Some History of Functional Programming Languages
(Invited Talk)

D.A. Turner

University of Kent & Middlesex University

Abstract. We study a series of milestones leading to the emergence of lazy, higher order, polymorphically typed, purely functional programming languages. An invited lecture given at TFP12, St Andrews University, 12 June 2012.

Introduction

A comprehensive history of functional programming languages covering all the major streams of development would require a much longer treatment than falls within the scope of a talk at TFP, it would probably need to be book length. In what follows I have, firstly, focussed on the developments leading to lazy, higher order, polymorphically typed, purely functional programming languages of which *Haskell* is the best known current example. Secondly, rather than trying to include every important contribution within this stream I focus on a series of snapshots at significant stages.

We will examine a series of milestones:

1. Lambda Calculus (Church & Rosser 1936)
2. LISP (McCarthy 1960)
3. Algol 60 (Naur et al. 1963)
4. ISWIM (Landin 1966)
5. PAL (Evans 1968)
6. SASL (1973–83)
7. Edinburgh (1969–80) — NPL, early ML, HOPE
8. Miranda (1986)
9. Haskell (1992 . . .)

1 The Lambda Calculus

The lambda calculus (Church & Rosser 1936; Church 1941) is a typeless theory of functions. In the brief account here we use lower case letters for variables: $a, b, c \cdots$ and upper case letters for terms: $A, B, C \cdots$.

A *term* of the calculus is a variable, e.g. x, or an application AB, or an abstraction $\lambda x.A$ for some variable x. In the last case $\lambda x.$ is a *binder* and free

H.-W. Loidl and R. Peña (Eds.): TFP 2012, LNCS 7829, pp. 1–20, 2013.

occurrences of x in A become *bound*. A term in which all variables are bound is said to be *closed* otherwise it is *open*. The motivating idea is that closed terms represent functions.

In writing terms we freely use parentheses to remove ambiguity.
The calculus has three rules

$$(\alpha) \quad \lambda x.A \rightarrow_\alpha \lambda y.[y/x]A$$
$$(\beta) \quad (\lambda x.A)B \rightarrow_\beta [B/x]A$$
$$(\eta) \quad \lambda x.Ax \rightarrow_\eta A \quad \text{if } x \text{ not free in } A$$

Here $[B/x]A$ means substitute B for free occurrences of x in A[1]. Rule α permits change of bound variable. Terms which are the same up to α-conversion, e.g. $\lambda x.x$ and $\lambda y.y$, are not usually distinguished.

The smallest reflexive, symmetric, transitive, substitutive relation on terms including $\rightarrow_\alpha$, $\rightarrow_\beta$ and $\rightarrow_\eta$, written $\Leftrightarrow$, is Church's notion of $\lambda-$*conversion*. If we omit symmetry from the definition we get an oriented relation, written $\Rightarrow$, called *reduction*.

An instance of the left hand side of rule β or η is called a *redex*. A term containing no redexes is said to be in *normal form*. A term which is convertible to one in normal form is said to be *normalizing*. There are non-normalizing terms, for example $(\lambda x.xx)(\lambda x.xx)$ which β-reduces to itself.

The three most important technical results are

Church-Rosser Theorem. If $A \Rightarrow B$ and $A \Rightarrow B'$ there is a term C such that $B \Rightarrow C$ and $B' \Rightarrow C$. An immediate corollary is that the normal form of a normalizing term is unique (up to α-conversion).

Second Church-Rosser Theorem. The normal form of a normalizing term can be found by repeatedly reducing its *leftmost outermost redex*, a process called normal order reduction.

Böhm's theorem. If A, B have distinct normal forms there is a context $C[]$ with $C[A] \Rightarrow \lambda x.(\lambda y.x)$ and $C[B] \Rightarrow \lambda x.(\lambda y.y)$.
This tells us that α, β, η-conversion is the *strongest possible* equational theory on normalizing terms — if we add any equation between non-convertible normalizing terms the theory becomes *inconsistent*, that is all terms are now interconvertible, e.g. we will have $x \Leftrightarrow y$.

The lambda calculus originates from an endeavour by Church, Curry and others to define an alternative foundation for mathematics based on functions rather than sets. The attempt foundered in the late 1920's on paradoxes analogous to those which sank Cantor's untyped set theory. What remained after the propositional parts of the theory were removed is a consistent equational theory of functions. Notwithstanding that it was devised before computers in the modern sense existed, the lambda calculus makes a simple, powerful and elegant programming language.

[1] Substitution includes systematic change of bound variables where needed to avoid *variable capture* — for details see any modern textbook, e.g. Hindley & Seldin (2008).

In the pure[2] untyped lambda calculus everything is a function — a closed term in normal form can only be an abstraction, $\lambda x.A$. An *applied lambda calculus* adds constants representing various types of data and primitive functions on them, for example natural numbers with $plus, times$ etc. and appropriate additional reduction rules — Church (1941) calls these δ-rules — this can be done while ensuring that Church-Rosser and other technical properties of the calculus are preserved. A type discipline can be imposed to prevent the formation of meaningless terms. There is thus a richly structured family of applied lambda calculi, typed and untyped, which continues to grow new members.

