SOME HISTORY OF

FUNCTIONAL

PROGRAMMING LANGUAGES

David Turner
University of Kent &
Middlesex University

CODEMESH 2017
ILEC Conference Centre
London 9.11.17

MILESTONES

λ calculus (Church 1936, 1941)

LISP (McCarthy1960)

ISWIM (Landin 1966)

PAL (Evans 1968)

SASL (1973...)

Edinburgh - NPL, early ML, HOPE

KRC & Miranda

Haskell

The λ-calculus (Church 1936, 1941) is a typeless theory of pure functions with three rules

[α] $\lambda x.e \Leftrightarrow \lambda y.[y/x]e$

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

[η] $(\lambda x.e\ x) \Rightarrow e$ if x not free in e

There are functional representations of natural numbers and other data.

1) Church-Rosser theorem

$A \Rightarrow B, A \Rightarrow B' \supset B \Rightarrow C, B' \Rightarrow C$

implies normal forms unique (upto α-conversion)

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

3) Böhm's theorem:
if A, B have distinct β,η-normal forms there is a context C[] with $C[A] \Rightarrow \lambda xy.x$, $C[B] \Rightarrow \lambda xy.y$

Implies α,β,η-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 λ 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 λ-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 λ 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 *(λ x · stuff) expr*
Offside rule for block structure

Assignment - and a generalisation of jumps, the J operator, which allows a program to capture its own continuation (see also Landin 1965).

ISWIM = sugared λ + 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 λ calculus, with no imperative features and no eval/quote distractions

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

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

LISP
cons(cons(car(car(x)),cons(car(car(cdr(x))),nil)),
cons(cons(car(cdr(car(x))),cons(car(cdr(car(cdr(
x)))),nil)),nil))

becomes

let *((a,b),(c,d)) = x* ***in*** *((a,c),(b,d))*

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 unknown number of Boolean arguments, we want to test if it is a tautology.

$$taut\ f \quad = \quad boolean\ f \rightarrow\ f\ ;\ taut\ (f\ True)\ \&\ taut\ (f\ False)$$

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

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))

I got this idea from John Darlington

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

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 ∞ 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 ***case***

type *tree*
niltree : tree
node : atom x tree x tree → tree

case *exp* ***of*** $pat_1 : exp_1;\ \ldots\ pat_n : exp_n;$

Darlington's **NPL** (1973-5) introduced multi-equation function defs over algebraic types

fib (0) ⇐ *1*
fib (1) ⇐ *1*
fib (n+2) ⇐ *fib (n+1) + fib (n)*

NPL also had "set expressions"

setofeven (X) ⇐ <: n : n in X & even(n) :>

NPL was used for 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

λ let letrec + references

types built using + x and type recursion
also <u>type abstraction</u>

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

KRC

KRC (Turner 1982) was a miniaturised version of SASL developed for teaching in 1979-1980, very simple, it had only top level equations (no ***where***) and a built in line editor

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

*sign x = 1, **if** x>0*
*= -1, **if** x<0*
*= 0, **if x == 0***

Combining pattern matching with guards gives significant gain in expressiveness

KRC also had list comprehension***s –*** with lazy lists these become very powerful

See ***krc-lang.org***
for papers and resurrected software

Miranda

Developed in 1983-86 Miranda is KRC with ***where*** put back in, plus algebraic types and polymorphic type system of Milner (1978)

Combining guards with ***where*** raises a puzzle about scope rules - the *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

Node x y = stuff

is pattern match, binds *x*, *y* to parts of *stuff* but

node x y = stuff

defines a function, *node*, of two arguments

Miranda is lazy, purely functional, has list comprehensions, polymorphic with type inference and optional type specifications - see “An Overview of Miranda” (Turner, 1986) for fuller description

Miranda was first released in 1985 and quite widely adopted (an interpreter, no compiler)

Papers and downloads at ***miranda.org.uk***

Haskell

Similar in many ways to Miranda, the most noticeable changes are

Switched guards to left hand side of equations

```
sign x  | x > 0  = 1
        | x < 0  = -1
        | x==0 = 0
```

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

```
map :: (* → **) → [*] → [**]
```

Miranda, becomes in Haskell

```
map :: (a → b) → [a] → [b]
```

Almost everything in Miranda is also present in Haskell but Haskell adds major new features

type classes, monadic I/O, a module system with two level names

Haskell has a richer syntax. e.g. it provides conditional expressions and guards, ***let*** and ***where*** pattern matching by equations and ***case*** etc.

