

Chapter 1

Introduction

Functional programming languages are essentially as old as the more well-known imperative programming languages like FORTRAN, PASCAL, C etc. The oldest functional programming language is LISP which was developed by John McCarthy in the 1950ies, i.e. essentially in parallel with FORTRAN. Whereas *imperative* or *state-oriented* languages like FORTRAN were developed mainly for the purpose of *numerical computation* the intended area of application for functional languages like LISP was (and still is) the algorithmic manipulation of *symbolic data* like lists, trees etc.

The basic constructs of imperative languages are commands which modify state (e.g. by an assignment $x:=E$) and conditional iteration of commands (typically by **while**-loops). Moreover, imperative languages strongly support *random access* data structures like arrays which are most important in numerical computation.

In *purely functional languages*, however, there is no notion of state or state-changing command. Their basic concepts are

- application of a function to an argument
- definition of functions either *explicitly* (e.g. $f(x) = x*x+1$) or *recursively* (e.g. $f(x) = \mathbf{if } x=0 \mathbf{ then } 1 \mathbf{ else } x*f(x-1) \mathbf{ fi}$).

These examples show that besides application and definition of functions one needs also basic operations on basic data types (like natural numbers or booleans) and a conditional for definition by cases. Moreover, all common functional programming languages like LISP, Scheme, (S)ML, Haskell etc. provide the facility of defining *recursive data types* by explicitly listing their constructors as e.g. in the following definition of the data type of binary trees

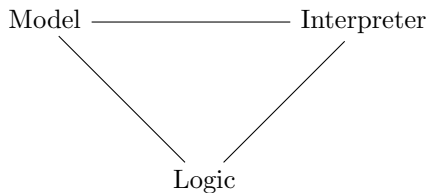
`tree = empty() | mk_tree(tree, tree)`

where `empty` is a 0-ary constructor for the empty tree with no sons and `mk_tree` is a binary constructor taking two trees t_1 and t_2 and building a new tree where the left and right sons of its root are t_1 and t_2 , respectively. Thus functional languages support not only the recursive definition of functions but also the recursive definition of data types. The latter has to be considered as a great advantage compared to imperative languages like PASCAL where recursive data types have to be implemented via pointers which is known to be a delicate task and a source of subtle mistakes which are difficult to eliminate.

A typical approach to the development of imperative programs is to design a *flow chart* describing and visualising the *dynamic behaviour* of the program. Thus, when programming in an imperative language the main task is to organize *complex dynamic behaviours*, the so-called *control flow*.

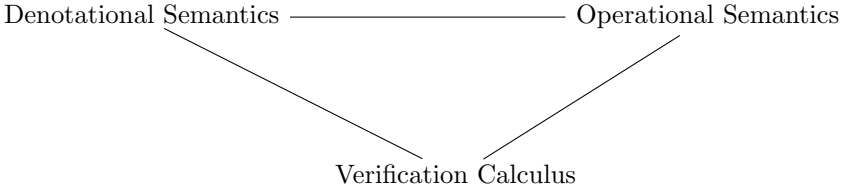
In functional programming, however, the dynamic behaviour of programs need not be specified explicitly. Instead one just has to *define* the function to be implemented. Of course, in practice these function definitions are fairly hierarchical, i.e. are based on a whole cascade of previously defined auxiliary functions. Then a *program* (as opposed to a function definition) usually takes the form of an application $f(e_1, \dots, e_n)$ which is *evaluated* by the interpreter¹. As programming in a functional language essentially consists of defining functions (explicitly or recursively) one need not worry about the dynamical aspects of execution as this task is taken over completely by the interpreter. Thus, one may concentrate on the *what* and forget about the *how* when programming in a functional language. However, when defining functions in a functional programming language one has to stick to the *forms of definition* as provided by the language and cannot use ordinary set-theoretic language as in everyday mathematics.

In the course of these lectures we will investigate functional (kernel) languages according to the following three aspects



¹But usually implementations of functional languages also provide the facility of compiling your programs.

or



respectively and, in particular, how these aspects interact.

First we will introduce a most simple functional programming language PCF (Programming Computable Functionals) with natural numbers as base type but no general recursive types.