However, the pure untyped lambda calculus is already computationally complete. There are functional representations of natural numbers, lists, and other data. One of several possibilities for the former are the Church numerals

$$\begin{aligned}
\overline{0} &= \lambda a.\lambda b.b \\
\overline{1} &= \lambda a.\lambda b.ab \\
\overline{2} &= \lambda a.\lambda b.a(ab) \\
\overline{3} &= \lambda a.\lambda b.a(a(ab)) \text{ etc. } \cdots
\end{aligned}$$

Conditional branching can be implemented by taking

$$\begin{aligned}
\overline{True} &\equiv \lambda x.(\lambda y.x) \\
\overline{False} &\equiv \lambda x.(\lambda y.y)
\end{aligned}$$

We then have

$$\begin{aligned}
\overline{True}AB &\Rightarrow A \\
\overline{False}AB &\Rightarrow B
\end{aligned}$$

Recursion can be encoded using $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ which has the property, for any term A

$$YA \Rightarrow A(YA)$$

With this apparatus we can code all the recursive functions of type $N \to N$ (using N for the set of natural numbers) but also those of type $N \to (N \to N)$, $(N \to N) \to N$, $(N \to N) \to (N \to N)$ and so on up.

It is the power to define functions of higher type, together with clean technical properties — Church-Rosser etc. — that make lambda calculus, either pure or applied, a natural choice as a basis for functional programming.

At first sight it seems a restriction that λ creates only functions of one argument, but in the presence of functions of higher type there is no loss of generality. It is a standard result of set theory that for any sets A, B, the function spaces $(A \times B) \to C$ and $A \to (B \to C)$ are isomorphic[3].

[2] By *pure* we here mean that variables are the only atomic symbols.

[3] Replacing the first by the second is called *Currying*, after H.B.Curry.

1.1 Normal Order Graph Reduction

At this point we temporarily break from the sequence of language milestones to trace an important implementation issue through to the present.

An implementation of λ-calculus on a sequential machine should use *normal order reduction*, otherwise it may fail to find the normal form of a normalizing term. Consider one reduction step, in applying rule β

$$(\lambda x.A)B \Rightarrow [B/x]A$$

we substitute B into the function body *unreduced*[4]. In general this will produce multiple copies of B, apparently requiring any redexes it contains to be reduced multiple times. For normal order reduction to be practical it is necessary to have an efficient way of handling this.

An alternative policy is to always reduce arguments before substituting into the function body — this is *applicative order reduction*, also known as parameter passing *by value*. Call-by-value is an unsafe reduction strategy for lambda calculus, at least if the measure of correctness is conformity with Church's theory of conversion, but efficient because the actual parameter is reduced only once.

All practical implementations of functional languages for nearly two decades from LISP in 1958 onwards used call-by-value.

The thesis of Wadsworth (1971, Ch 4) showed that the efficiency disadvantage of normal order reduction can be overcome by *normal graph reduction*. In Wadsworth's scheme the λ-term is a directed acyclic graph, and the result of β-reduction, which is performed by update-in-place of the application node, is that a *single copy* of the argument is retained, with pointers to it from each place in the function body where it is referred to. As a consequence any redexes in the argument are reduced at most once.

Turner (1979a) applied normal graph reduction to S, K combinators (Curry 1958) allowing a much simpler abstract machine. In Turner's scheme the graph may be cyclic, permitting a more compact representation of recursion. The combinator code is compiled from a high level functional language using a variant of Curry's abstraction algorithm (Turner 1979b). Initially this was SASL (Turner 1975) and in later incarnations of the system, Miranda (Turner 1986).

For an interpreter, a fixed set of combinators, S, K, C, B, I etc., each with a simple reduction rule, works well. But for compilation to native code on stock hardware it is better to use λ-abstractions derived from the program source as combinators with potentially bigger reduction steps. Extracting these requires a program transformation, λ-*lifting*, (Hughes 1984; Johnsson 1985). This method was used in a compiler for Lazy ML first implemented at Chalmers University in 1984 by Augustsson & Johnsson (1989). Their model for mapping combinator graph reduction onto von Neumann hardware, the G machine, has been refined by Simon Peyton Jones (1992) to the Spineless Tagless G-machine which underlies the Glasgow Haskell compiler, GHC.

[4] The critical case, which shows why normal order reduction is needed, is when B is non-normalizing but A contains no free occurrences of x.

Thus over an extended period of development normal order reduction has been implemented with increasing efficiency.

2 LISP

The first functional programming language and the second oldest programming language still in use (after FORTRAN), LISP began life in 1958 as a project led by John McCarthy at MIT. The aim was to create a system for programming computations over symbolic data, starting with an algorithm McCarthy had drafted for symbolic differentiation. The first published account of the language and theory of LISP is (McCarthy 1960).

The data on which LISP works is the **S-language**. This has a very simple structure, it consists of atoms, which are words like `X` or `TWO` and a pairing operation, written as a dot. Examples of S-expressions are

```
((X.Y).Z)
(ONE.(TWO.(THREE.NIL)))
```

S-expressions can represent lists, trees and so on — they are of variable size and can outlive the procedure that creates them. A far sighted decision by McCarthy was to refuse to clutter his algorithms with storage claim and release instructions. LISP therefore required, and had to invent, heap storage with a garbage collector.

