FUNCTIONAL PROGRAMMING[1]

by

William S. Shu

CHAPTER I: INTRODUCTION

The essence of what I mean
Functions on preliminaries,
A broad and general understanding.
—Shu WS

In notably end-user computing and software engineering, the term "functional" tends to suggest having lots of features ("functionality") or requirements to meet. Here, functional tends to imply the evaluation of expressions in the same way as one applies functions. That is, expressions are evaluated without side-effects—as expressions, as opposed to commands—and in this wise functional programming languages are contrasted with imperative programming languages. Functional programming languages are also called *applicative programming languages*, given that their key [defining] operation is applying functions to their arguments. They are of the declarative programming paradigm and non-procedural since they specify *what* is computed and not *how*.

RECAP: BASIC PROGRAMMING CONCEPTS

We recall the following concepts which are assumed in the rest of these notes, but which may also be revised to more technical, more precise, or more restrictive interpretations as the context of their use may warrant. They are: models of computation, including Turing machines; programming paradigms; and assorted programming language concepts (syntax, semantics and types). Other needed concepts are also discussed, either here or closer to where most relevant.

Models of Computation:

"A *computational model* is an abstract description of how a computation takes place. Informally, a computation is a mathematical description of carrying out operations that move a machine from one state to another until it reaches a final acceptance state, or fails to do so. A *state* is simply a configuration of values. [For programming purposes, we have] 3 broad models of computation" (CSC 207 notes), i.e., imperative, functional and logical.

Imperative model: It is "based on the notion of the Turing machine", of which there are many variants. Essentially, a Turing machine moves left or right and reads from or writes to (or overwrites) cells on an infinite (or semi-finite) tape as it moves from state to another. The possible transitions between states describes how the machine should function. "Initially, the tape starts with the input [data] of a problem written on it and, as computation progresses, the data gets gradually altered until the tape eventually

contains just the output [of the computation]. The machine halts when it finds an answer, but might never stop [its actions] if it does not" (CSC 207 notes).

Functional model: "The functional model of computation is based on the λ-calculus (lambda calculus) which seeks to formalise computation as the application and composition of mathematical functions over values. Necessarily, it requires that functions be expressed in terms of other functions (composition) and be seen as values that can also be manipulated during composition."

Logic programming model: "It is based on having a set of values, the relationships (rules) among these values, and the ability to make logical inferences over these values and relationships. In this model, a computation is essentially a proof. That is, a sequence of inferences to show that a property holds or can be inferred (decided) from some set of values, rules and/or properties".

Mathematical Formulations of Computation:

At an abstract, more theoretical level, the above models of computation have mathematically more precise formulations used to capture the notion of computation: these are the Turing machine, the lambda calculus, and logical inference respectively. Other mathematical formulations for computation exist. We state a few classic ones, which may help situate some of the concepts introduced: "

- *primitive recursive functions*: Loosely, this is the smallest class of [total] functions on the *natural numbers* (non-negative integers) that can be expressed recursively (in the PL sense) and that can be expressed as the mathematical composition of two or more similar functions (the general case) or a similar function using fewer of its arguments.
- *μ -recursive (mu-recursive) function*: Loosely, a primitive recursive function that has been *minimalised*; that is, a primitive recursive function that additionally has the smallest set of values [of possibly many values].
- *partially recursive functions*: they are equivalent to *partial μ-recursive functions*. Mathematically, a function is *partial* if some of its arguments (domain values) have no valid matching value in its codomain (or range). *e.g.*, division among integers is partial as 3÷2, say, is not an integer.
- *Post/semi-Thue/production/rewrite systems*: semi-Thue and production systems: which permit symbols in strings to be rewritten [to other strings].

Note that the lambda calculus (due to Church) is useful in mathematically describing programming languages, and many functional programming languages simply provide a thin veneer of syntactic sugar over it. However, the calculus was actually meant to model computation, not to describe algorithms [for computer systems]. Similarly, recursion theory (due to Kleene and Rosser) describes computable Mathematics. Combinatory logic (due to Curry), which constructs constants for operations in the lambda calculus, describes type theory in programming languages.

