Fritz Henglein
Inaugural lecture*
2011-02-24
Lundbeck Auditorium, Biocenter, U. Copenhagen

*as professor (without special obligations)
appView (slides, KU-design)

Fritz Henglein
Inaugural lecture*
2011-02-24
Lundbeck Auditorium, Biocenter, U. Copenhagen

*as professor (without special obligations)
Fritz Henglein
Inaugural lecture*
2011-02-24
Lundbeck Auditorium, Biocenter, U. Copenhagen

*as professor (without special obligations)
What is an inaugural lecture?
What is an inaugural lecture?

- Google search: “What is an inaugural lecture”?
- From answers.com:
What is an inaugural lecture?

- Google search: “What is an inaugural lecture”?
- From answers.com:

  **A:**

  An inaugural lecture is what a speech the president says when he/she is coming into office. It is also known as an inaugural address.
Fritz Henglein, DIKU
henglein@diku.dk
Why Functional Programming matters more now than ever

Fritz Henglein, DIKU
henglein@diku.dk
Abstract, Take 1

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 primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Communications of the ACM
August 1978
Volume 21
Number 8
Abstract, Take 2

Why Functional Programming Matters

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

This paper dates from 1984, and circulated as a Chalmers memo for many years. Slightly revised versions appeared in 1989 and 1990 as [Hug90] and [Hug89]. This version is based on the original Chalmers memo nroff source, lightly edited for LaTeX and to bring it closer to the published versions, and with one or two errors corrected. Please excuse the slightly old-fashioned type-setting, and the fact that the examples are not in Haskell!

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). Since modularity is the key to successful programming, functional languages are vitally important to the real world.
Good cards: Turing Awards related to FP

- John McCarthy, 1971, for LISP
- John Backus, 1977, for FORTRAN (!) and FP
- Ken Iverson, 1979, for APL
- Robin Milner, 1991, for ML type inference
But: Time for getting cold feet?

Functional Programming

Why no one uses functional languages

Editor: Philip Wadler, Bell Laboratories, Lucent Technologies; wadler@research.bell-labs.com

Philip Wadler

Advocates of functional languages claim they produce an order of magnitude improvement in productivity. Experiments don’t always verify that figure — sometimes they show an improvement of only a factor of four. Still, code that’s four times as short, four times as quick to write, or four times easier to maintain is not to be sniffed at. So why aren’t functional languages more widely used?
What is FP?
What is FP?

• Computation as transformation (mathematical function) from input data to output data

• Express what, not how

• Define, apply and compose functions
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
  for (j = 0; j < names.size(); j++)
    if accounts.get(i).id() == names.get(j).id() {
      result.add (new
        Pair(accounts.get(i), names.get(j)))
    }
```
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
    for (j = 0; j < names.size(); j++)
        if accounts.get(i).id() == names.get(j).id() {
            result.add (new Pair(accounts.get(i), names.get(j)))
        }
```
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
    for (j = 0; j < names.size(); j++)
        if accounts.get(i).id() == names.get(j).id() {
            result.add (new Pair(accounts.get(i), names.get(j)))
        }
```

Functionally (Haskell):

```
```
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
    for (j = 0; j < names.size(); j++)
        if accounts.get(i).id() == names.get(j).id() {
            result.add (new Pair(accounts.get(i), names.get(j)))
        }
```

Functionally (Haskell):

```haskell
accounts `X` names
`select`($id, $id) `in` eqInt
```
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
    for (j = 0; j < names.size(); j++)
        if accounts.get(i).id() == names.get(j).id() {
            result.add (new Pair(accounts.get(i), names.get(j)))
        }
```

Functionally (Haskell):

```haskell```
accounts `X` names
```
`select`($id, $id) `in` eqInt
```
Example

Imperatively (Java):

```java
for (i = 0; i < accounts.size(); i++)
    for (j = 0; j < names.size(); j++)
        if accounts.get(i).id() == names.get(j).
            result.add (new
                Pair(accounts.get(i), names.get(j))
        }
```

Functionally (Haskell):

```haskell
accounts `X` names
`select`($id, $id) `in` eqInt
```
FP language characteristics

- **Algebra and logic**: Amenable to proving properties ("programs as formulas")
  
  e.g., \texttt{let} \(x = \text{exp1} \) \texttt{in} \(\text{exp2} = \text{exp2}[/\text{exp1}/x]\)