The **M-language** defines computations on S-expressions. It has

(a) S-expressions
(b) function application, written $f[a; b; \ldots]$ with primitive functions *cons, car, cdr,* for creating and decomposing dotted pairs and *atom, eq,* which test for atoms and equality between atoms
(c) conditional expressions written [$\text{test}_1 \rightarrow \text{result}_1$; $\text{test} \rightarrow \text{result}_2$; ...]
(d) the ability to define recursive functions — example, here is a function to extract the leftmost atom from an S-expression:

```
first[x] = [atom[x] -> x; T -> first[car[x]]]
```

Note the use of atoms `T, F` as truth values. Function definitions introduce variables, e.g. x, which are lower case to distinguish them from atoms. The values to which the variables become bound are S-expressions. Function names are also lower case but don't get confused with variables because in the M-language function names cannot appear as arguments.

This is computationally complete. McCarthy (1960, Sect. 6) showed that an arbitrary flowchart can be coded as mutually recursive functions.

The M-language of McCarthy (1960) is first order, as there is no provision to pass a function as argument or return a function as result[5].

[5] There has been much confusion about this because McCarthy (1960) uses λ-abstraction — but in a completely different context from (Church 1941).

However, McCarthy shows that M-language expressions and functions can be easily encoded as S-expressions and then defines in the M-language functions, *eval* and *apply*, that correctly interpret these S-expressions.

Thus LISP allows meta-programming, that is treating program as data and vice versa, by appropriate uses of *eval* and *quote*. The 1960 paper gives the impression, quite strongly, that McCarthy saw this as removing any limitation stemming from the M-Language itself being first order.

It was originally intended that people would write programs in the M-language, in an Algol-like notation. In practice LISP programmers wrote their code directly in the S-language form, and the M-language became a kind of ghost that hovered in the background — theoretically important but nobody used it.

In LISP 1.5 (McCarthy et al. 1962) atoms acquired *property lists*, which serve several puposes and numbers appeared, as another kind of atom, along with basic arithmetic functions. This was the first version of LISP to gain a large user community outside MIT and remained in use for many years[6].

Many other versions and dialects of LISP were to follow.

Some Myths About LISP

Something called "Pure LISP" never existed — McCarthy (1978) records that LISP had assignment and **goto** before it had conditional expressions and recursion — it started as a version of FORTRAN I to which these latter were added. LISP 1.5 programmers made frequent use of *setq* which updates a variable and *rplaca*, *rplacd* which update the fields of a CONS cell.

LISP was not based on the lambda calculus, despite using the word "LAMBDA" to denote functions. At the time he invented LISP, McCarthy was aware of (Church 1941) but had not studied it. The theoretical model behind LISP was Kleene's theory of first order recursive functions[7].

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

To represent functions in closed form McCarthy uses $\lambda[[x_1;\ldots;x_n];e]$ and for recursive functions he uses ***label***[*identifier;function*].

However, these functional expressions can occur ONLY IN THE FUNCTION POSITION of an application $f[a;b;\ldots]$. This is clear in the formal syntax for the M-language in the LISP manual (McCarthy at al. 1962, p9).

That is, McCarthy's λ and *label* add no new functions to the M-language, which remains first order. They are introduced solely to allow M-functions to be written in closed form.

[6] When I arrived at St Andrews in 1972 the LISP running on the computer laboratory's IBM 360 was LISP 1.5.

[7] McCarthy made these statements, or very similar ones, in a contribution from the floor at the 1982 ACM symposium on LISP and functional programming in Pittsburgh. No written version of this exists, as far as I know.

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.

McCarthy (1978) reports that this problem (wrong binding for free variables) showed up very early in a program of James Slagle. At first McCarthy assumed it was a bug and expected it to be fixed, but it actually springs from something fundamental — that meta-programming is not the same as higher order programming. Various devices were invented to get round this FUNARG problem, as it became known[8].

Not until SCHEME (Sussman 1975) did versions of LISP with default static binding appear. Today all versions of LISP are lambda calculus based.

3 Algol 60

Algol 60 is not normally thought of as a functional language but its rules for procedures (the Algol equivalent of functions) and variable binding were closely related to those of λ-calculus.

The Revised Report on Algol 60 (Naur 1963) is a model of precise technical writing. It defines the effect of a procedure call by a copying rule with a requirement for systematic change of identifiers where needed to avoid variable capture — exactly like β-reduction.

Although formal parameters could be declared **value** the default parameter passing mode was *call by name*, which required the actual parameter to be copied unevaluated into the procedure body at every occurrence of the formal parameter. This amounts to normal order reduction (but not graph reduction, there is no sharing). The use of call by name allowed an ingenious programming technique: Jensen's Device. See `http://en.wikipedia.org/wiki/Jensen's_Device`

Algol 60 allowed textually nested procedures and passing procedures as parameters (but not returning procedures as results). The requirement in the copying rule for systematic change of identifiers has the effect of enforcing static (that is lexicographic) binding of free variables.