The Church-Turing Thesis:

This thesis asserts that "when a problem is expressible and computable in one model of computation, it is computable in other models of computation. … Of course, some problems are more conveniently expressed in some models of computation than in others. The main paradigms of programming are typically more convenient for problems natural to [their] models of computation" (CSC 207 notes).

Programming Language Presentations:

Syntax, Semantics, and Pragmas:

"A *programming language* is a collection of primitives (basic actions and control and data constructs), the rules governing how the primitives are combined and the meanings attributed to them (*i.e.*, the primitives and their combinations) in order to express more complex ideas [as algorithms]. *Syntax* refers to the symbols representing primitives and how they can be combined; the rules constitute the *grammar* of the language. It describes how components are correctly put together. *Semantics* refers to the concept represented or the meaning of the primitives. More precisely, it refers to the meaning associated with

entities (actions, constructs, objects, etc.). For a program, the basic actions (primitives) are the instructions recognised by the computer (or processor). [*Programming language*] *pragmas* (short for *pragmatics*) refers to options and techniques that permit one to efficiently use or implement a language. *e.g.*, how best to implement variable bindings in a compiler." (CSC 207 Notes.)

Programming Paradigms:
Programming languages are often classified, historically, according to their generations. However, they are also classified based on alternative approaches "to the programming process (the different paradigms) that are determined by the *model of computation* (abstract mathematical description of what the notion of computation is) used to establish the meaning of programs. As such, we have the following programming paradigms, based on 3 (or 4) models of computation" (CSC 208 notes): imperative or procedural; functional or applicative; logic programming; and declarative. The object-oriented paradigm is computationally seen as a variant of the imperative paradigm, as is scripting languages which some reject as a programming paradigm. See also CSC 208 notes. Note that the declarative paradigm is often said to be for declarative languages and so cover functional and logic programming paradigms.

The notion of programming paradigm is loosely defined as "a way of viewing the programming process". Thus, it is often associated with the underlying architectural models of machines designed for their execution, such as dataflow, parallel and distributed machines. Increasingly, new paradigms are "defined by the key features of particular programming language. … To avoid confusion with the classic paradigms of programming [and emphasise their affiliation to specific programming languages], we call these paradigms *Programming Language Paradigms*". Examples include aspect oriented programming, metaprogramming, generic programming, and probabilistic programming (CSC 208 notes).

More technically, "declarative paradigms focus on what to solve, rather than how (i.e. rather than the algorithm to solve the problem). Program development is the elaboration of a precise statement of the problem, rather than discovering an algorithm to solve the problem. The problem can then be solved using a general problem solving algorithm. Functional and logic paradigms are such general problem solving algorithms since they emphasis the "what" not "how" of problems. Note, however, that declarative languages are often designed for special purpose applications, such as simulations in *e.g.*, Economics, and need not be computationally "general"—strictly, computationally complete, being able to execute any algorithm—since they focus on the "what" and not the "how" of problems" (CSC 208).

Programming Language Types:
A "[*data*] *type* is essentially a set of values (objects) together with a set of permissible operations on them; that is, they determine what actions can be applied to the data values. Thus, for example, numbers (integers) allow for standard arithmetic operations on them; eggs allow for whisking (by a baker)" (CSC 207 notes). More generally, we see types as constraints imposed on how objects of a collection may be used. Thus, type-checking (see below) ensures that constraints on values and operations are respected and type-inference (see below) deduces the type of values to expect in a correct program. Furthermore, types are used to formally establish the above (c.f., program verification) by specifying invariants that a type must satisfy. (*Invariants* are properties that must not change or, if they do, must be restored by the time they are next verified or asserted; c.f., program correctness in CSC 207.)

