06-02552 *Princ. of Progr. Languages (and "Extended")*
Spring Semester 2019-20

The University of Birmingham
School of Computer Science

Handout 3: Extensional Functions and Types

In this handout, we are interested in understanding the concept of a "function".

1 Function as the "net effect"

In Computer Science, we tend to use the term "function" to mean a piece of code with a name and perhaps some parameters. This kind of thing is more properly called a **procedure**, a term that was used in the 1960s but has now fallen out of use. Mathematicians also call it the **rule** of the function. Another term used is **intensional function**..[1]

The term "function" was coined in mathematics in 17th and 18th centuries to mean the *net effect* of procedures or formulas.[2] We can write different procedures that behave the same way. For a simple example, consider:

$$\begin{aligned} f(x) &= x^2 - 4 \\ f'(x) &= (x+2) \cdot (x-2) \end{aligned}$$

These two "functions" use different primitive operations in different orders. Thus, they represent *different procedures*. However, they always have the same effect. Given any numerical value for x, both $f(x)$ and $f'(x)$ always give the same result. So, then, why should they be considered as different "functions?"

The same issue arises also for formulas describing numbers. For instance, the two formulas:

$$\begin{aligned} k &= 57 - 8 \\ k' &= 7^2 \end{aligned}$$

have exactly the same value, *viz.*, 49. We normally say that k and k' are "equal," meaning that they stand for the *same number*. Similarly, if f and f' are two "functions" that always give the same results, then they should be regarded as *same function*. That means that the idea of a "function" as an algorithm is inadequate. The idea of "sameness" for functions is different from the idea of "sameness" for algorithms.

Mathematicians grappled with this problem beginning in the 17th century, after the development of *analytical geometry*, which sets up a correspondence between algebra and geometry, allowing us to convert algebraic phenomena to geometric phenomena and *vice versa*. Using this correspondence, we can draw *graphs* for functions, putting the arguments (x) on one axis and the results (y) on another axis. Once we begin to draw graphs, we can visualize the input and output correspondences made by functions all at one go. We can notice that the functions f and f' mentioned above have exactly the *same graph*. This gave rise to the idea of functions as **graphs**, also called **extensional functions**.[3]

To describe the extensional function corresponding to f or f' *directly*, *i.e.*, without any reference to their internal primitive operations, we must focus on the *inputs* and *outputs* of the functions. We can, for instance, write out a table of the inputs and outputs as follows:

x	$f(x)$
0	-4
1	-3
2	0
3	5
$\vdots$	$\vdots$

(1)

[1]The term "intension" in this context means the way something is expressed, here by a rule of computation or piece of code.
[2]There is a Wikipedia article on the "History of the function concept", which you can read for the historical background.
[3]The term "extension" (or "extent") in this context means the collection of all input-output pairs that make up the function.

(Here, we have only considered integer inputs for f but the same idea applies for other kinds of numbers too.) The table for the function is necessarily *infinite*. We can never finish writing it out. However, we can imagine that *in principle* we can write out such a table. The *existence* of the table is all that matters. Such tables are referred to as **mappings**. (The "`Map`" classes of the Java collections library represent exactly this concept. They are extensional functions in a tabular form.)

What kind of an input-output table is a valid function? Note that (intensional) functions produce their outputs by applying the primitive operations in some specific order. So, for any *particular* value of x, they will always give *the same value* of $f(x)$. That means that the table for a function should have exactly one entry for every value of x. If we have two different entries for the same value of x, e.g., mapping 1 to -3 as well as 1 to 3, then it would not be a valid table for a function.

2 Set theory

The problem of formalizing extensional functions gave rise to set theory. Originally, the term "class" was used to refer to the concept. It was later replaced by the more refined term "set." Set theory provides the first example of subject we call **programming language semantics**. The language in question is that of functions defined by algebraic formulas. Set theory provides its "semantics," *i.e.*, gives an abstract characterization of what the algebraic definitions "mean."

