UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Applied Mathematics

Mathematical models of denotational semantics

Master's Thesis

Master's Degree in Mathematical Research

AUTHOR: Clement, Collin

Tutor: Rodríguez López, Jesús

ACADEMIC YEAR: 2023/2024

TRABAJO FINAL DE MÁSTER

MATHEMATICAL MODELS FOR DENOTATIONAL SEMANTICS

AUTOR : CLÉMENT COLLIN

TUTOR : JESÚS RODRÍGUEZ LÓPEZ

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN MATEMÁTICA
2023/2024

Contents

Chapter 1

Semantics of programming languages

1.1 Introduction

A programming language is a system of notation for writing computer programs. It is built from a *formal grammar* to which *semantic rules* are associated. In other words, a programming language is defined on the basis of two components :

Syntax

It lays down the rules that determine which combinations of symbols are correct expressions in the language. As Slonneger and Kurtz assert [24, Chapter 1], « syntax solely deals with the form and structure of symbols in a language without any consideration given to their meaning ». The syntax of a programming language is often defined using the BNF metalanguage introduced by John Backus in 1959, during the development of Algol 58, and later improved by Peter Naur [5].

Semantics

It assigns computational meaning to valid (that is, syntactically correct) phrases by establishing a systematic relationship between the inputs and the outputs of the programs written in the language.

Although both the above aspects are fundamental for a deep understanding of a programming language, in this work, our main interest is semantics.

Assigning meaning to expressions written in a programming language is fundamental for its design. In most programming languages, the meaning of commands is quite self-explanatory as we borrow words from natural language. For example, the command

```
    if A then B else C
```

means : « if expression A is true, then execute B and if A is false, then execute C » as one would naturally expect. Unfortunately, this "natural translation" is not always accurate, as it may encounter certain ambiguities. One may also want to explain how the programs written in a given programming language behave in concrete terms, prove their correctness, compare languages with each other, etc. However, this can't be done without a theory explaining how the language is interpreted by the computer. For all the reasons above, having at hand a formal definition of all the expressions in a programming language is essential. It provides an unambiguous description of their effects, a tool for design and analysis, and a guide to a program that produces the desired results. Such formal definition is brought by *semantic models*, i.e., descriptions of the semantics of the language using standardized, formal, and abstract terminology.

One could think that the semantics of a programming language depends on a particular compiler (which unequivocally translates programs into source code, itself converted into a series of physical operations). But, as Scott asserts [22], « this idea is wrong since the same language can have many different compilers » and they must always produce the same results (see also [13]).

The intent to come up with mathematical models for the semantics of a programming language, i.e., mathematical objects that « serve as a basis for understanding and reasoning about how programs behave [...] and useful for various kinds of analysis and verification » [29], is known as formal semantics, which has its roots in Floyd's work [11]. There are several approaches to formal semantics in the literature. We describe the most important ones in the following section.

1.2 Formal Semantics

In an article entitled *Assigning meanings to programs* [11] published in 1967, Robert W. Floyd defines the role of semantic models as being precisely to offer « a rigorous standard for proofs about computer programs, including proofs of correctness, equivalence, and termination ». Floyd's article gave rise to the now called axiomatic semantics, one of the main three approaches to formal semantics that flourished in the 1970s, the other two being operational and denotational semantics. These three visions are complementary : each of them has a different purpose. Quoting Winskel [29] :

> *A clear operational semantics is very helpful in implementation. Axiomatic semantics for special kinds of languages can give strikingly elegant proof systems, useful in developing as well as verifying programs. Denotational semantics provides the deepest and most widely applicable techniques, underpinned by a rich mathematical theory. Indeed, the different styles of semantics are highly dependent on each other.*
>
> *For example, showing that the proof rules of an axiomatic semantics are correct relies on an underlying denotational or operational semantics. To show an implementation correct, as judged against a denotational semantics, requires a proof that the operational and denotational semantics agree. And, in arguing about an operational semantics it can be an enormous help to use a denotational semantics, which often has the advantage of abstracting away from unimportant, implementation details, as well as providing higher-level concepts with which to understand computational behaviour.*

We now briefly describe these three different approaches to formal semantics ([13, 24, 29]).

1.2.1 Axiomatic semantics

Axiomatic semantics is a slight improvement on Hoare's logic. Just like it, it is a formal system with a set of logical rules for studying the correctness of computer programs. It was proposed by Tony Hoare in 1969 [15] and based on Floyd's work on flowcharts [11]. Its fundamental idea is to interpret program executions as *partial correction assertion triples* (also known as *Hoare triples*) of the form :

$$\{A\}\, c\, \{B\}$$

where A and B are predicate logic assertions, respectively called precondition and postcondition, and c is a program. This notation means that if A is true before the execution of c, and c terminates, then B is true. In order to be able to prove program properties, Hoare established several axioms about how programs modify logical predicates. To illustrate his method, we consider the example he himslef provided (cf. [15]) of an easy program that uses the method of successive subtraction for finding the quotient q and remainder r obtained by dividing x by y, which, in pseudo-code, can be written :

```
r := x;
q := 0;
while y <= r do:
    r := r-y;
    q := q+1;
```

Let $A = \{r = x \wedge q = 0\}$ and $B = \{(x = y \cdot q + r) \wedge (r < y)\}$ be our pre and postcondition. A necessary condition for the correctness of this program is that whenever A is true, B is true. The curious reader can see how to prove this using the axioms and rules proposed by Hoare, in his article *An axiomatic basis for computer programming* [15].

Some limitations of this technique are mentioned by Hoare himself : a formal proof can be excessively tedious to write, the axioms and rules quoted above give no basis for a proof that a program actually terminates, etc.

We don't give more details about this aspect of semantics, but more information can be found in [3, 24, 29].

1.2.2 Operational semantics

Operational semantics can be seen as a mathematical interpreter of programs written in a given language, describing how a computation is actually performed, that is, converted into physical operations. This is particularly important when writing compilers and language interpreters.

We only briefly describe the so-called *structural operational semantics* introduced by Plotkin [18], in which evaluation and execution relations are specified by rules in a syntax-oriented way.

It is based on the following idea. When a program is executed, the memory changes from an initial state to a final state, passing through several intermediate states. We can abstract a state of the machine as a function σ that provides the content of the locations of the memory. So if

x is a location, $\sigma(x)$ is the value of x in the σ state. Consider also the pairs of the form $\langle c,\sigma\rangle$ (called *configurations*) where c is a command and σ is a state. Operational semantics consists in defining relations $\langle c,\sigma\rangle \to \sigma'$ between a configuration $\langle c,\sigma\rangle$ and the state they induce σ', that is, the state obtained after the execution of c in state σ. In operational semantics, definitions are given in the form of inference rules, written as a series of premises (written from configurations) above a horizontal line and a conclusion below it :

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \text{premise}_3 \quad \dots \quad \text{premise}_n}{\text{conclusion}}$$

Once again, we won't go into further detail, but the curious reader can find more information in [24, 18].

1.2.3 Denotational semantics

Denotational semantics was introduced by Dana Scott [22] in 1970. It aims to represent programs' behavior by constructing mathematical objects called *denotations.* To be more precise, it assigns an element (a denotation) in a mathematical structure called *semantic domain* to each expression of the programming language. In other words, it is entirely determined by a semantic domain and a *semantic function* from the syntactic elements of the programming language to the semantic domain :

$$[\![\cdot]\!] : \text{abstract syntax} \to \text{semantic domain.}$$

For example, the denotation of a boolean expression is a boolean value, the denotation of an arithmetic expression is an integer, etc. Of course, meanings are often more complex, but we always represent them as mathematical objects. A few properties of denotational semantics that are important are *compositionality* (the meaning of a program is the composition of the meaning of all its sentences), *completeness* (the semantic function is surjective : every element in the semantic domain corresponds to at least one syntactic expression) and *soundness* (two programs make the same thing if and only if they have the same denotation).

Let's give an example. We consider a programming language $\mathscr{A}$ containing arithmetic expressions only. The abstract syntax of the language can be defined as follows :

$$\begin{aligned} <n> &::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \\ <\textbf{op}> &::= \textbf{sum} \mid \textbf{mul} \mid \textbf{sub} \\ <e> &::= n \mid \textbf{op}\ e\ e \end{aligned}$$

Let the ring of integers $(\mathbb{Z},+,\cdot)$ be our semantic domain and the semantic function

$$[\![\cdot]\!] : \mathscr{A} \to (\mathbb{Z},+,\cdot)$$

be defined by :

$$\begin{aligned} [\![n]\!] &= n \\ [\![\textbf{add}\, e_1\, e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\ [\![\textbf{mul}\, e_1\, e_2]\!] &= [\![e_1]\!] \cdot [\![e_2]\!] \\ [\![\textbf{sub}\, e_1\, e_2]\!] &= [\![e_1]\!] - [\![e_2]\!] \end{aligned}$$

This representation allows us to tell how a program written in the language $\mathscr{A}$ behaves and possibly prove that it converges, gives the expected results, etc.

One of the most important challenges at the origin of denotational semantics was to identify which types of mathematical structures are suitable as semantic domains. At first sight, algebraic structures come to mind. However, although they can be used as domains, theoretical aspects of computation lead to consider other mathematical objects. The first to notice it was Scott, in the early 1970s, as he was trying to formulate a mathematical theory of computation, that is, trying to find a mathematical structure that can imitate, in some sense, what programs can do.

Concretely, he was searching a semantic domain for λ-calculus, which is a model of computation that can be used to simulate any Turing machine. It was introduced by Church in the 1930s to provide a foundation for the mathematical theory of functions, in particular for studying functions of higher order, that is, functions that can be applied to other functions. Note that this is very common to do in programming but not in mathematics. Indeed, if X is a set of functions, and elements of X can be applied to themselves, this means that a function $x \in X$ should also be considered a function from X to X. Therefore, we should have a bijection between X and the family of self functions on X, which is not possible unless X has cardinal at most 1.

This was not the only issue awaiting resolution. Another important problem was that of recursive definitions (i.e. definitions of the elements in a set in terms of other elements in the set). The semantic domains must handle these definitions guaranteeing that at least one solution exists. All this lead Scott to consider ordered structures rather than algebraic structures as the prototypical semantic domains. The particular structure he invented is nowadays known as a *domain* [12] and has become the main structure for the mathematical modeling of notions like approximation and computation.

The main objective of this master's final project is to develop the basic theory of domains and explicitly show their usefulness as semantic domains for denotational semantics. To achieve this, we have organized this work as follows :

In chapter 2, we introduce some basic concepts about ordered sets, trying to indicate the motivation behind them from a computational perspective. We insist on Scott topology's fundamental role to model the notion of computability of an element through limits of nets.

In chapter 3, we give an introduction to λ-calculus, which is a formal system created by Alonzo Church in the 1930s to describe computations and study their properties. Scott's first objective was precisely to build semantic domains for the λ-calculus. We then construct a certain type of domains called D_∞, which provides an answer to the problem of the mathematical interpretation of programs that can "call" themselves.

Chapter 2

Domain theory

2.1 Introduction

Domains are the ordered structures most used for denotational semantics of programming languages. Their origins go back to Scott's work on the semantics of high-level computer languages [22]. Originally, he advocated the use of continuous lattices but it was soon noticed that the completeness condition of lattices was too strong from a theoretical and practical point of view. Thus, he changed it to a weaker one, i.e. directed completeness, giving rise to the concept of domain (a continuous directed complete partially ordered set) [12]. But how did Scott come to see these ordered structures as appropriate for denotational semantics ? Who better than Scott himself to explain it ? We will therefore follow the train of thought expressed in his article

> [22] D. Scott, *Outline of a mathematical theory of computation*, Technical Monograph PRG-2, Oxford University Computing Laboratory, 1970.

Recall that a function is computable if and only if it can be reproduced by an algorithm, that is, there exists an algorithm that takes any element in the function domain as input and returns the image of the element by the function. The Church-Turing thesis asserts that computable functions are those computable by a Turing machine, a theoretical model of computers. We recall its definition, originally given in [27] :

Definition 2.1.1 (Turing machine)

A Turing machine is a model describing a machine handling a theoretically infinite sequence of symbols given a set of rules. Formally, it is a quintuplet $(Q, \Sigma, \delta, q_0, F)$ that consists of :

* a finite set of states Q
* a finite set of symbols Σ containing a special blank symbol $\sqcup \in \Sigma$
* an initial state $q_0 \in Q$
* a transition function

$$\begin{aligned} \delta\colon Q \times \Sigma &\Longrightarrow Q \times \Sigma \times \{-1, 0, 1\} \\ (q_i, x) &\longmapsto \delta(q_i, x) = (q_j, y, ?) \end{aligned}$$

- $*$ a set of accepting states $F \subseteq Q$

Informally, a Turing machine can be seen as an infinite memory tape divided into cells containing symbols from Σ and having a read/write head moving to the left or to the right. Cells yet "untouched" are written with the blank symbol. In the state q_i, if the head reads x (at the current cell), then $\delta(q_i, x) = (q_j, y, i)$ determines what the machine does next : replace x by y, move the head according to i (-1 stands for moving to the left, 0 for halting and 1 for moving to the right), and change the state to q_j.

Let's see a Turing machine that increments binary numbers by 1. Let $\mathbb{T} = (Q, \Sigma, \delta, q_0, F)$ where $Q = (q_0, q_1, q_2, q_3)$, $\Sigma = \{0, 1, \sqcup\}$, $F = q_3$ and the transition function δ is defined by the following table :

Current State	Current Symbol	New State	New Symbol	Move Direction
q_0	0	q_0	0	right
q_0	1	q_0	1	right
q_0	$\sqcup$	q_1	$\sqcup$	left
q_1	0	q_3	1	halt
q_1	1	q_2	0	left
q_1	$\sqcup$	q_3	1	halt
q_2	0	q_2	1	left
q_2	1	q_2	0	left
q_2	$\sqcup$	q_3	$\sqcup$	halt

Table 2.1 : Transition table of a Turing machine incrementing binary numbers by 1

In figure 2.2, we describe the execution steps if the input is the binary number 101.

With all of this in mind, if the objective is to give mathematical meaning to programs implementing algorithms, we can be more precise by saying that our goal is to define a mathematical framework for computable functions. Moreover, although we may know that a function is calculable, it could be unclear how to calculate it. So an « *adequate theory of computation not only provides the abstractions (what is computable) but also their "physical" realizations (how to compute them)* » [22]. Taking into account that « *the mathematical meaning of a procedure ought to be a function from elements of the data type of the input variables to elements of the data type of the output* », it is necessary to determine a flexible mathematical structure allowing to represent the wide variety of data types.

Current state	Tape	Action
q_0	$\sqcup\underset{\uparrow}{1}01\sqcup$	$\delta(q_0, 1) = (q_0, 1, \text{right})$: move right
q_0	$\sqcup 1\underset{\uparrow}{0}1\sqcup$	$\delta(q_0, 0) = (q_0, 0, \text{right})$: move right
q_0	$\sqcup 10\underset{\uparrow}{1}\sqcup$	$\delta(q_0, 1) = (q_0, 1, \text{right})$: move right
q_0	$\sqcup 101\underset{\uparrow}{\sqcup}$	$\delta(q_0, \sqcup) = (q_1, \sqcup, \text{left})$: move left and change the state to q_1
q_1	$\sqcup 10\underset{\uparrow}{1}\sqcup$	$\delta(q_1, 1) = (q_2, 0, \text{left})$: write 0, move left and change the state to q_2
q_2	$\sqcup 1\underset{\uparrow}{0}0\sqcup$	$\delta(q_2, 0) = (q_2, 1, \text{left})$: write 1 and move left
q_2	$\sqcup\underset{\uparrow}{1}10\sqcup$	$\delta(q_2, 1) = (q_2, 1, \text{left})$: move left
q_2	$\underset{\uparrow}{\sqcup}110\sqcup$	$\delta(q_2, \sqcup) = (q_3, \sqcup, \text{halt})$: change the state to q_3, i.e. end the process

Table 2.2 : Execution steps of a Turing machine incrementing 101 by 1

A first simple idea is that a data type is only a set D. Nevertheless, data types, being based on the idea of approximation, have « more structure » than sets. For instance, consider the `float` data type, which represents the set of real numbers $\mathbb{R}$. Given two variables x, y of data type `float`, they can be considered completely different from each other, or as approximating the same number $r \in \mathbb{R}$. Then, it is natural to demand a relation order $\sqsubseteq$ such that $x \sqsubseteq y$ means that y is a better approximation than x to r. This inevitably leads to consider a data type as an ordered structure (a partially ordered set), and that, in a data type D, we have elements with incomplete information (*partially defined elements*) and maximal elements whose information contained cannot be improved by other elements (*totally defined elements*). To clarify this, let's take the following example.

Example 2.1.1 ([17])

Let $I(\mathbb{R})$ be the set of all nonempty closed and bounded real intervals with the binary relation given by

$$[a, b] \sqsubseteq [c, d] \text{ if and only if } [c, d] \subseteq [a, b]$$

where $[a, b], [c, d] \in I(\mathbb{R})$. We can interpret an interval $[a, b] \in I(\mathbb{R})$ as an approximation of all the real numbers it contains. If $a < b$, $\exists a < a' < b' < b$. So, $[a', b'] \subseteq [a, b] \implies [a, b] \sqsubseteq [a', b']$ and thus $[a, b]$ is not a maximal element but a partially defined element. Totally defined elements are degenerate intervals of the form $[a, a], a \in \mathbb{R}$. Moreover, if $[a, b]$ and $[c, d]$ both approximate r, that is, $r \in [a, b] \cap [c, d]$, and $[a, b] \sqsubseteq [c, d]$, then $[c, d]$ is a better approximation of r than $[a, b]$.

