SOME HISTORY OF
FUNCTIONAL
PROGRAMMING LANGUAGES

David Turner
University of Kent &
Middlesex University

TFP12
St Andrews University
12 June 2012
MILESTONES

\( \lambda \) calculus (Church 1936, 1941)

LISP (McCarthy 1960)

ISWIM (Landin 1966)

PAL (Evans 1968)

SASL (1973...)

Edinburgh - NPL, early ML, HOPE

Miranda

Haskell
The $\lambda$-calculus (Church 1936, 1941) is a typeless theory of pure functions with three rules

$[\alpha] \quad \lambda x.e \Leftrightarrow \lambda y.[y/x]e$

$[\beta] \quad (\lambda x.b) a \Rightarrow [a/x]b$

$[\eta] \quad (\lambda x.e \; x) \Rightarrow e \quad \text{if } x \text{ not free in } e$

There are functional representations of natural numbers and other data.

1) Church-Rosser theorem

\[ A \Rightarrow B, A \Rightarrow B' \; \Rightarrow \; B \Rightarrow C, B' \Rightarrow C \]

implies normal forms unique (upto $\alpha$-conversion)

2) second Church-Rosser theorem: repeatedly reducing leftmost redex is normalising

3) Böhm’s theorem:

if $A, B$ have distinct $\beta, \eta$-normal forms there is a context $C[]$ with $C[A] \Rightarrow K$, $C[B] \Rightarrow KI$

Implies $\alpha, \beta, \eta$-conversion is the strongest possible equational theory on normalising terms
Lazy Evaluation

2nd Church-Rosser theorem: to find normal form we must in general substitute actual parameters into function bodies *unreduced* (lazy evaluation).

Call-by-value is an *incorrect reduction strategy* for λ-calculus, but efficient because the actual parameter is reduced only once! Used from 1960.

Thesis of Wadsworth (1971) showed that the efficiency disadvantage of normal order reduction can be overcome by *graph reduction* on λ-terms.

Turner (1979) compiled a λ-calculus based language, SASL, to S,K combinators (Curry 1958) and did *graph reduction on combinators*.

Johnsson (1985) extracts program specific combinators from source (λ-lifting) to compile code for graph reduction on stock hardware.

Further developed by Simon Peyton Jones (1992) to Spineless Tagless G-machine which underlies the Glasgow Haskell compiler, GHC.
LISP

McCarthy 1960 (developed from 1958)

Computation over symbolic data: formed from atoms by pairing; S-expressions can represent lists, trees and graphs.

S-expressions are of variable size and can outlive the procedure that creates them - requires a heap and a garbage collector.

The M-language manipulates S-expressions: has cons, car, cdr, tests, conditional expressions and recursion. This is computationally complete. McCarthy showed an arbitrary flowchart can be coded as mutually recursive functions.

M-language is first order, cannot pass a function as argument or return as result. McCarthy’s model was Kleene’s theory of recursive functions.

M-language programs are coded as S-expressions and interpreted by eval. Allows meta-programming, by uses of eval and quote.
Some myths about LISP

“Pure LISP” never existed - LISP had assignment and goto before it had conditional expressions and recursion. LISP programmers made frequent use of replacar and replacdr.

LISP was not based on the $\lambda$ calculus, despite using the word “lambda” to denote functions. Based on first order recursion equations.

The M-language was first order, but you could pass a function as a parameter by quotation, i.e. as the S-expression for its code. But this gives the wrong binding rules for free variables (dynamic instead of lexicographic).

If a function has a free variable, e.g. $y$ in

$$f = \lambda x . x + y$$

$y$ should be bound to the value in scope for $y$ where $f$ is defined, not where $f$ is called.

Not until SCHEME (Sussman 1975) did versions of LISP with static binding appear. Today all versions of LISP are $\lambda$-calculus based.
Static binding and the invention of closures

Algol 60 allowed textually nested procedures and passing procedures as parameters (but not returning procedures as results). Algol 60 Report required static binding of free variables.

Randell and Russell (1964) implemented this by two sets of links between stack frames. The dynamic chain linked each stack frame, representing a function call, to the frame that called it. The static chain linked each stack frame to that of the textually containing function call, which might be much further down the stack. Free variables are accessed via the static chain.

If functions can be returned as results, a free variable might be held onto after the function call in which it was created has returned, and will no longer be present on the stack.

Landin (1964) solved this in his SECD machine. A function is represented by a closure, consisting of code for the function plus an environment for its free variables. Closures live in the heap.
ISWIM

In early 60's Peter Landin wrote a series of seminal papers on the relationship between programming languages and $\lambda$ calculus.

