w> = \text{unzip}(\text{state})$;
List$<\text{Weapon}>$ new$\_list = \text{addList}($"dagger"$)$;
$\text{zip}(c, \text{new}_list)$;
State

Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList("dagger");
zip(c,new_list);
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList("dagger");
zip(c, new_list);
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList("dagger");
zip(c,new_list);
a problem of expressiveness ("notational" overhead)

a problem of algorithmics ("slow" performance)
a problem of expressiveness ("notational" overhead)

solution 1: zip/unzip & functional data structures

da problem of algorithmics ("slow" performance)
a problem of expressiveness ("notational" overhead)

solution 1: zip/unzip & functional data structures

solution 2: **monads** and other fancy constructs

a problem of algorithmics ("slow" performance)
a problem of expressiveness ("notational" overhead)

solution 1: zip/unzip & functional data structures

solution 2: monads and other fancy constructs

solution 3: "bite the bullet" -- allow mutation in FP and FPLs

a problem of algorithmics ("slow" performance)
a problem of expressiveness

solution 1: *functional data structures* do not truly eliminate notational overhead
solution 1: functional data structures do not truly eliminate notational overhead

solution 2: monads gets close. The remaining type overhead is arguably an advantage. It helps tame side effects.

a problem of expressiveness
solution 1: functional data structures do not truly eliminate notational overhead

solution 2: monads gets close. The remaining type overhead is arguably an advantage. It helps tame side effects.

solution 3: mutation in FP and FPLs eliminates the problem as much as desired. Danger: it opens the flood gate for careless programmers.

a problem of expressiveness
a problem of algorithmics: theory

functional data structures: we have no proof that functional data structures are as efficient as imperative programming.
a problem of algorithmics: theory

functional data structures: we have no proof that functional data structures are as efficient as imperative programming.

monads: they are implemented imperatively. Period.
functional data structures: we have no proof that functional data structures are as efficient as imperative programming.

monads: they are implemented imperatively. Period.

assignments in FPLs: they eliminates the problem as much as desired. Danger: it tempts programmers to use mutation too much.
mix and match: people tend to combine monads or mutation with functional data structures.
mix and match: people tend to combine monads or mutation with functional data structures.

measuring end-to-end performance: efficiency is in practice indistinguishable from imperative programming.

da problem of algorithmics: in practice
mix and match: people tend to combine monads or mutation with functional data structures.

measuring end-to-end performance: efficiency is in practice indistinguishable from imperative programming.

catch: it takes experience to reach this point.
About Myself

I am not neutral.

I am not a purist.
About Myself

I am not neutral.

I am not a purist.

**research:** objects, assignment statements, design patterns, web servlets, continuations, modules, functional I/O, etc.

**programming:** mostly functional, but OO and imperative as needed
About Myself

I am not neutral.

I am not a purist.

**research:** objects, assignment statements, design patterns, web servlets, continuations, modules, functional I/O, etc.

**programming:** mostly functional, but OO and imperative as neede

**teaching:** start with functional programming in purely functional, strict languages for 10,000s of students, starting in 7th grade all the way to M.S.
Why Functional Programming?

Why a Functional Programming Language?
This was a project a friend took on. James P. Clarke, a
very talented artist and glass blower, was asked to design
a program that would be used by the police department. The
program was designed to help the officers with their daily
tasks. He came up with a design that was both functional
and aesthetically pleasing. The program was built using
Microsoft Access and Visual Basic.

In addition to the program, James also created a
series of posters that were used to promote the city's
culture. He used a variety of techniques to create these
posters, including screen printing and hand-painting.

The posters were displayed in local businesses and
community centers, where they were well-received.