Tautology in Haskell, first attempt

```
class Taut a where
  taut :: a->Bool

instance Taut Bool where
  taut b = b

instance Taut a => Taut (Bool->a) where    --problem here
  taut f = taut (f True) && taut (f False)
```

Tautology in standard Haskell

```
class Boolean b where
  fromBool :: Bool->b

instance Boolean Bool where
  fromBool t = t

class Taut a where
  taut :: a->Bool

instance Taut Bool where
  taut b = b

instance (Boolean a, Taut b) => Taut (a->b) where
  taut f = taut (f (fromBool True))
         && taut (f (fromBool False))
```

Tautology in SASL

```
taut f    = boolean f →   f ;
              taut (f True) & taut (f False)
```

REFERENCES (in date order)

Alonzo Church, J. B. Rosser ``Some Properties of conversion'', Transactions of the American Mathematical Society 39:472--482 (1936).

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

H. B. Curry, R. Feys ``Combinatory Logic, Vol I'', North-Holland, Amsterdam 1958.

John McCarthy``Recursive Functions of Symbolic Expressions and their Computation by Machine'', CACM 3(4):184--195, 1960.

B. Randell, L. J. Russell ``The Implementation of Algol 60'', Academic Press, 1964.

P. J. Landin ``The Mechanical Evaluation of Expressions'', Computer Journal 6(4):308--320 (1964).

Peter J. Landin ``A generalisation of jumps and labels'', Report, UNIVAC Systems Programming Research, August 1965. Reprinted in Higher Order and Symbolic Computation, 11(2):125--143, 1998.

Peter J. Landin ``The Next 700 Programming Languages'', CACM 9(3):157--165, March 1966.

A. Evans ``PAL - a language designed for teaching programming linguistics'', Proceedings ACM National Conference, 1968.

R. M. Burstall ``Proving properties of programs by structural induction'', Computer Journal 12(1):41-48 (Feb 1969).

John C. Reynolds ``GEDANKEN — a simple typeless language based on the principle of completeness and the reference concept'', CACM 13(5):308-319 (May 1970).

W. Burge ``Recursive Programming Techniques'', Addison Wesley, 1975.

D. A. Turner ``SASL Language Manual'', St. Andrews University, Department of Computational Science Technical Report CS/75/1, January 1975; revised version December 1976.

Gerald J Sussman, Guy L. Steele Jr. ``Scheme: An interpreter for extended lambda calculus'', MEMO 349, MIT AI LAB (1975).

R. M. Burstall, John Darlington ``A Transformation System for Developing Recursive Programs'', JACM 24(1):44--67, January 1977. Revised and extended version of paper originally presented at Conference on Reliable Software, Los Angeles, 1975.

R. Milner ``A Theory of Type Polymorphism in Programming'', Journal of Computer and System Sciences, 17(3):348--375, 1978.

M. J. Gordon, R. Milner, C. P. Wadsworth ``Edinburgh LCF'' Springer-Verlag Lecture Notes in Computer Science vol 78, 1979.

R. M. Burstall, D. B. MacQueen, D. T. Sanella ``HOPE: An experimental applicative language'', Proceedings 1980 LISP conference, Stanford, California, pp 136--143, August 1980.

D. A. Turner ``Recursion Equations as a Programming Language'', in Functional Programming and its Applications, pp 1--28, Cambridge University Press, January 1982 (editors Darlington, Henderson and Turner). Reprinted in LNCS 9600:459--478 Springer2016.

Philip Wadler ``Replacing a failure by a list of successes'', Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 1985 (Springer LNCS 201:113--128).

Thomas Johnsson ``Lambda Lifting: Transforming Programs to Recursive Equations'', Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture Nancy, France, Sept. 1985 (Springer LNCS 201).

D. A. Turner ``An Overview of Miranda'', SIGPLAN Notices, 21(12):158--166, December 1986.

R. Milner, M. Tofte, R. Harper, D. MacQueen ``The Definition of Standard ML'', MIT Press 1990, Revised 1997.

S. L. Peyton~Jones ``Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine'', Journal of Functional Programming, 2(2):127--202, April 1992.

Paul Hudak et al. ``Report on the Programming Language Haskell'', SIGPLAN Notices 27(5) 164 pages (May 1992).

S. L. Peyton~Jones ``Haskell 98 language and libraries: the Revised Report'', JFP 13(1) January 2003.

David Turner ``Some History of Functional Programming Languages'', Proceedings TFP 2012, Springer LNCS 7829:1--20.