A *type system*, then, defines a set of rules that attributes types to terms—that is, that make terms *well-typed*. Operationally, a type system rejects those terms it cannot attribute a type to. A language (or term) is said to be *untyped* if it has no types attributed to its terms, or if such types are not enforced. It is *strongly typed* if it enforces typing—the application of operations only to terms/objects of appropriate type. It is *statically typed* if one can establish the type of each term at compile time (in the program text), before the program is executed. The language is *dynamically typed* if the type can only be determined at runtime (when the program is executed).

Type-checking:
Type checking establishes if an expression or program is correctly typed and determines the type of any correctly typed expression; it signals error(s) for incorrectly-typed expressions. If type checking is done at compile-time, we say that the expression is *statically typed*. Languages such as Pascal, C, C++ effect static typing requiring the programmer attribute a type to each object of interest in the program. Alternatively, a type inference system (see below) may be included in a compiler and used to establish correct typing at compile time. Languages such as Ocaml, Miranda and Haskell use this approach.

Type checking done at runtime is said to be *dynamic*, when typing errors are detected. At runtime, terms (or objects or data values) tagged with typing information are checked for type compatibility and necessary type conversions. Lisp, Scheme and Smalltalk are example dynamically-typed languages.

Type Inference (or Type Reconstruction):
Type inference seeks to establish or attribute a type to a given object or term, possibly from partially typed objects and terms. In Pascal, it is in its simplest form in constant declarations (which inherit the type of the constants they name) and for-loop variables whose types are deduced from the types of the loop limits (i.e., the start and finish values of for loops). At the other extreme, statically typed languages such as Ocaml and Haskell need not declare the type of their terms (Aaby, 1996).

Type equivalence:
Type equivalence establishes whether two types are or could be considered the same, and so one could replace the other in their typed operations. "Two types T, T' are *name equivalent* iff T and T' have the same name. Informally, they are *structurally equivalent* iff T and T' have the same set of values". As often used, structural equivalence means their values are constructed in the same way; in its simplest form, for computability reasons, this means there is a correspondence between values and constructors used to construct corresponding values from T and T'.

INTRODUCTION: FUNCTIONAL PROGRAMMING

Functional vs Imperative Programming:
In an imperative programming language such as C, the sum to 42 may be given as: "for (i = 0, sum = 0; i <= 42; i++) sum += i;". It produces side-effects on—i.e., it changes the states (values) of—variables i and sum; we notice the effect of the for command on these variables. In contrast, a functional language may give the same result as sum[1..42] without using variables (and hence no discernible side effects). Even where variables are used as in "let rec sum n = (if n = 0 then 0 else n + (sum n-1)) in sum 50", the variable (here n) is the name of a value, as in mathematics, and so dose not changed after execution.

Thus, in functional programming one seeks to write programs as the mathematical composition of functions … and their execution as function application; functional languages support and encourage functional [style of] programming. Interest in "functional programming arises from the great ease of writing programs and specifying the values which they manipulate", notably mathematical constructions.

Why Functional Programming?:
"Functional programming encourages thinking at higher levels of abstraction by providing higher-order functions—functions that modify and combine existing programs" (Aaby, 1996). Functional programming is natural to certain implementation paradigms (e.g., concurrent programming). It is also desirable in application areas that require working at higher levels abstraction (e.g., in Artificial Intelligence, executable specifications, and implementation of prototypes) or where one wants to exploit mathematical formulations or structures.

Functional programming is closely related to theoretical computer science, since it is based on the lambda calculus, and so provides a framework for studying [and implementing] important concepts in

Computer Science. e.g., decidability and computability issues, as well as exploiting denotational semantics in program transformations.

In eliminating side effects functional programming eases understanding of code by making explicit the relationships between input and output. It also enhances the reuse and composition of functions. Also, this I/O relationship gives more leeway for optimisations peculiar to specific execution platforms.

