Why Functional Programming Matters

John Hughes
Mary Sheeran
...and Mary Sheeran (in absentia)
Functional Programming à la 1940s

• Minimalist: who needs booleans?
• A boolean just *makes a choice!*

\[
\begin{align*}
\text{true} \quad x \quad y & = x \\
\text{false} \quad x \quad y & = y
\end{align*}
\]

• We can *define* if-then-else!

\[
\text{ifte bool t e = bool t e}
\]
Who needs integers?

• A (positive) integer just \textit{counts loop iterations}!

\begin{align*}
two \ & f \ x = f \ (f \ x) \\
one \ & f \ x = f \ x \\
zero \ & f \ x = x
\end{align*}

• To recover a "normal" integer...

\texttt{\textasteriskcentered*Church> two \ (+1) \ 0} \\
2
Look, Ma, we can add!

- Addition by *sequencing* loops

\[
\text{add } m \ n \ f \ x = m \ f \ (n \ f \ x)
\]

- Multiplication by *nesting* loops!

\[
\text{mul } m \ n \ f \ x = m \ (n \ f) \ x
\]

*Church*> add one (mul two two) (+1) 0
5
Factorial à la 1940

\[
\text{fact } n = \\
\quad \text{ifte } (\text{iszero } n) \\
\quad \quad \text{one} \\
\quad \quad (\text{mul } n \ (\text{fact } (\text{decr } n)))
\]

*Church> fact (add one (mul two two)) (+1) 0
120
A couple more auxiliaries

• Testing for zero

\[
iszero \ n = \ n \ (\_ \rightarrow \ false) \ true
\]

• Decrementing...

\[
decr \ n = \ n \ (\m \ f \ x \rightarrow \ f \ (\m \ incr \ zero))
zero
(\x \rightarrow x)
zero
\]
Booleans, integers, (and other data structures) can be entirely replaced by functions!

"Church encodings"

Early versions of the Glasgow Haskell compiler actually implemented data-structures this way!

Alonzo Church
Before you try this at home...

Church.hs:27:35:
Occurs check: cannot construct the infinite type:
  t ~ t -> t -> t

Expected type:

```
(((t -> t -> t) -> t -> t) -> t -> t)
  -> (t -> t -> t) -> t -> t -> t
  -> (((t -> t -> t) -> t -> t) -> t -> t -> t)
  -> (((t -> t -> t) -> t -> t) -> t -> t -> t)
  -> (t -> t -> t)
  -> t
  -> t
  -> t
  -> t

```

Actual type:

```
(((t -> t -> t) -> t -> t) -> t -> t)
  -> (t -> t -> t) -> t -> t -> t
  -> (((t -> t -> t) -> t -> t) -> t -> t -> t)
  -> (((t -> t -> t) -> t -> t) -> t -> t -> t)
  -> (t -> t -> t)
  -> t
  -> t
  -> t

