Data-types as Objects Revisited
Insights and Proofs

Vaios-Rafail Michalakis
Reg. Num.: 7115142200013

Examination committee:
William Wadge, Emeritus Professor of Computer Science, University of Victoria, Canada.

Angelos Charalambidis (co-supervisor), Assistant Professor at the Department of Informatics and Telematics, Harokopio University, Greece.

Dr. Nikolaos Rigas, ALMA Graduate Program and Associate lecturer, Department of Information Technology, The American College of Greece.

Supervisor:
Panos Rondogiannis, Professor, Department of Informatics & Telecommunications, University of Athens.

Μεταπτυχιακό Πρόγραμμα «Αλγόριθμοι, Λογική και Διακριτά Μαθηματικά» – 2016

Contents

Abstract

English version

This thesis is concerned with an approach to giving the semantics of data-types in simple functional programming languages. The material is a thoroughly rewritten version of [SW77] completed with proofs and insights, which were largely absent in the original paper. The approach of [SW77] has been influential in several subsequent works and has been the basis for the development of the so-called *ideal-model* for recursive polymorphic types. The main idea of the aforementioned paper is to treat data-types as objects of the programming language; to do this, the authors developed a formal system in which substantial computation with data-types can be performed, including the recursive definition of data-types or their application to functions: in view of the Curry-Howard Isomorphism, this is to say that one can actually use computation with data-types to prove properties of programs. The resulting model allows the existence of subtypes and supertypes, as well as of logically qualified types and a limited form of dependent type families, notions which are much harder to define in traditional type systems. In the thesis we explore the formal definitions of the notions involved and present (with complete proofs) their basic properties, both at an abstract level, as well as at that of applications in type-checking real-world programs. To supplement the reader's understanding of the latter, scattered among the collection of theorems and the described techniques are a few illuminating examples, showing the flexibility and power of the theory under consideration.

Greek version

Η παρούσα διατριβή ασχολείται με μια προσέγγιση για την απόδοση της σημασιολογίας των τύπων δεδομένων σε απλές γλώσσες συναρτησιακού προγραμματισμού. Η περιεχόμενη ύλη είναι μια λεπτομερώς επαναδιατυπωμένη έκδοση του [SW77] που συμπληρώνεται με αποδείξεις και ιδέες, οι οποίες απουσίαζαν σε μεγάλο βαθμό από την αρχική εργασία. Η προσέγγιση του [SW77] έχει επηρεάσει αρκετές μεταγενέστερες εργασίες και αποτέλεσε τη βάση για την ανάπτυξη του λεγόμενου *μοντέλου ιδεωδών* για αναδρομικούς πολυμορφικούς τύπους. Η κύρια ιδέα της προαναφερθείσας δημοσίευσης είναι να αντιμετωπίζονται οι τύποι δεδομένων ως αντικείμενα της γλώσσας προγραμματισμού — για να γίνει αυτό, οι συγγραφείς ανέπτυξαν ένα τυπικό σύστημα στο οποίο μπορούν να γίνουν ουσιαστικοί υπολογισμοί με τύπους δεδομένων, συμπεριλαμβανομένου του αναδρομικού ορισμού των τύπων δεδομένων ή της εφαρμογής τους σε συναρτήσεις: υπό το πρίσμα του ισομορφισμού Curry-Howard, αυτό σημαίνει ότι μπορεί κανείς να χρησιμοποιήσει στην πραγματικότητα τον υπολογισμό με τύπους δεδομένων για να αποδείξει ιδιότητες των προγραμμάτων. Το μοντέλο που προκύπτει επιτρέπει την ύπαρξη υποτύπων και υπερτύπων, καθώς και λογικώς προσδιορισμένων τύπων και μιας περιορισμένης μορφής οικογενειών εξαρτημένων τύπων, έννοιες που είναι πολύ δυσκολότερο να οριστούν στα παραδοσιακά συστήματα τύπων. Στη διατριβή διερευνούμε τους τυπικούς ορισμούς των σχετικών εννοιών και παρουσιάζουμε (με πλήρεις αποδείξεις) τις βασικές ιδιότητές τους, τόσο σε αφηρημένο επίπεδο, όσο και σε αυτό των εφαρμογών στον έλεγχο τύπου πραγματικών προγραμμάτων. Προς καλύτερη κατανόηση του τελευταίου από τον αναγνώστη, ανάμεσα στη συλλογή των θεωρημάτων και των περιγραφόμενων τεχνικών υπάρχουν διάσπαρτα μερικά διαφωτιστικά παραδείγματα, που δείχνουν την ευελιξία και τη δύναμη της θεωρίας που εξετάζεται.

1 Introduction

1.1 About this Thesis

The material found here is a thoroughly rewritten version of [SW77], which will be our main reference. That paper is about a novel approach to give satisfactory denotational semantics to *data-types* such as found in several functional programming languages; it develops a short, new theory about what data-types do, or at least should, represent in a programming language's intended semantic domain. In fact, it goes further than that and actually constructs a formal system that allows one to perform meaningful *computation* with data-types, as though they were values (hence its name "Data Types as Objects"): in view of the *Curry-Howard Isomorphism*, this is to say that one can actually use computation with data-types to *prove* properties of programs. These insights come with a small but relatively comprehensive selection of theorems that can be used in several interesting examples, of which some we explore in the text.

Unfortunately, the original version, [SW77], was incomplete as it lacked the proofs to the theorems and the demonstrations of several different examples it mentioned. Thus, our main ambition here is to fill that gap and provide a more complete version of this theory by streamlining the presentation, by proving the theorems, correcting a few along the way and inserting a couple of our own, by filling the demonstrations of the important examples mentioned in the text, and by finding a few illuminating examples of our own. That is not to say that the (necessarily introductory) version of the theory we present here is anywhere near complete: to the contrary, we have reasons to believe that it can be developed much further, and that in fact when that is done, the resulting theory can find recognition as an efficient and ingenious alternative to the semantics of data-types, although our humble overview here will not be sufficient for that.

1.2 Generally about Semantics

Semantics is the study of *meaning*; in Computer Science, meaning of programs. There are different directions such a study can take: the one we are interested in in the context of this thesis is called *denotational semantics*; other possibilities include axiomatic semantics and operational semantics. To be brief, denotational semantics is to Programming Language Theory what Model Theory is to First Order Logic in that it tries to give the meaning of a program in the form of a model. Similarly (and using the same analogy between programming languages and First Order Logic) axiomatic semantics seems to be the equivalent of Proof Theory.

As an example, suppose we are given the following recursive program in a simple functional programming language:

$$F(n) \equiv \text{if } (n = 0) \text{ then } 1 \text{ else } n \times F(n-1) \ .$$

It shouldn't be difficult to see that its intended meaning is the factorial function; however, to give a complete, formal description of the meaning of such recursive programs we must specify two things: (a) what *kind of objects* the program needs to manipulate in order to make sense, and (b) what is the meaning of the program when it lives in the world of objects prescribed in (a). In our example, the "world" of the recursive program above is the set of natural numbers augmented with the two truth-values `tt` (true) and `ff` (false) –this is because we need to be able to make the comparison $n = 0$– and when interpreted in it, the recursive program does indeed model the factorial function. Actually, in order to be able to model non-termination, it is traditional to add a symbol $\perp$ to this semantic domain ordered below all other elements; this is understood to represent a state of non-knowledge. (Try to understand it thus: after a computation has terminated we know its exact value; however, before it finishes, we don't know what is the value of the result, and moreover, we generally don't even know if the computation will terminate at all! We use $\perp$ to model ongoing computations; an inequality like $\perp < 0$ is the way to say that if a computation finishes with value 0, one may "update" one's state of knowledge, and "lift" the value $\perp$ formerly used to represent that computation to the actual value 0.)

The study of semantics is of course too deep a subject to be treated here to any satisfactory degree; two excellent books on the subject are [Win93] and [Ten91]. Another very good account is given in [PWF12] (this is an online resource, and no guarantee can be given that it will continue to be accessible in the future.)

1.3 PCF: A model language

PCF stands for "Programming Computable Functions"; it is an ideal programming language developed as a framework for conducting research into typed functional programming languages. It is relatively minimalistic, and can be considered as an extended, Turing-complete version of typed λ-calulus, or as a simplified version of real-world typed functional programming languages.

In this thesis, we will work with a version of PCF allowing only *first-order functions*. Let us explain the term. A functional programming language usually has a few useful types built-in: these usually include integers, Booleans, and sometimes structures (which are called "products" in academic parlance). A function that only operates on those primitive, built-in types is said to be of the *first order*, otherwise it is of *higher order*; for example, higher-order functions may accept other functions as arguments, or return functions as values. A programming language that allows the definition of functions of higher-order is called a higher-order language, otherwise it is a first-order language; examples of higher-order languages include almost all known functional programming languages, including Lisp, Haskell, ML. (This terminology is not universally accepted.)