In their book "Algol 60 Implementation", Randell and Russell (1964, Sect. 2.2) handle this by two sets of links between stack frames. The dynamic chain links each stack frame, representing a procedure call, to the frame that called it. The static chain links each stack frame to that of the textually containing procedure, which might be much further away. Free variables are accessed **via the static chain**.

[8] When I started using LISP, at St Andrews in 1972–3, my programs failed in unexpected ways, because I expected λ-calculus like behaviour. Then I read the LISP 1.5 manual carefully — the penny dropped when I looked at the syntax of the M-language (McCarthy et al. 1962, p9) and saw it was first order. This was one of the main reasons for SASL coming into existence.

This mechanism works well for Algol 60 but in a language in which 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 the environment for its free variables. The environment is a linked list of name-value pairs. Closures live in the heap.

4 ISWIM

In early 60's Peter Landin wrote a series of seminal papers on the relationship between programming languages and lambda calculus. This includes (Landin 1964), already noted above, which describes a general mechanism for call-by-value implementation of lambda calculus based languages.

In "The next 700 programming languages", Landin (1966) describes an idealised language family, ISWIM, "If you See What I Mean". The sets of constants and primitive functions and operators of the language are left unspecified. By choosing these you get a language specific to some particular domain. But they all share the same design, which is described in layers.

There is an applicative core, which Landin describes as "Church without lambda". He shows that the expressive power of λ-calculus can be captured by using *where. let, rec* and saying $f(x) = \epsilon$ instead of $f = \lambda\ x\ .\ \epsilon$ and so on. Higher order functions are defined and used without difficulty.

In place of Algol's **begin** ... **end** the offside rule is introduced to allow a more mathematical style of block structure by levels of indentation.

The imperative layer adds mutable variables and assignment.
In a related paper, Landin (1965) defines a control mechanism, the J operator, which allows a program to capture its own continuation, permitting a powerful generalization of labels and jumps. In short,

ISWIM = sugared lambda + assignment + control

The ISWIM paper also has the first appearance of algebraic type definitions used to define structures. This is done in words, but the sum-of-products idea is clearly there.

At end of paper there is an interesting discussion, in which Christopher Strachey introduces the idea of a DL, that is a purely declarative or descriptive language and wonders whether it would be possible to program exclusively in one.

5 PAL

ISWIM inspired PAL (Evans 1968) at MIT and GEDANKEN (Reynolds 1970) at Argonne National Laboratory. These were quite similar. I was given the PAL tape from MIT when I started my PhD studies at Oxford in 1969.

The development of PAL had been strongly influenced by Strachey who was visiting MIT when Art Evans was designing it. The language was intended as a vehicle for teaching programming linguistics, its aims were:

(i) completeness — all data to have the same rights,
(ii) to have a precisely defined semantics (denotational).

There were three concentric layers:

R-PAL: this was an applicative language with sugared λ (let, rec, where) and conditional expressions: $test \rightarrow E_1 \ ! \ E_2$.
One level of pattern matching, e.g. *let x, y, z = expr*
L-PAL: this had everything in R-PAL but adds mutable variables & assignment
J-PAL: adds first class labels and **goto**

PAL was call-by-value and typeless, that is, it had run time type checking. The basic data types were: integer & float numbers, truthvalues, strings — with the usual infixes: $+ \ - \ etc.$ and string handling functions: *stem, stern, conc.*

Data structures were built from tuples, e.g. (a, b, c) these were vectors, not linked lists.

Functions & labels had the same rights as basic data types: to be named, passed as parameters, returned as results, included as tuple elements.

First class labels are very powerful — they allow unusual control structures — coroutines, backtracking and were easier to use than Landin's J operator.

6 SASL

SASL stood for "St Andrews Static Language". I left Oxford in October 1972 for a lectureship at St Andrews and gave a course on programming linguistics in the Autumn term. During that course I introduced a simple DL based on the applicative subset of PAL. This was at first intended only as a blackboard notation but my colleague Tony Davie surprised me by implementing it in LISP over the weekend! So then we had to give it a name.

Later in the academic year I was scheduled to teach a functional programming course to second year undergraduates, hitherto given in LISP. In preparation I started learning LISP 1.5 and was struck by its lack of relationship to λ-calculus, unfriendly syntax and the complications of the *eval/quote* machinery. I decided to teach the course using SASL, as it was now called, instead.

The implementation of SASL in LISP wasn't really robust enough to use for teaching. So over the Easter vacation in 1973 I wrote a compiler from SASL to SECD machine code and an interpreter for the latter, all in BCPL. The code of the first version was just over 300 lines — SASL was not a large language. It ran under the RAX timesharing system on the department's IBM 360/44, so the students were able to run SASL programs interactively on monitors, which they liked.

The language had `let ...in ...` and `rec ...in ...` for non-recursive and recursive definitions. Defining and using factorial looked like this:

```
rec fac n = n < 0 -> 1;
            n * fac (n-1)
in fac 10
```

For mutual recursion you could say `rec` *`def1`* `and` *`def2`* `and ... in ....`

The data types were: integers, truthvalues, characters, lists and functions. All data had same rights — a value of any of the five types could be named, passed to a function as argument, returned as result, or made an element of a list. Following LISP, lists were implemented as linked lists. The elements of a list could be of mixed types, allowing the creation of trees of arbitrary shape.