```

```
But wait, there’s more...

Relevant bindings include

(let n :: (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t) -> t) -> t) -> t
  in n)

But wait, there’s more...

(functions that comonadize, namely "mul", and "lift")

In the first argument of "mul", namely "n"
In the third argument of "lift", namely "mul n (fact (dec n))"
The type-checker needs a *little bit* of help

\[
\text{fact} :: \\
(\forall a. (a \rightarrow a) \rightarrow a \rightarrow a) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]
Factorial à la 1960

(LABEL FACT (LAMBDA (N)
    (COND ((ZEROP N) 1)
        (T (TIMES N (FACT (SUB1 N))))))))

Higher-order functions!

(MAPLIST FACT (QUOTE (1 2 3 4 5)))

(1 2 6 24 120)
Numbers implemented by functions (~1940)

LISP implemented by functions (1960)

Any programming language implemented by functions—
Christopher Strachey (~1964)
The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

“... today ... 1,700 special programming languages used to ‘communicate’ in over 700 application areas.”—Computer Software Issues, an American Mathematical Association Prospectus, July 1965.

Factorial in ISWIM

\[ \text{fac}(5) \]

where \( \text{rec} \ \text{fac}(n) = \)

\[ (n=1) \rightarrow 1; \]

\[ n \times \text{fac}(n-1) \]
Laws

\[(\text{MAPLIST } F \ (\text{REVERSE } L)) \equiv (\text{REVERSE } (\text{MAPLIST } F \ L))\]

What’s the point of two different ways to do the same thing?

Wouldn’t two facilities be better than one?

Expressive power should be by design, rather than by accident!
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Turing award 1977

Paper 1978
Conventional programming languages are growing ever more enormous, but not stronger.
Inherent defects at the most basic level cause them to be both *fat* and *weak*:
their inability to effectively use powerful combining forms for building new programs from existing ones
apply to all

\[ \alpha \mathcal{F} \]
construction

[f₁, f₂, f₃, f₄]
their lack of useful mathematical properties for reasoning about programs
\[ [f_1, f_2, \ldots, f_n] \cdot g \]
\[ [f_1, f_2, \ldots, f_n] \cdot g \]

\[ [f_1 \cdot g, f_2 \cdot g, \ldots, f_n \cdot g] \]
\[ \alpha \ f \bullet [g_1, g_2, \ldots, g_n] \]
\[ [f \cdot g_1, f \cdot g_2, \ldots, f \cdot g_n] \]
Laws

\[[f_1, f_2, \ldots, f_n] \cdot g = [f_1 \cdot g, f_2 \cdot g, \ldots, f_n \cdot g]\]

\[\alpha f \cdot [g_1, g_2, \ldots, g_n] = [f \cdot g_1, f \cdot g_2, \ldots, f \cdot g_n]\]
c := 0;

for i := 1 step 1 until n do
    c := c + a[i] \times b[i]
Def ScalarProduct =
  (Insert +) • (ApplyToAll ×) • Transpose
Def SP = (/ +) • (α x) • Trans
over \textit{(fish, rot (rot (fish)))}
\[ t = \text{over} \ (\text{fish}, \ \text{over} \ (\text{fish2}, \ \text{fish3})) \]

\[ \text{fish2} = \text{flip} \ (\text{rot45} \ \text{fish}) \]

\[ \text{fish3} = \text{rot} \ (\text{rot} \ (\text{rot} \ (\text{fish2}))) \]
\[ u = \text{over} \ (\text{over} \ (\text{fish2}, \ \text{rot} \ (\text{fish2})), \ \\
\text{over} \ (\text{rot} \ (\text{rot} \ (\text{fish2})), \ \\
\text{rot} \ (\text{rot} \ (\text{rot} \ (\text{fish2}))))) \]
quartet

cycle
v = cycle (rot(t))  quartet (v,v,v,v,v)
quartet(nil, nil, rot(t), t)

quartet(side1, side1, rot(t), t)

side1
quartet (nil, nil, nil, u)
corner1

quartet (corner1, side1, rot(side1), u)
squarelimit = nonet(
    corner, side, \[ rot(rot(rot(corner))) \],
    rot(side), u, \[ rot(rot(rot(side))) \],
    rot(corner), rot(rot(side)), \[ rot(rot(rot(corner))) \])
picture = function
picture = function
over \((p, q)\) \((a, b, c)\) = 
\[ p(a, b, c) \cup q(a, b, c) \]
over \((p,q)\) \((a,b,c)\) = 
\[ p(a,b,c) \cup q(a,b,c) \]

beside \((p,q)\) \((a,b,c)\) = 
\[ p(a,b/2,c) \cup q(a+b/2,b/2,c) \]
over \((p,q)\) \((a,b,c)\) = 
\[ p(a,b,c) \cup q(a,b,c) \]

beside \((p,q)\) \((a,b,c)\) = 
\[ p(a,b/2,c) \cup q(a+b/2,b/2,c) \]
\[
\text{rot}(p) (a,b,c) = p(a+b,c,-b)
\]
Laws

\[ \text{rot(above}(p,q)\text{))} = \text{beside(rot}(p),\text{rot}(q)) \]

It seems there is a positive correlation between the simplicity of the rules and the quality of the algebra as a description tool.
Whole values

Combining forms

Algebra as litmus test
Whole values

Combining forms

Algebra as litmus test
Haskell vs. Ada vs. C++ vs. Awk vs. ...
An Experiment in Software Prototyping Productivity*

Paul Hudak
Mark P. Jones
Yale University
Department of Computer Science