It appears that there is a profound reason why we restrict our attention to first-order languages: whereas in higher-order languages functions are no different from any other value, with types assuming the task of distinguishing between values of different nature, in this thesis our aim is to bring types to the same level as values, do computation with them &c., and thus it seems to us that functions *must* be a different kind of object. It would be a very interesting question to examine whether there is a way to treat data-types as objects in a higher-order language.

The discrepancy between the two kinds of languages is very evident in the case of denotational semantics. While programs in first-order languages have relatively simple models, to give a model of a program in a higher-order language, genuinely advanced mathematical tools and techniques are required. Although we will not discuss such languages here, we mention that the model of such a higher-order program is best understood in the language of Category Theory. The mathematical background necessary for venturing into that direction receives a fairly good treatment in [Ten91], and a presentation with most of the advanced mathematics streamlined into the text can be found in [Win93, Chapter 11].

To describe the programming language we will write the examples in (inspired by the language REC in [Win93, Chapter 9]), we will give the Backus Naur Form (BNF) of its syntax; in doing so, we will avoid going into details as the point is to develop a generally applicable method, not one tailored to a single language. Another point we shall leave intentionally vague is the *evaluation strategy*: we do not constrain ourselves to the exclusive study of a call-by-value or a call-by-name language.

Nothing is lost by thinking of our programming language as the functional fragment of Python or C (that is, a language with the same syntax as Python/C, but disallowing all imperative constructs such as: goto, for/while loops, (re-)assignments of variables, &c., and the higher-order function features in the case of Python), and therefore the confident reader may omit the rest of this section without loss of understanding.

Let $\texttt{Types}$ be the set of types the programming language supports; for our purposes, it is generally sufficient that $\texttt{Types} = \{\mathbb{B}, \mathbb{Z}, \texttt{List}\}$, where $\mathbb{B}$ is the type of Booleans, $\mathbb{Z}$ the type of integers, and $\texttt{List}$ the type of lists (which can be nested). The vocabulary of the language contains the constants $\texttt{tt}$, $\texttt{ff}$, $\texttt{0}$, $\texttt{1}$, and $\texttt{nil}$, the function-symbols $+$, $-$, $\times$, &, $\vee$, $=$, $\texttt{Cons}$, toghether with a countably infinite family

$(x_{\sigma,n})_{n\in\mathbb{N}}$ of variables of type σ for $\sigma \in \texttt{Types}$, and, for every $k \in \mathbb{N}$ and $\alpha \in \texttt{Types}^k$, a countably infinite family $(f_n^{\alpha,\sigma})_{n\in\mathbb{N}}$ of σ-valued function variables with arguments of types α.

The Backus Naur Form of the language is as follows:

$$\begin{aligned}
t ::\equiv\ & B \mid N \mid L \\
B ::\equiv\ & \texttt{tt} \mid \texttt{ff} \mid B \& B \mid B \vee B \mid \neg B \mid \\
& B = B \mid L = L \mid N = N \mid N < N \mid N > N \mid \\
& \text{if } B \text{ then } B \text{ else } B \ \mid x_{\mathbb{B},n} \mid F^{\alpha,\mathbb{B}}(t_{\alpha_1}, t_{\alpha_2}, \ldots, t_{\alpha_{|\alpha|}}) \\
N ::\equiv\ & \texttt{0} \mid \texttt{1} \mid N + N \mid N - N \mid N \times N \mid \text{if } B \text{ then } N \text{ else } N \ \mid \\
& x_{\mathbb{N},n} \mid F^{\alpha,\mathbb{N}}(t_{\alpha_1}, t_{\alpha_2}, \ldots, t_{\alpha_{|\alpha|}}) \\
L ::\equiv\ & \texttt{nil} \mid \texttt{Cons}(t, L) \mid \text{if } B \text{ then } L \text{ else } L \ \mid F^{\alpha,\texttt{List}}(t_{\alpha_1}, t_{\alpha_2}, \ldots, t_{\alpha_{|\alpha|}}) \\
F^{\alpha,\sigma} ::\equiv\ & f_n^{\alpha,\sigma}(t_{\alpha_1}, t_{\alpha_2}, \ldots, t_{\alpha_{|\alpha|}}) \mid \lambda t_{\alpha_1}.\lambda t_{\alpha_2}. \ldots .\lambda t_{\alpha_{|\alpha|}}.t_\sigma \quad (\sigma \in \texttt{Types}),
\end{aligned}$$

where t_σ is B, N, L according to whether σ is $\mathbb{B}$, $\mathbb{N}$, or $\texttt{List}$, respectively, and t_{α_i} is B, N, L according to whether α_i is $\mathbb{B}$, $\mathbb{N}$, or $\texttt{List}$, respectively. Informally, this says that a term t is either a Boolean (B), or a Number (N), or a List (L), and that Booleans, Numbers, and Lists are expressions that give the correct type. Note that $F^{\alpha,\sigma}$ is a generic expression for an $|\alpha|$-ary σ-valued function with arguments of types α.

At times, we may choose to work with subsets of the language we just described, ignoring, for example, lists and working only with Booleans and numbers.

1.4 Denotational Semantics of First-order Languages: A Brief Study

In this section we discuss the traditional approach to the semantics of first-order languages by way of example. The reader may benefit from a broad familiarity with fixed-point theorems (the relevant background material we need later in this thesis can be found in Chapter 2).

Suppose we are given the following simple program for the factorial function:

$$F(n) \equiv \text{if } (n = 0) \text{ then } 1 \text{ else } n \times F(n-1)\ .$$

The construction of a model for a program bears some analogies to that of a theory in first-order logic. We begin by choosing the semantic domain D; this is the ground set of values that the programming language (and consequently the program) are supposed to manipulate. The semantic domain must be one of a special kind of posets called *domains*; the precise definition of domains varies among authors, but in general it is sufficient to work with ω-cpos: posets with a least element such that every increasing ω-sequence has a supremum. (In subsequent chapters of this thesis we will work with a different notion, Definition 3.1, which is stronger.)

Next we must interpret function-symbols and constants: an *interpretation* of the function-symbols is, like in first-order logic, an assignment of an increasing (partial) function $f\colon D^n \rightharpoonup D$ to every n-ary function-symbol $\texttt{f}$ in the vocabulary; and similarly for constants. These two steps together are the analogues of choosing a structure in logic; however, in denotational semantics it is important that we interpret function-symbols as *increasing* functions. The poset of increasing functions from D to itself, pointwise ordered, is denoted by $[D \to D]$.

Finally, we construct an increasing function $D^n \rightharpoonup D$ that models the given program.

In our paticular example, we want the domain D to contain the truth-values and the integers. Its Hasse diagram is depicted in Figure 1.1. Notice that it is a *flat* domain: $x \sqsubseteq y$ iff $x = y$ or $x = \bot$.

Then interpret the function-symbols as the functions they usually stand for: e.g. $\vee$ is logical “or”, and $+$ is addition of natural numbers; the former only applies to Booleans, and the latter only to numbers. Hence the function-symbols are generally interpreted as partial-functions. This is actually a technicality, as we can lift the interpretations to total functions by declaring their undefined values to be $\bot$. These partial functions must still be increasing (if $\texttt{f}$ has arity n, $\text{dom}(\texttt{f}) \subseteq D^n$ is ordered pointwise). This means that, e.g. since $\bot \sqsubseteq \texttt{ff}$, we must have $\bot \vee \texttt{tt} \sqsubseteq \texttt{ff} \vee \texttt{tt} = \texttt{tt}$; hence the value of $\bot \vee \texttt{tt}$ must

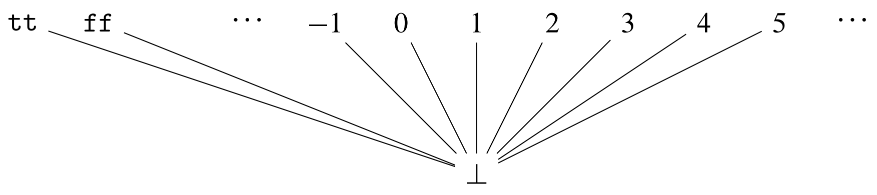

Figure 1.1: The semantic domain for the factorial function.

be either $\perp$ or tt. [And under different circumstances, it may make sense to define it as either value. It could be tt in a lazy language (which only evaluates its arguments if it needs them) or it could be $\perp$ in an eager language (note that in C, "or" is a short-circuiting operator – not a function: it doesn't evaluate the second argument if the first one is tt).]

The reason we are interested in increasing functions is because the ordering of D is supposed to be an information ordering, i.e., $x \sqsubseteq y$ is intended to mean that y is a more defined object than x. This should be obvious for flat domains, as the only non-trivial order relation is between $\perp$ (standing for an undefined, or as yet unknown, value) and other objects.