SASL was implemented using call-by-value, with run time type checking. It had two significant innovations compared with applicative PAL:

(i) strings, "...", were not a separate type, but an abbreviation for lists of characters
(ii) I generalised PAL's pattern matching to allow multi-level patterns, e.g. `let (a,(b,c),d) =` *stuff* `in ...`

SASL was and remained purely applicative. The only method of iteration was recursion — the interpreter recognised tail recursion and implemented it efficiently. The compiler did constant folding — any expression that could be evaluated at compile time, say (2 + 3), was replaced by its value. These two simple optimizations were enough to make the SASL system quite usable, at least for teaching.

As a medium for teaching functional programing, SASL worked better than LISP because:

(a) it was a simple sugared λ-calculus with no imperative features and no eval/quote complications
(b) following Church (1941), function application was denoted by juxtaposition and was left associative, making curried functions easy to define and use
(c) it had correct scope rules for free variables (static binding)
(d) multi-level pattern-matching on list structure made for a big gain in readability. For example the LISP expression[9]

```
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 in SASL

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

SASL proved popular with St Andrews students and, after I gave a talk at the summer meeting of IUCC, "Inter-Universities Computing Colloqium", in Swansea in 1975, other universities began to show interest.

[9] The example is slightly unfair in that LISP 1.5 had library functions for frequently occurring compositions of *car* and *cdr*, with names like $caar(x)$ for $car(car(x))$ and $cadr(x)$ for $car(cdr(x))$. With these conventions our example could be written $cons(cons(caar(x), cons(caadr(x), nil)), cons(cons(cadar(x), cons(cadadr(x), nil)), nil))$ However this is still less transparent than the pattern matching version.

6.1 Evolution of SASL 1974–84

The language continued in use at St Andrews for teaching fp and evolved as I experimented with the syntax. Early versions of SASL had an explicit λ, written `lambda`, so you could write e.g. $f = \lambda\ x\ .\ stuff$ as an alternative to $f\ x = stuff$. After a while I dropped λ, allowing only the sugared form. Another simplification was dropping `rec`, making `let` definitions recursive by default. SASL's list notation acquired right associative infixes, *":"* and *"++"*, for *cons* and *append*.

In 1976 SASL syntax underwent two major changes:

(i) a switch from `let` *defs* `in` *exp* to *exp* `where` *defs* with an offside rule to indicate nested scope by indentation.

(ii) allowing multi-equation function definitions, thus extending the use of pattern matching to case analysis. Examples:

```
length () = 0
length (a:x) = 1 + length x

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[10] (see Section 7 below). The second example above is Ackermann's function.

At the same time SASL underwent a major change of semantics, becoming *lazy*. For the implemention at St Andrews in 1976 I used a lazy version of Landin's SECD machine, following (Burge 1975), who calls it a "procrastinating machine".

On moving to the University of Kent in January 1977 I had two terms with very little teaching which I used to try out an idea I had been mulling over for several years — to apply the normal graph reduction of Wadsworth (1971) to SK combinators. I reimplemented lazy SASL by translation to SK combinators and combinator graph reduction (Turner 1979a, 1979b).

SASL continued to evolve gently, acquiring floating point numbers and list comprehensions[11] in 1983. The latter were inspired by Darlington's "set expressions" (see Section 7 below), but applied to lazy lists.

Burroughs Corporation adopted SASL for a major functional programming project in Austin, Texas, running from 1979 to 1986, to which I was a consultant. The team designed a hardware combinator reduction machine, NORMA (Scheevel 1984), of which two were built in TTL logic. NORMA's software, including compiler, operating system and verification tools was written in SASL.

By the mid-1980's lazy SASL had spread to a significant number of sites, see Table 1. There were three implementations altogether — the version at

[10] I didn't follow Darlington in using $(n + k)$ patterns because SASL's number type was integer rather than natural. I did later (1986) put *(n+k)* patterns into Miranda because Richard Bird wanted them for a book he was writing with Phil Wadler.

[11] I initially called these *ZF expressions*, a reference to Zermelo-Frankel set theory — it was Phil Wadler who coined the better term *list comprehension*.

St Andrews, using a lazy SECD machine, which was rewritten and improved by Bill Cambell; my SK combinator version; and the implementation running on NORMA at Burroughs Austin Research Centre.

6.2 Advantages of Laziness

Two other projects independently developed lazy functional programming systems in the same year as SASL — Friedman & Wise (1976), Henderson & Morris (1976). Clearly laziness was an idea whose time had arrived.

My motives in changing to a lazy semantics in 1976 were

(i) on a sequential machine, consistency with the theory of (Church 1941) requires normal order reduction
(ii) a non-strict semantics is better for equational reasoning
(iii) allows interactive I/O via lazy lists — call-by-value SASL was limited to outputs that could be held in memory before printing.
(iv) I was coming round to the view that lazy data structures could replace exotic control structures, like those of J-PAL.
  (a) lazy lists replace coroutines (e.g. equal fringe problem)
  (b) the *list of successes*[12] method replaces backtracking

6.3 Dynamic Typing

Languages for computation over symbolic data, such as LISP, POP2 and SASL, 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 the polymorphic type system of Milner (1978), the only convenient way to avoid specifying the type of the elements of a structure was to delay type checking until run time. SASL was dynamically typed for this reason.