- **Types**: Powerful type systems and type inference for safety, security, and efficient implementation

- **Bulk data**: (lists, sets, maps, matrices) oriented programming, instead of one-element-at-a-time processing

- **General purpose**: Turing-complete, any application
FP -- mature theory and technology

- Well-understood foundations: Lambda Calculus and Type Theory (1930s)
- Mature implementations of FPL families: LISP (since late 1950s), APL (1960s), ML (1970s), Miranda/Haskell (1980s), Erlang (1990s)
Pioneering FP research

• Ground-breaking programming language technology (PLT), later transferred to imperative programming languages

• Garbage collection (“managed” memory)

• Parametric polymorphism (“generics”)

• Language-integrated querying

• Safely embedded domain-specific languages (DSLs)
FP(L) applications

- Finance
- Military
- Telecom
- Formalization of mathematics
- High-performance distributed computing
  - Algorithmic skeletons (MapReduce etc.)
- ...

Declarative DSLs

- Declarative DSLs = special FPL, restricted to certain domain of application

- Benefits:
  - Problem-oriented specification: What, not how
  - Domain-oriented concepts (no “bits”, “registers”, etc.)
  - Optimization by exploiting “theory”, code engineering, and dynamic run-time context
  - Safety and security by limitation of expressive power
Declarative DSLs: Examples

- **Regular expressions** (regex): Matching strings against a pattern to check format and extract substrings.

- **Database queries** (SQL): Computing information by selecting, projecting, joining, and aggregating collections ("tables").

- **Spreadsheet expressions** (Excel): Functional dependencies of cells on other cells.

- **“Little” languages** for rendering, software configuration, workflow, data mining, cryptosystems...
von Neumann model

- Separate processing and memory modules
- Programming model:
  - Repeat sequentially:
    - Move small bits of data from Memory to CPU
    - Process data in CPU sequentially
    - Move data back to Memory
Modern computer

Highly parallel: Computation occurs “everywhere”

Grand challenge: How to program it?
(What is a good programming model?)
Parallel computer models

- Parallel RAM (PRAM), 80s: Fine-grained synchronous cores with global shared memory
- Bulk Synchronous Parallel (BSP) model [Valiant 90]
- SISD, SIMD, MISD, MIMD [Flynn 72]: Bulk data or bulk processing models
FP for parallelism

• Deterministic (data) parallelism: Model execution as transitions on (transformations of) parallel state

• Excellent fit with FP:
  • bulk data processing: parallel state = bulk data
  • no/little shared state (cause of race conditions, complex concurrency control, stalls, etc.)
FP: Chall’s & opportunities

• Challenge: FPLs historically built to execute well on von Neumann architectures
  • Like simulating a parallel architecture on a von Neumann architecture
  • Excellent correctness, robustness, reliability, and productivity, but Fortran and C++ typically faster

• Opportunity: Build to execute well on parallel architecture, exploit natural fit of FP with parallel state processing
IPL approaches to parallelism

• **Automatic parallelization** of sequential code

• **Hybrid approach**: Retrofit/develop sequential code with dedicated parallel libraries (Linpack, PLinq, etc.)
  
  • e.g., Python as glue language for HPC libraries

  • Develop natively parallel programming model: FP, ...

• Needed: Application domain with problem solutions “naturally” formulated functionally, and with simultaneous high-performance, transparency (“light-weight” correctness) and productivity requirements.
Hybrid approaches

- Retrofit IPL with dedicated parallel libraries
- SQL, Linpack, PLinq, etc.
- Host language: Script/glue languages (Python) or general-purpose language (C++, Java, C#, etc.)
- Gives performance plus backwards compatibility
- But what about reasoning? “Break” between embedded libraries and host language
- Security, safety, performance concerns
Who am I?