After deciding on the semantic domain and the interpretation of language symbols, we identify the thing whose least fixed point we are looking for: a functional. A functional is a function-taking, function-returning function: formally, a functional is a function

$$\tau\colon [D \to D] \to [D \to D].$$

One is naturally interested in increasing functionals. Furthermore, practically all functionals arising from recursive programs are continuous (Chapter 2), as can be proven by structural induction on τ, which means we can compute their fixed points using Lemma 2.5. We are interested in the least fixed point of functionals, i.e. the least solutions to recursive programs, as they are the ones with the least arbitrary information. As an example, for the recursive program under consideration, the two functions

$$f_1(n) = \begin{cases} n! & \text{if } n \geq 0, \\ \perp & \text{if } n < 0; \end{cases} \text{ and } f_2(n) = \begin{cases} n! & \text{if } n \geq 0, \\ 0 & \text{if } n < 0 \end{cases}$$

are solutions, but the latter makes arbitrary choises about the value of the solution at $n < 0$ (it cannot be "proved" that the function takes these values at $n < 0$).

The functional used in the recursive definition of the factorial is

$$\tau[F] := \lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n \times F(n-1)\ .$$

We want to show that the least fixed point of τ is a partial function $\mathbb{Z} \rightharpoonup \mathbb{Z}$ defined at all non-negative integers. Beginning with Ω, the nowhere defined function (constantly $\perp$) which is in $[D \to D]$, we successively evaluate the iterated applications of τ on Ω: one computes

$$\tau^1[\Omega](x) = \begin{cases} 1 & \text{if } x = 0, \\ \perp & \text{otherwise;} \end{cases}$$

$$\tau^2[\Omega](x) = \begin{cases} 1 & \text{if } x = 0, \\ 1 & \text{if } x = 1, \\ \perp & \text{otherwise;} \end{cases}$$

$$\vdots$$

$$\tau^n[\Omega](x) = \begin{cases} x! & \text{if } 0 \leq x < n, \\ \perp & \text{otherwise;} \end{cases}$$

$$\vdots$$

with least fixed point the limit $\tau^{\omega}[\Omega]$, which is a function with the desired property. All in all, we have shown that there is a function from the non-negative integers to the non-negative integers that is a fixed point of τ. Furthermore, it is the least fixed point of τ.

1.5 About the suggested approach

In type systems that follow the tradition of typed λ-calculus there is a number of ground types from which composite ones can be built using type formation operators: function-types can be built using the $\rightarrow$ operator, product-types (pairs) using $\times$, and union-types using $+$. Furthermore, every object in such types systems is assigned a unique type. Thus, for example, integers are not generally instances of real (floating point) numbers, and lists tend to be homogenous types (all their elements must be of the same type; in particular, they cannot be nested). Although this characteristic makes the theory mathematically robust, it is apparently also recognized as a limitation, or at least constraint, as ways to make such systems more flexible have been devised.

Some of these languages that stick closely to typed λ-calculus (e.g. Haskell) use instead Algebraic Data Types (ADT) and type inference to achieve the effects of polymorphism and creation of “heterogenous” data types (e.g. while tree-like structures can be simply represented as nested lists, in Haskell they have to be defined by custom-made data types; similarly the identity function `id` or the addition function `+` have type inferred from the context). Furthermore, for data types for which there is a conceivable notion of type “inheritance” (such as integers and floating-point numbers or lists and trees), specific type-casting functions are provided by the language or must be written by the programmer.

Other languages (e.g. Lisp) allow subtypes and supertypes (see fig. 1.2), and cast values of one type into values of supertypes using type coercion. This allows seemingly inhomogenous data-types such as nested lists, and type polymorphism. Technically, however, they are not inhomogenous, as all lists are considered lists of type `t` (`t` being the universal type in Common Lisp) and all function have type `t` $\rightarrow$ `t`.

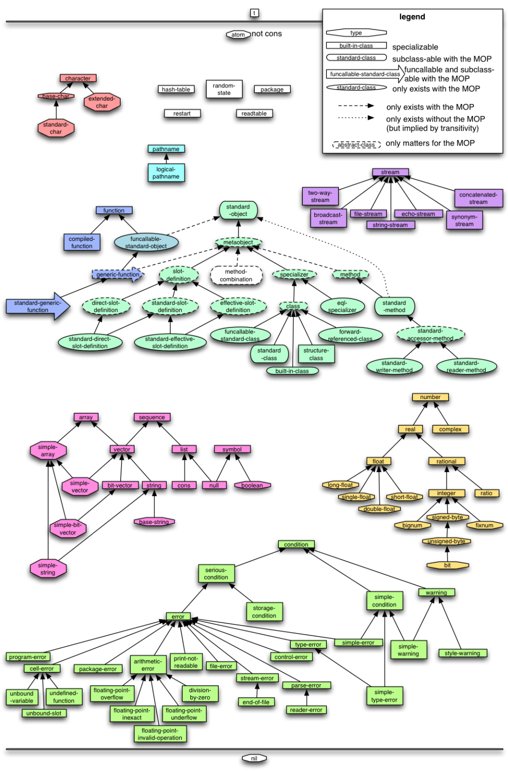

Figure 1.2: Types supported by Common Lisp. Source: `https://sellout.github.io/media/CL-type-hierarchy.png`, retrieved 30 Nov. 2024.

But even in languages that support super- and subtype relations, types themselves cannot be combined using the language's operations, and thus many intuitively natural and desirable computations such as

$$\textsc{Even} + \textsc{Odd} = \textsc{Odd}$$

cannot be expressed, or inferred, in a traditional type system.

Our approach is to incorporate the data-types into the domain of values/objects. An element in the resulting domain will serve two roles:

(i) it is a data-type on which functions can be defined, including of course functions that are the least fixed points of recursive programs;

(ii) it is the type of all object approximating it: in our system, the assertions $x \sqsubseteq y$, "x is of type y", and "any object of type x is also of type y" are equivalent.

Thus the ordering relation $\sqsubseteq$ in the extended domain will at the same time be a subtype inclusion relation, and an object-type instance relation.

For example, if the semantic domain of a programming language is as in Figure 1.1 and we want to add types for Booleans, Natural numbers, and Even and Odd integers, then we obtain the domain in Figure 1.3. Note that there is a universal type U.

In our system the types-as-objects perspective has its dual objects-as-types. Thus, not only is a data-type an object, but every object is a data-type. For example, if one defines a data-type Prime for prime numbers, one then has

$$\textsc{Even} \sqcap \textsc{Prime} = \tilde{2},$$

which says that the greatest lower bound (intersection) of even and prime numbers is the data-type $\tilde{2}$ in the objects-as-types interpretation; here, the data-type $\tilde{2}$ is the type whose only elements are 2 and $\perp$.

Monotonic functions in the original domain D can be extended to monotonic functions in the extended domain $\hat{D}$. We are very interested in the *least* extension, which we call the *tight extension*. The properties of tight extensions and related notions can be found in Chapter 4; they are going to be of constant use in the sequel.

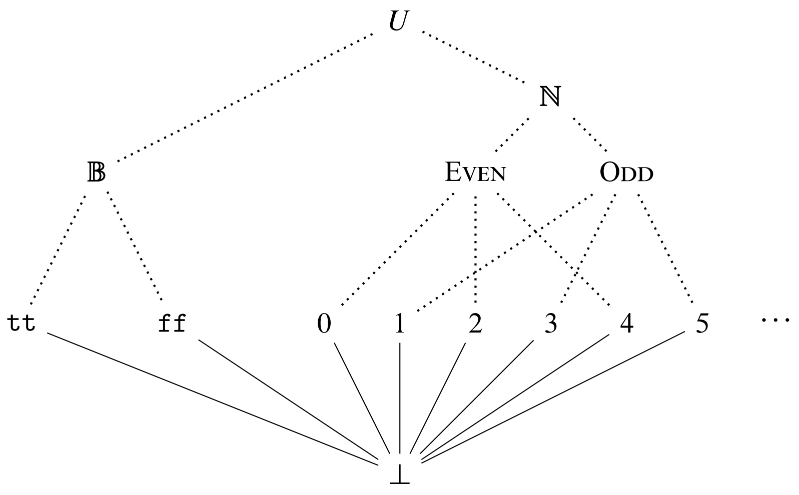

Figure 1.3: Domain D with extension $\hat{D}$. Solid lines show the ordering of elements of D, and dotted ones the object-type relations and type inclusions in the extension $\hat{D}$.

The simplest useful domain extension $\hat{D}$ is that depicted in Figure 1.3. In it many useful equations hold:

$$\textsc{Even} + 2 = \textsc{Even}$$
$$\textsc{Odd} + \textsc{Odd} = \textsc{Even},$$

when the functions $+, =$ are interpreted tightly.