Overall, James' work on this project was a success.
He was able to combine his artistic talents with his
knowledge of technology to create something that was
both useful and beautiful.
This was a project a friend took on. James P. Clarke, a fine art artist and glass blower with several diverse talents devised a great concept although he needed a bit of help. His goal was to create a parchment style real world and place his contact information realistically on the parchment scroll. His entire project was entirely devised in CoreDRAW because that application allowed placement of artwork anywhere and in whatever manner wanted. Additionally it allows bulk text to be placed in any chosen container and that is the goal of this project. In fact he’s even considered using an actual parchment made to order for this project. In any event he had done some work on this and was considering doing a much better job overall. He needed assistance in finishing the Jon Clarke’s original. Further it is noted that the parchment was laid over a really great artistic background. That or it could be the background art partially shown when you set the finished project. With the word object on a new layer in Photoshop, we’ll select a foreground palette color to apply old parchment. 255 R, 245 G, 160 B was the base color we chose although further manipulations will alter that color. After selecting the word object, we filled the text layer with the base foreground color. We also set the background layer to the grayscale version with the opacity set to 25%. Our text shadows were added as a new Adjustment layer. Judge for yourself, right, if it is all we were after. Create a parchment of your own dummy parchment. Create a parchment of your own dummy parchment.
This was a project a friend took on. James P. Clarke, a fine arts artist and glass blower with several different talents devised a great concept although he needed a bit of help. His goal was to create a parchment-type artifact and place the contact information realistically on the parchment itself. His entire project was entirely done in CorelDRAW because that application allowed placement of artistic windows and in whatever manner wanted. Additionally it allows the text to be placed in any chosen container and also can be placed in any portion as you like.

In the end, it worked out much better than expected. Once finished, the whole piece was placed in an envelope to be sent to the customer. It was a successful project, a testament to the perseverance of those who create something truly unique. The customer was very pleased with the outcome, as were the creators of the project.
This was a project friend took on. James P. Clarke, a fast data entry and class learner, wanted a series of image editing talents devised a great concept although he needed a bit of help. He went on to create a purchase a type of world and placed it in a shiny container that in the past of 10 years ago, it would have been impossible to imagine. This is a great basic for students to explore and learn about the various aspects of design. It is encouraging to see that the world of design is changing and evolving. In the past, design was more focused on the aesthetics and appearance of objects. However, in today's world, design is becoming more focused on functionality and user experience. This is evident in the design of the container that was created.
create

program

run

test

maintain
Programs must be written for people to read, and only incidentally for machines to execute. *from: Abelson & Sussman, SICP*
The cost of software is a function of the cost of programmer communication.
The cost of software is a function of the cost of programmer communication.

Functional programming and better functional programming languages greatly reduce the cost of communication and thus the cost of software.
There are many sides to the cost story: human, training, technical.
1995: DrScheme

15 yrs of FP in high schools
1995: DrScheme

15 yrs of FP in high schools

2005: Bootstrap for grades 6-8
1995: DrScheme

15 yrs of FP in high schools
Distributed Games, Chat Rooms

2005: Bootstrap for grades 6-8
Animations

Interactive Games
If these students can do FP, it is easy.
1995: DrScheme

15 yrs of FP in high schools

Interactive Games

Distributed Games, Chat Rooms

2005: Bootstrap for grades 6-8

Animations

If these students can do FP, it is easy

Why did these students improve so much in math? Why do they pass the state test?
Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math
Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

*only* 1/3 of the students get co-op positions that involve programming
Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001: conventional first-year introduction to OO (Java) programming and discrete math
only 1/3 of the students get co-op positions that involve programming

2006: functional-then-OO first-year introduction to programming and discrete math
Northeastern University:
5 year programs, including three 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

only 1/3 of the students get co-op positions that involve programming

2006:
functional-then-OO first-year introduction to programming and discrete math

over 2/3 of the students get co-op positions that involve programming
Northeastern University:
5 year programs, including three 6-month supervised co-op positions in industry

Graduate dean: “Industry asks, why can’t your MS students program as well as your undergraduates?”

- Only 1/3 of the students get co-op positions that involve programming
- Over 2/3 of the students get co-op positions that involve programming

conventional first-year introduction to OO (Java) and discrete math

functional-then-OO first-year introduction to programming and discrete math
Northeastern University:
2 year MS programs (one co-op)
now comes with a 4-month introduction
to Functional Program Design
called “Bootcamp”
Teaching FP has a highly beneficial effect on programmers even if they don’t end up programming that way.
Teaching FP has a highly beneficial effect on programmers even if they don’t end up programming that way.