Domain and codomain. The first application of sets is to specify the "domain" and "codomain" of functions. An extensional function must have the property that for *every* input value there is a *unique* output value. But, what can we possibly mean by "every" input value? To make sense of this, we must have a way to specify the range of input values there can be, *i.e.*, the collection of possible input values the function can accept. This is specified as a *set*. The set of possible input values is called the **domain** of the function. The possible output values produced by the function should again be a *set*. This is called the **codomain** of the function. We specify the function, its domain as well as its codomain by putting them in the notation "$f : A \to B$." Here, A and B are sets, which form the domain and the codomain of the function f. Informally, we read the statement as "f is a function from A to B."

Composition of extensional functions. If $f : A \to B$ and $g : B \to C$ are functions, where the codomain of the first function and the domain of the second function are *identical* then, and only then, we can *compose* the two functions. The composite function $g \circ f$ is of type $A \to C$. If we write out an input-output table such as (1) for the composite function $g \circ f$, it would show how the A-typed inputs are mapped to the C-typed outputs. The intermediate values (of type B) are gone. They play no role in the table for the extensional function $g \circ f$. This is the essence of extensional functions. They describe *what* the functions do. They have no concern with *how* the functions do it.

Two constructions on sets. Set theory goes further in formalizing extensional functions, by describing the functions themselves as *sets*. This is done using two constructions on sets:

1. If A and B are sets, their **cross product**, denoted $A \times B$ is a set whose elements are *pairs* of the form (x, y), where x is an element of A and y is an element of B. One might also write informally:

   $$A \times B = \{\, (x, y) \mid x \in A,\ y \in B \,\}$$

   For example, if A is the set $\{0, 1\}$ and B is the set $\{a, b\}$, then their cross product $A \times B$ is the set $\{(0, a),\ (0, b),\ (1, a),\ (1, b)\}$.

2. There is an obvious notion of a *subset* of a set. A set S is said to be a subset of another set A if every element of S is also an element of A. We write $S \subseteq A$ in this situation. For example, $\{0\} \subseteq \{0, 1\}$. All the subsets of a given set A can be put into another set, which is called the

powerset of A, denoted $\mathcal{P}(A)$. For example, if A is the set $\{0, 1, 2\}$, then the powerset $\mathcal{P}(A)$ contains the sets:

$$\{\},\ \{0\},\ \{1\},\ \{2\},\ \{0,1\},\ \{0,2\},\ \{1,2\},\ \{0,1,2\}$$

Functions as sets, functions as elements. Using these concepts, a function table such as (1) can be formalized as a set. First, the table can be seen as a *set* of its rows. The order in which the rows are listed does not matter. Secondly, every row of the table is a *pair* (x, y) where x is a possible input to the function and y is the corresponding output of the function. In the set-theoretic notation, the mapping table (1) can be expressed as follows:

$$f = \{(0, -4),\ (1, -3),\ (2, 0),\ (3, 5), \ldots\}$$

Therefore, a function $f : A \to B$ is a set of pairs, each pair being of type $A \times B$. The function itself is now a subset $f \subseteq A \times B$. Being a subset, it is also an *element of the powerset* $\mathcal{P}(A \times B)$.

However, a function is not any arbitrary element of $\mathcal{P}(A \times B)$. It has to be "functional," *i.e.*, for every possible input value there should be a unique output value. We can write this condition as a formula:

$$f \text{ is functional} \iff \forall x \in A.\, \exists \text{unique } y \in B.\, (x, y) \in f \tag{2}$$

So, a function of type $A \to B$ is an element of $\mathcal{P}(A \times B)$ satisfying the condition (2). Note that the function f displayed above is functional, *i.e.*, specifies a unique output for every input.

Functions as special relations. A closely related concept to function is that of **relation**, more specifically "binary" relation. A binary relation R between sets A and B is any arbitrary subset of $A \times B$. (No conditions attached.) For example $<$ is a binary relation between natural numbers. It includes such pairs as $(0, 1), (0, 2), (1, 2), \ldots$. We use the notation $R(x, y)$ to state that $x \in A$ and $y \in B$ are related by the relation R. We might also right xRy or $x[R]y$, depending on the context. Relations can be described writing logical formulas:

$$R(x, y) \iff \cdots$$

Now, functions from A to B can be seen as special kinds of binary relations beteen A and B. They are the relations that satisfy the "functional" condition (2). Note that $<$ is not a function, because 0 is related to many elements $(1, 2, \ldots)$. A function must relate an input value x to a *unique* output value y.