Things are a bit more complicated if list objects and a type for lists are allowed. If we have a type `List` and a corresponding constructor `cons` (usually abbreviated by $::$, or by simply writing the list constructed by `cons`), then we may choose to add to the domain extension $\hat{D}$ also all values that can arise from applying `cons` to types such as $\mathbb{B}$ or $\mathbb{N}$. Thus, we may opt to also add types $\mathtt{cons}(\mathbb{B}, \mathtt{cons}(\mathbb{B}, \mathtt{nil})) \equiv [\mathbb{B}, \mathbb{B}]$ and $\mathtt{cons}(\mathbb{B}, \mathtt{cons}(\mathbb{N}, \mathtt{cons}(3, \mathtt{nil}))) \equiv [\mathbb{B}, \mathbb{N}, 3]$.(1) As all the built-in functions of our language are interpreted as increasing functions in $\hat{D}$, we have

$$[\mathtt{tt}, \bot] \sqsubseteq [\mathbb{B}, \mathbb{B}], \text{ and } [\mathtt{ff}, 4, 3] \sqsubseteq [\mathtt{ff}, \textsc{Even}, 3] \sqsubseteq [\mathbb{B}, \mathbb{N}, 3].$$

Perhaps the most important function in a programming language is the conditional if-then-else; we have a special definition for it in our system:

Definition 1.1. For $b \sqsubseteq \mathbb{B}$ and elements x, y of the domain extension of D, we define:

$$\text{if } b \text{ then } x \text{ else } y \;=\; \begin{cases} \bot & \text{if } b = \bot; \\ x, & \text{if } b = \mathtt{tt}; \\ y, & \text{if } b = \mathtt{ff}; \\ x \sqcup y & \text{if } b = \mathbb{B}, \end{cases}$$

where $x \sqcup y$ stands for the lowest upper bound (supremum) of x, y. When we formalise the notion of extensions later, we will have to write $\tilde{\bot}, \widetilde{\mathtt{tt}}, \widetilde{\mathtt{ff}}$ instead of $\bot, \mathtt{tt}, \mathtt{ff}$. This definition of if-then-else is practically the tight extension to $\hat{D}$ of the if-then-else on the original domain D.

As an example,

$$\text{if } \mathbb{B} \text{ then } 3 \text{ else } \textsc{Even} \;=\; \mathbb{N}.$$

It turns out that even function-types can be encoded by certain functions defined on the extended domain $\hat{D}$. If $(x \to y)$ is the function defined on $\hat{D}$ by

$$\lambda z.\text{if } (z \sqsubseteq x) \text{ then } y \text{ else } U,$$

then $(x \to y)$ maps objects below type x to type y, and other objects to type U. As discussed in Chapter 5, a monotonic function f is below $(x \to y)$ in the domain $[\hat{D} \to \hat{D}]$ of monotonic functions from $\hat{D}$ to itself if it maps objects of type x to objects of type y; i.e. if it is a function of "type" $x \to y$.

We will use this arrow function to perform case analysis in Chapter 5, which we will apply later in Chapter 8 to deduce properties of the least fixed points of recursive programs.

Finally, although we will not examine these topics in this thesis, we discuss two more properties of the types-as-objects approach that we think deserve some attention.

The first is that in a types-as-objects system there is a uniform way to handle recursive definitions of types and objects. This works even for functions that use both (standard) values and types: for example the recursive definition

$$S(x, n) \equiv \text{if } (n \leq 0) \text{ then } \mathtt{nil} \text{ else } x :: S(x, n-1) \tag{1.2}$$

(where $::$ is list construction), defines the function S such that $S(x, n)$ is a list of n x's (if $n \leq 0$ it is the empty list `nil`). Then, over an appropriate domain and with the tight extensions of the functions involved, $S(\mathbb{B}, 3)$ is the type of lists of Booleans of length 3, $S(U, 3)$ is the type of lists of any elements of length 3, and $S(0, \textsc{Even})$ is the type of strings of 0's of any even length. In particular, this example shows that it is possible to define at least some *dependent type families.*

(1) We *don't have to*, but if we don't then the tight extension of certain functions would return non-meaningful information; e.g. if we don't add type $[\mathbb{B}, \mathbb{B}]$ then the computation $\overline{\mathtt{cons}}(\mathbb{B}, \overline{\mathtt{cons}}(\mathbb{B}, \widetilde{\mathtt{nil}}))$ would evaluate to the universal type U.

The second interesting possibility is that error messages can be added to the domain D, and they may be organized so as to give meaningful information about the error that occurred. For example, the new objects `DomainError`, `DivisionByZero` can be introduced for situations like $3 + \mathtt{tt}$ or $4/0$ respectively. More error messages can be introduced: candidates include the general `error`, as well as the more specialised `OutOfRange`, `NotInteger`, `IndexError`; furthermore, they may be given an elaborate structure to reflect the logical connections between the errors they represent: e.g. `IndexError` can be above `OutOfRange` and `NotInteger`, and `NotInteger` could also be below `DomainError`.

These error messages do not represent data-types, but rather objects of the domain D; however, since we intend to use a non-flat domain $\hat{D}$, their presence will not be at odds with the rest of the semantic domain $\hat{D}$.

2 Domains & Fixed Points

2.1 Posets and Fixed points

In this thesis, we shall use the mathematically familiar terms *supremum* and *infimum* for the least upper bound ("lub") and greatest lower bound ("glb") respectively.

Definition 2.1. Given a poset $(D, \sqsubseteq)$, and a subset S, we say that

- S is a *directed subset* of D if every pair of elements of S has an upper bound in S: that is, for every pair of elements $a, b \in S$, there is $c \in S$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$.
- D is a *directed-complete poset* if every directed subset of D has a supremum in D: that is, for every directed $S \subseteq D$, there is $e \in D$ such that $e = \sup S$;
- D is a *chain-complete poset* if every chain (totally ordered subset) has a supremum in D;
- D is a *complete lattice* if every subset of D has a supremum in D;

Note that every directed-complete poset is chain-complete as well, and that every complete lattice has a greatest and a least element, given by the supremum of the whole set and the empty set respectively. Another very important fact about complete lattices is that every subset of a complete lattice has an infimum; this is seen by taking the supremum of all lower bounds of the given set.

Of particular interest for semantics are fixed points of functions on posets. An element $x \in D$, is called a *prefixed point* of $f: D \to D$ if $f(x) \sqsubseteq x$, and a *fixed point* of f if $f(x) = x$.

In the following we will confine ourselves to the study of *increasing* (also called *order-preserving*) functions $f: D \to D$ and partial functions $f: D \rightharpoonup D$: those that satisfy $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ for all $x, y \in D$. It is easy to show that if x is the least prefixed point of an increasing function f, then it is a fixed point. By far, the most important theorem about fixed points is:

Theorem 2.2 (Knaster-Tarski)**.** Given a complete lattice $(D, \sqsubseteq)$ and an increasing function $f: D \to D$, the fixed points of f form a complete lattice; moreover, for every $a \in D$, the least fixed point b of f above a is

$$b = \min\{x \in D \mid a \sqsubseteq x \text{ and } f(x) \sqsubseteq x\};$$

that is, b is the least prefixed point of f above a. This least fixed point is denoted by $\mathit{lfp}_a(f)$. If D has a least element, we write $\mathit{lfp}(f)$ for the least fixed point of f.

Interestingly, there is a constructive and equally useful version of this theorem under weaker hypotheses:

Theorem 2.3 (Constructive Knaster-Tarski)**.** If D is a *chain-complete* poset with a least element $\bot$ and $f: D \to D$ is increasing, then f has a least fixed point, and in fact $\mathit{lfp}(f) = \sup_{\alpha \in \mathbf{Ord}} s_\alpha$, where $s_0 = \bot$ and for $\alpha > 0$,

$$s_\alpha = \sup_{\beta<\alpha} f(s_\beta) = \begin{cases} f(s_\gamma), & \text{if } \alpha = \gamma + 1 \text{ is a successor;} \\ \sup_{\gamma<\alpha} s_\gamma, & \text{if } \alpha \text{ is a limit.} \end{cases}$$

Here **Ord** stands for the class of ordinal numbers. Note that the ordinal-sequence $s: \mathbf{Ord} \to D$ is well-defined, and is easily seen to be increasing. Since there are more ordinals than any set has elements, it follows that the sequence $(s_\alpha)_{\alpha \in \mathbf{Ord}}$ is *eventually constant*, and hence the supremum exists.

Under more assumptions on f, we may even know how far in the world of ordinals we need to go to find that supremum.

Definition 2.4. A function $f\colon D \to D$ is called *continuous* if it preserves suprema of directed sets: for any directed $S \subseteq D$,
$$f[\sup S] = \sup\{f(x) \mid x \in S\}.$$

Note that a continuous function is also increasing. Continuous functions are of particular importance for order theory because of the following (trivial) observation.

Lemma 2.5. With the notation of Theorem 2.3, where $f\colon D \to D$ is now continuous, we have that
$$\mathit{lfp}(f) = \sup_{n\in\mathbb{N}} s_n.$$

Proof. By Theorem 2.3, it suffices to show that $\sup_{n\in\mathbb{N}} s_n$ is a fixed point. Put $A = \sup\{s_n \mid n \in \mathbb{N}\}$ and $\sigma = \sup A$. Then by the definitions and the continuity of f we have:
$$\begin{aligned} f(\sigma) = f(\sup A) &= \sup f[A] \\ &= \sup\{f(x) \mid x \in A\} \\ &= \sup\{f(s_n) \mid n \in \mathbb{N}\} \\ &= \sup\{s_{n+1} \mid n \in \mathbb{N}\} \\ &= \sigma. \end{aligned}$$ □

The least α for which $s_\alpha = \mathit{lfp}(f)$ is called the *closure ordinal* of f. So the results above show that every increasing function in a chain-complete poset with $\perp$ has a closure ordinal, and continuous functions have closure ordinal $\leq \omega$.