So far, we have defined data types as ordered structures and nothing more. It is still too general for our purposes and we must refine it. Suppose we have an algorithm that works iteratively producing a chain of better approximations to some result. In other words, we have a sequence

$(x_n)_{n\in\mathbb{N}}$ such that

$$x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$$

In this context, it is logical to expect this sequence of approximations to converge towards the desired result which should be $\sup_{n\in\mathbb{N}} x_n$, the supremum of the sequence. We will see that domains, that are directed complete partially ordered sets endowed in a natural way with a topology, meet these criteria and will therefore be our main objects of study. Before presenting them, let's give a brief example of an algorithm that could use such mathematical description.

A concrete example : the bisection method

The bisection method is an algorithm for finding the zeros of a continuous function

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto f(x) \end{aligned}$$

in an interval $[a,b] \subset \mathbb{R}$ such that $f(a)\cdot f(b) < 0$, that is $f(a)$ and $f(b)$ have opposite signs. Indeed, by the intermediate value theorem, the continuous function f must have at least one root in the interval $]a,b[$. Here is the description of the algorithm, written in pseudo-code.

```
if fa * fb > 0:
    print("f(a) and f(b) should have opposite signs!")
    return;

while abs(fc) > epsilon and abs(fa - fb) > delta do:
    c := (a+b)/2;
    fc := f(c);
    if fa * fc < 0:
        b := c;
        fb := fc;
    else:
        a := c;
        fa := fc;
```

In other words, as long as we are not sufficiently close to the solution and the interval is not too narrow (this condition prevents the algorithm from running too long, for example if the function becomes flatter near its roots), we iterate through the following steps : calculate the midpoint c_i of the current interval (a_i, b_i) (step 1), compute its image by f : $f(c_i)$ (step 2) and ultimately examine the sign of $f(c_i)$ and set the values of a_{i+1} and b_{i+1} accordingly) (step 3).

The key point here is that, as the algorithm progresses, the interval in which we know there is a zero crossing becomes narrower, i.e. the amount of information we have about its position increases. If we were to equip $\mathscr{P}([a,b])$ with a partial order $\preceq$ that reflects this idea that the narrower the interval, the greater the approximation, we would define it as :

$$\forall X, Y \subseteq [a,b], \quad Y \preceq X \quad \text{if } X \subseteq Y. \tag{2.1}$$

It would remain to be seen how to relate a topology to this relation order.

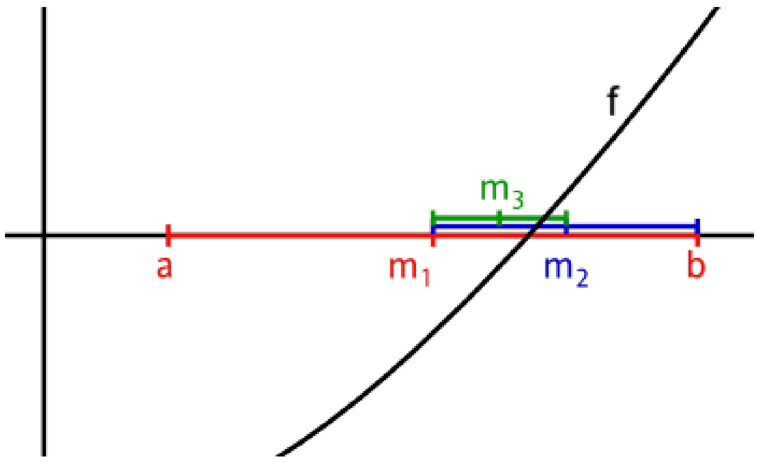

Figure 2.1 : An illustration of the bisection method (m_i indicates the interval in which we know the solution is at the i-th iteration)

2.2 Directed complete partial orders

In this section, we present some definitions related to ordered sets and the functions between them. The main notion is that of directed complete partially ordered sets. Adding an extra ingredient, we obtain the *domains* which turn out to be the prototypes for semantic domains. We mainly follow the monographs [2, 12].

Definition 2.2.1 (proset, [[12, Definition O-1.1.])

] Let P be a nonempty set. A binary relation $\leq$ on P which is :

* reflexive : $\forall x \in P, x \leq x$
* transitive : $\forall x, y, z \in P, x \leq y$ and $y \leq z \Longrightarrow x \leq z$

is called a *preorder*. A set P equipped with a partial order $\leq$ is called *preordered set*, or *proset* for short.

If $\{(P_i, \leq_i) : i \in I\}$ is a family of prosets, then we can endow the cartesian product $\prod_{i\in I} P_i$ with the poinwise preorder $\sqsubseteq$ given by

$$x \sqsubseteq y \text{ iff } x_i \leq_i y_i$$

where $x = (x_i)_{i\in I}, y = (y_i)_{i\in I} \in \prod_{i\in I} P_i$. We will always consider this preorder in a cartesian product of prosets.

Here's another definition, the one we're really interested in.

Definition 2.2.2 (poset, [[12, Definition O-1.6.])

] Let P be a nonempty set and $\leq$ be a preorder on P. If $\leq$ satisfies :

$$\forall x, y \in P, x \leq y \text{ and } y \leq x \Longrightarrow x = y$$

(we say that $\leq$ is anti-symmetric), then it is called a *partial order*. A set P equipped with a partial order is called *partially ordered set*, or *poset* for short.

Example 2.2.1

$*$ $(\mathbb{R}, \leq)$ is a poset.

$*$ The set of natural numbers equipped with the relation of divisibility is a poset.

$*$ Let $A = \{a, b\}$ be a set. The relation $\{(a,a),(a,b),(b,a),(b,b)\}$ is a preorder on A, but it's not a partial order.

Recall that in his seminal paper [22], Scott considers a data type as a partially ordered set $(D, \leq)$ in which the relation $x \leq y$ means that y can be considered a better version than x of what x is trying to approximate. So this means that y is consistent with x and possibly more accurate than x.

Definition 2.2.3 (upper bound)

Let $(P, \leq)$ be a poset and let $X \subseteq P$. If $\forall x \in X, x \leq u$ (resp. $\forall x \in X, l \leq x$), we say that u is an *upper bound* (resp. l is a *lower bound*) of X.

Definition 2.2.4 (supremum, [[12, Definition O-1.6.])

] Let $(P, \leq)$ be a poset and let $X \subseteq P$. If the set of upper bounds of X has a unique smallest element u (resp. the set of lower bounds of X has a unique largest element l), we call this element the *supremum* (resp. *infimum*) of X and we note $u = \sup X$ (resp. $l = \inf X$).

Note that a subset X of a poset $(P, \leq)$, although upper bounded, does not necessarily have a supremum. For instance, let's consider $\mathbb{R}_* = \mathbb{R} \setminus \{0\}$ (the set of all non-zero real numbers). Then, the interval $I_- = (-\infty, 0) = \{x \in \mathbb{R}_* \mid x < 0\}$ is upper bounded but its set of upper bounds $I_+ = (0, +\infty) = \{x \in \mathbb{R}_* \mid x > 0\}$ does not have a (unique) smallest element.

Definition 2.2.5 (directed subset, [[12, Definition O-1.1.])

] Let $(P, \leq)$ be a poset and let $\emptyset \neq \Delta \subseteq P$. If every subset of two elements in Δ has an upper bound (resp. lower bound) in Δ, i.e. if $\forall x, y \in \Delta, \exists z \in \Delta$ such that $x \leq z$ and $y \leq z$, then Δ is said to be *directed* (resp. *filtered*).

Also notice that we obtain an equivalent definition to the above if we replace the two-element subsets with finite subsets.

Directed subsets model the process of computation as it progresses through successive approximations. Each element in a directed set represents a stage of computation, i.e. a partial piece of information about the desired result. The directed property guarantees that any two stages of computation can be jointly extended to a more defined stage. This is crucial for ensuring that the computation converges towards a final result.

The main idea is that every computation has a purpose (it's always a computation $\underline{\text{of}}$ something). Essentially, there should always be a largest element that surpasses all the elements in the directed set representing the computation. This largest element is an upper bound of Δ. However, what we're really looking for is not any upper bound, but the supremum. This is the most accurate approximation we can make without superfluous information. This motivates the following definition.

Definition 2.2.6 (dcpo, cpo, [[12, Definition O-2.1.])

] Let $(P, \leq)$ be a poset. If every directed subset of P has a supremum, $(P, \leq)$ is called a *directed complete partially ordered set* (*dcpo* for short). If moreover $(P, \leq)$ has a least element (usually written $\perp$), then it's called a *complete partial ordered set* (*cpo*) or *pointed dcpo*.

Example 2.2.2

$*$ Let X be a nonempty set. The family $\mathscr{P}(X)$ of all subsets of X, endowed with the partial order induced by the inclusion, is a dcpo. Notice that, in this case, the supremum (resp. infimum) of a family of elements always exists and is their union (resp. intersection).

$*$ Given a topological space $(X, \mathscr{T})$, $(\mathscr{T}, \subseteq)$ is a dcpo, where the supremum of a family of elements is, as above, their union.

$*$ Let us consider the family $I(\mathbb{R})$ of all nonempty closed and bounded real intervals and the binary relation $\sqsubseteq$ as defined in the example 2.1.1. It is easy to check that $(I(\mathbb{R}), \sqsubseteq)$ is a poset. Moreover, if $\Delta = \{[x_i, y_i] : i \in J\} \subseteq I(\mathbb{R})$ is a directed subset, if $[x_i, y_i], [x_j, y_i] \in \Delta$ then we can find $[x_k, y_k] \in \Delta$ such that $[x_i, y_i] \sqsubseteq [x_k, y_k]$ and $[x_j, y_j] \sqsubseteq [x_k, y_k]$, that is, $[x_k, y_k] \subseteq [x_i, y_i] \cap [x_j, y_j]$. Hence, the intersection of two intervals of Δ is nonempty. Moreover, a straightforward calculation shows that $\sup \Delta = [\sup_{i \in i} x_i, \inf_{i \in I} y_i]$ so $(I(\mathbb{R}), \sqsubseteq)$ is a dcpo.

$*$ Let $X \rightharpoonup Y$ be the set of partial functions between two sets X, Y, endowed with the following partial order

$$f \leq g \iff (\mathrm{dom}(f) \subseteq \mathrm{dom}(g) \text{ and } \forall x \in \mathrm{dom}(f), f(x) = g(x))$$

where $\mathrm{dom}(f)$ denotes the domain of f. $(X \rightharpoonup Y, \leq)$ has a least element $\perp$ which is the everywhere undefined function. Indeed, $\forall h \in X \rightharpoonup Y, \mathrm{dom}(\perp) = \emptyset \subseteq \mathrm{dom}(h) \subseteq X$ and $\forall x \in \emptyset, f(x) = g(x)$. Moreover, if $\Delta \subseteq X \rightharpoonup Y$ is directed, that is $\forall f, g \in \Delta, \exists h \in \Delta$ such that $f \leq h$ and $g \leq h$, which is only possible if $\forall f, g \in \Delta, \forall x \in \mathrm{dom}(f) \cap \mathrm{dom}(g), f(x) = g(x)$, Δ has a supremum defined as $\sup \Delta = \bigcup_{f \in \Delta} \{(x, y) \mid x \in \mathrm{dom}(f) \text{ and } f(x) = y\}$. Hence, $X \rightharpoonup Y$ is a cpo. In particular, it's a dcpo.

Example 2.2.3 ([2, Example 1.1.6])

Every arbitrary nonempty set M can be easily converted into a cpo by considering $M_\perp = M \cup \{\perp\}$ and endowing it with the partial order $\preceq$ given by

$$a \preceq b \text{ if and only if } (a = b \text{ or } a = \perp)$$

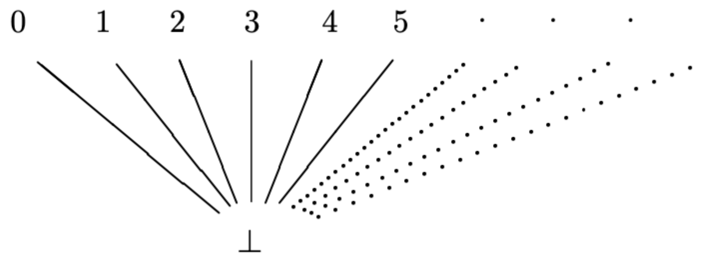

Figure 2.2 : An illustration of the cpo $\mathbb{N}_\perp$ (straight lines represent the relation order on $\mathbb{N}_\perp$)

for all $a, b \in M_\perp$. Indeed :

$\preceq$ is a partial order

1. By definition, $\preceq$ is reflexive.
2. Let $a, b \in M_\perp$. If $a \preceq b$ and $b \preceq a$, then ($a = \perp$ and $b \in M$ and $b = \perp$ and $a \in M$) or $a = b \Longrightarrow a = b$.
3. Let $a, b, c \in M_\perp$ such that $a \preceq b$ and $b \preceq c$. If $a = \perp$, then $a \preceq c$. Otherwise $a \in M$. Thus, $a \preceq b \Longrightarrow a = b \neq \perp$. Thus, $b \preceq c \Longrightarrow b = c \Longrightarrow a = c \Longrightarrow a \preceq c$.

every directed subset of $M_\perp$ has a supremum

The only directed subsets of $M_\perp$ are one-member sets and pairs $\{\perp, n\}$ with $n \in M$ and both of these have obvious suprema.

The cpos constructed in this way are called flat. When $M = \mathbb{N}$ we obtain the flat cpo $\mathbb{N}_\perp$ which will be useful in section 3.2.

A well-known and stronger notion than dcpo is the following.

Definition 2.2.7 ([12, Definition O-2.1.])

Let $(P, \leq)$ be a poset. If every subset of P (and not just every directed subset) has a supremum and an infimum, P is called a *complete lattice.*

Example 2.2.4

$*$ Let X be a set. $(\mathscr{P}(X), \subseteq)$, the power set of X ordered by inclusion, is a complete lattice. The supremum is given by the union and the infimum by the intersection of subsets.

$*$ The ideals of a ring ordered by inclusion form a complete lattice. The supremum is given by the sum of ideals and the infimum by the intersection.

2.2.1 Functions between posets and dcpos

We next define the natural functions between posets, that is, the functions that preserve the order structure.

Definition 2.2.8 (monotony, [2, Definition 1.1.12])

Let $(P, \leq)$ and $(P', \preccurlyeq)$ be posets. A function $f : P \to P'$ is called *monotone* or *order preserving* if

$$\forall x, y \in P, \quad x \leq y \Longrightarrow f(x) \preccurlyeq f(y).$$

We can interpret the previous definition from a computational point of view. Suppose that $(P, \leq)$ and $(P', \preccurlyeq)$ are two data types, that is, two partially ordered sets as Scott interprets them [22]. If $f : P \to P'$ is a function determined by a program, it is natural to expect that whenever $x, y \in P$ and y is more accurate than x, i.e. $x \leq y$, then $f(y)$ is more accurate than $f(x)$, i.e. $f(x) \preccurlyeq f(y)$. So monotone functions are also natural from this perspective.

Notice that a monotone function maps directed sets to directed sets. Indeed, if $\Delta \subseteq P$ is directed and $f : P \to P'$ is monotone, then $\forall u, v \in f(\Delta), \exists x, y \in \Delta$ such that $f(x) = u$ and $f(y) = v$. Since Δ is directed $\exists z \in \Delta$ such that $x \leq z$ and $y \leq z$. Thus $w = f(z) \in f(\Delta)$, and since f is monotone we deduce that $u = f(x) \preccurlyeq f(z) = w$ and $v = f(y) \preccurlyeq f(z) = w$, that is, $f(\Delta)$ is directed.

When we consider dcpos instead of posets, we are guaranteeing the existence of suprema for directed sets. So it is natural to require that functions preserving the structure of dcpos additionally satisfy the following property.

Definition 2.2.9 (function preserving directed suprema, [2, Definition 1.1.12])

Let $(D, \leq)$ and $(D', \preccurlyeq)$ be dcpos. A function $f : D \to D'$ is said to *preserve directed suprema* if

$$\forall \Delta \subseteq D \text{ directed}, \quad f\left(\sup \Delta\right) = \sup f(\Delta).$$

Observe that if $f : (D, \leq) \to (D', \preccurlyeq)$ preserves directed suprema, then it is monotone. Indeed, given $x, y \in D$, with $x \leq y$ then $\Delta = \{x, y\}$ is directed and $\sup \Delta = y$. Since f preserves directed sups then $\sup f(\Delta) = \sup f(\{x, y\}) = f(\sup \Delta) = f(y)$. Hence $f(x) \preccurlyeq f(y)$.

We present a theorem that is the key to the interpretation of recursively defined programs and will help us understand the order theoretical limit construction of D_∞ spaces in chapter 3, section 3.2.

Theorem 2.2.1 (Kleene fixed-point, [2, Proposition 1.1.7])

Let $(D, \leq)$ be a cpo and $f : D \to D$ be a function preserving directed suprema. Then f has a fixed point.

Proof. It is clear that $\perp \leq f(\perp)$. Consider the monotone sequence $(f^n(\perp))_{n \in \mathbb{N}}$. Since $\Delta = \{f^n(\perp) : n \in \mathbb{N}\}$ is directed it has a supremum $x = \sup \Delta = \sup\{f^n(\perp) : n \in \mathbb{N}\}$. Moreover,

$$f(x) = f(\sup \Delta) = \sup f(\Delta) = \sup\{f^{n+1}(\perp) : n \in \mathbb{N}\} = \sup\{f^n(\perp) : n \in \mathbb{N}\} = \sup \Delta = x,$$

so x is a fixed point. □

2.3 Scott topology

In the beginning of the chapter, we introduced an iterative algorithm producing a sequence $(x_n)_{n\in\mathbb{N}}$ of partial elements approximating a final result x, which should naturally be the supremum of the sequence. This gives the intuition that topology should come into play. More precisely, it would be good to equip the dcpo with a topology that allows to express elements as limits of its approximate elements and to say "how far from the result we are". Such a topology was introduced by Scott (see [22, 12]) and this is the main topic of this section.