Time to look at some technical points.
From mathematical models to programs

$$f(x,y,z) = \ldots \int f(g(x), h(y), i(z,x)) \, dx \ldots$$
From mathematical models to programs

\[ f(x, y, z) = \]
\[ \ldots \int f(g(x), h(y), i(z, x)) \, dx \ldots \]

Program

\[ f(x, y, z) = \]
\[ \ldots \text{integrate}(f(g(x), h(y), i(z, x))) \ldots \]
From mathematical models to programs

Mathematics

\[ f(x,y,z) = ... \int f(g(x), h(y), i(z,x)) \, dx ... \]

Program

\[ f(x,y,z) = ... \integrate(f(g(x), h(y), i(z,x))) ... \]

Yes, they basically look the same and it is easy to convince yourself that they mean the same.
From algebraic types to functions

```
type Contract =
    zero
|  scale of Contract * Double
|  and of Contract * Contract
|  until of Contract * Observation
|  ...
```
From algebraic types to functions

```
type Contract =
    zero
  | scale of Contract * Double
  | and of Contract * Contract
  | until of Contract * Observation
  | ...
```

fun Number value(Contract c, Model m) =
case c
    case zero             -> ...
  | scale(base,fac)      -> . value(base,m) .
  | and(c1,c2)          -> . value(c1,m) ...
    . value(c2,m) ...
  | until(base,obs)      -> . value(base,m) ...
type Contract =
    zero
  | scale of Contract * Double
  | and of Contract * Contract
  | until of Contract * Observation
  | ...

fun Number value(Contract c, Model m) =
case c
    zero            -> ...
  | scale(base,fac) -> . value(base,m) .
  | and(c1,c2)      -> . value(c1,m) ...
    . value(c2,m) ...
  | until(base,obs) -> . value(base,m) ...
  | ...

Imagine all the OO design patterns you need in Java.

algebraic types translate directly into a function outline
type State = Color \times Time

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
From function signatures to understanding

Imperative

type State = Color x Time

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }

Functional

type State = Initial U Intermediate U Final

Initial setToRed()
State nextColor(State current)
Image renderLight(State current)
State setTime(State current)
boolean atMidnight(State current) : Final
Image renderWarning(Final current)
From function signatures to understanding

**Imperative**

```java
type State = Color \times Time

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

**Functional**

```java
type State = Initial U Intermediate U Final

Initial setToRed
State nextColor(State current)
Image renderLight(State current)
State setTime(State current)
boolean atMidnight(State current) : Final
Image renderWarning(Final current)
```

*Type signatures convey a lot of information.*
From function signatures to understanding

**Type signatures convey a lot of information.**

**VOID conveys nothing.**
type State = Color x Time

void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }

Functional

Type signatures convey a lot of information.

Imperative

VOID conveys nothing.

Subtract $10 for every VOID return type in your programmers code.
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime();
}

Imperative
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime();
}

Functional

test case {
    compare(setTime(someState), expectedState);
}

Imperative
From functions to testing

**Imperative**

test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime()
}

**Functional**

test case {
    compare(setTime(someState), expectedState);
}

state is transferred explicitly
and can be understood in isolation
Tests in the functional world become "one liners". And that works for compositions, too.

state is transferred explicitly and can be understood in isolation
Function Composition in Action

```
search_good_solution(
    criteria,
    generate_all_solutions(model, state0));
```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));

Yes, on demand. Lazy data structures enable a powerful, yet simple compositional style

```java
search Winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));

evaluate some nodes, not all
Imperative OOP can express this idea, but only in extremely ugly ways, too ugly for this slide.
Function composition is pervasive, even in the strict world.
Financial Contracts as Functional Composition

Combinator DSL

type Contract ... Observation ... Currency

fun Contract zero() ...  
fun Contract one(Currency c) ...  
fun Contract when(Obs t, Contract c) ...  
fun Contract scale(Double s, Contract c) ...  
fun Observation at(Date d) : Obs ...
Financial Contracts as Functional Compostion

Combinator DSL

type Contract ... Observation ... Currency

fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d) : Obs ...

fun zero_coupon_discount_bond(t, x, k) =
    when (at t) (scale (konst x) (one k))
Financial Contracts as Functional Composition

**Combinator DSL**

```text
type Contract ... Observation ... Currency

fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d): Obs ...
```

**One Contract**

```text
fun zero_coupon_discount_bond(t, x, k) =
  when (at t) (scale (konst x) (one k))
```
Financial Contracts as Functional Composition

**Combinator DSL**

```haskell
-- type Contract ... Observation ... Currency

fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c) ...
fun Observation at(Date d) : Obs ...

fun zero_coupon_discount_bond(t, x, k) =
  when (at t) (scale (konst x) (one k))
```

Simon Peyton Jones

One Contract
Simple functions represent basic ideas.
Simple functions represent basic ideas.