The fact that functions are special kinds of relations allows us to *specify* the effect of functions by describe the relations between the inputs and outputs. For instance, we might write, for a function f from natural numbers to natural numbers:

$$f(x) = y \implies x^2 = y + 4$$

(Can you tell what the function f is?) This means that the function f regarded as a set of pairs in $A \times B$ is a *subset* of the specified relation R. Or, more simply, $f \subseteq R$.

So a **specification** of a function is in general a relation.

Composition of extensional functions. Let us use this definition of functions to work out what composition means. If $f : A \to B$ is a function, it has pairs of the form (x, y), and, if $g : B \to C$ is a function, it has pairs of the form (y, z). The function $g \circ f : A \to C$ now has a pair (x, z) precisely when there is a pair (x, y) in f and a pair (y, z) in g, for some $y \in B$. Writing it out as a set-theoretic definition, we have:

$$g \circ f = \{\, (x, z) \mid \exists y {\in} B.\, (x, y) \in f,\ (y, z) \in g \,\}$$

Note that the intermediate value y does not appear in the mapping of $g \circ f$. Only x and z appear. This is the *essence* of extensional functions: *Only the inputs and outputs of functions are captured in the mappings. The intermediate computations are dissolved.*

Can we argue that this set is "functional," *i.e.*, satisfies the condition (2)? Indeed we can. Suppose x is an arbitrary element of A. Since f is a function, there is a unique $y \in B$ such that $(x, y) \in f$. Since g is a function, there is a unique $z \in C$ such that $(y, z) \in g$. Thus, there will be a unique z such that $(x, z) \in g \circ f$.

Function space. We can now see that the functions from A to B in turn form a set, which we call a **function space**, and denote it by $(A \to B)$.

$$(A \to B) \;=\; \{\, f \in \mathcal{P}(A \times B) \mid f \text{ is functional} \,\}$$

A function $f \in (A \to B)$ has exactly one pair of the form (x, y) for any given value $x \in A$. The corresponding y value for the input x is called the **image** of x under f. We denote it simply as "$f(x)$."

An example isomorphism. We work through an example isomorphism involving function spaces to illustrate the idea of function spaces as sets:

$$\phi \quad : \quad (A \to B \times C) \;\cong\; (A \to B) \times (A \to C) \quad : \quad \psi \tag{3}$$

To show that the two sets are isomorphic, we need to provide a function from left to right (which we denote ϕ) and a function from right to left (which we denote ψ), and show that $\phi \circ \psi$ as well as $\psi \circ \phi$ are both identities.

The function ϕ can be defined as follows: Given a function $f \in (A \to B \times C)$, we need to produce two separate functions of types $(A \to B)$ and $(A \to C)$. They can be produced by looking through the mapping table of the function f:

$$\begin{aligned} \phi(f) = \quad & (\{\, (x, y) \mid \exists z.\, (x, (y, z)) \in f \,\}, \\ & \;\{\, (x, z) \mid \exists y.\, (x, (y, z)) \in f \,\}) \end{aligned} \tag{4}$$

The definition says that, if f maps x to a pair (y, z) then the first function maps x to y, and the second function maps x to z.

Pause: In the description of the first set, why did we include an existential quantifier ($\exists$) for z but not for x and y? What would happen if we did not include this existential quantifier at all?

In the reverse direction, the function ψ does the opposite. Given functions $g \in (A \to B)$ and $h \in (A \to C)$, it combines their mappings to produce a pair of results for each $x \in A$:

$$\psi(g, h) = \{\, (x, (y, z)) \mid (x, y) \in g,\; (x, z) \in h \,\} \tag{5}$$

To prove that $\psi \circ \phi = \textbf{id}$, we argue as follows. Consider an arbitrary function $f \in (A \to B \times C)$. Let $\phi(f) = (g, h)$, *i.e.*, use the names g and h for the two functions produced by ϕ. Then we need to show that $\psi(g, h) = f$ again. We do this in two steps: $f \subseteq \psi(\phi(f))$ and $\psi(\phi(f)) \subseteq f$. For the first step, we argue as follows:

1. Suppose $(x, (y, z))$ is a mapping pair in f.
2. Then, by (4), (x, y) is a mapping pair in g and (x, z) is a mapping pair in h.
3. By (5), $(x, (y, z))$ is a mapping pair in $\psi(g, h)$.