We start with a definition that will be used right after.

Definition 2.3.1 (upper set)

Let $(P, \leq)$ be a poset and let $A \subseteq P$. Define

$$\uparrow A = \{x \in P : a \leq x \text{ for some } a \in A\};$$
$$\downarrow A = \{x \in P : x \leq a \text{ for some } a \in A\}.$$

We say that A is an upper set (resp. lower set) if $A = \uparrow A$ (resp. $A = \downarrow A$).

Lemma 2.3.1

Any union or intersection of upper sets (resp. lower sets) is an upper set (resp. lower set).

Proof. Let's prove the two assertions one by one.

A union of upper sets is an upper set.

Let $(P, \leq)$ be a poset and $(A_i)_{i\in I}$ be a family of upper sets in P. Let $y \in P$ such that $\exists x \in \bigcup_{i\in I} A_i$ and $x \leq y$. Therefore, there exists $i \in I$ such that $x \in A_i$ which is an upper set. Hence, $y \in A_i \subseteq \bigcup_{i\in I} A_i$. Thus $\bigcup_{i\in I} A_i$ is an upper set.

An intersection of upper sets is an upper set.

Let $(P, \leq)$ be a poset and $(A_i)_{i\in I}$ be a family of upper sets in P. Let $y \in P$ such that $\exists x \in \bigcap_{i\in I} A_i$ and $x \leq y$. Therefore, $\forall i \in I, x \in A_i$ and A_i is an upper set. Hence, $\forall i \in I, y \in A_i \Longrightarrow y \in \bigcap_{i\in I} A_i$. Thus $\bigcap_{i\in I} A_i$ is an upper set. □

Any poset $(P, \leq)$ can be "topologized" in a natural way using upper sets as open sets. Such topology is called the *Alexandrov topology*. It is, indeed, a topology since $\emptyset$ and P are upper sets and, by 2.3.1, the infinite union and the (in)finite intersection of upper sets is an upper set. Hence, this topology satisfies an extra condition : arbitrary intersection of open sets is open. This type of topology was introduced by Alexandrov in 1937 [1] under the name of *discrete spaces*. These topological spaces have been well studied in the literature (see [4, 20, 26]) and are very related to preordered sets.

Notice that, in a poset $(P,\leq)$, if $\Delta \subseteq P$ is directed, then $(d)_{d\in\Delta}$ is a net on P (see definition 2.5.1). Moreover, suppose that $(d)_{d\in\Delta}$ converges to an upper bound s of Δ, in the Alexandrov topology. Then s must be the supremum of Δ. Let's show this. Let u be an upper bound of Δ. Since $\uparrow\{s\}$ is an open set containing the limit s, there exists $d \in \Delta$ such that $d \in \uparrow\{s\}$, that is, $s \leq d$. Since u is an upper bound of Δ then $d \leq u$ so $s \leq u$, showing that s is the supremum.

In general, we cannot assure that an arbitrary net $(d)_{d\in\Delta}$ converges to its supremum in the Alexandrov topology, but it seems a natural requisite for the topology we're looking for, since, in a dcpo, the supremum of a directed set always exists. The Scott topology was introduced to fill this gap and does this by imposing an additional condition on open sets (besides being upper sets).

Definition 2.3.2 (Scott topology,[12, 22, 23])

Let D be a dcpo. A subset $A \subseteq D$ is called Scott open if :

1. $A = \uparrow A$
2. if $x \in A$, $\Delta \subseteq D$ is directed and $x = \sup\Delta$, then $\Delta \cap A \neq \emptyset$

The collection $\Omega_S(D)$ of Scott open sets is a topology over D called the Scott topology.

Let's verify that $\Omega_S(D)$ is a topology.

1. D and $\emptyset$ trivially check the two conditions.

2. Given $n \in \mathbb{N}$, if the sets $A_1, A_2, \ldots, A_n$ are Scott open, then, by 2.3.1, $\bigcap_{i\leq n} A_i$ is an upper set. Moreover, if Δ is a directed set, $x = \sup\Delta$ and $x \in \bigcap_{i\leq n} A_i$, then $\forall i \leq n, \exists y_i \in \Delta \cap A_i$. But Δ is directed, therefore, by induction, $\exists y \in \Delta$ such that $y_i \leq y$ but A_i is an upper set $\Longrightarrow y \in A_i$. Hence, $y \in \Delta \cap \bigcap_{i\leq n} A_i$.

3. For any infinite set I, if $\forall i \in I$, A_i is Scott open, then, by 2.3.1, $\bigcup_{i\in I} A_i$ is an upper set. What's more, if Δ is a directed set, $x = \sup\Delta$ and $x \in \bigcup_{i\in I} A_i$, then $\exists i \in I$ such that $x \in A_i \Longrightarrow \exists y \in \Delta \cap A_i \subseteq \Delta \cap \bigcup_{i\in I} A_i$.

Example 2.3.1

$*$ Let us consider the dcpo $(\mathbb{R},\leq)$. Then

$$\Omega_S(\mathbb{R}) = \Big\{(a,+\infty) : a \in \mathbb{R}\Big\} \cup \{\mathbb{R},\emptyset\}.$$

$*$ Let us consider the dcpo $(\mathscr{P}(X),\subseteq)$. Then a nonempty family $\mathcal{O}$ of subsets of X is open in the Scott topology $\Omega_S(\mathscr{P}(X))$ if and only if for every $O \in \mathcal{O}$ there exists a finite subset F of O such that $F \in \mathcal{O}$.

We've just shown that a poset can be endowed, in general, with two different topologies : the Alexandrov topology and the Scott topology. Of course, these are not the only topologies that can be defined (think of discrete or trivial topologies), but the former have the particularity of being defined using the order structure of the set. Now, let's see how to retrieve the initial order from one of these two topologies, using a particular construction.

Definition 2.3.3 (specialization preorder)

Let (X, Ω_X) be a topological space. The binary relation $\leq_X \subseteq X \times X$, defined as

$$x \leq_X y \iff \forall A \in \Omega_X, x \in A \implies y \in A$$

is a preorder on X called the specialization preorder associated with Ω_X.

Example 2.3.2

Let X be a nonempty set.

$*$ Let us consider the topological space (X, Ω_I) where Ω_I is the indiscrete topology. Its specialization preorder is given by $x \leq_{X_I} y$ if and only if $x, y \in X$, i. e. $\leq_{X_I} = X \times X$.

$*$ Let us consider the topological space (X, Ω_D) where Ω_D is the discrete topology. Its specialization preorder is given by $x \leq_{X_D} y$ if and only if $x = y$, i. e. $\leq_{X_D} = \Delta_X = \{(x, x) : x \in X\}$.

$*$ Let us consider a T_1 topology Ω_X on X. Then, as above, the specialization preorder of (X, Ω_X) is equality. In fact, given two different points $x, y \in X$, since Ω_X is T_1, we can find an open set A such that $x \in A$, $y \notin A$. Hence $x \not\leq_X y$.

Lemma 2.3.2

A topology is T_0 if and only if its specialization preorder is a partial order.

Proof. Let (X, Ω_X) be a topological space.

$$\begin{aligned}
X \text{ is } T_0 &\iff \forall x, y \in X, x \neq y \implies \exists A \in \Omega_X \text{ such that } (x \in A \wedge y \notin A) \vee (x \notin A \wedge y \in A) \\
&\iff (x \notin A \vee y \in A) \wedge (x \in A \vee y \notin A), \forall x, y \in X, A \in \Omega_X \implies x = y \\
&\iff (x \in A \implies y \in A) \wedge (y \in A \implies x \in A), \forall x, y \in X, A \in \Omega_X \implies x = y \\
&\iff x \leq_X y \text{ and } y \leq_X x, \forall x, y \in X, A \in \Omega_X \implies x = y \\
&\iff \leq_X \text{ is a partial order}
\end{aligned}$$

□

Lemma 2.3.3

Let $f : (X, \Omega_X) \to (Y, \Omega_Y)$ be a continuous function between two topological spaces. Then $f : (X, \leq_X) \to (Y, \leq_Y)$ is monotone, where $\leq_X$ and $\leq_Y$ are the respective specialization preorders.

Proof. Let $x, y \in X$ such that $x \leq_X y$. Let $f(x) \in B \in \Omega_Y$. Since f is continuous, the antecedent of an open set by f is an open set. Hence, $x \in f^{-1}(f(x)) \subseteq f^{-1}(B) \in \Omega_X$ but $x \leq_X y \iff \forall A \in \Omega_X, x \in A \implies y \in A$. Therefore, $y \in f^{-1}(B) \implies f(y) \in B$. This remains true for any other $x, y \in X, B \in \Omega_Y$ so $f(x) \leq_Y f(y)$. □

Observe that, in the previous lemma, if Ω_X, Ω_y are Alexandrov, that is, are the Alexandrov topologies induced by the preorders $\leq_X$, $\leq_y$, then the implication becomes an equivalence [20, Theorem 8.3.6].

Lemma 2.3.4 ([2, Proposition 1.2.3])

Let $(D, \leq)$ be a dcpo equipped with the Scott topology $\Omega_S(D)$. The specialization preorder associated with $\Omega_S(D)$ is $\leq$. In particular, it's a partial order. Hence, $\Omega_S(D)$ is T_0.

Proof. Let $(D, \leq)$ be a dcpo equipped with the Scott topology $\Omega_S(D)$ and let $\leq_D$ be the specialization order associated with $\Omega_S(D)$. $\forall x, y \in D, x \leq y \Longrightarrow (\forall A \in \Omega_S(D), x \in A \Longrightarrow y \in \uparrow A = A) \Longleftrightarrow x \leq_D y$. Conversely, if $\forall A \in \Omega_S(D), x \in A \Longrightarrow y \in \uparrow A = A$, in particular, $x \in D \setminus \downarrow \{y\} \Longrightarrow y \in D \setminus \downarrow \{y\}$. Suppose now that $D \setminus \downarrow \{y\} \in \Omega_S(D)$ and $x \not\leq y$, then $x \in D \setminus \downarrow \{y\}$ but $y \notin D \setminus \downarrow \{y\}$ so $x \not\leq_D y$. We only need to prove that $D \setminus \downarrow \{y\} \in \Omega_S(D)$. If $x \in D \setminus \downarrow \{y\}$ and $x \leq z$, then $x \not\leq y \Longrightarrow z \not\leq y$ so $z \in D \setminus \downarrow \{y\}$. Moreover, if $\Delta \subseteq D$ is directed and $\sup \Delta \in D \setminus \downarrow \{y\}$, then $\sup \Delta \not\leq y$ so not all elements in Δ are less than y, i.e. $\exists d \in \Delta$ such that $d \not\leq y \Longrightarrow d \in D \setminus \downarrow \{y\}$. Hence, $D \setminus \downarrow \{y\} \in \Omega_S(D)$. □

The above lemma is also true if we consider the Alexandrov topology instead of the Scott topology [20, Theorem 8.3.3]. In fact, it is possible to enclose all the topologies on a poset whose specialization order is equal to the partial order. These topologies are coarser than the Alexandrov topology and finer than the upper topology [12, Definition 0-5.4].

In our search for an adequate topology to model computational properties using dcpos, we considered Scott topology because not only does the specialization preorder coincide with the original partial order, but also the suprema of directed sets are limits in the topology. It is also natural to ask that Scott topology makes functions preserving directed suprema (2.2.9) coincide with continuous functions. We'll see that it does indeed satisfy this property.

Proposition 2.3.1 ([2, Proposition 1.2.4])

Let $(D, \leq)$ and $(D', \leqslant)$ be dcpos. A function $f : (D, \leq) \to (D', \leqslant)$ preserves directed suprema if and only if $f : (D, \Omega_S(D)) \to (D', \Omega_S(D'))$ is continuous, where $\Omega_S(D), \Omega_S(D')$ are the Scott topologies of $(D, \leq)$, $(D', \leqslant)$ respectively.

Proof. Let f be continuous with respect to the Scott topologies. Let's show that f preserves directed suprema. By lemmas 2.3.3 and 2.3.4, f is monotone. Let $\Delta \subseteq D$ be a directed set. Suppose that $f(\sup \Delta) \not\leqslant \sup f(\Delta)$. It is easy to check that $B = D' \setminus \downarrow \{\sup f(\Delta)\}$ is open in $\Omega(D')$ and $f(\sup \Delta) \in B$ but $\sup f(\Delta) \notin B$. Hence, $\sup \Delta \in f^{-1}(B)$ which is an open set since f is continuous. By definition of Scott open sets, $\exists d \in \Delta$ such that $d \in f^{-1}(B) \Longrightarrow f(d) \in B \Longrightarrow f(d) \not\leqslant \sup f(\Delta)$ which is a contradiction. To see the other inequality, note that $\forall y \in f(\Delta), \exists x \in \Delta$ such that $y = f(x) \leqslant f(\sup \Delta)$ by monotony of f. Thus, $\sup f(\Delta) \leqslant f(\sup \Delta)$.

Conversely, let $f : (D, \leq) \to (D', \leqslant)$ preserves directed suprema. Let $A \in \Omega_S(D')$. Let's show that $f^{-1}(A) \in \Omega_S(D)$. If $x \in \uparrow f^{-1}(A)$ there exists $y \in f^{-1}(A)$ with $y \leq x$. Since f is monotone then $f(y) \leqslant f(x)$. Hence $f(x) \in \uparrow f(y) \subseteq \uparrow A = A$ since A is Scott open. Therefore, $\uparrow f^{-1}(A) = f^{-1}(A)$. In addition, let $\Delta \subseteq D$ be directed such that $\sup \Delta \in f^{-1}(A)$. Since f preserves directed sups then $f(\sup\Delta) = \sup f(\Delta) \in A$, so there exists $d \in \Delta$ such that $f(d) \in A$, that is, $d \in f^{-1}(A)$. This completes the proof. □

Definition 2.3.4 ([12, Definition II-2.2.])

A function $f : (D, \leq) \to (D', \preccurlyeq)$ between two dcpos is called Scott continuous if it satisfies the equivalent conditions of the previous lemma. We will denote by $D \to D'$ the family of all Scott continuous functions.

Lemma 2.3.5 ([2, Proposition 1.4.4])

Let $(D, \leq)$ and $(D', \preccurlyeq)$ be two dcpos. Let $f, g \in D \to D'$. We say that f is less than g point-wise, and we write $f \preccurlyeq g$, if $f(x) \preccurlyeq g(x), \forall x \in D$. In this way, $(D \to D', \preccurlyeq)$ is a dcpo and if D' is a cpo, then $D \to D'$ is a cpo.

Proof. It is easy to check that $(D \to D', \preccurlyeq)$ is a poset. Moreover, let $\mathscr{D} = \{f_i : i \in I\}$ be a directed subset of $D \to D'$. Given $x \in D$, then $\{f_i(x) : i \in I\}$ is directed so $\sup_{i\in I} f_i(x)$ exists. Define $f : D \to D'$ as

$$f(x) = \sup_{i\in I} f_i(x)$$

for all $x \in X$. Let's check that f is Scott continuous. If $x \leq y$ then $f_i(x) \preccurlyeq f_i(y)$ for all $i \in I$, since f_i is Scott continuous so monotone. Hence $f(x) \preccurlyeq f(y)$ so f is also monotone.

Furthermore, let Δ be a directed subset of D. Since $d \leq \sup\Delta \forall d \in \Delta$ and f is monotone then $f(d) \preccurlyeq f(\sup\Delta)$. Hence $\sup_{d\in\Delta} f(d) = \sup f(\Delta) \preccurlyeq f(\sup\Delta)$.

Moreover,

$$f(\sup\Delta) = \sup_{i\in I} f_i(\sup\Delta) = \sup_{i\in I}(\sup f_i(\Delta)) \preccurlyeq \sup f(\Delta).$$

Consequently, $\sup f(\Delta) = f(\sup\Delta)$ so f preserves directed suprema. By proposition 2.3.1, f is Scott continuous. Besides, it is clear that $f = \sup_{i\in I} f_i$ so $(D \to D', \preccurlyeq)$ is a dcpo.

If D' is a cpo and $\bot'$ is its bottom element, then the constant function with value $\bot'$ is clearly Scott continuous and it is the bottom of $D \to D'$. □

The following results concerning Scott continuity will be very important in the next chapter for constructing a model of the λ-calculus.

Proposition 2.3.2 ([12, Lemma II-2.8],[2, Proposition 1.4.3])

Let $(D, \leq), (D', \preccurlyeq), (E, \preceq)$ be dcpos. A function $f : D \times D' \to E$ is Scott continuous if and only if f is separately continuous, that is, the functions $f_x : D \to E, f_y : D' \to E$ given by

$$f_x(d) = f(d, x) \qquad\qquad f_y(d') = f(y, d')$$

for all $d \in D, d' \in D'$ are continuous for every $x \in D', y \in D$.

Proof. Suppose that f is continuous and let $x \in D'$ be fixed but arbitrary and Δ be a directed subset of D. Then

$$f_x(\sup\Delta) = f(\sup\Delta, x) = f(\sup\Delta_x) = \sup f(\Delta_x) = \sup f_x(\Delta)$$

where we have used that f is Scott continuous and where $\Delta_x = \{(d,x) : d \in \Delta\}$ is a directed subset of $D \times D'$. Hence f_x is Scott continuous. A similar argument shows that f_y is Scott continuous for every $y \in D$.