Some Concepts Important to Functional Programming:
Functional programming exploits the mathematical notion of functions and function application, and is typically made up of the following sets or items (Aaby, 1996; CSC 207 notes on language constructs): A set of primitive or basic functions assumed to be part of the language; a set of functional forms which are the accepted format or structures for functions that may take or return other functions; the [*function*] *application* operation so one can apply one function to another; a set of data objects and associated functions (c.f, data types); and an abstraction mechanism to bind names to functions.

Some important (and general) FP concepts include:

- The notion of functions as first-class entities: they can be treated as other values of the language (as parameters, return values, etc.) and as other computations as well.
- Multiple evaluation semantics ease the construction, use and reuse of programs: lazy (deferred evaluation), call-by-name, call-by-need (only when needed), call-by-value (eager evaluation).
- Basic but useful functionalities: fold, iteration, map; translation (homomorphic mapping); expression of powerful mathematical concepts.
- Nature of values: immutable values limit negative impact of side effects. Special cases are handled uniformly via the unit value (contrast handling of null references with pointers).

Families of Functional Languages:
Families of functional languages are typically classified based on their ancestry, which type of functional programming language they derive from. E.g., Ocaml derives from the ML family of programming languages. However, such families usually sit on, or reflect, certain key technical features..

Some Defining Characteristics:
For example, some classify functional programming languages based on the following [combination of] characteristics, though the first two are typically dominant when characterising FP families:

Without side effects (pure) or with side effects (impure): A pure functional programming language is one whose functions, when applied to the same input arguments two or more times, always give the same result. (Technically, its functions are *referentially transparent*.) In practical terms, it does not have side-effects (change of state) nor imperative programming language features (e.g., while loops): everything is a computation and how it is carried out is unimportant unlike for imperative programming languages. Examples include Miranda where a "program is a sequence of equations defining functions and data structures", and Haskell where even its I/O system uses monads. (Crudely, monads encapsulate side-effects and so mask them from computations beyond (i.e., outside the monads). In contrast, impure languages such as Lisp and ML exploit imperative language features, notably change of state and significance in the order of evaluation of expression (i.e., flow of control).

Static vs Dynamic Typing: Typing permits one to verify whether the type of a function's argument indeed matches that of its formal parameter. If this verification can be done before program execution, at compile time, and is called *static typing*. It can be carried out once and for all and so cannot slow down program execution. This is the case in the ML language and its dialects such as Ocaml: only correctly typed programs, i.e., those accepted by the type verifiers, can be compiled [and then executed]. The verification can also be done during program execution, and called *dynamic typing*. If type errors occur the program will halt in a consistent state. This is the case in the language Lisp.

Strict vs Lazy Evaluation: Strictly, there are many evaluation modes, but these are the most common distinguishing evaluation characteristics. In strict function evaluation, a function is undefined if any of its argument is undefined (or unavailable) and so cannot be applied; its arguments are evaluated as encountered and passed to the function. An advantage is that function application is efficient. In lazy evaluation, a function defers the evaluation of its argument(s), often doing so only when it is necessary. The upside is that possibly infinite data structures (e.g., I/O streams) can be easily implemented, and certain computations avoided if not necessary or erroneous. (Note that immediate argument evaluation is called eager evaluation. Technically, strict and lazy evaluations need not be mutually exclusive; in practical terms they almost certainly are.)

Type of Polymorphism: Informally, polymorphism is the ability of functions or operators to carry out the same kind of tasks (e.g., sorting) on, or be interpreted differently (e.g., addition used to add integers but to concatenate strings) for, values of different types.

Families of Functional Programming Languages.
Some broad families of functional languages identified are given below, though opinions may vary where classification does not reckon with ancestry. (Some terms used are defined later in the course).
ML family: The ML family has two main branches Caml (from which descends OCaml) and Standard ML (SML) with descendants such as SML/NJ (i.e., SML/New Jersey) and mosml. Typing is static and polymorphism is top-level parametric—the so-called ML-style polymorphism.