(We are using the term "mapping pair" just to avoid confusion with the pairs in $B \times C$. But the "mapping pairs" are just pairs.) Thus we have shown that $f \subseteq \psi(g, h)$ where $(g, h) = \phi(f)$.

For the second step $\psi(g, h) \subseteq f$, we argue as follows:

1. Suppose $(x, (y, z))$ is a mapping pair in $\psi(g, h)$.

2. Then, by (5), $(x, y) \in g$ and $(x, z) \in h$.
3. By (4), (x, y) can be in g only if, for some z', there is a mapping pair $(x, (y, z'))$ in f. Similarly, (x, z) can be in h only if, for some y', there is a mapping pair $(x, (y', z))$ in f. Since f is a *functional* mapping, the mapping pair of the form $(x, (y, z))$ in both the cases must be the same. In other words, $y' = y$ and $z' = z$.

Thus, we have shown that the original mapping pair $(x, (y, z))$ of $\psi(g, h)$ is also in f, *i.e.*, $\psi(g, h) \subseteq f$.

We can write the whole proof compactly as follows:

$$\begin{array}{rcl}
\psi(\phi(f)) & = & \psi(g,h) \text{ where } g = \{\,(x,y) \mid \exists z.\,(x,(y,z)) \in f\,\} \text{ and } h = \{\,(x,z) \mid \exists y.\,(x,(y,z)) \in f\,\})) \\
& = & \{\,(x,(y,z)) \mid (x,y) \in g,\ (x,z) \in h\,\} \text{ where } g = \ldots \text{ and } h = \ldots \\
& = & \{\,(x,(y,z)) \mid \exists y', z'.\,(x,(y,z')) \in f,\ (x,(y',z)) \in f\,\} \\
& = & \{\,(x,(y,z)) \mid (x,(y,z)) \in f\,\} \\
& = & f
\end{array}$$

The verification of $\phi \circ \psi = \mathbf{id}$ is similar and left to the reader.

3 Church's lambda notation

In the 1930's, Alonzo Church (who is famous for the "Church's thesis" on computable functions) set out to formulate a calculus for functions which is more convenient than the set-theoretic notation. Instead of writing

$$\{\,(x, y) \mid x \in A,\ y = E(x)\,\}$$

Church proposed to write:

$$\lambda x \in A.\, E(x)$$

We can read $\lambda x \in A.\, E(x)$ in very computational terms: it is *the function that, given an argument x of type A, produces $E(x)$ as its result.* When the type A is clear from the context, we can also abbreviate it to $\lambda x.\, E(x)$.

This is not only a notational simplification. It also *limits* the kind of functions that we can express. In the set-theoretic notation, we can define a function by stating a property of the output instead of giving a formula for computing it. For example, consider:

$$\{\,(x, y) \mid x \in A,\ y \in B,\ P(x, y)\,\} \tag{6}$$

where $P(x, y)$ is some predicate involving x and y. For this to be a well-defined function, we would also need to show that the condition of functional mappings (2) is satisfied. However, we do not need to give a formula $E(x)$ for computing the result y. Such functions will be in general *non-computable*. For example, we can define a function for the halting problem. Let **halts**(x, y) stand for the predicate that means that the Turing machine x halts when run with an input tape y. Then, the function:

$$H \;=\; \{\,(x, b) \mid b = \text{true} \iff \forall y.\, \textbf{halts}(x, y)\,\}$$

is a perfectly good set-theoretic function. However, it is non-computable. There is no formula $E(x)$ that can describe the valid output $H(x)$ for all Turing machines x.

In contrast, in Church's notation, functions are described by formulas. So, only *computable functions* can be expressed. Thus, Church's notation serves well for Computer Science, though it is not adequate for the concerns of mathematics. In Computer Science, we call a set-theoretic definition of a function such as (6) a **specification**. It fully and accurately describes the *desire* for a particular program. However, to describe the program itself, we need to come up with a **program**, i.e., a formula or algorithm for computation.