Conversely, suppose that f_x, f_y are Scott continuous for every $x \in D', y \in D$. Let Δ be a directed subset of $D \times D'$. Defining $\Delta_D = \{d \in D : (d,d') \in \Delta \text{ for some } d' \in D'\}$ and $\Delta_{D'} = \{d' \in D' : (d,d') \in \Delta \text{ for some } d \in D\}$, it is obvious that $\sup\Delta = (\sup\Delta_D, \sup\Delta_{D'})$. Then

$$\begin{aligned} f(\sup\Delta) = f(\sup\Delta_D, \sup\Delta_{D'}) &= \sup_{d\in\Delta_D} f(d, \sup\Delta_{D'}) \\ &= \sup_{d\in\Delta_D} \sup_{d'\in\Delta_{d'}} f(d,d') \preceq \sup f(\Delta). \end{aligned}$$

Moreover, since f_x, f_y are Scott continuous for every $x \in D', y \in D$ it is clear that f is monotone. This obviously implies that $\sup f(\Delta) \preceq f(\sup\Delta)$ which proves that f is Scott continuous. □

Proposition 2.3.3 ([2, Exercise 1.4.7],[12, Theorem II-2.10])

Let $(D,\leq), (D',\leqslant), (E,\preceq)$ be dcpos. If $f : D \times D' \to E$ is Scott continuous then $\Lambda(f) : D \to (D' \to E)$ is Scott continuous where

$$(\Lambda(f)(y))(x) = f(y,x)$$

for all $y \in D, x \in D'$.

Proof. Let Δ be a directed subset of D. Then given $x \in D'$

$$\Lambda(f)(\sup\Delta)(x) = f(\sup\Delta, x) = f_x(\sup\Delta) = \sup f(\Delta, x) = \sup\Lambda(f)(\Delta)$$

where we have used Scott continuity of f_x as assured by Proposition 2.3.2. Consequently, $\Lambda(f)$ is Scott continuous. □

Proposition 2.3.4

Let $(D,\leq), (E,\leqslant), (E',\preceq)$ be dcpos. Then $f : D \to E, g : D \to E'$ are Scott continuous if and only if $f \times g : D \to E \times E'$ is Scott continuous where

$$(f \times g)(d) = (f(d), g(d)) \qquad \forall d \in D.$$

Proof. We first suppose that f, g are Scott continuous. Let Δ be a directed subset of D. Then

$$(f \times g)(\sup\Delta) = (f(\sup\Delta), g(\sup\Delta)) = (\sup f(\Delta), \sup g(\Delta)) = \sup(f \times g)(\Delta).$$

Therefore, $f \times g$ is continuous.

The converse is obvious since projections are Scott continuous functions so $f = \pi_E \circ (f \times g), g = \pi_{E'} \circ (f \times g)$ are continuous. □

Proposition 2.3.5 ([12, Lemma II-2.9])

Let $(D,\leq),(E,\preccurlyeq)$ be dcpos. Then the evaluation map $ev:(D\to E)\times D\to E$ given by

$$ev(f,d)=f(d) \qquad \forall f\in D\to E, d\in D$$

is Scott continuous.

Proof. Let $f\in D\to E$ be fixed and Δ be a directed subset of D. Since f is Scott continuous then

$$ev_f(\sup\Delta)=f(\sup\Delta)=\sup f(\Delta)=\sup ev_f(\Delta)$$

so ev_f is Scott continuous.

On the other hand, for a fixed $d\in D$, given a directed subset $\mathscr{F}$ of $D\to E$ then

$$ev_d(\sup\mathscr{F})=(\sup_{f\in\mathscr{F}} f)(d)=\sup_{f\in\mathscr{F}} f(d)=\sup_{f\in\mathscr{F}} ev_d(f)$$

so ev_d is Scott continuous. We conclude by Proposition 2.3.2 that ev is Scott continuous. $\square$

In short, we've seen how important Scott topology is, mostly because it allows us to talk properly about limits and continuity. We've also seen how this topology is "the most natural" for building semantic models. However, there are still a few structural properties to be added to achieve spaces that perfectly meet our expectations.

2.4 Scott domains

In this section, we present Scott domains, which are the main semantic domains for denotational semantics. First, we need the following definition.

Definition 2.4.1 (way-below relation, [12, Definition I-1.1.])

Let $(P,\leq)$ be a poset. We say that x is *way below* y, and we write $x\ll y$, if and only if for all directed subsets $\Delta\subseteq P$ for which $\sup\Delta$ exists, the relation $y\leq\sup\Delta$ always implies the existence of a $d\in\Delta$ with $x\leq d$.

In simpler terms, $x\ll y$ means that x is not only below y in the partial order, but also that x can be "reached" from any directed set that has y as its supremum. In other words, any computation of y involves an element d greater than x. If $x\ll y$, we also say that "x approximates y".

Note that the way-below relation is not necessarily a preorder. Indeed, although it is *transitive* (if $x\ll y$, $y\ll z$, Δ is directed and $s=\sup\Delta, z\leq s\implies\exists d\in\Delta$ such that $y\leq d\leq s\implies\exists d'\in\Delta$ such that $x\leq d'$), it is not always *reflexive* since $x\leq\sup\Delta$ does not necessarily imply the existence of $d\in\Delta$ with $x\leq d$. For example, if $x=\sup\Delta\notin\Delta, x\leq\sup\Delta$ but $\nexists x\leq d\in\Delta$ since otherwise $x\leq d$ and $d\leq x\implies x=d\in\Delta$ which contradicts our hypothesis.

Definition 2.4.2 (compacity, [2, Definition 1.1.10])

Let $(D, \leq)$ be a dcpo. An element $d \in D$ is called *compact* if $d \ll d$, that is, for every directed subset $\Delta \subseteq D$, the following implication holds :

$$d \leq \sup \Delta \Longrightarrow \exists x \in \Delta,\ d \leq x.$$

We denote $\mathcal{K}(D)$ the collection of compact elements of D.

Example 2.4.1

$*$ In $\mathbb{R}$ with the usual order $\leq$, there are no compact elements. Indeed, if $x \in \mathbb{R}$, $\Delta = \left\{x - \frac{1}{n} : n \in \mathbb{N}\right\}$ is a directed set such that $x = \sup \Delta$, but $x \not\leq x - \frac{1}{n}$ for all $n \in \mathbb{N}$, so x is not compact.

$*$ Let M be a nonempty set and consider the flat cpo $M_\perp$ defined in Example 2.2.3. Then $\mathcal{K}(M_\perp) = M_\perp$. It is obvious that $\perp \ll \perp$. On the other hand, let $x \in M$ and Δ be a directed set such that $x \leq \sup \Delta$. The only directed sets verifying this condition are $\{x\}$ and $\{\perp, x\}$. In any case, $x \in \Delta$ so x is also compact.

Let P be a poset. In analogy with the notation we used for upper sets, we write :

$$\twoheaduparrow x = \{u \in P \mid x \ll u\} \quad \text{and} \quad \twoheaddownarrow x = \{v \in P \mid v \ll x\}$$

Now let's look at two important definitions that use this relation.

Definition 2.4.3 (continuity, [12, Definition I-1.6])

A poset $(P, \leq)$ is said to be continuous if
$*$ $\twoheaddownarrow x$ is directed
$*$ $x = \sup \twoheaddownarrow x$
for all $x \in P$.

Definition 2.4.4 (algebraicity, [2, Definition 1.1.12])

A dcpo $(D, \leq)$ is called *algebraic*, if for all $x \in D$ the set

$$\{d \in \mathcal{K}(D) \mid d \leq x\}$$

is directed and has a supremum x.

The following proposition establishes that algebraicity is a stronger property than continuity.

Proposition 2.4.1 ([12, Proposition I-4.3.])

A poset $(P, \leq)$ is algebraic if and only if it is continuous and

$$x \ll y \text{ if and only if there is a compact element} k \in P \text{ such that } x \leq k \leq y.$$

With all this in hand, we can finally define Scott domains.

Definition 2.4.5 (Scott domain, [2, Definition 1.4.9])

A dcpo $(D, \leq)$ is called *bounded complete*, if every upper bounded subset $S \subseteq D$ has a supremum. Bounded complete and algebraic cpos are called *Scott domains* (or simply *domains*).

Example 2.4.2

If M is a nonempty set then $(M_\perp, \preceq)$ is a domain. Notice that the only upper bounded subsets of $M_\perp$ are of the form $\{\perp, x\}$ where $x \in M_\perp$ so $(M_\perp, \preceq)$ is bounded complete. Given $x \in M_\perp$ then $\{d \in \mathcal{K}(M_\perp) \mid d \preceq x\} = \{\perp, x\}$, by Example 2.4.1. Obviously, this set is directed and its supremum is x. Hence $(M_\perp, \preceq)$ is algebraic.

We would like to draw the reader's attention to the use of the word *domain*. Its meaning depends on the paper or monograph. In the first version of the book [12], domains were continuous lattices, but domains in the previous sense are continuous lattices without a top (see [12, Proposition I-1.25]). In the new version of the book, domains are "just" continuos dcpos [12, Definition I-1.6].

The following theorem tells us that, under certain conditions, the structural properties of domains extend to the set of continuous functions between them. Its demonstration can be found in [12].

Theorem 2.4.1 ([12, Theorem II-2.12])

If $(D, \leq)$ is a continuous dcpo (resp. an algebraic dcpo) and $(D', \leqslant)$ is a continuous lattice (resp. an algebraic lattice) then $D \to D'$ is a continuous lattice (resp. an algebraic lattice).

As we will see, domains provide the appropriate framework for denotational semantics.

2.5 Scott convergence

It is well-known that every topology has an associated notion of convergence, and the knowledge of this convergence helps deal with several aspects of the topology. However, obtaining a concise characterization of the topological convergence is not always possible. In this section, we analyze the convergence in the Scott topology. Our development is based on [12, Chapter II]. We first recall some basic notions that can be consulted in topological monographs like [10].

Definition 2.5.1 (net)

A net on a nonempty set X is a function $x : (I, \leqslant) \to X$ whose domain $(I, \leqslant)$ is a directed proset. We will denote $x(i)$ by x_i and the net x by $(x_i)_{i \in I}$.

Moreover, given $A \subseteq X$ we say that $x_i \in A$ eventually if there exists $i_0 \in I$ such that $x_i \in A$ for all $i \geqslant i_0$.

We next recall how to define a convergence of nets in a topological space.

Definition 2.5.2 (convergence of nets in topological spaces)

Let (X, Ω_X) be a topological space. A net $(x_i)_{i \in I}$ on X is aid to be convergent to $x \in X$ if

$$\forall A \in \Omega_X \text{ such that } x \in A, x_i \in A \text{ eventually.}$$

Equivalently,

$$\forall U \in \mathcal{N}_x, x_i \in U \text{ eventually}$$

where $\mathcal{N}_x$ is the neighborhood system at x.

Let's recover a characterization of a particular class of continuous dcpos to motivate the subsequent definition of Scott convergence.

Proposition 2.5.1 ([12, Theorem II-1.14.])

Let $(D, \leq)$ be a complete semilattice, that is, a dcpo such that every nonempty subset has infimum. Then $(D, \leq)$ is continuous if and only if

$$x = \sup\left\{\inf A \mid x \in A \in \Omega_S(D)\right\}$$

for every $x \in D$.

We have already mentioned that, in a dcpo, suprema of directed sets are limits in the Scott topology. The above result shows that, in a continuous complete semilattice, every element is a supremum of elements below it. This is one of the most important ideas behind the definition of continuity (see 2.4.3) and it leads naturally to the question of whether, in the Scott topology, each element is the limit of elements below it. We discuss this in the following.

Suppose that $(x_i)_{i \in I}$ is a net converging to x in the Scott topology of a continuous complete semillatice $(D, \leq)$. If A is a Scott open set with $x \in A$, then we can find $i_0 \in I$ such that $x_i \in A$ for all $i \geqslant i_0$. Then

$$\inf A \leq \inf\{x_i \mid i_0 \leqslant i\}$$

so by Proposition 2.5.1

$$x = \sup\left\{\inf A \mid x \in A \in \Omega_S(D)\right\} \leq \sup\left\{\inf\{x_i \mid j \leqslant i\} \mid j \in I\right\}.$$

This leads to the following definition.

Definition 2.5.3 ([12, Definition II-1.1])

Let $(D, \leq)$ be a complete semilattice. Given a net $(x_i)_{i \in I}$ its lower limit is defined as

$$\liminf_{i \in I} x_i = \sup_{j \in I} \inf_{i \geqslant j} x_i.$$

Then we say that $(x_i)_{i\in I}$ is Scott convergent to x if

$$x \leq \liminf_{i\in I} x_i.$$

The following result establishes a relationship between the Scott convergence and the convergence in the Scott topology.

Proposition 2.5.2

Let $(D, \leq)$ be a complete semilattice. If $(x_i)_{i\in I}$ is a Scott convergent net on D then it is also convergent in the Scott topology.

Proof. Suppose that $(x_i)_{i\in I}$ is Scott convergent to $x \in D$, that is, $x \leq \liminf_{i\in I} x_i$. Let A be a Scott open set such that $x \in A$. Since $x \leq \liminf_{i\in I} x_i$ and A is an upper set then

$$\liminf_{i\in I} x_i = \sup_{j\in I} \inf_{i\geqslant j} x_i \in A.$$

Notice that $\Delta = \left\{\inf_{j\leqslant i} x_i \mid j \in I\right\}$ is directed. In fact, for all pair of elements $\inf_{j_1\leqslant i} x_i$ and $\inf_{j_2\leqslant i} x_i$ in Δ, $\exists j_3 \in I$ such that $j_1 \leqslant j_3$ and $j_2 \leqslant j_3$ which implies $\inf_{j_1\leqslant i} x_i \leq \inf_{j_3\leqslant i} x_i$ and $\inf_{j_2\leqslant i} x_i \leq \inf_{j_3\leqslant i} x_i$.

Since $\sup \Delta = \liminf_{i\in I} x_i \in A$, which is Scott open, we can find $j_0 \in I$ such that $\inf_{i\geqslant j_0} x_i \in A$. Using again that A is an upper set, we deduce that $x_i \in A$ for all $i \geqslant j_0$. Therefore, $(x_i)_{i\in I}$ converges to x in the Scott topology. □

It would be nice if the above proposition provided a characterization of the convergence in the Scott topology, that is, if the converse of the above statement was true. However, it does not hold in general. Equivalence is only true when D is continuous.

Theorem 2.5.1 ([12, Theorem II-1.9])

A dcpo $(D, \leq)$ is continuous if and only if the Scott convergence is equal to convergence in the Scott topology.

Proof. The proof of this theorem can be found in [12]. □

Chapter 3

Models of the untyped λ-calculus

In mathematics, it is usual to interpret a function f between two sets A and B as a graph, i.e. a subset G_f of the cartesian product $A \times B$ such that, if $(a,b),(a,c) \in G_f$, then $b = c$, where $a \in A, b, c \in B$.

However, we can also interpret a function f as a rule for transforming elements of A into elements of B. This approach is nearer to the computational interpretation of procedures and programs. In this way, λ-calculus can be seen as an appropriate framework for the study of functions as rules rather than graphs. This theory is attributed to Alonzo Church, who designed it in the 1930s [7, 8], although other researchers like Schönfinkel and Curry also participated in its development.

The motivation for this theory was to provide a formulation for the theory of types introduced by Russell to avoid the contradictions encountered in the beginnings of set theory [8], like the famous Russell's paradox, which can be stated as follows.

Let S be the set of all sets that are not members of themselves, that is, $S = \{x | x \notin x\}$. From this, we deduce that

$$S \in S \Longleftrightarrow S \notin S,$$

which is a contradiction. This paradox comes from the original naive set theory and the axiom of unrestricted comprehension assumed in its origins, which asserts that, given a predicate P, there exists a set A containing all the objects satisfying P (if $P(x) =$ "$x \notin x$", we obtain the set S that appears in Russell's paradox).

How to prevent this paradox? We see that the reason for it is that, in the definition of S, we are not specifying the universe to which x belongs, so x can be arbitrary. Zermelo proposed a solution by replacing the axiom schema of unrestricted comprehension with the axiom schema of specification (also known as the axiom schema of separation). This axiom asserts that any definable subclass of a set is a set. In this case, this means that you cannot define a set as all objects satisfying a property. Rather, you can always construct the subset of a set satisfying certain properties. With this axiom, we cannot construct S. In fact, we can construct $S' = \{x \in A | A \text{ is a set}, x \notin x\}$, but this does not lead to a paradox. If $S' \in S'$ since, we have that $S' \in A$ and

$S' \not\in S'$, from which we conclude that $S' \notin A$. Conversely, if $S' \in A$ and $S' \not\in S'$ leads to $S' \in S'$ which is a contradiction so S' can not belong to A. This avoids the paradox.