New Haven, CT 06518
{hudak-paul, jones-mark}@cs.yale.edu

July 4, 1994
Time 40.0:
commercial aircraft: (100.0, 43.0)
  -- In engageability zone
  -- In tight zone
hostile craft: (210.0, 136.0)
  -- In carrier slave doctrine
Functions as Data

> type Region = Point -> Bool

> circle :: Radius -> Region
> outside :: Region -> Region
> (\_) :: Region -> Region -> Region

> annulus :: Radius -> Radius -> Region
> annulus r1 r2 = outside (circle r1) \ circle r2
Including 29 lines of inferable type signatures/synonyms

A student, given 8 days to learn Haskell, w/o knowledge of Yale group

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of code</th>
<th>Lines of documentation</th>
<th>Development time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Haskell</td>
<td>85</td>
<td>465</td>
<td>10</td>
</tr>
<tr>
<td>(2) Ada</td>
<td>767</td>
<td>70</td>
<td>23</td>
</tr>
<tr>
<td>(3) Ada9X</td>
<td>800</td>
<td>50</td>
<td>28</td>
</tr>
<tr>
<td>(4) C++</td>
<td>1105</td>
<td>130</td>
<td>-</td>
</tr>
<tr>
<td>(5) Awk/Nawk</td>
<td>250</td>
<td>150</td>
<td>-</td>
</tr>
<tr>
<td>(6) Rapide</td>
<td>157</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>(7) Griffin</td>
<td>251</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>(8) Proteus</td>
<td>293</td>
<td>79</td>
<td>26</td>
</tr>
<tr>
<td>(9) Relational Lisp</td>
<td>274</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>(10) Haskell</td>
<td>156</td>
<td>112</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 3: Summary of Prototype Software Development Metrics
Reaction...

“too cute for its own good”

...higher-order functions just a trick, probably not useful in other contexts
Lazy Evaluation (1976)

Henderson and Morris
A lazy evaluator

Friedman and Wise
CONS should not evaluate its arguments
"The Whole Value” can be $\infty$!

- The *infinite list* of natural numbers $[0, 1, 2, 3 \ldots]$
- All the iterations of a function $\text{iterate } f \ x = [x, f \ x, f (f \ x), \ldots]$
- A consumer for numerical methods $\text{limit } \epsilon \ ps \ xs = \text{<first element of } xs \text{ within } \epsilon \ ps \text{ of its predecessor>}$
Some numerical algorithms

- Newton-Raphson square root
  \[
  \sqrt{a} = \lim_{\epsilon \to 0} \text{iterate next } 1.0
  \]
  where next \( x \) = \( (x + a/x) / 2 \)

- Derivatives
  \[
  \text{deriv } f \ x = \lim_{\epsilon \to 0} \text{map slope (iterate (1/2) 1.0)}
  \]
  where slope \( h \) = \( (f \ (x+h) - f \ x) / h \)

 Same convergence check  Different approximation sequences
Speeding up convergence

The smaller $h$ is, the better the approximation

$A + B \times h^n$

The right answer

An error term
Eliminating the error term

- Given:
  
  \[ A + B \cdot h^n \]
  
  \[ A + B \cdot (h/2)^n \]

  Two successive approximations

- Solve for A and B!

  improve \(n \) xs \( converges \) faster than \( xs \)
Really fast derivative

deriv f x =
  limit eps
  (improve 2
    (improve 1
      (map slope (iterate (/2) 1.0))))

The approximations

The improvements

The convergence check

Everything is programmed separately and easy to understand—thanks to "whole value programming"
Why
Functional Programming Matters

John Hughes
The University, Glasgow

1990
Lazy producer-consumer

- Consumer demands values from producer.
- Producer provides values to consumer.
- Convergence test demands numerical approximations.
- Numerical approximations provide values to Convergence test.
Lazy producer-consumer

consumer

producer

demands

Search strategy

Search space

values

demands

values
Why
Functional Programming
Matters

John Hughes
The University, Glasgow
The Design of a Pretty-printing Library

John Hughes

Chalmers Tekniska Högskola, Göteborg, Sweden.

Selection criterion for the best layout

Ways to lay out a document

demands

values

1995
if \ x < 0 \ then \ -x \ else \ x \\

VS.

if \ n == 0 \ then \ 1 \ else \ n * fac (n-1)
if n == 0
    then 1
else n * fac (n-1)

a $$ b $$ c
if n == 0

then 1

delse n * fac (n-1)

da <> b <> c
if \ n == 0
then 1

\text{a} \ <\nest\ k\ \ b\ =\ \text{nest}\ k\ (a\ <\ b)
(a<>b)<>c = a<> (b<>c)

(a$$b) $$c = a$$ (b$$c)

(a$$b)<>c = a$$ (b<>c)

(a<>b) $$c ≠ a<> (b$$$$c)
A set of possible layouts

\[ \text{sep } [a,b,c] = \]
\[ a \leftrightarrow b \leftrightarrow c \]
\[ U \]
\[ a \quad \$\$ \quad b \quad \$\$ \quad c \]
e.g. Pretty-printing trees