"The next 700 programming languages"(1966) describes an idealised language family (can choose constants and basic operators). Ideas:

“Church without lambda”

**let, rec, and, where**

so we can say e.g

```
expr where f x = stuff
```

instead of $(\lambda x \cdot stuff) expr$

Offside rule for block structure

assignment; and a generalisation of jumps, the J operator - allowed a program to capture its own continuation (see also Landin 1965).

ISWIM = sugared $\lambda$ + assignment + control

also first appearance of algebraic type defs

At end of paper: Strachey discussion of DL
ISWIM inspired PAL (Evans 1968) and GEDANKEN (Reynolds 1970)

**PAL** (MIT 1968)

applicative PAL = sugared λ (let, rec, where) and conditional expressions, allowed one level of pattern matching, e.g.

```
let x, y, z = expr
```

imperative PAL adds mutable variables & assignment; and first class labels

data types: integer & floating point numbers, truth values, strings, tuples, functions, labels

“typeless”, i.e. runtime type checking

first class labels allowed unusual control structures - coroutines, backtracking

coroutine example - *equal fringe* problem
backtracking example - parsing
SASL - St Andrews Static Language

I left Oxford in 1972 for a lectureship at St Andrews and gave a course on programming language theory in the Autumn term.

During that course I invented a simple DL based on the applicative subset of PAL. Tony Davie implemented it in LISP then I implemented it BCPL by an SECD machine (Easter 1973).

Two changes from applicative PAL
- multi-level pattern matching
- string as list of char
SASL was and remained purely applicative

call by value, runtime typing, let and rec (no λ)
curried functions with left associative appln

data types: int, truthvalue, char, list, function
all data types had same rights

Used for teaching functional programming, instead of LISP.
Advantages of SASL over LISP for teaching fp

1) pure sugaring of $\lambda$ calculus, with no imperative features and no eval/quote nonsense

2) has correct scope rules for free variables (static binding)

3) multi-level pattern matching makes for huge improvement in readability

LISP

\[
\text{cons(cons(car(car(cdr(x))),cons(car(cdr(car(cdr(x))))),nil)),cons(cons(car(car(x)),cons(car(cdr(car(x))),nil)),nil))}
\]

becomes

\[
\text{let } ((a,b),(c,d)) = x \text{ in } ((c,d),(a,b))
\]

in 1973 SASL was probably unique in these properties
Why runtime typing?

LISP and other languages for computation over symbolic data worked on lists, trees and graphs.

This leads to a need for structural polymorphism - a function which reverses a list, or traverses a tree, doesn’t need to know the type of the elements.

Before Milner (1978) the only way to handle this was to delay type checking until run time.

SASL example

let $f$ be a curried function of some number of Boolean arguments., we want to test if it is a tautology.

\[
\text{taut } f = \text{logical } f \rightarrow f;
\]

\[
\text{taut } (f \text{ True}) \& \text{taut } (f \text{ False})
\]

runtime typing still has followers - Erlang, LISP
evolution of SASL 1973-83

dropped rec allowing recursion as default, switched from let to where

in 1976 SASL became lazy and added multi-equation pattern matching for case analysis

\[ A 0 \ n = n+1 \]
\[ A \ m \ 0 = A \ (m-1) \ 1 \]
\[ A \ m \ n = A \ (m-1) \ (A \ m \ (n-1)) \]

I got this idea from John Darlington

implemented at St Andrews in 1976 by lazy version of SECD machine (Burge 1975)

there was also an implementation by Bill Campbell

at Kent in 1977 reimplemented by translation to SK combinators and combinator graph reduction

added floats and list comprehensions
Why laziness?

for consistency with Church 1941 - second Church Rosser theorem

better for equational reasoning

allows interactive I/O via lazy lists and programs using $\infty$ data structures

renders exotic control structures unnecessary

- lazy lists replace coroutines (equal fringe problem)

- list of successes method replaces backtracking

the list of successes method is in my 1976 SASL manual, but didn’t have a name until Wadler 1985
SASL sites, circa 1986

California Institute of Technology, Pasadena
City University, London
Clemson University, South Carolina
Iowa State U. of Science and Technology
St Andrews University
Texas A & M University
Universite de Montreal
University College London
University of Adelaide
University of British Columbia
University of Colorado at Denver
University of Edinburgh
University of Essex
University of Groningen, Netherlands
University of Kent
University of Nijmegen, Netherlands
University of Oregon, Eugene
University of Puerto Rico
University of Texas at Austin
University of Ulster, Coleraine
University of Warwick
University of Western Ontario
University of Wisconsin-Milwaukee
University of Wollongong