Different solutions to this paradox were provided by other researchers. If we analyze Russell's paradox, the difficulty comes from the fact that we are applying a predicate to itself. Concretely, if S is the set of objects x that make $P(x)$ true, $P(S)$ violates the principle of non-contradiction. This is not a particularity of this paradox. Others, like the Burali-Forti paradox, have similar characteristics. As Russell asserted in [21]

> *In all the above contradictions there is a common characteristic, which we may describe as self-reference or reflexiveness. [...] In each contradiction something is said about all cases of some kind, and from what is said a new case seems to be generated, which both is and is not of the same kind as the cases of which all were concerned in what was said.*

The solution proposed by Russell in the paper *Mathematical logic as bases of the theory of types* [21] published in 1908 and in the *Principia Mathematica* [28] was completely different from the axiomatic view of Zermelo. He introduced the *ramified theory of types* by establishing concrete domains (*ranges of significance*) for the variables appearing in predicates, removing the possibility of applying a predicate (function) to itself. These ranges of significance form a hierarchy of types. The focus of type theory is the concept of function, but not from a set-theoretic perspective, where functions are graphs, but from more procedural point of view, where functions are rules with domains over which they act. This theory was informally explained by Russell and we can summarize it as follows. There are objects called *individuals*, forming the lowest type of objects (say of type 0) which are neither propositions nor functions. Propositions and functions taking individuals as input are of type 1, propositions and functions taking propositions and functions of type 1 as input are of type 2, and so on... In this way, a propositional function $P(x)$ is valid if and only if takes an object of a lower type than itself. In this way, $P(x)$ must have a *type*. Notice that the previous discussion is closely related to the usual concept of function and arguments of functions : a function is different from a standard object and functions whose arguments are functions are different from functions acting on standard objects.

However, Russell's theory encountered several objections and difficulties. The ramified theory of types was refined by Chwistek and Ramsey in the 1920s resulting in the *simple type theory*. We will not deal with type theory in this work, but the interested reader can consult [28, 21, 19, 9, 16].

Later, as previously mentioned, Church [8] gave a formulation of the simple type theory by introducing the now-called λ-calculus, that can be considered a formal system for modeling functions as rules. It fills the gap left by Zermelo-Fraenkel's set-theoretic concepts of function and application, which are unable to describe processes such as recursive functions, which are very common in computer science. Originally, λ-calculus was typed. This mimics type theory where functions can only be applied to certain types of data. However, there also exists the *untyped λ-calculus* that can express more things than the typed one. This is the one we will consider. To see exactly how λ-calculus is related to type theory, one can consult [16, 9].

Church also introduced the concept of a λ-computable function (λ-definability), based on λ-calculus. In his famous paper *On computable numbers, with an application to the Entschei-*

dungsproblem [27], Alan Turing noticed that these λ-computable functions are exactly the computable functions by a Turing machine (Church was Turing's doctoral advisor). This allows us to claim that λ-calculus is a powerful model of computation, which can be viewed as an abstract paradigmatic programming language. In fact, programming languages such as Algol'60, Pascal or LISP have properties similar to λ-calculus (procedures can be arguments of procedures) [6]. This is why establishing a denotational semantics for λ-calculus is crucial and gives a denotational semantics for other programming languages.

Models of λ-calculus are mathematical structures that interpret the terms and operations of the λ-calculus (notions which have yet to be defined) in a way that satisfies its axioms and rules. It is Scott's original goal [23] to find these mathematical structures and this is precisely what motivated the entire development of Domain theory.

In this chapter, we will give some elements of λ-calculus, try to understand the originality of this theory, and then, we will construct Scott's D_∞ model [23, 14] which underpins the concept of "function" belonging to its own domain. Our basic references are [14, 6, 2].

3.1 Introduction to λ-calculus

Consider the everyday mathematical expression "$x - y$". This can be thought of as a function f of the variable x defined as $f(x) = x - y$ or as a function g of y defined as $g(y) = x - y$ which can also be written as $f : x \to x - y$ and $g : y \to x - y$. The major drawbacks are that we have to name each new function differently, and that the way we name the function is not systematic (we can call a function anything we like). This can get very messy when dealing with higher-order functions (functions that act on other functions). This was one of Church's motivations for creating the λ-notation, i.e. the syntax of the λ-calculus. Although it seems clumsier at first, it has the advantage of being systematic.

We mentioned the syntax of the λ-calculus. Indeed, like any programming language, the λ-calculus is defined on the basis of two components : syntax and semantics.

3.1.1 Syntax of the λ-calculus

The syntax of the untyped λ-calculus is given by the following grammar:

$$M ::= x \mid (M_1\ M_2) \mid (\lambda x.M_1)$$

where x is called a *variable*, $(M_1 M_2)$ is called an *application* and $(\lambda x.M_1)$ is called an *abstraction*. In other words, whether or not a λ-calculus expression is valid or not is determined by the inductive definition above. A valid λ-calculus expression is called a "λ-term" and the set of all λ-terms is denoted by Λ.

Parentheses are very often omitted according to the rule of association to the left. For example, $MNPQ$ denotes the λ-term $(((MN)P)Q)$. Other abbreviations very frequently used are: $\lambda x.PQ$ for $(\lambda x.(PQ))$ and $\lambda x_1 x_2 \ldots x_n.M$ for $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$.

Syntactic identity of λ-terms

If M and N are λ-terms, $M \equiv N$ means that M is exactly the same term as N. By identification, if $MN \equiv PQ$ then $M \equiv P$ and $N \equiv Q$, and if $\lambda x.M \equiv \lambda y.P$ then $x \equiv y$ and $M \equiv P$. Note also that variables are distinct from constants, applications are distinct from abstractions, etc.

3.1.2 Semantics of the λ-calculus

Let's now look at the meaning of λ-terms. Before diving into more complex material, let's explain what their three main components are, namely variables, applications and abstractions.

Variables

Variables here correspond to both mathematical and computer variables. If we think of the λ-calculus as a formal system, it's clear that its variables are symbols representing a mathematical object. But if we see λ-calculus as a programming language, a variable can be considered an abstract storage location (we say abstract because the physical location is not necessarily known until the variable is actually used) associated with a symbolic name. It can be seen as a container for a particular set of bits referred to as the variable's value. Recall that the set of possible values a variable can hold is precisely its data type. Moreover, t he set of variables in λ-calculus is countable.

Applications

An application can be thought of as a function call. It is then clear what M_1 and M_2 mentioned above correspond to: the former is the function called and the latter is its argument, i.e. what is "passed" into the function. So, for example, fx is the application of the λ-term f to the λ-term x, which happens to be a variable but could have been a "composite" λ-term.

Abstractions

Abstractions (or function abstractions) correspond to function definitions. They are made up of two elements: a formal parameter (located between the λ and . symbols) and a body (what appears next). Note that all function abstractions are anonymous.

Let's look at some examples of λ-terms, the functions to which they correspond and their implementation in pseudo-code.

Bound and free variables

In the λ-calculus, like in other programming languages, there are two kinds of variable occurrences: *variable declaration* and *variable use.* This can easily be seen in the following piece of pseudo-code.

```
def f(x): return x + y;
```

λ-expression	**corresponding function**	**pseudo-code implementation**
$\lambda x.x$	the identity function: $f(x) = x$	`def f(x):  return x;`
$\lambda y.y$	once again, the identity function	`def f(y):  return y;`
$\lambda x.y$	the function of x constantly equal to y: $f(x) = y$	`def f(x):  return y;`
$\lambda x.\lambda y.y$	the function of x that returns the identity function	`def g(y):  return y; def f(x):  return g;`

Figure 3.1 : Comparison of λ-terms, their corresponding functions, and their implementation in pseudo-code

which is written $\lambda x.x + y$ using the λ-notation. In this function, there are two occurrences of the variable x. The first occurrence, enclosed within parentheses, serves as the declaration of the parameter x, introducing the variable into the program. The second occurrence of x serves as a use, specifically as an operand for an addition operation.

In most programming languages, a variable declaration establishes a scope for that variable, defining where it can be used within the program. In the provided example, the scope of the variable x is limited to the body of the function. This concept naturally extends to λ-calculus.

Definition 3.1.1 (scope, [14, Definition 1.9])

For a particular occurrence of $\lambda x.M$ in a term, the occurrence of M is called the scope of the occurrence of λx on the left.

Definition 3.1.2 (bound, binding, free occurrences, [14, Definition 1.9])

Let x be a variable in a λ-term P. If an occurrence of x is in the scope of some λx, it is said to be bound. The occurrence of x in λx is called a binding occurrence (which is also considered as bound). Otherwise, x is a free occurrence in P.

Definition 3.1.3 (bound, free variables, [14, Definition 1.9])

If x occurs bound at least once in P (if and only if there is at least once binding occurrence of x), it is called a bound variable in P. If x occurs free at least once, it is called a free variable. The set of free variables in P is denoted by $FV(P)$.

In the short program above, the declaration of x within parentheses is its binding occurrence. Consequently, any use of x within the scope of this declaration is bound to it. However, the occurrence of y remains free in the body of the function, as there is no corresponding binding occurrence of y within this scope.

To avoid ambiguity in programming, each occurrence of a variable must be bound to a specific declaration. This is typically governed by a binding scheme defined by the programming language. The λ-calculus, like many modern programming languages, employs static binding wherein each variable use is bound to the declaration of the same name in the nearest enclosing abstraction. Let's give an example.

Consider the λ-expression: $\lambda y.(\lambda x.x(yx)) \equiv \lambda yx.x(yx)$. This λ-expression is a λ-abstraction (i.e. a function definition) with parameter y and body the application of the identity function to the application (yx). In this expression, the leftmost occurrence of y is the binding occurrence of the variable y: its scope encompasses the entire expression $(\lambda x.x(yx))$. In particular, the rightmost occurrence of y is bound to this leftmost declaration. However, if the first occurrence of x is a declaration and the second occurrence is bound to the first, the third occurrence is free because it lies outside the scope of the declaration (which is only the λ-abstraction $\lambda x.x$). So if we mark binding occurrences in yellow-green, bound variables in blue and free variables in red, we obtain: $\lambda yx.x(yx)$. This illustrates the possibility of variables being both free and bound within the same expression.

α-conversion

When a variable x occurs bound in a λ-term, it means that x appears within the body of the corresponding function and refers to a parameter of the function. Therefore, the value of x is completely determined by and is equal to the argument which is passed into the function when it's called. This is why, although the variable x is different from the variable y, the λ-abstractions $\lambda x.x$ and $\lambda y.y$ are exactly the same. The process of systematically renaming bound variables within λ-expressions while preserving their meaning, which allows us to rename $\lambda y.y$ into $\lambda x.x$, is called the α-conversion. In particular, all free variables must remain free after an α-conversion.

Consider the following expression: $\lambda x.yx$, i.e. the application of the λ-abstraction that always return y to the variable x. In this expression, the first occurrence of x is a binding occurrence but the occurrence of y and the second occurrence of x are free. We can perfectly α-convert this expression to $\lambda z.yx$: both expressions "return" the value y and the free variables remain free. However, we can't substitute the variable y for the first occurrence of x, as this would change the meaning of the expression (it would return x instead of y) and the variable y in the body of the abstraction would go from being free to bound (when this happens we say that the variable is captured). This naturally leads to the following definition.

Definition 3.1.4 (congruence, [14, Definition 1.17])

Let P and Q be two λ-terms. We say that P is congruent to Q, and we write $P \equiv_\alpha Q$, if and only if P α-converts to Q, that is P can be changed to Q by a finite (or empty) series of α-conversions.

For example,

$$\begin{aligned}\lambda xy.x(xy) &\equiv \lambda x.(\lambda y.x(xy))\\ &\equiv_\alpha \lambda x.(\lambda v.x(xv))\\ &\equiv_\alpha \lambda u.(\lambda v.u(uv))\\ &\equiv \lambda uv.u(uv)\end{aligned}$$

Lemma 3.1.1

The binary relation $\equiv_\alpha$ is a relation of equivalence. That is, for all λ-terms P, Q, R, we have:

* $P \equiv_\alpha P$: reflexivity
* $P \equiv_\alpha Q, Q \equiv_\alpha R \Longrightarrow P \equiv_\alpha R$: transitivity
* $P \equiv_\alpha Q \Longrightarrow Q \equiv_\alpha P$: symmetry

Proof.

reflexivity

P can trivially be changed to itself by an empty series of α-conversions.

transitivity

If $P \equiv_\alpha Q$ and $Q \equiv_\alpha R$, then there is a finite (or empty) series of α-conversions which transforms P into Q and another series which converts Q into R so the concatenation of these two series is a finite or empty series which transforms P into R, i.e. $P \equiv_\alpha R$.

symmetry

If P goes to Q by a change of bound variables, further changes can be found that bring Q back to P. Indeed, any α-conversion α_i can easily be reversed (say we substituted the variable y for the variable x, we just need to substitute x for y). Let's write this procedure α_i^{-1}. If we denote by $(\alpha_1, \alpha_1, \ldots, \alpha_n)$ the (finite) series of α-conversions transforming P into Q, then its common sense that $(\alpha_n^{-1}, \alpha_{n-1}^{-1}, \ldots, \alpha_1^{-1})$ takes Q back to P.

□

In a sense, we can say that $\equiv_\alpha$ extends the syntactic identity $\equiv$. Most times, it proves to be more relevant than it.

Substitution-based model of evaluation

So far, we have seen the syntax of λ-calculus and the meaning of λ-expressions. We also explained the process of renaming the parameter in a function abstraction, called α- conversion. We now have all the tools we need to calculate with λ-terms, i.e. substitute values for variables, which reminds us of executing function calls. Let's give a formal procedure for doing so.

Definition 3.1.5 (substitution, [14, Definition 1.12])

For any variable x and λ-terms M and N, define $[N/x]M$ to be the result of substituting N

for every free occurrence of x in M and performing the proper α-conversions (i.e. changing bound variables) to avoid clashes. Formally, this means:

a. $[N/x]x \equiv N$
b. $[N/x]a \equiv a$ for all variables $a \not\equiv x$
c. $[N/x](P\,Q) \equiv ([N/x]P\,[N/x]Q)$
d. $[N/x](\lambda x.P) \equiv \lambda x.P$
e. $[N/x](\lambda y.P) \equiv \lambda y.P$ if x doesn't occur free in P
f. $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ if x occurs free in P and y doesn't occur free in N
g. $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ if x occurs free in P and y occurs free in N

Let's take a closer look at the last clause. Informally, this means that if x occurs free in P and y occurs free in N, to avoid the free occurrences of y in N from depending on the binding occurrence of y in $\lambda y.P$, we must first α-convert y to z in this abstraction.

As an example, let us evaluate the following substitutions:

$$\begin{aligned}
[(uv)/x](\lambda y.x(\lambda w.vwx)) &\equiv [(uv)/x]\lambda y.x\,[(uv)/x](\lambda w.vwx) && \text{by c.}\\
&\equiv \lambda y.[(uv)/x]x\,(\lambda w.[(uv)/x]vwx) && \text{by f.}\\
&\equiv \lambda y.uv\,(\lambda w.vw(uv)) && \text{by a. and c.}
\end{aligned}$$

$$\begin{aligned}
[(\lambda y.xy)/x](\lambda y.x(\lambda x.x)) &\equiv [(\lambda y.xy)/x]\lambda y.x\,[(\lambda y.xy)/x](\lambda x.x) && \text{by c.}\\
&\equiv \lambda y.[(\lambda y.xy)/x]x\,(\lambda x.x) && \text{by f. and d.}\\
&\equiv \lambda y.(\lambda y.xy)\,(\lambda x.x) && \text{by a.}
\end{aligned}$$

$$\begin{aligned}
[(\lambda y.vy)/x](y(\lambda v.xv)) &\equiv [(\lambda y.vy)/x]y\,[(\lambda y.vy)/x](\lambda v.xv) && \text{by c.}\\
&\equiv y\,(\lambda z.[(\lambda y.vy)/x][z/v](xv)) && \text{by b. and g.}\\
&\equiv y\,(\lambda z.((\lambda y.vy)z)) && \text{by c. and a.}
\end{aligned}$$

In the substitution above, if $[(\lambda y.vy)/x](\lambda v.xv)$ was evaluated by f, we would get $\lambda v.((\lambda y.vy)\,v)$ and the second occurrence of v would be bound when it shouldn't be. Also note that the act of replacing $\lambda x.M$ by $\lambda y.[y/x]M$ (assuming that y doesn't occur free in M) is what we previously defined as an α-conversion.

Lemma 3.1.2 ([14, Lemma 1.21])
$M \equiv_\alpha M'$ and $N \equiv_\alpha N' \Longrightarrow [N/x]M \equiv_\alpha [N'/x]M'$

Proof. We won't give the proof here but it can be found in [14] on page 279. □

This lemma tells us that the operation of substitution is well-behaved with respect to congruence: if the λ-terms involved in a substitution are congruent to others then the output of the substitution is congruent to the substitution involving these other terms. So from now on, when a bound variable in a term P threatens to make a particular substitution complicated, we will consider it perfectly fine to replace it with a new "harmless" variable.

Now that we have introduced the substitution, we can give the proof of the following result, which we'll need later.

Lemma 3.1.3 ([14, Lemma 1.19])

Let P and Q be two λ-terms. If $P \equiv_\alpha Q$ then $FV(P) = FV(Q)$.

Proof. α-converting P to Q means replacing a term of the form $\lambda x.M$ in P by $\lambda y.[y/x]M$, where $y \not\equiv x$, $y \notin FV(xM)$ and x and y are not bound in M. In this case, $[y/x]M$ is obtained by simply changing x to y throughout M. Indeed, looking at definition 3.1.5, we can see that we find ourselves in one of the three cases a, b and c (eventually leading to a and/or b) which boil down to simply substituting y for x throughout M. Hence, if $P \equiv_\alpha Q$, there exists a finite (perhaps empty) series of α-conversions changing P to Q, and each of these conversions only involve the replacement of some bound occurrences of a variable x_i by another variable x_{i+1}. Therefore, any free variable in P remains free in Q and conversely. $\square$

β-reduction

Now that we know how to perform a substitution, we can define a rule for evaluating applications, which is at the heart of λ-calculus. This rule is called β-reduction and an expression to which the rule can be applied, a β-redex (which is short for β-reduction expression). We will see that a β-redex is a λ-expression of a specific form, namely, an application in which the first term is an abstraction.