But languages with dynamic typing also have a flexibility which is hard to emulate with a static type system. This example comes from SASL's 1976 manual. Let f be a curried function of some unknown number of Boolean arguments — we want to test if f is a tautology (the predicate *logical* tests if its argument is a truthvalue):

```
taut f = logical f -> f;
         taut (f True) & taut (f False)
```

There are still active user communities of languages with run time typing including LISP, which is far from disappearing and, a rising newcomer, Erlang (Cesarini & Thompson 2009).

[12] An example of list of successes method — for the 8 queens problem — is in the 1976 SASL manual, but the method didn't have a name until (Wadler 1985).

Table 1. Lazy SASL sites, circa 1986

California Institute of Technology, Pasadena
City University, London
Clemson University, South Carolina
Iowa State U. of Science & Technology
St Andrews University, UK
Texas A & M University
Université de Montréal, Canada
University College London
University of Adelaide, Australia
University of British Columbia, Canada
University of Colorado at Denver
University of Edinburgh, UK
University of Essex, UK
University of Groningen, Netherlands
University of Kent, UK
University of Nijmegen, Netherlands
University of Oregon, Eugene
University of Puerto Rico
University of Texas at Austin
University of Ulster, Coleraine
University of Warwick, UK
University of Western Ontario, Canada
University of Wisconsin-Milwaukee
University of Wollongong, Australia
Burroughs Corporation, Austin, Texas
MCC Corporation, Austin, Texas
Systems Development Corporation, PA
(24 educational + 3 commercial)

7 Developments in Edinburgh, 1969–1980

Burstall (1969), in an important early paper on structural induction, extended ISWIM with algebraic type definitions — still defined in words — and *case expressions* to analyse data structure.

John Darlington's NPL, "New Programming Language", developed with Burstall in the period 1973-5, replaced case expressions with multi-equation function definitions over algebraic types, including natural numbers, e.g.

```
fib (0)   <=  1
fib (1)   <=  1
fib (n+2) <=  fib (n+1) + fib (n)
```

Darlington got this idea from Kleene's recursion equations.

NPL was implemented in POP2 by Burstall and used for Darlington's work on program transformation (Burstall & Darlington 1977). The language was first order, strongly (but not polymorphically) typed, purely functional, call-by-value. It also had "set expressions" e.g.

```
setofeven (X)  <=  <:x : x in X & even(x):>
```

NPL evolved into HOPE (Burstall, MacQueen & Sannella, 1980), this was higher order, strongly typed with explicit types and polymorphic type variables, purely functional. It kept multi-equation pattern matching but dropped set expressions.

Also in Edinburgh during 1973-78 the programming language ML emerged as the meta-language of Edinburgh LCF (Gordon et al 1979) a programmable verification system for Scott's logic for computable functions, PPLAMBDA.

This early version of ML had

λ, *let* and *letrec*
references and assignment
types built using $+$, $\times$ and type recursion.
type abstraction
polymorphic strong typing with *type inference* (NB!)
used `* ** ***` *etc.* as an alphabet of type variables

The language was higher order, call-by-value and allowed assignment and mutable data. It lacked pattern matching — structures were analysed by conditionals, tests e.g. *isl, isr* and projection functions.

Standard ML (Milner et al. 1990), which appeared later, in 1986, is the confluence of the HOPE and ML streams, thus has both pattern matching and type inference, but is not pure — it has references and exceptions.

8 Miranda

Developed in 1983-86, Miranda is essentially SASL plus algebraic types and the polymorphic type discipline of Milner (1978). It retains SASL's policies of no explicit λ's and, optional, use of an offside rule to allow nested program structure by indentation (both ideas derived from Landin's ISWIM). The syntax chosen for algebraic type definitions resembles BNF:

```
tree * ::= Leaf * | Node (tree *) (tree *)
```

The use of `* ** ***` ... as type variables followed the original ML (Gordon et al. 1979) — standard ML had not yet appeared when I was designing Miranda.

For type specifications I used "`::`" because following SASL ":" was retained as infix *cons*.

A lexical distinction between variables and constructors was introduced to distinguish pattern matching from function definition. The decision is made on the initial letter of an identifier — upper case for constructors, lower case for variables. Thus

```
Node x y = stuff
```