Example isomorphism revisited. To get insight into how Church's notation works, let us rework the isomorphism (3) using his lambda notation. The two functions

$$\begin{array}{rcl} \phi & : & (A \to B \times C) \longrightarrow (A \to B) \times (A \to C) \\ \psi & : & (A \to B) \times (A \to C) \longrightarrow (A \to B \times C) \end{array}$$

involved in the isomorphism can be expressed as follows:

$$\begin{array}{rcl} \phi(f) & = & (\lambda x \in A.\, \mathbf{fst}(f(x)),\ \lambda x \in A.\, \mathbf{snd}(f(x))) \\ \psi(g,h) & = & \lambda x \in A.\, (g(x), h(x)) \end{array}$$

where **fst** $: B \times C \to B$ and **snd** $: B \times C \to C$ are functions for extracting the first and second components of a pair: $\mathbf{fst}(y,z) = y$ and $\mathbf{snd}(y,z) = z$.

To prove $\psi \circ \phi = \mathbf{id}$, we argue as follows:

$$\begin{array}{rcl} \psi(\phi(f)) & = & \psi(\lambda x.\, \mathbf{fst}(f(x)),\ \lambda x.\, \mathbf{snd}(f(x))) \\ & = & \lambda x.\, (\mathbf{fst}(f(x)),\ \mathbf{snd}(f(x))) \\ & = & \lambda x.\, f(x) \\ & = & f \end{array}$$

In the first step, we just used the definition of $\phi(f)$. In the second step, we used the definition of $\psi(g, h)$ where $g = \lambda x.\, \mathbf{fst}(f(x))$ and $h = \lambda x.\, \mathbf{snd}(f(x))$. Note that $g(x)$ is now nothing but $\mathbf{fst}(f(x))$, and $h(x)$ is nothing but $\mathbf{snd}(f(x))$. In the third step, we used an inherent property of the functions **fst** and **snd**:

$$(\mathbf{fst}(y,z),\ \mathbf{snd}(y,z)) = (y,z) \tag{7}$$

In the last step, we used an inherent property of the lambda notation:

$$\lambda x.\, f(x) \ = \ f \tag{8}$$

We can prove the reverse direction $\phi \circ \psi = \mathbf{id}$ in an equally straightforward way:

$$\begin{array}{rcl} \phi(\psi(g,h)) & = & \phi(\lambda x.\, (g(x), h(x))) \\ & = & (\lambda x.\, \mathbf{fst}(g(x), h(x)),\ \lambda x.\, \mathbf{snd}(g(x), h(x))) \\ & = & (\lambda x.\, g(x),\ \lambda x.\, h(x)) \\ & = & (g, h) \end{array}$$

In the third step, we have used the definitions of the functions **fst** and **snd** to simplify the expressions. In the last step, we used the property (8) twice, for g and h respectively.

Working with anonymous functions. The normal notation for functions involves giving them names. We say something like "let $f : A \to B$ be a function given by $f(x) = E(x)$" where $E(x)$ is a formula involving the variable x. Church's lambda notation allows us to express the same function as $\lambda x \in A.\, E(x)$ *without giving the function a name*. In Computer Science, we tend to call such functions expressed using the lambda notation **anonymous functions**. Despite the change of notation, the functions still have exactly the same meaning as the traditional notation. We can *apply* an anonymous function to an argument a by writing simply

$$(\lambda x \in A.\, E(x))\,(a)$$

Here, the anonymous function $(\lambda x \in A.\, E(x))$ is applied to the argument a. The result will be of course $E(a)$, *i.e.*, the value of the formula $E(x)$ where we replace the symbol x by a. We write this as an equation:

$$(\lambda x \in A.\, E(x))\,(a) \ = \ E(a) \tag{9}$$

and call it the β**-equivalence** of anonymous functions. If F is a term that represents a function of type $A \to B$, then the anonymous function $\lambda x \in A.\, F(x)$ behaves the same way as the term F. If we apply either of these functions to an argument a, we get $F(a)$. This justifies the equation:

$$(\lambda x \in A.\, F(x)) \;=\; F \tag{10}$$

This is called the η**-equivalence** of anonymous functions. Note that we have already used a special case of this equivalence in the equation (8).