MCC Corporation, Austin Texas
Systems Development Corporation, Pennsylvania
Burroughs Corporation

(24 educational + 3 commercial)
meanwhile in Edinburgh

Burstall (1969) extends ISWIM with algebraic type defs and \textit{case}

\footnotesize
\begin{verbatim}
type tree
  niltree : tree
  node : atom \times tree \times tree \rightarrow tree
\end{verbatim}
\normalsize

\textit{case} \(\text{pat}_1 : \text{exp}_1 \ldots \text{pat}_n : \text{exp}_n\)

Darlington’s \textbf{NPL} (1973-5) introduced multi-equation function defs over algebraic types

\footnotesize
\begin{verbatim}
\begin{align*}
\text{fib} \ (0) & \leftarrow 1 \\
\text{fib} \ (1) & \leftarrow 1 \\
\text{fib} \ (n+2) & \leftarrow \text{fib} \ (n+1) + \text{fib} \ (n)
\end{align*}
\end{verbatim}
\normalsize

\textbf{NPL} also had “set expressions”

\footnotesize
\begin{verbatim}
\begin{align*}
\text{setofeven} \ (X) & \leftarrow \langle x : x \in X \& \text{even}(x) \rangle
\end{align*}
\end{verbatim}
\normalsize

\textbf{NPL} was used for Darlington’s work on program transformation (Burstall & Darlington 1977) first order, strongly typed, purely functional, call-by-value
NPL evolved into **HOPE** (1980) higher order, strongly typed with explicit types, polymorphic type variables, purely functional - kept multi-equation p/m but dropped set expressions

also in Edinburgh (1973-78) **ML** developed as meta-language of Edinburgh LCF (Gordon et al 1979) this had

\[ \lambda \text { let letrec} + \text { references} \]

types built using \(+\ x\) and type recursion

type abstraction

call-by-value, no pattern matching, structures analysed by conditionals and e.g. *isl, isr*

polymorphic strong typing with **type inference**

* * * * * * * * * ... as type variables

**Standard ML** (1990) is the confluence of the HOPE and ML streams, but not pure - has references and exceptions
Miranda

Developed in 1983-86 Miranda is essentially SASL plus algebraic types and polymorphic type system of Milner (1978)

An important change - switched from conditional expressions to conditional equations, with guards, example

\[ \text{sign } x = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \end{cases} \]

Combining pattern matching with guards gives significant gain in expressiveness

Guards first appeared in KRC (Turner 1982) a miniaturised version of SASL I designed in 1980 for teaching - very simple, it had only top level equations (no \textit{where}) and built in line editor

Putting guards into a language with \textit{where} raised a puzzle about scope rules - the \textit{where} clause has to govern a whole rhs rather than one expression
Another necessary change was the introduction of a lexical distinction between variables and constructors, in order to be able to distinguish pattern matching from function definition

\[ \text{Node} \ x \ y = \text{stuff} \]

is pattern match, binds \( x, y \) to parts of \( \text{stuff} \) but

\[ \text{node} \ x \ y = \text{stuff} \]

defines a function, \( \text{node} \), of two arguments

Miranda is lazy, purely functional, has list comprehensions, polymorphic with type inference and optional type specifications - see Turner (1986) for fuller description - papers and downloads at www.miranda.org.uk
Haskell

Similar in many ways to Miranda, the main innovations are

Switched guards to left hand side of equations

\[
\text{sign } x \begin{cases} 
  x > 0 & = 1 \\
  x < 0 & = -1 \\
  x == 0 & = 0
\end{cases}
\]

Extended Miranda’s var/constructor distinction to types, allowing lower case tvars, upper case tcons

\[
\text{map} :: \ (\star \rightarrow \star \star) \rightarrow \left[ \star \right] \rightarrow \left[ \star \star \right] \quad \text{Miranda}
\]
\[
\text{map} :: \ (a \rightarrow b) \rightarrow \left[ a \right] \rightarrow \left[ b \right] \quad \text{Haskell}
\]

Almost everything in Miranda is also present in Haskell but Haskell adds

\textit{type classes}, \textit{monadic I/O}, \textit{a module system with two level names}

Haskell has a richer syntax. e.g. it provides conditional expressions and guards, \textit{let} and \textit{where} pattern matching by equations and \textit{case} etc.
REFERENCES (in date order)

A. Church ``The calculi of lambda conversion'', Princeton University Press, 1941.


Philip Wadler \”Replacing a failure by a list of successes”,