is a pattern match, binding x, y to a, b if *stuff* $=$ *Node* a b whereas

```
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 Turner (1986) for fuller description — papers and downloads at `www.miranda.org.uk`

An important change from SASL — Miranda had, instead of conditional expressions, conditional equations with guards. Example:

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

Combining pattern matching with guards gives a significant gain in expressive power. Guards of this kind first appeared in KRC, "Kent Recursive Calculator" (Turner 1981, 1982), a miniaturised version of SASL which I designed in 1980–81 for teaching. Very simple, KRC had only top level equations (no *where*) with pattern matching and guards; and a built in line editor — a functional alternative to BASIC. KRC was also the testbed for list comprehensions, from where they made their way into SASL and then Miranda.

Putting *where* into a language with guards raised a puzzle about scope rules, forcing a rethink of part of the ISWIM tradition. The solution is that a *where*-clause now governs a whole rhs, including guards, rather than an expression. That is *where* becomes part of definition syntax, instead of being part of expression syntax (a decision that is retained in Haskell).

Miranda was a product of Research Software Ltd, with an initial release in 1985, and subsequent releases in 1987 and 1989. It was quite widely taken up with over 200 universities and 40 companies taking out licenses.

Miranda was by no means the only project combining Milner's polymorphic type system with a lazy, purely functional language in this period (mid 1980's).

Lazy ML, first implemented at Chalmers in 1984 was, as the name suggests, a pure, lazy version of ML, used by Lennart Augustsson and Thomas Johnsson as both source and implementation language for their work on compiled graph reduction which we referred to in Section 1.1, see (Augustsson 1984; Augustsson & Johnsson 1989).

At Oxford, Philip Wadler developed Orwell, a simple equational language for teaching functional programming, along much the same lines as Miranda. Orwell and Miranda were able to share a text book. Bird & Wadler (1988) used a mathematical notation for functional programming — e.g. Greek letters for type variables — that could be used with either Miranda or Orwell (or indeed other functional languages).

Clean, a lazy language based on graph reduction, with *uniqueness types* to handle I/O and mutable state. was developed at Nijmegen from 1987 by Rinus Plasmeijer and his colleagues (Plameijer & van Eekelen 1993).

9 Haskell

Designed by a committee which started work in 1987, version 1.2 of the Haskell Report was published in SIGPLAN Notices (Hudak et al. 1992). The language continued to evolve, reaching a declared standard for long term support in Haskell 98 (Peyton Jones 2003).

Similar in many ways to Miranda, being lazy, higher order, polymorphically typed with algebraic types, pattern matching and list comprehensions, the most noticeable syntactic differences are:

Switched guards to left hand side of equations

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

Change of syntax for type declarations — Miranda

```
bool ::= True | False
string == [char]
```

becomes in Haskell

```
data Bool = False | True
type String = [Char]
```

Extension of Miranda's var/constructor distinction by initial letter to types, giving lower case type variables, upper case type consts — Miranda

```
map ::  (*->**)->[*]->[**]
filter :: (*->bool)->[*]->[*]
zip3 :: [*]->[**]->[***]->[(*,**,***)]
```

becomes in Haskell

```
map :: (a->b)->[a]->[b]
filter :: (a->Bool)->[a]->[a]
zip3 :: [a]->[b]->[c]->[(a,b,c)]
```

Haskell has a richer and more redundant syntax. e.g. it provides conditional expressions and guards, *let*-expressions and *where*-clauses, *case*-expressions and pattern matching by equations, λ-expressions and *function-form = rhs* etc. ...

Almost everything in Miranda is also present in Haskell in one form or another, but not vice versa — Haskell includes a system of type classes, monadic I/O and a module system with two level names. Of these the first is particularly noteworthy and the most innovative feature of the language. An account of the historical relationship between Haskell and Miranda can be found in (Hudak et al. 2007, s2.3 and s3.8).

```
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)
```

Fig. 1. Class *Taut*

This is not the place for a detailed account of Haskell, for which many excellent books and tutorials exist, but I would like to close the section with a simple example of what can be done with type classes. Let us try to recover the variadic tautology checker of Section 6.3.

Figure 1 introduces a class *Taut* with two instances to cover the two cases. Unfortunately the second instance declaration is illegal in both Haskell 98 and the current language standard, Haskell 2010. Instance types are required to be generic, that is of the form $(Ta_1 \ldots a_n)$. So $(a \to b)$ is allowed as an instance type but not $(Bool \to a)$.