The proofs of the theorems in this section can be found in any classical book on order theory; a good introduction to the field is [DP02].

3 Formal Construction of Domain Extension

3.1 Domains and Data Types

Domains are special posets; they play a very important role in denotational semantics, as we usually require all models of programs to be domains. There are several slightly different notions of domains, useful in different circumstances. Here by a *domain* we shall mean:

Definition 3.1. A *domain* is a poset $(D, \leq)$ with a least element (usually denoted $\perp$) such that every directed subset of D has a supremum (in D).

Note that this does *not* say that the supremum of directed set X is in X (which would mean that every directed subset has a greatest element).

Definition 3.2. For a domain D, a *data-type* (or simply a *type*) over D is a non-empty subset X of D which is closed downwards, and closed under suprema of D-directed subsets: that is, if $x \sqsubseteq y$ and $y \in X$ then also $x \in X$, and if $S \subseteq X$ is directed *in* D, then $\sup S \in X$. Such sets are also called *ideals*.(1)

Standard types found in most programming languages include the integers, the truth-values (Booleans), characters and strings, lists and arrays, among others. In our system, as we will see later, it is very easy to incorporate new data-types, which allows us to consider and work with such unique types as the type of even or odd integers, or of positive integers, or of strings without 0's; this is a great help for the type-checking and analysis of programs.

To every $d \in D$ we associate the set $\tilde{d}$ defined by

$$\tilde{d} = \{a \in D : a \sqsubseteq d\}, \tag{3.3}$$

which is evidently a data-type. As mentioned, we shall work with an extension $\hat{D}$ of D. Formally, this means that $\hat{D}$ will contain an isomorphic copy of D; the $\tilde{d}$'s will play the role of this copy.

Definition 3.4. A *type structure* over D is a collection T of data-types over D such that:

(i) $\tilde{d} \in T$ for every $d \in D$;

(ii) the universal type U, the set of all elements of D, is in T (so technically $U = D$); and

(iii) T contains the intersection of any non-empty family of types in T.

Our extended domain then will be $\hat{D} = (T, \subseteq)$ for some type structure T, which we can choose as convenient. In practice, we describe the data types we are interested in, and then define T to be the least type structure containing those data types (which usually means completing T with all intersections, as prescribed by (iii) in Definition 3.4).

Theorem 3.5. For any type structure T over a domain D, the extension $\hat{D} = (T, \subseteq)$ satisfies:

(i) $\hat{D}$ is a complete lattice;

(ii) $\tilde{\perp} = \{\perp\}$ is the least element of $\hat{D}$;

(iii) for any $x, y \in D$, $x \sqsubseteq_D y$ iff $\tilde{x} \sqsubseteq_{\hat{D}} \tilde{y}$;

(1) See, for example, [DP02].

(iv) if S is a directed subset of D with supremum e, then $S' = \left\{\tilde{d} : d \in S\right\}$ is a directed subset of $\hat{D}$ with supremum $\tilde{e}$.

Proof. We begin by showing (i), that $\hat{D}$ is a complete lattice. Let S be any non-empty subset of $\hat{D}$. Let $\mathcal{U}$ be the set of all *upper bounds* of S; it is not empty because the universal type $U \in \mathcal{U}$. By (iii) in Definition 3.4 (of type structures), the intersection $\bigcap \mathcal{U}$ is a type in $\hat{D}$. We claim this is the desired supremum of S. This is actually obvious: it is an upper bound because for every $s \in S$ and $u \in \mathcal{U}$ we have $s \subseteq u$, implying that $s \subseteq \bigcap \mathcal{U}$. Hence the intersection $\bigcap \mathcal{U}$ is in $\mathcal{U}$, which means that it is the least element of $\mathcal{U}$, and hence the least upper bound.

For (ii), note that every element of $\hat{D}$ is non-empty, and, being closed downwards, it therefore contains $\bot$. So $\tilde{\bot} = \{\bot\}$ is less than all other elements of $\hat{D}$.

If $x \sqsubseteq_D y$ then obviously $\tilde{x} \sqsubseteq_{\hat{D}} \tilde{y}$. Conversely, if $\tilde{x} \sqsubseteq_{\hat{D}} \tilde{y}$ then $x \in \tilde{y}$, and thus $x \sqsubseteq y$, which proves (iii).

Finally we prove (iv). Suppose S is directed in D. Then S' is directed in $\hat{D}$, for if $\tilde{x}, \tilde{y} \in S'$ then S contains an upper bound b for x, y, which implies $\tilde{x}, \tilde{y} \subseteq \tilde{b}$ with $\tilde{b} \in S'$.

Furthermore, if e is the supremum of S, then for any $s \in S$ we have $s \sqsubseteq e$; thus $\tilde{s} \subseteq \tilde{e}$, giving that $\tilde{e}$ is an upper bound for S'. Finally, if A is an upper bound for S', for all $s \in S$ we have that $\tilde{s} \subseteq A$, hence $s \in A$, implying that $S \subseteq A$. Because S is directed and A is closed under suprema of directed subsets (cf. Definition 3.2 of data types), we conclude $e = \sup S \in A$, giving the required result $\tilde{e} \subseteq A$. □

The significance of this theorem is that $\hat{D}$ is indeed an extension of D that is adequate for our purposes, in the sense that we can apply fixed point theorems to it. Property (iv) is particularly important when we consider least fixed points, because it ensures that the structure of suprema of *directed* subsets is preserved. (Note, however, that this may not be true of general supremum-possessing subsets of D.)

Usually, there are many different ways to extend a domain; for example, we may take the set of all ideals of D. However, experience suggests that smaller extensions are easier to work with than larger ones (and also more intuitive), and thus we usually restrict the added types to those actually needed.

On closing this section, we ask the reader to be henceforth mindful of the following

CONVENTION: *Although D is not a subset of $\hat{D}$, in statements of theorems we will usually identify D with its isomorphic copy in $\hat{D}$; in proofs and definitions, however, we will normally make the distinction.*

3.2 Axiomatic Characterisation of the extension

On reading the introduction the reader may have formed the impression that any extension will serve our purposes, and consequently may have been baffled by the preceding discussion. Why do we need such a complicated construction? It is natural to think of data types as sets, but why do we need to think of elements of our original domain as subsets of itself, as in the theorem and proof above? Furthermore, even the first point may cause some confusion, since in the introduction (and in the examples we have given thus far) we have treated types as abstract objects of our system, thereby stripping them of their set-theoretic content.

The answer is that we want data types to respect the important property of *extensionality*: that is, types are determined by their elements. This is generally expected of types, as we tend to think of them as sets, but in general it may not hold in some other systems, and especially in type theory, where types are the primary object of study.(2)

In a domain extension $\hat{D}$, we want two types with exactly the same elements of D to coincide. Moreover, we want to introduce a natural inclusion ordering on types, so that if all instances of type s are also instances of type t then $s \sqsubseteq t$. The construction given above is particularly convenient as it addresses both issues.

(2) In fact, in type theory every object is usually assigned a *unique* type, which makes it impossible even to formulate extensionality, as the only type sharing *even a single* element with a given type is itself.

3.3 Examples

Example 3.6. Let $D = \{\mathtt{tt}, \mathtt{ff}, 0, 1, 2, 3, \dots\}_\perp$ be a flat domain with values `tt` (true), `ff` (false), and the natural numbers. Suppose we want to extend D to $\hat{D}$ by adding data types for the Booleans, the Naturals, and the Even and Odd integers. First, we find the isomorphic copy of D in $\hat{D}$: this is given by the elements of the form $\tilde{n} = \{n, \perp\}$ for natural n, together with $\widetilde{\mathtt{tt}} = \{\mathtt{tt}, \perp\}$, $\widetilde{\mathtt{ff}} = \{\mathtt{ff}, \perp\}$, and $\widetilde{\perp} = \{\perp\}$. Next we define the new data-types:

$$\begin{aligned}
\mathbb{N} &= \{0, 1, 2, 3, \dots\} \cup \{\perp\},\\
\textsc{Even} &= \{0, 2, 4, 6, \dots\} \cup \{\perp\},\\
\textsc{Odd} &= \{1, 3, 5, 7, \dots\} \cup \{\perp\},\\
\mathbb{B} &= \{\mathtt{tt}, \mathtt{ff}, \perp\},\\
U &= \{\mathtt{tt}, \mathtt{ff}, 0, 1, 2, 3, 4, \dots\} \cup \{\perp\}.
\end{aligned}$$

The result appears in the form of a Hasse diagram in Figure 1.3.