Recall that a λ-term of form $(\lambda x.M)N$, which we previously called an application, represents the action of an operator $\lambda x.M$ on an argument N. We also know that $\lambda x.M$ reads "the function of x that returns M". So, we would like $(\lambda x.M)N$ to be calculated by substituting N for x in M, that is evaluating $[N/x]M$. This process is captured in the following definitions.

Definition 3.1.6 (β-redex, β-contraction, β-reduction, [14, Definition 1.24])

Any λ-term of the form $(\lambda x.M)N$ is called a β-redex and the term $[N/x]M$ is called its contractum. We say that a term P β-contracts to P', and we note $P \triangleright_{1\beta} P'$, if and only if P contains an occurrence of $(\lambda x.M)N$, we replace that occurrence by its contractum $[N/x]M$ and the result is P'. We say that a term P β-reduces to Q and we note $P \triangleright_\beta Q$ if and only if it can be changed to Q by a finite (or empty) series of β-contractions.

Definition 3.1.7 (β-normal form, [14, Definition 1.26])

A λ-term Q which contains no β-redexes is called a β-normal form (or β-nf for short). If P β-reduces to a β-nf Q, we say that Q is a β-normal form of P.

For example, let us reduce the following λ-terms.

$$\begin{aligned}(\lambda x.xy)(\lambda u.vuu) &\rhd_{1\beta} [(\lambda u.vuu)/x](xy)\\ &\equiv (\lambda u.vuu)y\\ &\rhd_{1\beta} [y/u](vuu)\\ &\equiv vyy\end{aligned}$$

$$\begin{aligned}(\lambda xy.yx)uv &\equiv (\lambda x.(\lambda y.yx)u)v\\ &\rhd_{1\beta} ([u/x](\lambda y.yx))v\\ &\equiv (\lambda y.[u/x](yx))v\\ &\equiv (\lambda y.yu)v\\ &\rhd_{1\beta} [v/y](yu)\\ &\equiv vu\end{aligned}$$

$$\begin{aligned}(\lambda x.x(x(yz))x)(\lambda u.uv) &\rhd_{1\beta} [(\lambda u.uv)/x](x(x(yz))x)\\ &\equiv (\lambda u.uv)((\lambda u.uv)(yz))(\lambda u.uv)\\ &\rhd_{1\beta} (\lambda u.uv)([yz/u](uv))(\lambda u.uv)\\ &\equiv (\lambda u.uv)((yz)v)(\lambda u.uv)\\ &\rhd_{1\beta} [((yz)v)/u](uv)(\lambda u.uv)\\ &\equiv ((yz)v)v(\lambda u.uv)\\ &\equiv yzvv(\lambda u.uv)\end{aligned}$$

$$\begin{aligned}(\lambda x.xxy)(\lambda y.yz) &\rhd_{1\beta} [(\lambda y.yz)/x](xxy)\\ &\equiv (\lambda y.yz)(\lambda y.yz)y\\ &\equiv ((\lambda y.yz)(\lambda y.yz))y\\ &\rhd_{1\beta} ([(\lambda y.yz)/y](yz))y\\ &\equiv ((\lambda y.yz)z)y\\ &\rhd_{1\beta} ([z/y](yz))y\\ &\equiv zzy\end{aligned}$$

Note that, each time, we stopped when we reached a β-normal form. In particular, $yzvv(\lambda u.uv)$ is a β-nf since it contains no β-redex which is of the form $(\lambda x.M)N$ but not $N(\lambda x.M)$. We would also like to point out that some β-reductions require α-conversions during their process, as in the following.

$$\begin{aligned}(\lambda xy.xyy)(\lambda u.uyx) &\equiv (\lambda x.(\lambda y.xyy))(\lambda u.uyx)\\ &\rhd_{1\beta} [(\lambda u.uyx)/x](\lambda y.xyy)\\ &\equiv \lambda z.[(\lambda u.uyx)/x][z/y](xyy)\\ &\equiv \lambda z.[(\lambda u.uyx)/x](xzz)\\ &\equiv \lambda z.(\lambda u.uyx)zz\\ &\rhd_{1\beta} \lambda z.[z/u](uyx)z\\ &\equiv \lambda z.zyxz\end{aligned}$$

Sometimes, a λ-term has several β-redexes in it. It is legitimate to ask if we obtain the same λ-nf depending on the order in which we perform the β-contractions. As we expect, the answer is yes. Indeed, an application is meant to represent a computation and how we evaluate a computation does not change its result.

The Church-Rosser theorem, presented below, demonstrates that the normal form of a term is indeed unique, as long as we disregard changes in bound variables. This theorem is arguably the most frequently cited theorem in λ-calculus.

3.1.3 Church–Rosser theorem

Note : In the follwing section, not all proofs have been given. They can be found in [14] at the location indicated in the heading of the lemmas/theorems concerned.

Let's start by giving two lemmas. The first one says that no free variable can appear through β-reduction and the second one that the reducibility relation $\rhd_\beta$ is preserved by substitution.

Lemma 3.1.4 ([14, Lemma 1.30])

Let P and Q be λ-terms. If $P \rhd_\beta Q$, then $FV(Q) \subseteq FV(P)$.

Proof. For this proof, we'll need two intermediate results (step 1 and 2) which have their own interest.

Step 1: $x \notin FV(M) \implies [N/x]M \equiv M$

The only two possibilities for x to not be free in M is for x to not occur in M outside the scope of some λx or for x to not occur in M at all. Looking at the corresponding cases in definition 3.1.5, namely cases b, d and c (eventually leading to b or d), we see that $[N/x]M \equiv M$.

Step 2: $x \in FV(M) \implies FV([N/x]M) = (FV(N) \cup (FV(M) - \{x\}))$

Let x be a free variable in M.

Let y be a free variable in $[N/x]M$. Therefore, either y occurs free in M and is different from x (in which case it will be replaced by N), or y occurs free in N and somehow remains free after the substitution. Hence, $FV([N/x]M) \subseteq (FV(N) \cup (FV(M) - \{x\}))$.

Conversely, if y occurs free in M and is different from x, then looking at the only two possible cases (a and c eventually leading to a) in 3.1.5, we see that y remains a free variable after the substitution. In the same way, if y occurs free in N, since x occurs free in M, looking at the only two possible cases (a and g) in 3.1.5, we see that y remains a free variable after the substitution. Thus, $FV([N/x]M) \supseteq (FV(N) \cup (FV(M) - \{x\}))$.

Which is why $x \in FV(M) \implies FV([N/x]M) = (FV(N) \cup (FV(M) - \{x\}))$.

Step 3: $P \rhd_{1\beta} Q \implies FV(Q) \subseteq FV(P)$

$P \rhd_{1\beta} P' \implies P$ contains an occurrence of the form $(\lambda x.M)N$ (and we replace that occurrence by $[N/x]M$). But if $x \notin FV(M)$, then, by step 1, $[N/x]M \equiv M \implies FV([N/x]M) = FV(M) \subseteq$

$FV((\lambda x.M)N)$ (in general we would have $FV(M) \subseteq FV((\lambda x.M)N) \cup \{x\}$ but in this case $x \notin FV(M)$). And if $x \in FV(M)$, then, by step 2, $FV([N/x]M) = (FV(N) \cup (FV(M) - \{x\})) \subseteq FV((\lambda x.M)N)$. Since α-conversions do not change the set of free variables by lemma 3.1.3, if $P \rhd_\beta Q$ and we note $P, P_1, P_2, \ldots, Q$ the finite (or empty) series of β-contractions reducing P to Q, then we have $FV(P) \supset FV(P_1) \supset FV(P_2) \ldots FV(Q)$. □

Lemma 3.1.5 ([14, Lemma 1.40])

If $P \rhd_\beta P'$ and $Q \rhd_\beta Q'$, then $[P/x]Q \rhd_\beta [P'/x]Q'$.

Definition 3.1.8 (confluent relation)

A binary relation $\lhd$ between λ-terms is said to be confluent if and only if, for all terms P, M, and N, it holds that $P \lhd M$ and $P \lhd N \implies$ there exists a λ-term T such that $M \lhd T$ and $N \lhd T$.

One way to visualize it is in figure 3.2 (where the straight lines represent the $\lhd$ relations assumed to exist, and the dotted lines the inferred relations of same nature):

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[dr] & \\
M \arrow[dr, dashed] & & N \arrow[dl, dashed] \\
 & T &
\end{tikzcd}$$

Figure 3.2 : Diagram illustrating the fundamental property of a confluent relation

That being said, we can formulate the Church–Rosser theorem.

Theorem 3.1.1 (Church–Rosser theorem, [14, Theorem 1.41])

The reducibility relation $\lhd_\beta$ between λ-terms is confluent.

3.2 D_∞ spaces

Let's recall a few ideas already mentioned in the introduction, but which deserve to be emphasized.

In the 1920s when λ-calculus was invented by Church, logicians did not automatically think of a function as a subset of a cartesian product. In fact, there are other ways of representing functions. One such representation is that of a function as an operation-process which can be defined by giving a set of rules describing how it acts on an input object (it may not produce an output for certain inputs). In reality, every time we define a function by a formula, e.g. $f(x) = 2x + 3$, rather than by extension (i.e. by giving all the ordered pairs $\{(0,3), (\frac{1}{2}, 4), (1,5), \ldots\}$

belonging to f), we refer to this concept of function as an operator-process.

This representation is particularly evident in computer programs, although it was not originally intended to have the finiteness and effectiveness limitations that are involved with computation. To avoid confusion, let's call « operator » the function-as-operation-process concept, and « function » or « map » the set-of-ordered-pairs concept.

One of the key distinctions between operators and functions is that an operator can be defined solely by its action, without specifying the set of inputs for which this action is valid (that is, without defining its domain). In this way, operators can be seen as « partial functions ». Another significant difference is that some operators have no domain restrictions: they can accept themselves as input, a property that is inherently impossible for functions.

In the previous chapter, we saw how operators can be formally defined using the untyped λ-calculus. A natural question to ask is whether this can also be done within the framework of standard ZF set theory, despite the obvious restrictions encountered with functions. The first answer to this question was provided by Dana Scott in 1969, when he proposed to interpret operators as infinite sequences of functions. This gave rise to the D_∞ spaces we shall now discuss.

3.2.1 Intuition

We mentioned the main limitation of functions in the ZF set theory, i.e. the impossibility of belonging to their own domain. This entails that classical mathematical spaces of functions are not useful for modeling the operator theory provided by the λ-calculus. More formally, what we seek to define models of the untyped λ-calculus is a cpo D isomorphic to the set $D \to D$ of functions from D to itself, that is, $D \cong D \to D$ (this equation is one of the most important domain equations [12, Chapter IV-7], [2, Chapter 7]). Obviously, $D = \{\bot\}$ is a trivial solution since there is exactly one function $f : \{\bot\} \to \{\bot\}$ but we are interested in a non-trivial solution D with more than just one element (otherwise, all λ-terms would be identified).

This problem was one of Scott's original motivations for developing the theory of domains [22, 23]. He brought a solution which can be illustrated using the Kleene fixed-point theorem (cf. 2.2.1). Let $(D, \leq)$ be a cpo and a continuous function $f : D \to D$. Since continuity implies monotony (cf. proposition 2.3.1), the following holds :

$$\bot \leq f(\bot) \leq f^2(\bot) \leq f^3(\bot) \leq \ldots$$

Then $\sup\{f^n(\bot) : n \in \mathbb{N}\} = \lim_{n\to+\infty} f^n(\bot) = x$ is a fixed point of f, i. e. $f(x) = x$.

Now consider a function $F : \mathsf{CPO} \to \mathsf{CPO}$, where CPO denotes all the cpos, such that $F(D) = D \to D$, i.e. the image of D by F is the family of self-functions on D. If we found a fixed-point $D_0 \in \mathsf{CPO}$ of f such that $F(D_0) = D_0$, then we would have a candidate for a solution to the previous equation. This immediately recalls the Kleene fixed-point theorem. Notice that F can be considered monotone since we can think of D as being included in $D \to D$ by assigning, to every element $d \in D$, the constant function equal to D. For this candidate to be a solution, it still needs to preserve directed suprema (which merits further comment in this context). To achieve it, a concept of limit of cpos is required.

To tackle these challenges, Scott considered $F(D) = D \to D$ as the family of all (Scott) continuous functions and the limit as an inverse limit (or projective limit) of cpos. With all these tools, he constructed the first non-trivial solution D_∞ to the domain equation $D \cong D \to D$ to serve as a model for the λ-calculus.

We develop this theory in the next sections based on [2, 14, 12, 23].

3.2.2 Projective limit

Taking into account the above comments, it is natural to construct a solution to the equation $D \cong D \to D$ as the limit of a sequence of cpos. But to do this, we need a concept of limit of cpos. This role will be played by the projective limit, which is usual in category theory and has several applications [12, Chapter IV-4]. A lot of constructions like terminal objects, products, equalizers, etc. are examples of such limits. However, we don't need this level of generality here so we give a simple definition of projective limit where the category of the domain is a directed set I and the codomain is the category of dcpos with Scott continuous functions.

Proposition 3.2.1 ([12, Proposition IV-4.3])

Let I be a directed set and $\mathscr{D} = \{D_i, p_{ij} : i, j \in I, j \geq i\}$ be a family where D_i is a dcpo for all $i \in I$ and $p_{ij} : D_j \to D_i$ is a (Scott) continuous function, for every $i, j \in I, j \geq i$. Let

$$D_\infty := \varprojlim \mathscr{D} := \left\{ (x_i)_{i\in I} \in \prod_{i\in I} D_i : p_{ij}\left(x_j\right) = x_i \text{ whenever } i \leq j \text{ in } I \right\}.$$

Then D_∞ is a dcpo called the projective limit (or the inverse limit) of $\mathscr{D}$.

Proof. D_∞ is endowed with the poitwise partial order $\sqsubseteq$ inherited from $\prod_{i\in I} D_i$.

Let Δ be a directed set in D_∞. Given $i \in I$, the set $\{d_i : d \in \Delta\}$ is directed in D_i so it has a supremum that we denote by s_i. Let $s = (s_i)_{i\in I}$. Notice that

$$p_{ij}(s_j) = p_{ij}(\sup\{d_j : d \in \Delta\}) = \sup\{p_{ij}(d_j) : d \in \Delta\} = \sup\{d_i : d \in \Delta\} = s_i$$

whenever $i, j \in I, i \leq j$. Consequently, $s \in D_\infty$. It is clear that $s = \sup \Delta$ so D_∞ is a dcpo. □

The above definition of projective limit also exists for other spaces than dcpos. The only difference relies on the property that the functions p_{ij} must satisfy. For example, in topological spaces, continuity is required.

Example 3.2.1 ([10, Examples 2.5.3, 2.5.4])

$*$ Let $\{D_n | n \in \mathbb{N}\}$ be a family of dcpos. For each $n \in \mathbb{N}$ define $E_n = D_1 \times \ldots \times D_n$. Moreover, if $n \leq m$ consider $p_{nm} : E_m \to E_n$ as $p_{nm}(d_1, \ldots, d_m) = (d_1, \ldots, d_n)$ for all $(d_1, \ldots, d_m) \in E_m$. If $\mathscr{E} = \{E_n, p_{nm} : n, m \in \mathbb{N}, n \leq m\}$ then

$$E_\infty = \varprojlim \mathscr{E} = \prod_{n\in\mathbb{N}} D_n.$$

$*$ Let $\{D_n | n \in \mathbb{N}\}$ be a family of dcpos such that $D_{n+1} \subseteq D_n$ for all $n \in \mathbb{N}$. Given $n \leq m$ consider

$p_{nm} : D_m \to D_n$ as the inclusion. If $\mathscr{D} = \{D_n, p_{nm} : n, m \in \mathbb{N}, n \leq m\}$ then

$$D_\infty = \varprojlim \mathscr{D} = \Big\{(x_1, x_2, \dots) \in \prod_{n\in\mathbb{N}} D_n : x_n = x_{n+1} \forall n \in \mathbb{N}\Big\} = \Big\{(x, x, \dots) \in \prod_{n\in\mathbb{N}} D_n : x \in \bigcap_{n\in\mathbb{N}} D_n\Big\}.$$

Hence D_∞ is isomorphic to $\bigcap_{n\in\mathbb{N}} D_n$.

3.2.3 Construction of D_∞

The first concept we need for constructing a solution to the equation $D \cong D \to D$ as a projective limit of cpos is that of injection-projection pair.

Definition 3.2.1 (retraction, injection-projection pair, [14, Definition 3.1.2])

Let $(D, \leq)$ and $(E, \leqslant)$ be two cpos. A retraction pair between D and E is a pair $(\phi : D \to E, \psi : E \to D)$, usually written $(\phi, \psi) : D \to_{\mathrm{r}} E$ such that $\psi \circ \phi = \mathrm{id}_D$. If also $\phi \circ \psi \leq \mathrm{id}_E$, then the pair $(\phi : D \to E, \psi : E \to D)$ is called an injection-projection pair and we write $(\phi, \psi) : D \to_{\mathrm{ip}} E$. Retraction and injection-projection pairs are composed component-wise: $(i_1, j_1) \circ (i_2, j_2) = (i_1 \circ i_2, j_1 \circ j_2)$.

Example 3.2.2

$*$ Let $(E, \leq)$ be a cpo and $D \subseteq E$. Consider $i : D \to E$ the inclusion and $p : E \to D$ given by

$$p(e) = \begin{cases} e & \text{if } e \in D, \\ \perp & \text{if } e \notin D, \end{cases}$$

for every $e \in E$. Then (i, p) is an injection-projection pair.