The *operational semantics* of PCF will be given by an *inductively* defined *evaluation relation*

$$E \Downarrow V$$

specifying which expressions E *evaluate* to which values V (where values are particular expressions which cannot be further evaluated). For example if $E \Downarrow V$ and E is a closed term of the type **nat** of natural numbers then V will be an expression of the form \underline{n} , i.e. a canonical expression for the natural number n (usually called *numeral*). It will turn out as a property of the evaluation relation \Downarrow that $V_1 = V_2$ whenever $E \Downarrow V_1$ and $E \Downarrow V_2$. That means that \Downarrow is *deterministic* in the sense that \Downarrow assigns to a given expression E at most one value. An operational semantics as given by an (inductively defined) evaluation relation \Downarrow is commonly called a “Big Step Semantics” as it abstracts from intermediary steps of the computation (of V from E).² Notice that in general there does not exist a value V with $E \Downarrow V$ for arbitrary expressions E , i.e. not every program terminates. This is due to the presence of *general recursion* in our language PCF guaranteeing that *all* computable functions on natural numbers can be expressed by PCF programs.

Based on the big step semantics for PCF as given by \Downarrow we will introduce a notion of *observational equality* for closed PCF expressions of the same type where E_1 and E_2 are considered as observationally equal iff for all contexts $C[\]$ of base type **nat** it holds that

$$C[E_1] \Downarrow \underline{n} \iff C[E_2] \Downarrow \underline{n}$$

²For sake of completeness we will also present a “Small Step Semantics” for PCF as well as an abstract machine serving as an interpreter for PCF.

for all natural numbers $n \in \mathbb{N}$. Intuitively, expressions E_1 and E_2 are observationally equal iff the same observations can be made for E_1 and E_2 where an *observation* of E consists of observing that $C[E] \Downarrow n$ for some context $C[\]$ of base type **nat** and some natural number n . This notion of observation is a mathematical formalisation of the common practice of *testing of programs* and the resulting view that programs are considered as (observationally) equal iff they pass the same tests.

However, this notion of observational equality is not very easy to use as it involves quantification over all contexts and these form a collection which is not so easy to grasp. Accordingly there arises the desire for more convenient criteria sufficient for observational equality which, in particular, avoid any reference to (the somewhat complex) syntactic notions of evaluation relation and context.

For this purpose we introduce a so-called *Denotational Semantics* for PCF which assigns to every closed expression E of type σ an element $\llbracket E \rrbracket \in D_\sigma$, called the *denotation* or *meaning* or *semantics* of E , where D_σ is a previously defined structured set (called “semantic domain”) in which closed expressions of type σ will find their interpretation.

The idea of denotational semantics was introduced end of the 1960ies by Ch. Strachey and Dana S. Scott. Of course, there arises the question of what is the nature of the mathematical structure one should impose on semantical domains. Although the semantic domains which turn out as appropriate can be considered as particular topological spaces they are fairly different³ in flavour from the spaces arising in analysis or geometry. An appropriate notion of semantic domain was introduced by Dana S. Scott who also developed their basic mathematical theory to quite some extent of sophistication. From the early 1970ies onwards various research groups all over the world invested quite some energy into developing the theory of semantic domains—from now on simply referred to as *Domain Theory*—both from a purely mathematical point of view and from the point of view of Computer Science as (at least one) important theory of meaning (semantics) for programming languages.

Though discussed later into much greater detail we now give a preliminary account of how the domains D_σ are constructed in which closed terms of type σ find their denotation. For the type **nat** of natural numbers one puts $D_{\mathbf{nat}} = \mathbb{N} \cup \{\perp\}$ where \perp (called “bottom”) stands for the denotation

³In particular, as we shall see they will not satisfy Hausdorff’s separation property requiring that for distinct points x and y there are disjoint open sets U and V containing x and y , respectively.

of terms of type **nat** whose evaluation “diverges”, i.e. does not terminate. We think of $D_{\mathbf{nat}}$ as endowed with an “information ordering” \sqsubseteq w.r.t. which \perp is the least element and all other elements are incomparable. The types of PCF are built up from the base type **nat** by the binary type forming operator \rightarrow where $D_{\sigma \rightarrow \tau}$ is thought of as the type of (computable or continuous) functionals from D_σ to D_τ , i.e. $D_{\sigma \rightarrow \tau} \subseteq D_\tau^{D_\sigma} = \{f \mid f : D_\sigma \rightarrow D_\tau\}$. In particular, the domain $D_{\mathbf{nat} \rightarrow \mathbf{nat}}$ will consist of certain functions from $D_{\mathbf{nat}}$ to itself. It will turn out as appropriate to define $D_{\mathbf{nat} \rightarrow \mathbf{nat}}$ as consisting of those functions on $\mathbb{N} \cup \{\perp\}$ which are monotonic, i.e. preserve the information ordering \sqsubseteq . The clue of Domain Theory is that domains are not simply sets *but sets endowed with some additional structure* and $D_{\sigma \rightarrow \tau}$ will then accordingly consist of all *structure preserving* maps from D_σ to D_τ . However, for higher types (i.e. types of the form $\sigma \rightarrow \tau$ where σ is different from **nat**) it will turn out that it is not sufficient for maps in $D_{\sigma \rightarrow \tau}$ to preserve the information ordering \sqsubseteq . One has to require in addition some form of *continuity*⁴ which can be expressed as the requirement that certain suprema are preserved by the functions. The information ordering on $D_{\sigma \rightarrow \tau}$ will be defined pointwise, i.e. $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D_\sigma$.

