Why Functional Programming Matters

John Hughes & Mary Sheeran
Functional Programming à la 1940s

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

true \ x \ y = x
false \ x \ y = y

• We can define if-then-else!

ifte bool t e =
    bool t e
Who needs integers?

• A (positive) integer just counts loop iterations!

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

• To recover a ”normal” integer...

\[
*\text{Church}> \quad \text{two} \ (+1) \ 0
\]

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

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

• Decrementing...

\[ \text{decr } n = \]
\[ n (\_ \rightarrow f \rightarrow f (\_ \rightarrow x) x) \]
\[ \text{zero} \]
\[ \text{zero} \]
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:
\( t \sim t \rightarrow t \rightarrow t \)

Expected type:

\[
(((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \\
\rightarrow (t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \rightarrow (t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (t \rightarrow t \rightarrow t) \\
\rightarrow t \\
\rightarrow t \\
\rightarrow t)
\]

Actual type:

\[
(((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \\
\rightarrow (t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \rightarrow (t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (((t \rightarrow t \rightarrow t) \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \rightarrow t) \\
\rightarrow (t \rightarrow t \rightarrow t) \\
\rightarrow t \\
\rightarrow t \\
\rightarrow t)
\]
Relevant bindings include

n :: (((((t -> t -> t) -> t -> t) -> t -> t) -> t -> t) -> t) -> t

But wait, there’s more...

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

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

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

(maplist fact (quote (1 2 3 4 5)))
(1 2 6 24 120)

Higher-order functions!
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 fac}(n) = \begin{cases} 1; & (n=1) \end{cases} \]

\[ n \times \text{fac}(n-1) \]
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
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:
Word-at-a-time
their inability to effectively use powerful combining forms for building new programs from existing ones
apply to all

αf
construction

\[ \{f_1, f_2, f_3, f_4\} \]
their lack of useful mathematical properties for reasoning about programs
\([f_1, f_2, \ldots, f_n] \cdot g\)
\([f_1, f_2, \ldots, f_n] \cdot g\)

\([f_1 \cdot g, f_2 \cdot g, \ldots, f_n \cdot g]\)
c := 0;
for i := 1 step 1 until n do
  c := c + a[i] × b[i]
Def IP = (/ +) • (α x) • Trans
Peter Henderson, Functional Geometry, 1982
fish
over (fish, rot (rot (fish)))
\[ t = \text{over} \ (\text{fish}, \ \text{over} \ (\text{fish2}, \ \text{fish3})) \]

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

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

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

side1
quartet (nil, nil, nil, nil, u)  quartet (corner1, sidel, rot(sidel), u)
squarelimit = nonet(
    corner,       side,       rot(rot(rot(rot(corner)))),
    rot(side),    u,           rot(rot(rot(rot(side)))),
    rot(corner),  rot(rot(rot(side))), rot(rot(rot(rot(corner))))
)
picture = function
picture = function
over \((p, q)\) \((a, b, c)\) =
\[p(a,b,c) \cup q(a,b,c)\]
\[ \text{beside } (p,q) \ (a,b,c) = p(a, b/2, c) \cup q(a+b/2, b/2, c) \]
\[ \text{rot}(p) \ (a, b, c) = p(a+b, c, -b) \]
Laws

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

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

Functions as representations
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
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>750</td>
<td>23</td>
</tr>
<tr>
<td>(3) Ada9X</td>
<td>800</td>
<td>800</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 ∞!

• The infinite list of natural numbers
  [0, 1, 2, 3 …]

• All the iterations of a function
  iterate \( f \ x = [x, f \ x, f \ (f \ x), \ldots] \)

• A consumer for numerical methods
  \[
  \text{limit } \epsilon \ \text{xs} = \text{<first element of xs within } \epsilon \text{ of its predecessor>}
  \]
Some numerical algorithms

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

- Derivatives
  \[
  \text{deriv } f \ x = \lim_{\varepsilon \to 0} \text{map } \varepsilon \text{ slope } \frac{f (x + h) - f \ x}{h}
  \]
  where
  \[
  \text{slope } h = \frac{f (x + h) - f \ x}{h}
  \]

Same convergence check  Different approximation sequences
Speeding up convergence

Differentiation Integration

The smaller \( h \) is, the better the approximation

\[ A + B^* h^n \]

The right answer

An error term
Eliminating the error term

• Given:

\[ A + B^* h^n \]
\[ A + B^* (h/2)^n \]

• Solve for A and B!

Two successive approximations

improve \( n \) \( \times s \) converges faster than \( \times s \)
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
Lazy producer-consumer

consumer \quad demands \quad values \quad demands \quad values

producer

Convergence test

Numerical approximations
Lazy producer-consumer
demands
consumer
values
producer
demands
Search
strategy
values
Search
space
Why
Functional Programming
Matters

John Hughes
The University, Glasgow

\[ \alpha \beta \]

demands

values
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
Improved on by...
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 + unit delays

• Inherits many combining forms and laws

• Good for reasoning about alternative designs
Example law: “retiming”
“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

→ FPGA layouts on Xilinx chips

Satnam Singh
Xilinx

Semantics
Placement
Four adder trees—no placement
Four adder trees—Lava
Intel

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0
Intel

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0

Flawed Pentium

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 256
$475 million
Lazy functional language, 1000s users

• Design
• High-level specification
• Scripting
• Implementation of formal verification tools and theorem provers
• Object language for theorem proving

Thanks to Carl Seger (formerly of Intel)
Selection criterion for lowest power

Ways to lay out a parallel prefix circuit

demands

values
Solid Formal Link with Good Return of the Investment

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

Haskell-like language (architecture) + atomic transition rules (H/W modelling)

Frees designers to explore better algorithms, making 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 FPGA!

A Generic Synthesisable Test Bench (Naylor and Moore, Memocode 2015)
two  \( f \, x = f \, (f \, x) \)

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

zero  \( f \, x = x \)
Whole values
Simple laws
Combining forms
Functions as representations