Combinator functions combine ideas.
Simple functions represent basic ideas.

Combinator functions combine ideas.

With function composition programmers create and communicate programs in combinator DSLs.
Functional programming languages in the LISP tradition use a “template” approach to DSLs in addition to combinators (Scheme, Clojure, Racket, Template Haskell).
The last part of the functional story: parallelism.

Compilers think, too.
Remember the Definition

some expression

another expression

your program

implicit communication
some expression

another expression

aFun

your program

implicit communication
your program

aFun(()

some expression, another expression

run in parallel
In the purely functional world, the compiler does not need proof of non-interference. It is built into the programming language.
Implicit parallelism is free in functional programming languages.
Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.
Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.

In the imperative world mutation creates too few opportunities for automatic parallel execution.

In the functional world a lack of dependencies means too many opportunities for automatic parallel execution.
Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.

In the imperative world mutation creates too few opportunities for automatic parallel execution.

In the functional world a lack of dependencies means too many opportunities for automatic parallel execution.

The imperative world will see explicit parallel programming and the big battle against race condition bugs.
Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.

In the imperative world mutation creates too few opportunities for automatic parallel execution.

In the functional world a lack of dependencies means too many opportunities for automatic parallel execution.

The imperative world will see explicit parallel programming and the big battle against race condition bugs.

The functional world will provide explicit parallel programming with fewer race conditions.
25 years of research on parallelism for FORTRAN calls for mostly functional intermediate compiler representations (PDGs, SSAs).

Explicit parallelism is easy in functional programming languages.
25 years of research on parallelism for FORTRAN calls for mostly functional intermediate compiler representations (PDGs, SSAs).

Explicit parallelism is easy in functional programming languages.

Functional programming languages make the dependencies explicit and thus facilitate the compiler’s reasoning task.
So what is my favorite functional language?
What is my favorite functional programming language?
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The Lazy Racket language
- streams
- lazy trees
The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The *Lazy* Racket language
- streams
- lazy trees

The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming
The _Typed_ Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The _Lazy_ Racket language
- streams
- lazy trees

The _FrTime_ language
- functional reactive programming

The _Racket_ language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The _Web_ language

The _Typed_ Racket language

The _Scribble_ language

The _Slideshow_ language
The **Racket language**
- pattern matching et al.
- classes
- cross-platform GUIs
- rich web programming

The **Typed Racket language**
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The **Lazy Racket language**
- streams
- lazy trees

The **FrTime language**
- functional reactive programming

The **Web language**

The **Scribble language**

The **Slideshow language**

The **Foundation (10 core constructs)**
The *Racket* language
- pattern matching et al.
- classes
- cross-platform GUIs
- rich web programming

The *Web* language
- streams
- lazy trees

The *FrTime* language
- functional reactive programming

The *Lazy* Racket language
- streams
- lazy trees

The *Scribble* language

The *Slideshow* language

The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The Foundation (10 core constructs)

powerful DSL Framework
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The Typed Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The Lazy Racket language
- streams
- lazy trees

The FrTime language
- functional reactive programming

The Web language

The Lazy Racket language
- streams
- lazy trees

The Scribble language

The Slideshow language

powerful DSL Framework

Matthew Flatt, UUtah

The Foundation (10 core constructs)
Summary
Functional programming is about clear, concise communication between programmers.

Functional programming languages keep you honest about being functional.

A good transition needs training, but training pays off.
Though Smalltalk came from many motivations, ... one was to find a more flexible version of assignment, and then to try to eliminate it altogether.

Alan Kay,
*History of Smalltalk* (1993)

Favor immutability.
Joshua Bloch,
*Effective Java* (2001)

Use value objects when possible.
Kent Beck,
*Test Driven Development* (2001)