Example 3.7. Suppose now that D is that of Example 3.6 together with all *rational numbers*, and that we want to have types for the Booleans, the Even and Odd integers, a type $\mathbb{Q}$ for rationals, and another $\textsc{Primary}$ for primary objects, namely Booleans and natural numbers. The data-types $\textsc{Even}, \textsc{Odd}, \mathbb{B}$ and the $\tilde{n}$, $\widetilde{\mathtt{tt}}$, $\widetilde{\mathtt{ff}}$ are defined as before. The new entrants are $\widetilde{p/q} = \{p/q, \perp\}$ for naturals p, q ($q \neq 0$), and

$$\begin{aligned}
\textsc{Primary} &= \{\mathtt{tt}, \mathtt{ff}, 0, 1, 2, 3, 4, \dots\} \cup \{\perp\},\\
\mathbb{Q} &= \{0, 1, 2, 3, \dots, 1/2, 1/3, \dots, 5/324, \dots\} \cup \{\perp\}.
\end{aligned}$$

But then $\hat{D}$ is not a valid extension, as it is not closed under intersections; indeed, the system lacks $\textsc{Primary} \cap \mathbb{Q}$, which happens to be the set $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\} \cup \{\perp\}$. If, however, we incorporate the data-type $\mathbb{N}$ to the ones mentioned above, then $\hat{D}$ becomes a valid extension.

4 Function Domains

4.1 Tight Functions and Tight Extensions

Despite the fact that our system makes no distinction between objects and (primary) data types, functions are treated as a different kind of object.[1] For the sake of simplicity, we shall only consider functions of a single argument, as the extension to multivariate arguments is straightforward, either by generalizing the results, or by techniques such as currying.

If D, E are domains, we write $[D \to E]$ for the set of all increasing functions from D to E. It is well known that with the induced pointwise ordering, $[D \to E]$ is a domain.

Now we focus on the setting we study in this thesis. Let D be a domain, and let $\hat{D}$ be an extension, as defined in Chapter 3.

Definition 4.1. A function $\hat{f}$ in $[\hat{D} \to \hat{D}]$ is an *extension* of a function f in $[D \to D]$ if they agree on D: more formally, for all $a, b \in D$,

$$\text{if } f(a) = b \text{ then } \hat{f}(\tilde{a}) = \tilde{b}.$$

Functions in $[\hat{D} \to \hat{D}]$ that are extensions of functions in $[D \to D]$ are called *conservative.*

Of course, not every function in $[\hat{D} \to \hat{D}]$ is conservative. Conservative functions play an important role in our system, as we are usually interested in extending functions in $[D \to D]$ to functions in $[\hat{D} \to \hat{D}]$. In fact, often we will be interested in the least such extensions, and accordingly we give them a name:

Definition 4.2. A function g in $[\hat{D} \to \hat{D}]$ is called *tight* if:

$$g(x) = \sup_{d \in x} g\left(\tilde{d}\right) = \sup\left\{g\left(\tilde{d}\right) : d \in x\right\}$$

for all $x \in \hat{D}$. (The supremum exists because $\hat{D}$ is a complete lattice.)

In other words, *a tight function is the least extension of its restriction on D*; in particular this means that it is determined by its values on D. A word of caution is due here: although the notion of tightness may sound reminiscent of that of continuity, as we will see towards the end of this chapter the two are not equivalent.

Example. Let's suppose our programming language can manipulate Booleans and natural numbers, and that it has an operand + for the addition of natural numbers. Suppose further that we want to extend this domain of values (which happens to be flat) with the types shown in in Figure 1.3. Now write $\mathtt{Add}_m(n)$ for $m + n$ (generally known as "Currying" of addition); for this example, we let $m = 3$. As usual, we assume that $\mathtt{Add}_3(\mathtt{tt}) = \mathtt{Add}_3(\mathtt{ff}) = \mathtt{Add}_3(\perp) = \perp$. Now let $\mathtt{Add}'_3$ be an extension of $\mathtt{Add}_3$ to all of $\hat{D}$ by setting $\mathtt{Add}'_3(\textsc{Even}) = \textsc{Odd}$, $\mathtt{Add}'_3(\textsc{Odd}) = \textsc{Even}$, $\mathtt{Add}'_3(\mathbb{N}) = \mathbb{N}$, and $\mathtt{Add}'_3(U) = U$. Then $\mathtt{Add}'_3$ is a tight function: for example, indeed

$$\mathtt{Add}'_3(\textsc{Even}) = \textsc{Odd} = \sup_{n \in \textsc{Even}} \mathtt{Add}'_3(\tilde{n}),$$

which says $3 + \textsc{Even} = \textsc{Odd}$. The remaining checks are similar. ◇

[1] Unlike some other type systems (e.g. Haskell) where functions too are primary objects of their own types (*in exactly the same way* that `3` is an object of type $\mathbb{N}$, or `true` of type `Bool`), and can be passed as arguments to, or returned as values from, other functions like any other object. For more on this, the reader is steered towards studying models of λ-calculus.

In order to understand better the claim made above that a tight function is the least extension of its restriction on D, we make the following two definitions, which are useful in their own right:

Definition 4.3. Given a function h in $[\hat{D} \to \hat{D}]$, its *tightening* $\bar{h}$ is the function in $[\hat{D} \to \hat{D}]$ with

$$\bar{h}(x) = \sup_{d \in x} h\left(\tilde{d}\right) = \sup\left\{h\left(\tilde{d}\right) : d \in x\right\}$$

for all $x \in \hat{D}$. (The supremum exists because $\hat{D}$ is a complete lattice.)

Theorem 4.4. For any h in $[\hat{D} \to \hat{D}]$,

(i) $\bar{h}$ is the least function in $[\hat{D} \to \hat{D}]$ which agrees with h on D: $\bar{h}(\tilde{d}) = h(\tilde{d})$ for all $d \in D$;

(ii) $\bar{h}$ is a tight function; and

(iii) h is tight iff $\bar{h} = h$.

Proof. We begin with (i). First note that $\bar{h}$ indeed agrees with h on D. Let $d \in D$. Since h is increasing, $e \sqsubseteq d$ implies $\tilde{e} \subseteq \tilde{d}$ and thus $h(\tilde{e}) \subseteq h(\tilde{d})$. Hence

$$\sup_{e \sqsubseteq d} h(\tilde{e}) = h\left(\tilde{d}\right),$$

as $h(\tilde{d})$ itself contributes to the supremum; this in turn shows that $h(\tilde{d}) = \bar{h}(\tilde{d})$ by the definition of $\bar{h}(\tilde{d})$. It is also obvious that $\bar{h}$ is the least such function in $[\hat{D} \to \hat{D}]$: if g is another function in $[\hat{D} \to \hat{D}]$ which agrees with h on D, then (since g is increasing) $d \in x \Rightarrow \tilde{d} \subseteq x \Rightarrow g(\tilde{d}) \subseteq g(x)$, whence

$$g(x) \supseteq \sup_{d \in x} g\left(\tilde{d}\right) = \sup_{d \in x} h\left(\tilde{d}\right) = \bar{h}(x),$$

which concludes the proof of (i).

For (ii), $\bar{h}$ is a tight function by definition (while reading the definition, recall that $\bar{h}$ agrees with h on D). Finally for (iii), if h is tight then $h = \bar{h}$ as $h, \bar{h}$ agree on D and the values of a tight function are determined by its values on D. Conversely, if $\bar{h} = h$ then h is tight (being equal to a tight function). □

Definition 4.5. Given a function f in $[D \to D]$, the *tight extension* $\bar{f}$ of f to $\hat{D}$ is the least extension of f; in other words, it is the function defined by

$$\bar{f}(x) = \sup_{e \in x} \widetilde{f(e)} = \sup\left\{\widetilde{f(e)} : e \in x\right\}.$$

for all $x \in \hat{D}$. (The supremum exists because $\hat{D}$ is a complete lattice.)

Example. This is a follow-up to the previous example. We have

$$\begin{aligned}\overline{\texttt{Add3}}(\textsc{Even}) &= \sup_{n \in \textsc{Even}} \widetilde{\texttt{Add3}(n)} \\ &= \sup_{n \in \textsc{Even}} \widetilde{3 + n} \\ &= \sup\left\{\tilde{1}, \tilde{3}, \tilde{5}, \dots\right\} = \textsc{Odd}. \diamond\end{aligned}$$

As the notations for the tightening and the tight-extension are the same, care must be paid to understand what is meant: $\bar{f}$ means the tightening of f if f is a function $\hat{D} \to \hat{D}$, and the tight extension of f if f is a function $D \to D$.

Theorem 4.6. For f in $[D \to D]$,

(i) $\bar{f}$ is a tight function;

(ii) $\bar{f}$ is the least extension of f;

(iii) for any other extension g of f, $\bar{f} = \bar{g}$ (where $\bar{g}$ is the tightening of g).

Proof. To show that $\bar{f}$ is tight, we have to show that for every x in $\hat{D}$, $\bar{f}(x) = \sup_{d \in x} \bar{f}(\tilde{d})$. But for every x in $\hat{D}$ we have:

$$\begin{aligned}
\bar{f}(x) &= \sup_{d \in x} \widetilde{f(d)} && \text{(by definition)} \\
&= \sup_{d \in x} \sup_{e \sqsubseteq d} \widetilde{f(e)} && \text{(because } \sup_{e \sqsubseteq d} \widetilde{f(e)} = \widetilde{f(d)}\text{)} \\
&= \sup_{d \in x} \sup_{e \in \tilde{d}} \widetilde{f(e)} \\
&= \sup_{d \in x} \bar{f}\left(\tilde{d}\right),
\end{aligned}$$

as wanted.