Lisp family: This family includes the Scheme language. It differs from ML family mainly due to its its dynamic typing. They have some imperative programming language features. They are based on LISP (for LISt Processing), by John McCarthy in 1958, and considered the first functional programming language. LISP processes expressions as lists of operations in recursive functions calls. It was intended to handle symbolic computation in areas such as theorem proving, artificial intelligence and natural language processing. Historically, its implementation with dynamic scoping is inadvertent. Scheme is a smaller, modern version of LISP, which uses static scoping.

Miranda Family: This family is typified by Miranda though Haskell, which is heavily influenced by Miranda, is more recent and more popular. It is characterised by lazy evaluation, statical typing and being purely functional. Miranda allows for list comprehension, polymorphic type inference and optional type specifications (Turner, D A). The language Haskell was named after the logician Haskell B. Curry and seeks to continuously include beneficial ideas from functional language research that could be used for teaching, research and applications. Haskell allows for overloading in its polymorphic type system, purely functional implementations of I/O and arrays, and data abstraction and information hiding.

Other: Some families or groups of languages are based on their application areas. For instance we have communication languages typified by ERLANG which is a dynamically typed language used for concurrent programming, and SCOL which is used for communication and construction of "3D worlds". ERLANG was developed by Ericsson Corporation with telecommunications applications in mind.

Functional Programming: Historical: TODO
?? History (c.f., D. A. Turner, "Some History of Functional Programming Languages"; Barendregt, 1990 (introd.)). Situate historically and give: theoretical underpinning, language influences and key concepts (e.g., lazy, higher order, polymorphism, pure FP languages, static vs dynamic), practical implementation.

"In the 1930s Alonso Church [and Stephen Cole Kleene] developed the lambda-calculus as an alternative to set theory for the foundations of mathematics and Haskell B. Curry developed combinatory logic for the same reason. While their goal was not realised, the lambda-calculus and combinators capture the most general formal properties of the notion of a mathematical function, [and] are abstract models of computation equivalent to the Turing machine, recursive functions, and Markov chains".

The λ-calculus, unlike the Turing machine which is sequential, retains the implicit parallelism in mathematical expressions. Semantically, the typed λ-calculus could be seen as a meta-language whose meaning is formally captured by λ-calculus objects modelled in domain theory. This led to the use of denotational semantics for formal descriptions of programming languages, as started by Peter Landin, Christopher Strachey and others in the 1960's. Dana Scott (1969) discovered the first mathematical model [as opposed to theory] for the type-free lambda-calculus. Concepts influenced by the lambda-calculus include: the *call by name* parameter passing mechanism of Algol-60, textual substitutions in macros, and the LISP programming language, though "the theoretical model behind LISP was Kleene's theory of first order recursive functions".

Models of the lambda-calculus:
"Lambda theories are equational extensions of the untyped λ-calculus that are closed under derivation" (Antonino Salibra, "Lambda Calculus: Models and Theories"). Syntactically, a λ-theory may match a "possible operational (observational) semantics of the lambda calculus" and, semantically, it "may induce models of the λ-calculus," mostly based on topological constructions. Models of the typed lambda calculus are typically based on [semantic] domains which are partially ordered structures, variations of them classified by their [category-theoretic] representable functions (Salibra); partially recursive functions (i.e., recursive function theory or Turing machine computations); partial combinatory algebras which generalise recursive functions (Mitchell, 1993); and cartesian closed categories (CCC). Henkin models correspond to, and Kripke lambda models are, special cases of CCC based on functor categories. Henkin models have a set of elements for each type; "a Kripke model has a family of sets indexed by a partially-ordered set of 'possible worlds'" (Mitchell,1993).

Implementation(?) TODO: FP languages are usually implemented as a sequence of translations to virtual machines: translations into the λ-calculus, into combinatorial logic and then into the code for a graph reduction machine (Aaby, 1996). Hardware designs can directly execute the λ-calculus and combinators, with full support for parallel executions. ?? What of graph reductions implementations?