Denotational semantics provides a purely *extensional* view of functional programs as closed expressions of type $\sigma \rightarrow \tau$ will be interpreted as particular *functions* from D_σ to D_τ which are considered as equal when they deliver the same result for all arguments. In other words the meaning of such a program is fully determined by its input/output behaviour. Thus, denotational semantics just captures *what* is computed by a function (its extensional aspect) and abstracts from *how* the function is computed (its intensional aspect as e.g. time or space complexity).

When a programming language like PCF comes endowed with an operational and a denotational semantics there arises the question how good they fit together. We will now discuss a sequence of criteria for “goodness of fit” of increasing strength.

Correctness

Closed expressions P and Q of type σ are called *semantically* or *denotationally equal* iff $\llbracket P \rrbracket = \llbracket Q \rrbracket \in D_\sigma$. We call the operational semantics *correct* w.r.t. the denotational one iff P and V are denotationally equal whenever $P \Downarrow V$, i.e. when evaluation preserves semantical equality. In particular for

⁴which is in accordance with the usual topological notion of continuity when the domains D_σ and D_τ are endowed with the so-called *Scott topology* which is defined in terms of the information ordering

programs, i.e. closed expressions P of base type **nat**, correctness ensures that $\llbracket P \rrbracket = n$ whenever $P \Downarrow n$, i.e. the operational semantics evaluates a program in case of termination to the number which is prescribed by the denotational semantics.

Completeness

On the other hand it is also desirable that if a program denotes n then the operational semantics evaluates program P to the numeral \underline{n} or, more formally, $P \Downarrow \underline{n}$ whenever $\llbracket P \rrbracket = n$ in which case we call the operational semantics *complete* w.r.t. the denotational semantics.

Computational Adequacy

In case the operational semantics is both correct and complete w.r.t. the denotational semantics, i.e.

$$P \Downarrow \underline{n} \iff \llbracket P \rrbracket = n$$

for all programs P and natural numbers n , we say that the denotational semantics is *computationally adequate*⁵ w.r.t. the operational semantics.

Computational adequacy is sort of a minimal requirement for the relation between operational and denotational semantics and holds for (almost) all examples considered in the literature. Nevertheless, we shall see later that the proof of computational adequacy does indeed require some mathematical sophistication.

If the denotational semantics is computationally adequate w.r.t. the operational semantics then closed expressions P and Q are observationally equal if and only if $\llbracket C[P] \rrbracket = \llbracket C[Q] \rrbracket$ for all contexts $C[\]$ of base type, i.e. observational equality can be reformulated without any reference to an operational semantics.

The denotational semantics considered in the sequel will be *compositional* in the sense that from $\llbracket P \rrbracket = \llbracket Q \rrbracket$ it follows that $\llbracket C[P] \rrbracket = \llbracket C[Q] \rrbracket$ for all contexts $C[\]$ (not only those of base type). Thus, for compositionally computationally adequate denotational semantics from $\llbracket P \rrbracket = \llbracket Q \rrbracket$ it follows that P and Q are observationally equal. Actually, this already entails

⁵One also might say that “the operational semantics is computationally adequate w.r.t. the denotational semantics” because the denotational semantics may be considered as conceptually prior to the operational semantics. One could enter an endless “philosophical” discussion on what comes first, the operational or the denotational semantics. The authors have a slight preference for the view that denotational semantics should be conceptually prior to operational semantics (the *What* comes before the *How*) being, however, aware of the fact that in practice operational semantics often comes before the denotational semantics.

completeness of the denotational semantics as if $\llbracket P \rrbracket = n = \llbracket n \rrbracket$ then P and n are observationally equal from which it follows that $P \Downarrow n \Leftrightarrow n \Downarrow n$ and, therefore, $P \Downarrow n$ as $n \Downarrow n$ does hold anyway. Thus, under the assumption of correctness for a compositional denotational semantics computational adequacy is equivalent to the requirement that denotational equality entails observational equality.