$*$ Let $f : X \to Y$ be a surjective function. Define $F : \mathscr{P}(X) \to \mathscr{P}(Y)$ and $F^{-1} : \mathscr{P}(Y) \to \mathscr{P}(X)$ as

$$F(A) = \big\{f(a) : a \in A\big\}, \quad F^{-1}(B) = \big\{x \in X : f(x) \in B\big\}$$

for every $A \in \mathscr{P}(X)$ and $B \in \mathscr{P}(Y)$. Then (F^{-1}, F) is an injection-projection pair between the cpos $(\mathscr{P}(Y), \subseteq)$ and $(\mathscr{P}(X), \supseteq)$.

We can also show that if D and E are cpos, the injection ϕ from D to $D +_c E$ ($+_c$ denotes the coalesced sum, the definition of which is given below 3.2.2), defined by $\phi(\perp_D) = \perp$ and $\phi(x) = (1, x)$ if $x \neq \perp$, is the first component of an injection-projection pair.

Definition 3.2.2 (coalesced sum)

Let $(D, \leq)$ and $(E, \leqslant)$ be two cpos. Their coalesced sum $D +_c E$ is the set defined by:

$$D +_c E = \{(1, x) \mid x \in D \setminus \{\perp_D\}\} \cup \{(2, y) \mid y \in E \setminus \{\perp_E\}\} \cup \{\perp\},$$

ordered as follows: $z_1 \preceq z_2$ if

* $z_1 = \bot$,
* $z_1 = (1, x_1)$ and $z_2 = (1, x_2)$ with $x_1 \leq x_2$,
* $z_1 = (2, y_1)$ and $z_2 = (2, y_2)$ with $y_1 \leqslant y_2$.

Let $\psi : D +_c E \to D$ be defined as follows: $\psi(\bot) = \bot_D$, $\psi(1, x) = x$ and $\psi(2, y) = \bot_D$. Now let's prove that ϕ and ψ form an injection-projection pair, that is $\psi \circ \phi = \mathrm{id}_D$ and $\phi \circ \psi \preceq \mathrm{id}_{D+_cE}$.

$\underline{\psi \circ \phi = \mathrm{id}_D}$

$$(\psi \circ \phi)(x) \quad = \quad \begin{cases} \psi(\phi(\bot_D)) = \psi(\bot) = \bot_D & \text{if } x = \bot_D \\ \psi(\phi(x)) = \psi((1, x)) = x & \text{otherwise} \end{cases} \quad = \quad \mathrm{id}_D(x)$$

$\underline{\phi \circ \psi \preceq \mathrm{id}_{D+_cE}}$

Let's examine three cases.

* if $z = \bot \in D +_c E$, then $(\phi \circ \psi)(\bot) = \phi(\bot_D) = \bot \preceq \bot = Id_{D+_cE}(\bot)$
* if $z = (1, x) \in D +_c E$, then $(\phi \circ \psi)(z) = \phi(\psi(z)) = \phi(x) = (1, x) \preceq (1, x)$ since $x \leq x$
* if $z = (2, y) \in D +_c E$, then $(\phi \circ \psi)(z) = \phi(\psi(z)) = \phi(\bot_D) = \bot \preceq (2, y)$

So $(\phi \circ \psi)(z) \preceq z, \forall z \in D +_c E \Longleftrightarrow \phi \circ \psi \preceq \mathrm{id}_{D+_cE}$.

Definition 3.2.3 (isomorphism, [14, Definition 16.20])

Let $(D, \leq)$ and $(E, \leqslant)$ be cpos. We say D is isomorphic to E, and write $D \cong E$, if and only if there exist $\phi \in (D \to E)$ and $\psi \in (E \to D)$ such that $\psi \circ \phi = \mathrm{id}_D$ and $\phi \circ \psi = \mathrm{id}_E$.

In our context, the main interest of an injection-projection pair $(\phi, \psi) : D \to_{\text{ip}} E$ is that it makes D isomorphic to the set $\phi(D) \subseteq E$ and it makes the bottom members of D and E correspond : $\phi(\bot_D) = \bot_E$ and $\psi(\bot_E) = \bot_D$. Let's prove this.

We already know that $\psi \circ \phi = \mathrm{id}_D$. If $y \in \phi(D) \subseteq E$, $\exists x \in D$ such that $\phi(x) = y \Longrightarrow (\phi \circ \psi)(y) = (\phi \circ \psi \circ \phi)(x) = \phi(x) = y$ so $\phi \circ \psi = \mathrm{id}_{\phi(D)}$. $\forall x \in D, \phi(\bot_D) \leq \phi(x)$ since ϕ is monotone. In particular, $\phi(\bot_D) \leqslant \phi(\psi(\bot_E)) = (\phi \circ \psi)(\bot_E) = \bot_E \Longrightarrow \phi(\bot_D) \leqslant \bot_E$. Conversely, $\forall y \in E, \psi(\bot_E) \leq \psi(y)$. In particular, $\psi(\bot_E) \leq \psi(\phi(\bot_D)) = (\psi \circ \phi)(\bot_D) = \bot_D \Longrightarrow \psi(\bot_E) = \bot_D$.

Lemma 3.2.1 ([14, Proposition 3.1.5])

Let $(D, \leq)$ and $(E, \leqslant)$ be cpos. If $(\phi, \psi) : D \to_{\text{ip}} E$, then ϕ determines ψ as follows:

$$\psi(x) = \sup\{y \in D \mid \phi(y) \leqslant x\} \qquad \forall x \in D.$$

Proof. We first show that, in an injection projection pair, (ϕ, ψ), ϕ determines ψ. Let's suppose that (ϕ, ψ') is another injection-projection pair. Then $\psi' = \mathrm{id}_D \circ \psi' = \psi \circ \phi \circ \psi' \leqslant \psi \circ \mathrm{id}_E = \psi$ and symmetrically $\psi \leqslant \psi'$. Since $\leqslant$ is a partial order, $\psi = \psi'$.

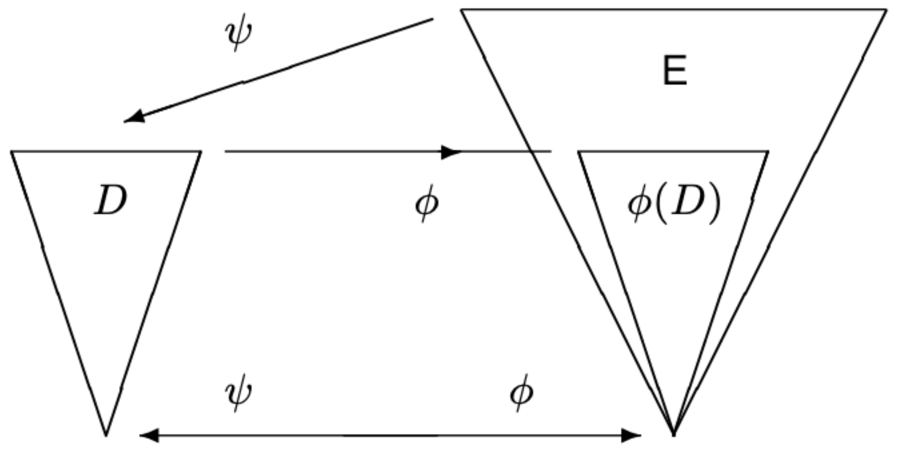

Figure 3.3 : An illustration of the action of an injection-projection pair [14]

We next check that (ϕ, ψ) is an injection-projection pair, where ψ is defined as in the statement. Well obviously, $(\psi \circ \phi)(x) = \psi(\phi(x)) = \bigvee\{y \in D \mid \phi(y) \leqslant \phi(x)\} = x$ and $(\phi \circ \psi)(y) = \phi(\bigvee\{z \in D \mid \phi(z) \leqslant y\}) \leqslant y$. □

Lemma 3.2.2 ([14, Definition 3.1.8, Definition 3.1.10])

Let $(D, \leq)$ and $(E, \leqslant)$ be cpos. Let $(\phi, \psi) : D \to_{\text{ip}} E$ be an injection-projection pair.
1. We can define a new injection-projection pair $(m, n) = (\phi, \psi) \to (\phi, \psi) : (D \to D) \to_{\text{ip}} (E \to E)$ by:

$$\begin{cases} m(u) = \phi \circ u \circ \psi \\ n(v) = \psi \circ v \circ \phi \end{cases}$$

for every $u \in (D \to D), v \in (E \to E)$.
2. The pair $(i_s, j_s) : D \to_{\text{ip}} (D \to D)$ defined by:

$$\begin{cases} i_s(x) = x \\ j_s(f) = f(\bot) \end{cases}$$

is also an injection-projection pair called the standard injection-projection pair.

Proof. 1. Let $\mathscr{D}$ be a directed subset of $(D \to D)$, which, by Lemma 2.3.5, is a cpo so $\sup\mathscr{D} = s$ exists. For each $e \in E$ we have

$$(m(s))(e) = (\phi \circ s \circ \psi)(e) = \phi(\sup\mathscr{D}(\psi(e))) = \phi(\sup_{d \in \mathscr{D}} d(\psi(e))) = \sup_{d \in \mathscr{D}} \phi(d(\psi(e))) = (\sup_{d \in \mathscr{D}} \phi \circ d \circ \psi)(e).$$

where we have used Scott continuity of ϕ. Hence m preserves directed suprema so it is Scott continuous by 2.3.1. A similar argument shows that n is also Scott continuous.

Moreover,

$$(n \circ m)(u) = n(\phi \circ u \circ \psi) = \psi \circ \phi \circ u \circ \psi \circ \phi = \mathrm{id}_D \circ u \circ \mathrm{id}_D = u$$

for all $u \in (D \to D)$ and

$$(m \circ n)(v) = m(\psi \circ v \circ \phi) = \phi \circ \psi \circ v \circ \phi \circ \psi \leqslant v \circ \phi \circ \psi \leqslant v,$$

the last inequality being deduced from $\phi \circ \psi \leqslant \mathrm{id}_E$ and v's monotony. Therefore, (m, n) is indeed an injection-projection pair between $D \to D$ and $E \to E$.

2. $(j_s \circ i_s)(x) = j_s(i_s(x)) = (i_s(x))(\bot) = x$ and $(i_s \circ j_s)(f) = i_s(j_s(f)) = i_s(f(\bot)) = \mathbb{1} \cdot f(\bot)$ which is of course point-wise less than f since f is continuous, in particular monotone. □

Definition 3.2.4 (D_∞ space, [14, Definition 3.1.10])

Let $(D, \leq)$ be a cpo and let $(D_n)_{n \in \omega}$ and $((i_n, j_n))_{n \in \omega}$ be the sequences given respectively by:

$$\begin{cases} D_{n+1} = D_n \to D_n, & D_0 = D \\ (i_{n+1}, j_{n+1}) = (i_n, j_n) \to (i_n, j_n), & (i_0, j_0) = (i_s, j_s) \end{cases}$$

so that $(i_n, j_n) : D_n \to_{\mathrm{ip}} D_{n+1}$ for all n. We define the D_∞ space as the inverse limit of $\{D_n, j_n : n \in \omega\}$, that is

$$D_\infty = \Big\{(x_0, \ldots, x_n, \ldots) \in \prod_{n \in \omega} D_n \mid x_n = j_n(x_{n+1}) \forall n \in \omega\Big\}.$$

Since $(D_\infty, \sqsubseteq)$ is a projective limit of cpos, then it is a cpo. It is usual to consider in this construction the cpo D_0 as the flat cpo $\mathbb{N}_\bot$ considered in Example 2.2.3 (see, for example, [14, Definition 16.22]).

We will see in the next section that D_∞ is indeed a solution to the domain equation, i.e. that we can write $D_\infty \cong D_\infty \to D_\infty$.

3.2.4 Properties of D_∞

From now on, we write y_n for the n-th component of $y \in D_\infty$.

Proposition 3.2.2 ([2, Lemma 3.1.12])

Let $x \in D_n$. There is a natural way to define an injection-projection pair $(i_{n\infty}, j_{n\infty}) : D_n \to_{\mathrm{ip}} D_\infty$ as :

$$\begin{cases} i_{n\infty}(x) = (k_{n0}(x), k_{n1}(x), k_{n2}(x), \ldots) \\ j_{n\infty}(y) = y_n \end{cases}$$

where $k_{nm} : D_n \to D_m$ is defined by :

$$k_{nm} = \begin{cases} j_m \circ \ldots \circ j_{n-2} \circ j_{n-1} & \text{if } n > m \\ \mathrm{id}_{D_n} & \text{if } n = m \\ i_{m-1} \circ \ldots \circ i_{n+1} \circ i_n & \text{if } n < m \end{cases}$$

Proof. The mappings k_{nm} are continuous (as compositions of continuous mappings). Moreover,

$$j_{n\infty} \circ i_{n\infty}(x) = j_{n\infty}\big((k_{n0}(x), k_{n2}(x), k_{n2}(x), \ldots)\big) = k_{nn}(x) = \mathrm{id}_{D_n}(x) = x$$

and

$$i_{n\infty} \circ j_{n\infty}(y) = i_{n\infty}(y_n) = (\ldots, j_{n-1}(y_n), y_n, i_n(y_n), \ldots) \sqsubseteq y$$

since $\forall m \geq 0, k_{nm}(y_n) \leq_n y_n$ (either $n \leq m$ and $k_{nm}(y_n) = y_m$ or $n > m$ and we can show by recurrence that $k_{nm}(y_n) \leq_m y_m$). Hence, $(i_{n\infty}, j_{n\infty}) : D_n \to_{\mathrm{ip}} D_\infty$ is, indeed, an injection-projection pair. □

In this way, given $n \in \omega$, we can embed D_n into D_∞ using $i_{n\infty}$. So $x \in D_n$ can be identified to its image $i_{n\infty}(x)$ and we can consider that x belongs to D_∞. Then we shall freely write x for $i_{n\infty}(x)$. Under this abuse of notation, the following holds $x \in D_n \Longrightarrow x = x_n$ (the first x refers to x as a member of D_n and the second to x as a member of D_∞).

That being said, given $x \in D_\infty$ we have that

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \qquad \text{and} \qquad x = \sup_{n\in\omega} x_n.$$

This matches Scott's idea of the computation of an element x as a sequence of successive approximations.

Lemma 3.2.3 ([2, Lemma 3.1.13])

The following properties hold:

1. $\forall n \leq p, x \in D_{n+1}, y \in D_p \quad x(y_n) = x_{p+1}(y)$
2. $\forall n \leq p, x \in D_{p+1}, y \in D_n \quad x_{n+1}(y) = x(y_p)_n$

Proof. The case $p = n$ is trivial according to what we've just said. We prove the other cases by recurrence over p. Suppose the equalities true for $n \leq p$.

1. Let $x \in D_{n+1}, y \in D_{p+1}$. Then $x_{p+2}(y) = (i_{p+1}(x_{p+1}))(y) = i_p \circ x_{p+1} \circ j_p(y) = i_p(x_{p+1}(y_p))$. Identifying $i_p(x_{p+1}(y_p)) \in D_{p+1}$ to

$$\begin{aligned} i_{(p+1)\infty}(i_p(x_{p+1}(y_p))) &= (k_{(p+1)0}(i_p(x_{p+1}(y_p))), \ldots, k_{(p+1)(p+1)}(i_p(x_{p+1}(y_p))), \ldots, k_{(p+1)m}(i_p(x_{p+1}(y_p))), \ldots) \\ &= (k_{p0}(x_{p+1}(y_p)), \ldots, k_{pp+1}(x_{p+1}(y_p)), \ldots, k_{pm}(x_{p+1}(y_p)), \ldots), \end{aligned}$$

that is, $x_{p+1}(y_p)$. So we have :

$$\begin{aligned} x_{p+2}(y) = i_p(x_{p+1}(y_p)) = x((y_p)_n) &= x(j_{n\infty}((k_{p0}(y_p), \ldots, k_{pn}(y_p), \ldots, k_{pp}(y_p), \ldots))) \\ &= x(k_{pn}(y_p)) \\ &= x((j_n \circ \cdots \circ j_{p-1})(y_p)) \\ &= x((j_n \circ \cdots \circ j_{p-1} \circ j_p)(y)) \\ &= x(k_{(p+1)n}(y)) \\ &= x(y_n) \end{aligned}$$

2. We proceed in the same way. □

Lemma 3.2.4 ([14, Definition 3.1.15, Lemma 3.1.16])

Let $\begin{array}{lll} \bullet : D_\infty \times D_\infty & \longrightarrow D_\infty \\ (x,y) & \longmapsto x \bullet y = \sup_{n\geq 0} x_{n+1}(y_n) \end{array}$

Then the following properties hold :

1. If $x \in D_{n+1}$, then $x \bullet y = x(y_n)$, so $x \bullet y = (x \bullet y)_n$.
2. If $y \in D_n$, then $(x \bullet y)_n = x_{n+1}(y)$.

Proof.

1. Using lemma 3.2.3 (1), we have :

$$x \bullet y = \sup_{p>n} x_{p+1}(y_p) = \sup_{p>n} x((y_p)_n) = x(y_n)$$

Hence, $(x \bullet y)_n = (x(y_n))_n = x(y_n) = x \bullet y$.