Ocaml:
"Caml is a general-purpose programming language ... developed and distributed by INRIA, a French research institute in computer science and applied mathematics, since 1985" (http://caml.inria.fr/). It was "designed with program safety and reliability in mind" and supports the functional, imperative, and object-oriented programming paradigms. "*Caml* was originally an acronym for Categorical Abstract Machine Language" which is a pun on CAM, the Categorical Abstract Machine, and ML, its language family, even though its more modern implementations, such as current Ocaml, bear no relation to CAM.

OCaml features include: ML-style static typing and type inference, even in the presence of advanced object-oriented programming idioms (such as types, parametric classes and type specialisation); a powerful module system; polymorphic methods, labelled and optional function arguments, and polymorphic variants". Ocaml has a native-code compiler for high performance execution; "a bytecode compiler, for increased portability; and an interactive loop, for experimentation and rapid development."

Ocaml has steadily gained popularity, with a significant user base from the late 1990's. Programs in Ocaml and by its user community include "high-quality libraries, frameworks and tools in areas ranging from graphical user interfaces and database bindings to Web and network programming, cross-language interoperability and static program analysis". The "core OCaml development team actively maintains the base system", and improves implementations and ports to "the latest architectures and systems."

Ocaml programming: See Hickey (2008), and the Ocaml reference manual, v. 4.02 or later (Leroy et al, 2014).

CHAPTER II:
DIGRESSIONS or BACKGROUND CONCEPTS

Oh Yes! It's neither fate nor fake!
Context is not for Context's sake,
So do not Context forsake,
For it does define the stakes!

—Shu WS

[***Note***: *This part of the notes presents basic notions and concepts that may be useful in understanding other parts of these notes. It is not taught, need not be in prerequisite courses, and will not be examined directly unless used in other parts of the notes. It is added for comprehensiveness, disambiguation, and easier understanding of some of the recurrent concepts in computer science, so the student need not search through alternative usage to grasp those relevant to this course, or to Computer Science in general. We generally remain very informal, largely seeking to convey an intuitive understanding*!]

***DIGRESSION*: PROGRAMMING LANGUAGE THEORY**:
Functional programming is often studied under programming language theory, a subdiscipline of Computer Science. *Programming language theory* (*PLT*) "deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features. It [depends on and affects] Mathematics, Software Engineering, Linguistics and even Cognitive science" (wiki). PLT exploits many "branches of mathematics, including computability theory, category theory, and set theory" (wiki).

Subfields of Programming Language Theory:
Some subfields with active research in programming language theory (from wiki) include the following:

- *Formal semantics*: Formal semantics states in mathematical terms the meaning of computer programs and programming languages, and hence how they should behave. Such meanings are typically given in terms of denotational, operational and axiomatic semantics.
- *Type theory*: Type theory studies types and type systems in order to mechanically classify program terms according to the values they compute and the behaviours they exhibit.
- *Program analysis and transformation*: *Program analysis* examines a program in order to establish key properties relevant to an issue (e.g., control follow or absence of certain errors). Program transformation 'translates' a program from one language form into another.
- *Comparative programming language analysis*: "Comparative programming language analysis seeks to classify programming languages into different types based on their characteristics; broad categories of programming languages are often known as programming paradigms" (wiki?).
- *Generic and metaprogramming*: "Metaprogramming is the generation of higher-order programs which, when executed", generate programs in possibly different languages (wiki). Generic programming, also called polytypic or type-indexed programming, essentially seeks "to provide the programmer with the ability to define a function by induction on the structure of types" (Hinze et al. 2004). Loosely, a polytypic function is one that "can be instantiated on many data types to obtain datatype-specific functionality" (ibid.). Conceptually, "type-indexing" conveys the notion that we use different data types to identify (index) the required functionalities. The above definition focuses on lower-level mechanisms to implement genericity in programming languages, such as templates in C++. A variant interpretation is based on higher-level view of generic programming: In it generic programming is a paradigm for software decomposition where "fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalized as concepts, analogously to the abstraction of algebraic theories in abstract algebra" (wiki). "Generic programming is about abstracting and