Full Abstraction

For those people who think that operational semantics is prior to denotational semantics the notion of observational equality is more basic than denotational equality because the former can be formulated without reference to denotational semantics. From this point of view computational adequacy is sort of a “correctness criterion” as it guarantees that semantic equality entails the “real” observational equality (besides the even more basic requirement that denotation is an invariant of evaluation).

However, one might also require that denotational semantics is complete w.r.t. operational semantics in the sense that observational equality entails denotational equality, in which case one says that the denotational semantics is *fully abstract* w.r.t. the operational semantics. At first sight this may seem a bit weird because in a sense denotational semantics is more abstract than operational semantics as due to its extensional character it abstracts from intensional aspects such as syntax. However, observational equivalence—though defined *a priori* in operational terms—is more abstract than denotational equality under the assumption of computational adequacy guaranteeing that denotational equality entails observational equality. Accordingly, a fully abstract semantics induces a notion of denotational equality which is “as abstract as reasonably possible” where “reasonable” here means that terms are not identified if they can be distinguished by observations.

Notice, moreover, that under the assumption of computational adequacy full abstraction can be formulated without reference to operational semantics as follows: closed expressions P and Q (of the same type) are denotationally equal already if $C[P]$ and $C[Q]$ are denotationally equal for all contexts $C[\]$ of base type. A denotational semantics satisfying this condition is fully abstract w.r.t. an operational semantics iff it is computationally adequate w.r.t. this operational semantics.

Whereas computational adequacy holds for almost all models of PCF this is not the case for full abstraction as exemplified by the (otherwise sort of canonical) Scott model. Though the Scott model (and, actually, also

all other models considered in the literature) is fully abstract for closed expressions of first order types $\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \dots \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}$ full abstraction fails already for the second order type $(\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}) \rightarrow \mathbf{nat}$.

However, the Scott model is fully abstract for an extension of PCF by a *parallel*, though deterministic, language construct $\mathbf{por} : \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}$, called “parallel or”, which gives 0 as result if its first or its second argument equals 0, 1 if both arguments equal 1 and delivers \perp as result in all other cases. This example illustrates quite forcefully the *relativity* of the notion of full abstraction w.r.t. the language under consideration. The only reason why the Scott model fails to be fully abstract w.r.t. PCF is that it distinguishes closed expressions E_1 and E_2 of the type $(\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}) \rightarrow \mathbf{nat}$ although these cannot be distinguished by program contexts $C[\]$ expressible in the language of PCF. However, E_1 and E_2 can be distinguished by the context $[\](\mathbf{por})$. In other words whether a denotational semantics is fully abstract for a language strongly depends on the expressiveness of this very language. Accordingly, a lack of full abstraction can be repaired in two possible, but different ways

- (1) keep the model under consideration but extend the language in a way such that the extension can be interpreted in the given model and denotationally different terms can be separated by program contexts expressible in the extended language (e.g. keep the Scott model but extend PCF by \mathbf{por}) *or*
- (2) keep the language and alter the model to one which is fully abstract for the given language.

Whether one prefers (1) or (2) depends on whether one gives preference to the model or to the syntax, i.e. the language under consideration. A mathematician’s typical attitude would be (1), i.e. to extend the language in a way that it can grasp more aspects of the model, simply because he is interested in the structure and the language is only a secondary means for communication. However, (even) a (theoretical) computer scientist’s attitude is more reflected by (2) because for him the language under consideration is the primary concern whereas the model is just regarded as a tool for analyzing the language. Of course, one could now enter an endless discussion on which attitude is the more correct or more adequate one. The authors’ opinion rather is that each single attitude when taken absolutely is somewhat disputable as (i) why shouldn’t one take into account various different models instead of stubbornly insisting on a particular “pet model” and (ii) why should one take the language under consideration as

absolute because even if one wants to exclude `por` for reasons of efficiency why shouldn't one allow⁶ the observer to use it?

Instead of giving a preference to (1) or (2) we will present both approaches. We will show that extending PCF by `por` will render the Scott model fully abstract and we will present a refinement of the Scott model, the so-called *sequential domains*, giving rise to a fully abstract model for PCF which we consider as a final solution to a—or possibly *the*—most influential open problem in semantic research in the period 1975–2000. The solution via sequential domains is mainly known under the name “relational approach” because domains are endowed with (a lot of) additional relational structure which functions between sequential domains are required to preserve in addition to the usual continuity requirements of Scott's Domain Theory.