```haskell
data Tree a =
    Leaf a | Branch (Tree a) (Tree a)

pretty (Leaf a) =
    text "(Leaf "++show a++")"
pretty (Branch l r) =
    sep [text "(Branch",
        nest 2 (pretty l),
        nest 2 (pretty r)<>text "")"
```

The implementation uses the laws to transform into the best layout
The “best” layout

maximum page width

maximum text width
Improved on by...

erlang /lib/syntax_tools/src/pretypr.erl
prop_reverse() -> ?FORALL(Xs, list(int()), reverse(reverse(Xs)) == Xs).

3> eqc:quickcheck(qc:prop_reverse()).
.......................................................
...............................................
OK, passed 100 tests
true
prop_wrong() ->
    ?FORALL(Xs, list(int()), reverse(Xs) == Xs).

4> eqc:quickcheck(qc:prop_wrong()).
Failed! After 1 tests.
[-36,-29,20,31,-47,-63,80,-7,93,-87,-29,33,64,58]
Shrinking xx.x.x..xx(4 times)

[0,1]
false

minimal counterexample
QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

QuickCheck search strategy

Space of all possible tests

random

systematic
muFP—Circuits as values

• Backus FP + one-clock-cycle delays
  • (+ feedback)

• Many of the same combining forms, same laws

• Good for reasoning about ”retiming”
Users!

Plessey
Users!

Plessey
Plessey designers write

Using muFP, the array processing element was described in just one line of code and the complete array required four lines of muFP description. muFP enabled the effects of adding or moving data latches within the array to be assessed quickly.


work with Plessey done by G. Jones and W. Luk
Lava

muFP + Functional Geometry

Capture *semantics* of a circuit + relative *placement*

Programmer control of geometry!

Satnam Singh at Xilinx generated FPGA layouts in VHDL
Four adder trees from Lava
Without Layout Information
Lava was implemented as a Haskell library
Satnam gave Xilinx customers Haskell binaries

Here’s a layout generator for your problem
(Don’t ask what’s inside)
Intel

4195835.0 - 3145727.0 \times \left(\frac{4195835.0}{3145727.0}\right) = 0
Intel

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0

Flawed Pentium

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 256
$475 million
Intel

Forte System  1000s users

fl — lazy functional language

built-in decision procedures
HW symbolic simulator

Thanks to Carl Seger (Intel)
Design language
High-level specification language
Scripting language
Implementation language for formal verification tools and theorem provers
Object language for theorem proving

Thanks to Carl Seger (Intel)
Bluespec—FP for hardware

Architecture spec: pure functional Haskell-like programming language

Behaviour spec: guarded atomic transition rules —lets the compiler find parallelism

Generates Verilog for further synthesis

“Abandon sequential von Neumann legacy”
Bluespec

Often BEATS hand-coded RTL code

Designers can use *better algorithms*

Refinement, evolution, major architectural change EASY

Types, Functional Programming and Atomic Transactions in Hardware Design
Rishiyur Nikhil  LNCS 8000
Bluecheck

• QuickCheck in Bluespec!

• Generates and shrinks tests on the chip!

_a Generic Synthesisable Test Bench (Naylor and Moore, Memocode 2015)_
QuickCheck on a chip
two \( f \ x = f \ (f \ x) \)

one \( f \ x = f \ x \)

zero \( f \ x = x \)
Whole values

Combining forms

Simple laws

Functions as representations