```
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))
```

Fig. 2. Variadic *taut* in Haskell 2010

GHC, the Glasgow Haskell compiler, supports numerous language extensions and will accept the code in Figure 1 if language extension *FlexibleInstances* is

enabled. To make the example work in standard Haskell is more trouble; we have to introduce an auxilliary type class, see Figure 2. One could not claim that this is as simple and transparent as the SASL code shown earlier.

When what was to become the Haskell committee had its first scheduled meeting in January 1988 a list of goals for the language, which did not yet have a name, was drawn up, see (Hudak et al. 2007, s2.4), including

It should be based on ideas which enjoy a wide consensus.

Type classes cannot be said to fall under that conservative principle: they are an experimental feature, and one that pervades the language. They are surprisingly powerful and have proved extremely fertile but also add greatly to the complexity of Haskell, especially the type system.

Acknowledgements. I am grateful to the programme committee of TFP 2012 for inviting me to give a lecture on the history of functional programming at TFP 2012 in St Andrews. This written version of the lecture has benefitted from comments and suggestions I received from other participants in response to the lecture. In particular I am indebted to Josef Svenningsson of Chalmers University for showing me how to code the variadic *taut* function in Haskell using type classes.

Section 1 draws on material from my essay on *Church's Thesis and Functional Programming* in (Olszewski et al. 2006, 518-544).

References

Augustsson, L.: A compiler for Lazy ML. In: Proceedings 1984 ACM Symposium on LISP and Functional Programming, pp. 218–227. ACM (1984)

Augustsson, L., Johnsson, T.: The Chalmers Lazy-ML Compiler. The Computer Journal 32(2), 127–141 (1989)

Bird, R.S., Wadler, P.: Introduction to Functional Programming, 293 pages. Prentice Hall (1988)

Burge, W.: Recursive Programming Techniques, 277 pages. Addison Wesley (1975)

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

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

Burstall, R.M., MacQueen, D., Sanella, D.T.: HOPE: An experimental applicative language. In: Proceedings 1980 LISP Conference, Stanford, California, pp. 136–143 (August 1980)

Cesarini, F., Thompson, S.: Erlang Programming, 498 pages. O'Reilly (June 2009)

Church, A., Rosser, J.B.: Some Properties of conversion. Transactions of the American Mathematical Society 39, 472–482 (1936)

Church, A.: The calculi of lambda conversion. Princeton University Press (1941)

Curry, H.B., Feys, R.: Combinatory Logic, vol. I. North-Holland, Amsterdam (1958)

Evans, A.: PAL - a language designed for teaching programming linguistics. In: Proceedings ACM National Conference (1968)

Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. In: Proceedings 3rd Intl. Coll. on Automata Languages and Programming, pp. 256–284. Edinburgh University Press (1976)
Gordon, M., Wadsworth, C.P., Milner, R.: Edinburgh LCF. LNCS, vol. 78. Springer (1979)
Henderson, P., Morris, J.M.: A lazy evaluator. In: Proceedings 3rd POPL Conference, Atlanta, Georgia (1976)
Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction, 2nd edn., 360 pages. Cambridge University Press (August 2008)
Hudak, P., et al.: Report on the Programming Language Haskell. SIGPLAN Notices 27(5), 164 pages (1992)
Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A History of Haskell: Being Lazy with Class. In: Proceedings 3rd ACM SIGPLAN History of Programming Languages Conference, San Diego, California, pp. 1–55 (June 2007)
Hughes, J.: The Design and Implementation of Programming Languages, Oxford University D. Phil. Thesis (1983); (Published by Oxford University Computing Laboratory Programming Research Group, as Technical Monograph PRG 40 (September 1984)
Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, Springer, Heidelberg (1985)
Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal 6(4), 308–320 (1964)
Landin, P.J.: A generalization of jumps and labels. Report, UNIVAC Systems Programming Research (August 1965); Reprinted in Higher Order and Symbolic Computation 11(2), 125–143 (1998)
Landin, P.J.: The Next 700 Programming Languages. CACM 9(3), 157–165 (1966)
McCarthy, J.: Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. CACM 3(4), 184–195 (1960)
McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 Programmer's Manual, 106 pages. MIT Press (1962); 2nd edn. 15th printing (1985)
McCarthy, J.: History of LISP. In: Proceedings ACM Conference on History of Programming Languages I, pp. 173–185 (June 1978),
`www-formal.stanford.edu/jmc/history/lisp/lisp.html`
Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17(3), 348–375 (1978)
Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. MIT Press (1990) (revised 1997)
Naur, N.: Revised Report on the Algorithmic Language Algol 60. CACM 6(1), 1–17 (1963)
Olszewski, A., et al. (eds.): Church's Thesis after 70 years, 551 pages. Ontos Verlag, Berlin (2006)
Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. Journal of Functional Programming 2(2), 127–202 (1992)
Peyton Jones, S.L.: Haskell 98 language and libraries: the Revised Report. JFP 13(1) (January 2003)
Plasmeijer, R., Van Eekelen, M.: Functional Programming and Parallel Graph Rewriting, 571 pages. Addison-Wesley (1993)
Randell, B., Russell, L.J.: The Implementation of Algol 60. Academic Press (1964)
Reynolds, J.C.: GEDANKEN a simple typeless language based on the principle of completeness and the reference concept. CACM 13(5), 308–319 (1970)

Scheevel, M.: NORMA: a graph reduction processor. In: Proceedings ACM Conference on LISP and Functional Programming, pp. 212–219 (August 1986)

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

Turner, D.A.: SASL Language Manual, St. Andrews University, Department of Computational Science Technical Report CS/75/1 (January 1975); revised December 1976; revised at University of Kent, August 1979, November 1983

Turner, D.A.: A New Implementation Technique for Applicative Languages. Software-Practice and Experience 9(1), 31–49 (1979a)

Turner, D.A.: Another Algorithm for Bracket Abstraction. Journal of Symbolic Logic 44(2), 267–270 (1979b)

Turner, D.A.: The Semantic Elegance of Applicative Languages. In: Proceedings MIT/ACM conference on Functional Languages and Architectures, Portsmouth, New Hampshire, pp. 85–92 (October 1981)

Turner, D.A.: Recursion Equations as a Programming Language. In: Darlington, Henderson, Turner (eds.) Functional Programming and its Applications, pp. 1–28. Cambridge University Press (January 1982)

Turner, D.A.: An Overview of Miranda. SIGPLAN Notices 21(12), 158–166 (1986)

Wadler, P.: Replacing a failure by a list of successes. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)

Wadsworth, C.P.: The Semantics and Pragmatics of the Lambda Calculus. D.Phil. Thesis, Oxford University Programming Research Group (1971)