That $\bar{f}$ is an extension of f follows from the following:

$$\bar{f}(\tilde{a}) = \sup_{e \in \tilde{a}} \widetilde{f(e)} = \sup_{e \sqsubseteq a} \widetilde{f(e)} = \sup \left\{ \widetilde{f(e)} : e \sqsubseteq a \right\} = \widetilde{f(a)},$$

because $f(a)$ is the greatest element of that set (as f is increasing). This shows, as wanted, that $\bar{f}(\tilde{a}) = \widetilde{f(a)}$.

That f is the *least* extension is obvious from the definition, as $\bar{f}(x)$ must be at least as big as $\bar{f}(\tilde{e})$ for all $\tilde{e} \subseteq x$, which (as we just showed) is equivalent to $\widetilde{f(e)} \sqsubseteq \bar{f}(x)$ for all $e \in x$; now take the supremum of this over all $e \in x$.

Finally, for any extension g of f we have:

$$\bar{g}(x) = \sup_{d \in x} g\left(\tilde{d}\right) = \sup_{d \in x} \widetilde{f(d)} = \bar{f}(x),$$

by the definitions of tightening and extension. □

The theorems in this chapter are so fundamental for the sequel, that the reader is advised to memorise them and think of them as *tools of first resort*: *we will be using them constantly without reference.*

We conclude this chapter with a remark about tight functions. A tight function is practically the least extension to all of $\hat{D}$ of an incresing function in D; and as we saw in the introduction, section 1.5, we are usually interested in the least extension as they are often the most intuitive extensions. Despite the usefulness of tight extensions, however, non-tight functions play an important role, and we have to take them into account, because eliminating them would leave us incapable of performing basic tasks:

Example. The composition of tight functions may not be tight.

Demonstration. We will give two functions on the domain D with extension $\hat{D}$ shown in Figure 1.3. Define $g, h \colon D \to D$ by

$$g(n) = \begin{cases} \bot, & \text{if } n = \bot, \mathtt{tt}, \mathtt{ff} \\ 2n, & \text{if } n = 0, 2, 4, \dots \text{ (i.e. an } \textit{even} \text{ integer)} \\ 4n, & \text{if } n = 1, 3, 5, \dots \text{ (i.e. an } \textit{odd} \text{ integer)} \end{cases}$$

and

$$h(n) = \begin{cases} \bot, & \text{if } n = \bot, \mathtt{tt}, \mathtt{ff} \\ \mathtt{tt}, & \text{if } n \equiv 0 \pmod 4 \\ \mathtt{ff}, & \text{if } n \not\equiv 0 \pmod 4. \end{cases}$$

Now consider the tight extensions of g, h on $\hat{D}$; recall that

$$\bar{g}(x) = \sup_{d \in x} \widetilde{g(d)} \quad \text{and} \quad \bar{h}(x) = \sup_{d \in x} \widetilde{h(d)}.$$

We claim that $\left(\bar{h} \circ \bar{g}\right)$ is not tight; in particular, we show that $\left(\bar{h} \circ \bar{g}\right)(\mathbb{N}) \neq \sup_{n \in \mathbb{N}} \left(\bar{h} \circ \bar{g}\right)(\tilde{n})$. Indeed,

$$\bar{g}(\mathbb{N}) = \sup_{n \in \mathbb{N}} \widetilde{g(n)} = \sup \left\{\tilde{0}, \tilde{4}, \tilde{4}, \tilde{8}, \widetilde{12}, \widetilde{12}, \dots\right\} = \text{Even},$$

and

$$\bar{h}(\text{Even}) = \sup_{n \in \text{Even}} \widetilde{h(n)} = \sup\{\widetilde{h(0)}, \widetilde{h(2)}, \widetilde{h(4)}, \dots\} = \sup\{\widetilde{\mathtt{tt}}, \widetilde{\mathtt{ff}}, \widetilde{\mathtt{tt}}, \dots\} = \mathbb{B}.$$

Thus, $\left(\bar{h} \circ \bar{g}\right)(\mathbb{N}) = \bar{h}\left(\bar{g}\left(\mathbb{N}\right)\right) = \bar{h}\left(\text{Even}\right) = \mathbb{B}$. On the other hand, $\bar{g}(\tilde{n}) = \widetilde{g(n)}$ (as $\bar{g}$ is an extension of g; similarly for h) and thus:

$$\sup_{n \in \mathbb{N}} \left(\bar{h} \circ \bar{g}\right)(\tilde{n}) = \sup_{n \in \mathbb{N}} \bar{h}\left(\bar{g}\left(\tilde{n}\right)\right) = \sup_{n \in \mathbb{N}} \widetilde{h(g(n))} = \sup_{n \in \mathbb{N}} \widetilde{\mathtt{tt}} = \widetilde{\mathtt{tt}}. \qquad \square$$

This example also shows that there is a functional τ whose least fixed point is not tight: one can take $\tau[F] = h \circ g$.

Example. It is also not difficult to prove that there is a domain D and an extension $\hat{D}$ such that a continuous function $f \colon D \to D$ does not have any tight continuous extension in $[\hat{D} \to \hat{D}]$.

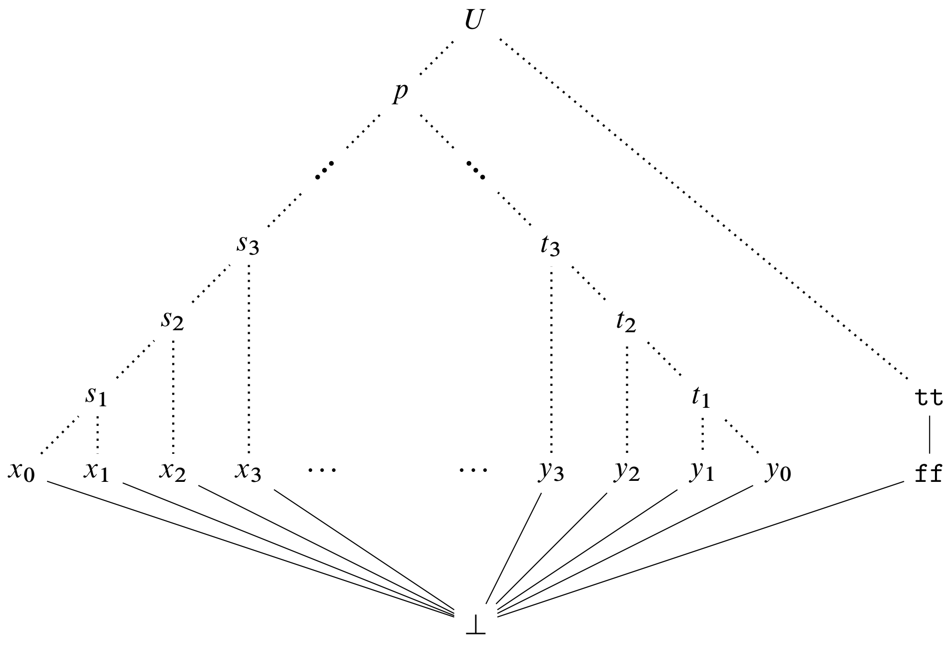

Figure 4.1: Domain D with extension $\hat{D}$ in the example.

Demonstration. Consider the domain D and its extension $\hat{D}$ depicted in Figure 4.1. Solid lines connect elements of D, while dotted ones represent the ordering of the extension $\hat{D}$. [A realistic construction of such a domain extension would be to take $x_i = 2i + 1$, $y_i = 2i$, and add types $s_j = \text{Odd}_{\leq 2j+1}$, $t_j = \text{Even}_{\leq 2j}$ and $p = \mathbb{N}$ ($i \geq 0, j \geq 1$); note that $\mathtt{ff} \sqsubseteq \mathtt{tt}$ and there is no Boolean type $\mathbb{B}$.]

Now let the function $f \colon D \to D$ be defined by mapping $x_i \mapsto \mathtt{ff}$ and $y_i \mapsto \mathtt{tt}$ (and $\perp, \mathtt{tt}, \mathtt{ff}$ to $\perp$). It is easy to see that $S = \{s_1, s_2, \dots\}$ and $T = \{t_1, t_2, \dots\}$ are both *directed* sets in $\hat{D}$ with the same supremum p. Therefore, f cannot have a tight, continuous extension, as the tight extension would send $s_i \mapsto \mathtt{ff}$ and $t_i \mapsto \mathtt{tt}$, and this would make it discontinous at S.