A competing and, actually, more influential approach is via *game semantics* where types are interpreted as games and programs as strategies. However, this kind of models is never extensional and, accordingly, not fully abstract for PCF as by Milner's Context lemma extensional equality entails observational equality. However, the “extensional collapse” of games models turns out as fully abstract for PCF. But this also holds for the term model of PCF and in this respect the game semantic approach cannot really be considered as a genuine solution of the full abstraction problem at least according to its traditional understanding. However, certain variations of game semantics are most appropriate for constructing fully abstract models for non-functional extensions of PCF, e.g. by control operators or references, as for such extensions the term models obtained by factorisation w.r.t. observational equivalence are *not extensional* anymore and, therefore, the inherently extensional approach via domains is not applicable anymore.

Notice that there is also a more liberal notion of sequentiality, namely the *strongly stable domains* of T. Ehrhard and A. Bucciarelli where, however, the ordering on function spaces is not pointwise anymore.

Universality

In the Scott model one can distinguish for every type σ a subset $C_\sigma \subseteq D_\sigma$ of *computable* elements without any reference to PCF-definability such that all PCF-definable elements of D_σ are already contained in C_σ . Now, if one has fixed such a semantic notion of computability for a model then there arises the question whether *all computable elements of the model do*

⁶as for example in cryptology where the attacker is usually assumed to employ as strong weapons as possible

arise as denotations of closed PCF terms in which case the model is called *universal*.⁷

A language universal for the Scott model can be obtained from PCF by adding `por` (“parallel or”) and Plotkin’s *continuous existential quantifier* \exists of type $(\mathbf{nat} \rightarrow \mathbf{nat}) \rightarrow \mathbf{nat}$ which is defined as follows: $\exists(f) = 0$ if $f(n) = 0$ for some $n \in \mathbb{N}$, $\exists(f) = 1$ if $f(\perp) = 1$ and $\exists(f) = \perp$ in all other cases.

Notice, however, that \exists cannot be implemented within PCF+`por` from which it follows that universality is a stronger requirement than full abstraction. But universality entails full abstraction as there is a theorem saying that a model of PCF is fully abstract iff all its “finite” elements are PCF definable and as these “finite” elements are subsumed by any reasonable notion of computability.

We conclude this introductory chapter by discussing the relevance of denotational semantics for **logics of programs**, i.e. calculi where properties of programs can be expressed and verified.

First of all denotational models of programming languages are needed for defining validity of assertions about programs as can be expressed in a logic for this programming language. In case of PCF the family $(D_\sigma)_{\sigma \in \text{Type}}$ provides the carriers for a many-sorted structure in which one can interpret the terms of the program logic LCF (Logic of Computable Functionals)⁸ whose terms are expressions of the programming language PCF and whose formulas are constructed via the connectives and quantifiers of first order logic from atomic formulas $t_1 \sqsubseteq t_2$ stating that the meaning of t_1 is below the meaning of t_2 w.r.t. the information ordering as given by the denotational model. Notice, however, that the term language PCF is not first order as it contains a binding operator λ needed for explicit definitions of functions. However, this does not cause any problems for the interpretation of LCF. Instead of first order logic one might equally well consider higher

⁷Calling this property “universal” is in accordance with the common terminology where a programming language L is called “Turing universal” iff all partial recursive functions on \mathbb{N} can be implemented by programs of L . The property “universal” as defined above is stronger since it requires that computable elements of *all* types can be implemented within the language under consideration. But in both cases “universal” means that one has already got an implementation for all possible computable elements (of a certain kind).

⁸The calculus LCF was introduced by D. Scott in an unpublished, but widely circulated and most influential manuscript dating back to 1967. In the 1970ies a proof assistant for LCF was implemented by R. Milner who for this very purpose developed and implemented the functional programming language ML (standing for “Meta-Language”) whose refined versions SML and OCAML today constitute the most prominent typed call-by-value functional programming languages.

order logic over a model of PCF which has the advantage that higher order logic allows one to express inductively defined predicates which are most useful for the purposes of program verification.

In principle one could interpret LCF also in the structure obtained by factorizing the closed PCF terms modulo observational equality. However, such a structure is not very easy to analyze as it is too concrete. Denotational models have the advantage that simple and strong proof principles like *fixpoint induction*, *computational induction* and *Park induction*, which are indispensable for reasoning about recursively defined functions and objects, can be easily verified for these models as they are actually derived from some obvious properties of these models.