2. By continuity of $j_{n\infty}$ and by lemma 3.2.3 (2), we have :

$$(x \bullet y)_n = \sup_{p>n}(x_{p+1}(y_p))_n = \sup_{p>n} x_{n+1}(y) = x_{n+1}(y).$$

□

Theorem 3.2.1 ([14, Theorem 3.1.17])

Let $\begin{array}{rll} F : D_\infty & \longrightarrow (D_\infty \to D_\infty) \\ x & \longmapsto F_x : y \mapsto x \bullet y \end{array}$ and $\begin{array}{rll} G : (D_\infty \to D_\infty) & \longrightarrow D_\infty \\ f & \longmapsto G(f) = \sup_{n\geq 0} G_n(f) \end{array}$

for all $x, y \in D_\infty$, where $G_n \in D_{n+1}$ is defined by $G_n(f)(y) = f(y)_n$. Then F and G are inverse isomorphisms between D_∞ and $D_\infty \to D_\infty$, that is, $D_\infty \cong D_\infty \to D\infty$.

Proof. By lemma 3.2.4, we have $G_n(F(x)) = x_{n+1}$. Hence $G(F(x)) = \sup_{n>0} x_{n+1} = x$. Now let's see that $F(G(f)) = f$, that is, $G(f) \bullet x = f(x)$ for any $f : D_\infty \to D_\infty$ and $x \in D_\infty$. It is easy to check that $\bullet$ is jointly continuous so we have $G(f) \bullet x = \sup_{n>0}(G_n(f) \bullet x)$. Since $G_n(f) \bullet x = G_n(f)(x_n)$ by lemma 3.2.4, we have $G(f) \bullet x = \sup_{n>0} f(x_n)_n$. On the other hand, we have $f(x) = \sup_{n>0} f(x_n)$ by continuity, hence $f(x) = \sup_{n>0, p>n} f(x_n)_p$. Finally, observing that $f(x_n)_p < f(x_p)_p$, we have : $G(f) \bullet x = \sup_{n>0} f(x_n)_n = \sup_{n>0,p>n} f(x_p)_p = f(x)$. □

We have thus obtained a solution to the equation $D \cong D \to D$.

3.2.5 Models of the λ-calculus

In the introduction, we said that a model of the λ-calculus, or λ-model, is a mathematical structure that serves to interpret the formalism of the λ-calculus. A model must imitate the behavior of λ-terms in such a way that objects can act not only as arguments of functions but also as functions. One more time, this is why the model D should be isomorphic to the family of self-functions on D, which is usually not possible in classical set theory (unless D contains only one element) and lead Scott to consider D_∞ as a solution to this problem. But the concept of λ-model has yet to be formally defined. There exist different but equivalent ways to define it : via syntactic λ-models [14, Definition 15.3], [25, Definition 2.3], syntax free λ-models [14, Definition 15.19], [25, Definition 2.6], Scott-Meyer λ-models [14, Definition 15.22], functional λ-models [2, Definition 3.2.2][14, Discussion 15.26][6, Section 5.4], and many more.

Of all these models, we have chosen to focus on functional λ-models which, as their name suggest, emphasize the functional aspect of λ-calculus. We begin with a few preliminary definitions.

Definition 3.2.5 (applicative structure, set of representable functions)

An *applicative structure* (or *magma*) $(X,\star)$ is a nonempty set X equipped with a closed binary operation $\star$, i.e. such that $a, b \in X \Longrightarrow a \star b \in X$.

The set of *representable functions* $X \to_{\text{rep}} X$ on $(X,\star)$ is the set of functions from X to X defined by :

$$X \to_{\text{rep}} X = \left\{ f \in X \to X \mid \exists y \in X, \forall x \in X, f(x) = y \star x \right\}.$$

The element y will be called a representative of f.

Definition 3.2.6 (pre-reflexive, reflexive domain, [2, Definitions 3.2.1, 3.2.4])

Let $(D,\star)$ be an applicative structure and let $F : D \to (D \to_{\text{rep}} D)$ and $G : (D \to_{\text{rep}} D) \to D$ be two functions such that $F \circ G = \text{id}_{D \to_{\text{rep}} D}$ and $F(D) = (D \to_{\text{rep}} D)$. Then $(D,\ F,\ G)$ is called a *pre-reflexive domain.*

If D is a cpo, F and G are continuous and $(D \to_{\text{rep}} D) = (D \to_{\text{cont}} D)$ (the representable functions are exactly the continuous functions), then $(D,\ F,\ G)$, is called a *reflexive domain.*

Observe that, in a reflexive domain $(D,\ F,\ G)$, F is a left inverse of G. In category theory terminology, we say that $D \to_{\text{cont}} D$ is a retraction of D (see [6, Definition 5.4.1]).

Moreover, if $(D,\ F,\ G)$ is a pre-reflexive domain, given $d \in D$ there exists $F_d \in D$ such that $F(d)(x) = F_d \star x$ for all $x \in D$. Of course, the element F_d could be not unique. This naturally leads to consider an equivalence relation $\sim$ given by $a \sim b$ if and only if $a \star x = b \star x$ for all $x \in D$, called extensional equivalence [14, Discussion 15.8]. Hence, we can interpret the function F as an assignment to an extensional equivalence class.

Reciprocally, given a representable function f with representative a, we have that $G(f) \in D$ and $F(G(f)) = f$. This means that the extensional equivalence class associated with $G(f)$ is exactly the extensional equivalence class of a. Moreover, since $F \circ G = \text{id}_{D \to_{\text{rep}} D}$, G is injective so $G(f)$ can be considered as a selection of an element in D determining the extensional equivalence class of f.

Proposition 3.2.3

$(D_\infty,\ F,\ G)$ is a reflexive domain.

Proof. We have already seen that D_∞ is a cpo (see Proposition 3.2.1). Moreover, $(D_\infty, \bullet)$ is an applicative structure and F, G are inverse isomorphisms between D_∞ and $D_\infty \to D_\infty$ (see Theorem 3.2.1). For all $x \in D_\infty$, $F(x)$ is a representable function on $(D_\infty, \bullet)$, so (D_∞, F, G) is a reflexive domain. □

Definition 3.2.7 ([14, Notation 14.1], [6, Definition 5.1.2])

A mapping ρ from *Vars*, the set of all variables in the λ-calculus to a nonempty set D is called a valuation (of variables) or an environment. Given $d \in D$ and $x \in$ *Vars*, $\rho[d/x]$ denotes the valuation which is equal to ρ except in x where it is equal to d.

We next give the definition of a (functional) λ-model.

Definition 3.2.8 (λ-model, [2, Definition 3.2.2])

A λ-model, is a pre-reflexive domain (D, F, G), such that for any valuation ρ, the mapping $[\![\cdot]\!]_\rho$ that assign to each λ-term M an element $[\![M]\!]_\rho$ of D given by the following semantic equations is correctly defined.

a. $[\![x]\!]_\rho = \rho(x)$
b. $[\![PQ]\!]_\rho = F\big([\![P]\!]_\rho\big)\big([\![Q]\!]_\rho\big)$
c. $[\![\lambda x.P]\!]_\rho = G\big(\lambda d.[\![P]\!]_{\rho[d/x]}\big)$

The idea of the previous definition is that every λ-term P has an interpretation $[\![P]\!]_\rho$ in the pre-reflexive domain D, that can be seen as an applicative structure, for every valuation ρ. Moreover, the three main components of λ-calculus (variables, applications and abstractions) must be interpreted in the model. In this way, the conditions of the above definition are natural requirements to model these components. Requirement (a) establishes that the interpretation of variables is given by the valuation. Condition (b) asserts that the interpretation of the application of a λ-term P to a λ-term Q is given by the traditional application of the function $F([\![P]\!]_\rho)$ to the interpretation $[\![Q]\!]_\rho$ of the λ-term Q in the model. Therefore condition (b) means $[\![PQ]\!]_\rho = F_{[\![P]\!]_\rho} \star [\![Q]\!]_\rho$: an application of the λ-calculus is interpreted as a *multiplication* in the model.

Finally, property (c) gives the interpretation of an abstraction in λ-calculus. We can motivate it as follows. The interpretation of $\lambda x.P$ should be the interpretation of a function depending on the meaning of the term P. Then $[\![\lambda x.P]\!]_\rho \in D$ should determine a representable function f which is naturally given by $[\![\lambda x.P]\!]_\rho \star d = [\![P]\!]_{\rho[d/x]}$ for every $d \in D$. This last equality means that $[\![\lambda x.P]\!]_\rho$ acts like a function whose value is calculated by interpreting x as d. Consequently, we have a function $f : D \to D$ given by

$$f(d) = [\![P]\!]_{\rho[d/x]} \qquad \forall d \in D.$$

If we denote this function in λ-calculus terminology as $\lambda d.[\![P]\!]_{\rho[d/X]}$ we have that

$$f(d) = \lambda d.[\![P]\!]_{\rho[d/X]} \qquad \forall d \in D.$$

Since $[\![\lambda x.P]\!]_\rho$ is the representative of this function we arrive at

$$[\![\lambda x.P]\!]_\rho = G(\lambda d.[\![P]\!]_{\rho[d/x]})$$

which is precisely property (c). This discussion assumes that $f(d)$ is a representable function.

We next prove that main result of this section showing that reflexive domains are λ-models.

Theorem 3.2.2 ([2, Proposition 3.2.5])

A reflexive domain is a λ-model.

Proof. Let (D, F, G) be a reflexive domain. To ensure that the semantic equations of a λ-model given in Definition 3.2.8 are correct, we only have to check that $\lambda d.[\![P]\!]_{\rho[d/x]}$ is continuous.

Suppose that P does not contain any abstraction, that is, we cannot find a variable x and a λ-term M such that $\lambda x.M$ occurs in P. Therefore, M is a « sequence » of variables. Let's check that $\lambda d.[\![P]\!]_{\rho[d/x]}$ is continuous by induction on the length of P. Suppose that the length is 1, that is, P contains only one variable. If this variable y is different from x then

$$[\![P]\!]_{\rho[d/x]} = [\![y]\!]_{\rho[d/x]} = \rho[d/x](y) = \rho(y)$$

for every $d \in D$. Hence $\lambda d.[\![P]\!]_{\rho[d/x]}$ is constant, so continuous.

Moreover, if $P = x$ then

$$[\![P]\!]_{\rho[d/x]} = [\![x]\!]_{\rho[d/x]} = \rho[d/x](x) = d$$

for every $d \in D$. Hence $\lambda d.[\![P]\!]_{\rho[d/x]}$ is the identity, so continuous. Consequently, if the length of P is 1 we have the continuity of the function.

Suppose that $\lambda d.[\![P]\!]_{\rho[d/x]}$ is continuous when the length of P is n. If P has length $n+1$ then we can find a variable z and a λ-term P' such that $P = P'x$. Then

$$[\![P'z]\!]_{\rho[d/x]} = F([\![P']\!]_{\rho[d/x]})([\![z]\!]_{\rho[d/x]}).$$

Since z has length 1, we have already proven that $f(d) = \lambda d.[\![z]\!]_{\rho[d/x]}$ is Scott continuous. Furthermore, P' has length n so $g(d) = [\![P']\!]_{\rho[d/x]}$ is also Scott continuous. Since

$$([\![P']\!]_{\rho[d/x]}) = F(g(d))$$

and F is Scott continuous, $([\![P']\!]_{\rho[d/x]})$ also is.

Then we have the following diagram :

$$\begin{tikzcd}
 & D \arrow[r, "F"] & (D \to D) \arrow[dr, "i"] & & \\
D \arrow[ur, "g"] \arrow[dr, "f"'] & & & (D \to D) \times D \arrow[r, "ev"] & D \\
 & & D \arrow[ur, "i"'] & &
\end{tikzcd}$$

where i denotes inclusion. In this way,

$$[\![P'z]\!]_{\rho[d/x]} = ev(F \circ g, f(d))$$

for all $d \in D$. Since $F \circ g, f, ev$ are Scott continuous (see Proposition 2.3.5) we deduce that $\lambda d.[\![P]\!]_{\rho[d/x]}$ is Scott continuous.

Suppose now that $P = \lambda x_1 \dots x_n.M$. We make induction on the set of variables $\{x_1, \dots, x_n\}$. When it has length 1 then fixing $d \in D$ we have that

$$[\![P]\!]_{\rho[d/x]} = [\![\lambda x_1.M]\!]_{\rho[d/x]} = G(\lambda e.[\![M]\!]_{\rho[d/x, e/x_1]}).$$

As above, we can check that $\lambda e.[\![M]\!]_{\rho[d/x, e/y]}$ is Scott continuous so $G(\lambda e.[\![M]\!]_{\rho[d/x, e/x_1]})$ is well defined. Since G is also continuous we have Scott continuity of $[\![P]\!]_{\rho[d/x]}$.

Suppose that, if $P = \lambda \vec{x}.M$, then $\lambda d.[\![\lambda \vec{x}.M]\!]_{\rho[d/x]}$ is Scott continuous, where $\vec{x} = x_1 \dots x_n$ is a vector of n variables. Then, if $P = \lambda \vec{x}.M$ and $\vec{x} = x_1 \dots x_{n+1}$ has $n+1$ variables, then

$$\begin{aligned}[\![P]\!]_{\rho[d/x]} = [\![\lambda x_1 \dots x_n x_{n+1}.M]\!]_{\rho[d/x]} &= [\![\lambda x_1.(\lambda x_2 \dots x_n x_{n+1}.M)]\!]_{\rho[d/x]} \\ = G\Big(\lambda e.[\![\lambda x_2 \dots x_{n+1}.M]\!]_{(\rho[d/x])[e/x_1]}\Big) &= G\Big(\lambda e.f(d,e)\Big)\end{aligned}$$

where $f : D \times D \to D$ is given by $f(d,e) = [\![\lambda x_2 \dots x_{n+1}.M]\!]_{(\rho[d/x])[e/x_1]}$ for every $d, e \in D$. By induction hypothesis, f is separately continuous so continuous by Proposition 2.3.2. Moreover, $\Lambda(f)$ is continuous (see Proposition 2.3.3). Since

$$[\![P]\!]_{\rho[d/x]} = G\Big((\Lambda(f))(d)\Big)$$

and G is continuous, we can conclude that $\lambda d.[\![P]\!]_{\rho[d/x]}$ is continuous, which finishes the proof. $\square$

This last demonstration convinces us once and for all that D_∞ models, being reflexive, serve as a λ-model.

Bibliography

[1] P. Alexandrov, *Diskrete Räume*, Mat. Sb. **44** (1937), no. 3, 501–519.

[2] R. M. Amadio and P.-L. Curien, *Domains and lambda-calculi*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[3] K. R. Apt and E.-R. Olderog, *Fifty years of Hoare's logic*, Form. Asp. Comput. **31** (2019), no. 6, 751–807.

[4] F. G. Arenas, *Alexandroff spaces*, Acta Mathematica Universitatis Comenianae **LXVIII** (1999), no. 1, 17–25.

[5] J. W. Backus, *The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference*, IFIP Congress, 1959.

[6] H. P. Barendregt, *The lambda calculus. Its syntax and semantics*, North Holland, 1984.

[7] A. Church, *A set of postulates for the foundation of logic*, Ann. of Math. (2) **34** (1933), no. 4, 839–864.

[8] ______, *A formulation of the simple theory of types*, J. Symbolic Logic **5** (1940), 56–68.

[9] J. Collins, *A history of the theory of types: Developments after the second edition of 'Principia Mathematica'*, Lambert Academic Publishing, 2012.

[10] R. Engelking, *General topology*, Sigma Series in Pure Mathematics, vol. 6, Heldermann Verlag Berlin, 1989.

[11] R. W. Floyd, *Assigning meanings to programs*, Proceedings of Symposium on Applied Mathematics **19** (1967), 19–32.

[12] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott, *Continuous lattices and domains*, Cambridge University Press, 2003.

[13] C. A. Gunter, *Semantics of programming languages. structures and techniques*, The MIT Press, 1992.

[14] J. R. Hindley and J. P. Seldin, *Lambda calculus and combinators, an introduction*, Cambridge University Press, 2008.

[15] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580.

[16] T. D. L. Laan, *The evolution of type theory in logic and mathematics*, Dissertation, Technische Universiteit Eindhoven, Eindhoven, 1997.

[17] R. E. Moore, *Methods and applications of interval analysis*, SIAM Studies in Applied and Numerical Mathematics, Cambride University Press, 1979.

[18] G. Plotkin, *A structural approach to operational semantics*, The Journal of Logic and Algebraic Programming **60-61** (2004), 17–139.

[19] F. P. Ramsey, *The foundations of mathematics*, Proceedings of the London Mathematical Society **s2-25** (1926), no. 1, 338–384.

[20] T. Richmond, *General topology*, De Gruyter GmbH, 2020.

[21] B. Russell, *Mathematical Logic as Based on the Theory of Types*, Amer. J. Math. **30** (1908), no. 3, 222–262.

[22] D. S. Scott, *Outline of a Mathematical Theory of Computation*, Tech. Report PRG–2, Oxford, England, November 1970.

[23] ______, *Continuous lattices*, Toposes, algebraic geometry and logic (Conf., Dalhousie Univ., Halifax, N.S., 1971), Lecture Notes in Math., vol. Vol. 274, Springer, Berlin-New York, 1972, pp. 97–136. MR 404073

[24] K. Slonneger and B. L. Kurtz, *Formal syntax and semantics of programming languages*, Addison-Wesley Publishing Company, 1995.

[25] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor, *Mathematical theory of domains*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1994.

[26] R. E. Stong, *Finite topological spaces*, Transactions of the American Mathematical Society **123** (1966), no. 2, 325–340.

[27] A. M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Soc. (2) **42** (1936), no. 3, 230–265. MR 1577030

[28] A. N. Whitehead and B. Russell, *Principia mathematica to* ^{*56}, Cambridge University Press, New York, 1962. MR 130163

[29] G. Winskel, *The formal semantics of programming languages. An introduction*, The MIT Press, 1993.