4 Polymorphism

The history of mathematics shows that "functions," which started out being intensional objects (formulas or programs), transitioned into being extensional objects (mappings). Most mathematicians would agree that this transition has been for the good. Extensional functions caputre the input-output behaviour and abstract away from the irrelevant details of the internal calculations (if at all we have any such calculations). We have mentioned that mathematicians also care about extensional functions which are not computable and, so, cannot have inensional representations at all.

From the Computer Science point of view, however, this "goodness" of extensional functions is open to question. The problem is that extensional functions deal with "polymorphism" awkwardly.

A **polymorphic function** (or **generic function**) is an intensional function that can be given many types. For example, the function **fst** given by the rule:

$$\textbf{fst}(y, z) = y \tag{11}$$

selects the first component of a pair. It does this no matter what the types of the two components are. So, the **fst** function has the type $B \times C \to B$ *for every type* B *and every type* C.

Extensional functions have a fixed domain and codomain. They are mappings that have unique pairs of the form (x, y) for every element x of the *domain* and some element y of the *codomain*. Without fixing the domain and codomain, it is not possible to talk about extensional functions. Thus, the function **fst** given by (11) is not an extensional function.

For every selection of types B and C, we can define an extensional function $\textbf{fst}_{B,C} : B \times C \to B$, given by the rule

$$\textbf{fst}_{B,C}(y, z) \;=\; y$$

Note that there are an infinite number of such extensional functions, one for each combination of types B and C! For example, there is a **fst** function for the combination $B = \textbf{Int}$ and $C = \textbf{Int}$, another for $B = \textbf{Bool}$ and $C = \textbf{Bool}$, yet another for $B = \textbf{Int}$ and $C = \textbf{Bool}$. and so on. Computer Scientists find this kind of proliferation of extensional functions curious and amusing, not to say pointless. Intensional functions such as those given by (11) seem perfectly adequate to them.

The proliferation of extensional polymorphic functions also seems to be technically wrong. If we ask a set-theorist how many polymorphic functions of type $B \times C \to B$ are there, he/she has to say that there an infinite number of them. First of all, there are an infinite number of types one can choose for B and C. Morever, even for a fixed choice, say $B = \textbf{Int}$ and $C = \textbf{Int}$, there are an infinite number of functions of type $\textbf{Int} \times \textbf{Int} \to \textbf{Int}$. So, the number of possible polymorphic functions of type $B \times C \to B$ is definitely infinite.

On the other hand, there is *exactly one* intensional polymorphic function of this type, *viz.*, the one defined in (11). We can argue intuitively as follows. The function **fst** is given a pair (y, z) of type $B \times C$ where B and C are both *unknown types*. It is asked to produce a result of type B. Since B is an unknown type, the only knowledge the **fst** function has about B is the fact it has an element y, which has been provided as the first component of the argument. All that it can do is to return this y as the result. *There is no other choice*!

How can we combine the convenience of intensional functions for polymorphism with the behavioral elegance of extensional functions? John Reynolds (1935-2013) produced a theory called **relational parametricity** which brings the theory of extensional functions in line with what is intensionally possible. According to him, polymorphic functions like $\mathbf{fst}_{B,C} : B \times C \to B$ cannot do arbitrary things for different instances of types B and C. They must all act the "same way." His theory says that the different instances of the **fst** function, say $\mathbf{fst}_{B_1,C_1}$ and $\mathbf{fst}_{B_2,C_2}$, must preserve *all possible relationships* between the different type instantiations. This means that, if $R \subseteq B_1 \times B_2$ and $S \subseteq C_1 \times C_2$ are two arbitrary relations, then the corresponding **fst** functions must satisfy the following property:

$$(y_1, y_2) \in R \wedge (z_1, z_2) \in S \Longrightarrow (\mathbf{fst}_{B_1,C_1}(y_1, z_1), \mathbf{fst}_{B_2,C_2}(y_2, z_2)) \in R$$

Using this principle, it is possible to prove that there is exactly one relationally parametric family of extensional **fst** functions, which corresponds to what we know about the possibility of intensional functions.

We will discuss the theory of relational parametricity in further detail when we consider the issues of data abstraction and information hiding.