Tenured
Hall of Famer
RBI (Research Buck$ in)
At Bats (Papers Written)
H-Index
Wings (Ph.D. Students Graduated)
Losers (Ph.D. Students Dropped Out)
Home Runs (Paper in Science or Nature)

Academic Stats

<table>
<thead>
<tr>
<th>RBI</th>
<th>20M</th>
<th>2B</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>300</td>
<td>3B</td>
<td>62</td>
</tr>
<tr>
<td>H</td>
<td>19</td>
<td>SP</td>
<td>13</td>
</tr>
<tr>
<td>W</td>
<td>53</td>
<td>IP</td>
<td>221</td>
</tr>
<tr>
<td>L</td>
<td>53</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>HR</td>
<td>3</td>
<td>NG</td>
<td>2</td>
</tr>
</tbody>
</table>


Doubles (Two Papers on the Same Topic)
Triples (Three Papers Using the Same Data Set)
Stolen Post-docs
Innings Pitched (Invited Lectures)
Career Awards

# of Times Featured in National Geographic

www.phdcomics.com
Early work

- Efficient type inference (incl. lower bounds)
- Polymorphic recursion, System F
- Type-based program analysis:
  - Binding-time inference, dynamic typing
- Applications:
  - Hunting Y2K problems in COBOL code
Recent work I

• Foundations:
  • Generic sorting, partitioning, and querying
  • Regular expressions

• Theme: Abstraction and performance and programming
Recent work II

• Applications
Regular expressions

(From xkcd.com)
Sorting and partitioning

• Can you sort and partition in linear time, without breaking abstraction?
  • Yes, by generic discrimination
    [Henglein, ICFP 2008, PEPM 2010]
• How do I know that a function ``sorts''?
• If it is an intrinsically parametric permutation function [Henglein, JLAP 2009]
Querying

- Can you perform generic SQL-like querying and get both good performance and abstraction without indexes and complex query optimization?
  - Yes, combine generic discrimination with lazy data structures [Henglein/Larsen, WGP 2010]
  - Saves space, shaves off a linear factor from naive, easily programmed
Recent work II

• Applications:
  • TrustCare.eu: Trustworthy processes in eHealth
  • 3gERP: Next-generation Enterprise Resource Planning systems

• Theme:
  • Declarative DSLs for processes (contracts, workflow), information ("business intelligence"), legal/business rules

• Purpose: Eliminate "programming" (low-level coding)
Next stop: HIPERFIT

• Strategic Research Center for Functional High-Performance Computing for Financial Information Technology (HIPERFIT)

• See hiperfit.dk

• Aspects:
  • high performance
  • transparency (light-weight “correctness”)
  • “declarative” high-level models of problems
FP and DSLs in Finance

- Financial contract specification (Danske Bank, Lexifi, Nordea, Simcorp, Standard Chartered, Credit Suisse, ...) Infinitely many contracts expressible

- Execution, pricing, risk... (your analysis here) ... implemented for all of them once and for all.

- High Frequency Trading (Jane Street Capital)
Not so bold predictions

- All computing is parallel in the future: Parallelism is the default, sequentiality the exception. (Reason: physics)
- Serious systems require built-in light-weight verification: type systems
- Some optimizing needs to be reclassified as pessimizing; e.g. aggressive memory reuse
- The end of (global) heaps and (global) garbage collection
Why FP now?

• Noncommitment to von Neumann model, natural fit with (data) parallel computing models: Bug turned into a feature

• Support for reasoning for serious systems: Both light-weight (type inference), algebraic (transformations, calculi) and heavy-weight (dependent types, program logic etc.)

• Maturity: Past both the hype and the disappointment stages.

• Simplicity: Orthogonality, small trusted computing base
FP(L): What is missing?

- Finer-grained memory model (unboxed data, no random access, no global heap)
- Robust parallel cost model
- No built-in laziness (!)
- Better reasoning support (e.g. inductive types)
- Starting with high performance rather than expressiveness:
  - Tools and technologies for GPGPUs, etc
Threat 1

- The sequential mind set:
  - for (i = 0; i <= l.size; i++) {
    ... i++ ...
  }

Threat 2

If we build our software with no bugs, we can make a 10% return on our investment.

But if we do a poor job, we can make a 40% return by selling upgrades and service.

But don’t worry. We only have the budget for a poor job.

I can’t remember if we’re cheap or smart.

PHEW!
Threat 3

• “Does it fit *directly* into our (existing, complex) system?”
Threat 3

• “Does it fit directly into our (existing, complex) system?”

“Simplicity is not an add-on feature”, attributed to Ken Olsen, founder of Digital Equipment Corporation
Questions?