Given that f has both a tight extension (namely $\bar{f}$) and a continuous one (e.g. extend it by mapping $s_i, t_i, p, U \mapsto U$), this example shows that *the notions of tightness and continuity are not equivalent.* $\square$

Following the idea in the last sentence of the example, it is not difficult to find conditions on $\hat{D}$ forcing every continuous $f\colon D \to D$ to have a continuous extension in $[\hat{D} \to \hat{D}]$. Such a condition is, for example, that D is a *down-set* of $\hat{D}$, in the terminology of [DP02], i.e. if $s \sqsubseteq d$ for some $s \in \hat{D}$ and $d \in D$, then $s \in D$ (formally: if $s \subseteq \tilde{d}$ for some $d \in D$, then $s = \tilde{e}$ for some $e \in D$, but we identify the embedding of D into $\hat{D}$ as the inclusion map).

Indeed, we can then map all $s \in \hat{D} \setminus D$ to U; this extension is obviously increasing and continuous. (To see this, note that an element $s \notin D$ cannot be the supremum of a directed subset of D, as that would be *directed also in* D, and thus D, being a domain, would contain the supremum s.)

Note that all flat domains satisfy this condition; how this extends to functions of higher arities, however, remains unclear.

Open Question. In [SW77], the authors claim that there are examples of domains D with extension $\hat{D}$ and a continuous function $f\colon D \to D$ such that f has no continuous extension in $[\hat{D} \to \hat{D}]$, but the author of the current thesis has not been able to find such an example.

Example. One may try to give a more general definition of tight extensions for functions of many arguments without using Currying. For increasing $f\colon D^n \to D$ write $\bar{f}$ for the increasing function $\hat{D}^n \to \hat{D}$ defined by

$$\bar{f}(A_1, A_2, \dots, A_n) = \sup\{\widetilde{f(b_1, b_2, \dots, b_n)} : b_i \in A_i\}. \tag{4.7}$$

With this definition, the if-then-else from Definition 1.1 is the tight extension of the if-then-else on D.

When we write $\bar{\mathtt{f}}$ for a multiargument function $\mathtt{f}$ we will usually have definition (4.7) in mind, unless something else is stated. However, (4.7) and the tight extension of the corresponding Curried function (i.e. the one tightly extended argument-by-argument) coincide in most cases.

Example. Let D be a domain with values $\mathtt{tt}, \mathtt{ff}$, natural numbers, and lists. Suppose that we work in a lazy (call-by-name) language, so that terms like $\mathtt{cons}(\bot, \mathtt{nil})$ are not equal to $\bot$; thus D is not a flat domain. Let $\hat{D}$ be the domain extension obtained by adjoining to D types Even, Odd, $\mathbb{N}, \mathbb{B}$, $\mathtt{List}$, with the obvious meaning. Then for the tight extension $\overline{\mathtt{cons}}$ of $\mathtt{cons}$ we have:

$$\begin{aligned}
\overline{\mathtt{cons}}(\mathbb{B}, \widetilde{\mathtt{nil}}) &= \sup_{b \in \mathbb{B}} \widetilde{\mathtt{cons}(b, \mathtt{nil})} \\
&= \sup \left\{ \widetilde{\mathtt{cons}(b, \mathtt{nil})} : b \in \mathbb{B} \right\} \\
&= \sup \left\{ \begin{array}{c} \{\bot, \bot :: \bot, \bot :: \mathtt{nil}\}, \\ \{\bot, \bot :: \bot, \bot :: \mathtt{nil}, \mathtt{tt} :: \bot, \mathtt{tt} :: \mathtt{nil}\}, \\ \{\bot, \bot :: \bot, \bot :: \mathtt{nil}, \mathtt{ff} :: \bot, \mathtt{ff} :: \mathtt{nil}\} \end{array} \right\} \\
&= \mathtt{List}.
\end{aligned}$$

The point here is that the only ideal in $\hat{D}$ that contains the set

$$A = \{\bot, \bot :: \bot, \bot :: \mathtt{nil}, \mathtt{tt} :: \bot, \mathtt{tt} :: \mathtt{nil}, \mathtt{ff} :: \bot, \mathtt{ff} :: \mathtt{nil}\}$$

is $\mathtt{List}$. This is not the end of the matter, however: since A is an ideal itself, if one adds it to $\hat{D}$ then then $\overline{\mathtt{cons}}(\mathbb{B}, \widetilde{\mathtt{nil}}) = A$. In other words, $\overline{\mathtt{cons}}(\mathbb{B}, \widetilde{\mathtt{nil}})$ is the type of all lists of Booleans of length 1. The reasoning can be extended lists of higher lengths.

5 Arrow Functions

The aim of this chapter is to define a method for performing *case-analysis*. (Conceptually, case analysis is a method for deriving properties about functions by distinguishing cases about their arguments. For example, one may refine one's knowledge that the function $n \mapsto 3n + 1$ is of type $\mathbb{N} \to \mathbb{N}$ by further analysing cases to obtain that is also of type (EVEN $\to$ ODD) $\sqcap$ (ODD $\to$ EVEN); i.e. it maps even natural numbers to odd, and odd to even.)

Before attempting this, it will be beneficial if we study the properties of the arrow operator $(\cdot \to \cdot)$ we introduced informally earlier.

5.1 Arrow Operator

Definition 5.1. Given $x, y \in \hat{D}$, we define the function $(x \to y) : \hat{D} \to \hat{D}$ to be:

$$(x \to y) \overset{\text{def}}{=} \lambda z.(\text{if } z \sqsubseteq x \text{ then } y \text{ else } U\ ).$$

Obviously, $(x \to y)$ is increasing, so it is in $[\hat{D} \to \hat{D}]$. As argued previously, $(x \to y)$ represents the "function-type" of functions of type $x \to y$, that is functions that given an argument of type x return an argument of type y (and given an argument not of type x, return an argument of the universal type U). We now make this precise:

Theorem 5.2. For any x, y in $\hat{D}$ and any h in $[\hat{D} \to \hat{D}]$ the following are equivalent:

(i) $h \sqsubseteq_{[\hat{D} \to \hat{D}]} (x \to y)$;

(ii) $h(x) \sqsubseteq y$;

(iii) $h(z) \sqsubseteq y$ whenever $z \sqsubseteq x$.

Care must be given to the fact that in (i), $\sqsubseteq$ refers to the pointwise function ordering induced by $\sqsubseteq$ on $\hat{D} \to \hat{D}$. We usually follow the common mathematical practice and suppress such indices on the understanding that no confusion will be caused to the attentive reader.

Proof. Obviously (iii) implies (ii), and conversely, (ii) implies (iii) because h is monotonic. By definition, if $h \sqsubseteq (x \to y)$ then for any $z \sqsubseteq x$ we have $h(z) \sqsubseteq (x \to y)(z) = y$; so (i) implies (iii). Finally, if (iii) holds then for any z in $\hat{D}$, if $z \sqsubseteq x$ then $h(z) \sqsubseteq (x \to y)(z)$, and if not $h(z) \sqsubseteq (x \to y)(z) = U$, showing that $h \sqsubseteq (x \to y)$. So (iii) implies (i). □

We can intuitively understand the arrow operator as a knowledge ordering. Under this interpretation, $\to$ is increasing in its second argument, but decreasing in the first.

Theorem 5.3. For x, x', y, y' in $\hat{D}$,

$$\text{if } x' \sqsubseteq x \text{ and } y \sqsubseteq y', \text{ then } (x \to y) \sqsubseteq (x' \to y').$$

Proof. For z in $\hat{D}$, if $z \not\sqsubseteq x$ then $(x \to y)(z) = U$ and $(x' \to y')(z) = U$, as also $z \not\sqsubseteq x'$. If $x' \not\sqsubseteq z \sqsubseteq x$ then $(x \to y)(z) = y$ and $(x' \to y')(z) = U$. Finally, if $z \sqsubseteq x'$ then $(x \to y)(z) = y$ and $(x' \to y')(z) = y'$. In all cases, $(x \to y)(z) \sqsubseteq (x' \to y')(z)$, as wanted. □

Remark 5.4. In most type systems, types are domains in their own right, and thus for any two types X, Y, the function-type $[X \to Y]$ is itself a domain, which is increasing in both X and Y. In our system, however, the arrow operator is decreasing in x. This behavioural difference is due to the fact in our system, a type can be a subtype of another type.

5.2 Case Analysis

Using the arrow operator, it is relatively easy to construct compound function-types. To be precise, this is achieved by taking the (pointwise) infimum of functions (which we shall abbreviate with the symbol "$\sqcap$" for finitely many functions), as the following theorem shows.

Theorem 5.5. For x, x', y, y', z in $\hat{D}$, we have:

(i) If either $x \sqsubseteq y$ or $y \sqsubseteq z$, then $(x \to y) \sqcap (y \to z) \sqsubseteq (x \to z)$; and

(ii) $(x \to y) \sqcap (x' \to y') \sqsubseteq (x \sqcap x') \to (y \sqcap y')$.

Proof. For (i), we procceed by case analysis. Let $f = (x \to y) \sqcap (y \to z)$. Then for any w,

- if $w \sqsubseteq x$ and $w \sqsubseteq y$, then $f(w) = y \sqcap z$ and $(x \to z)(w) = z$;
- if $w \sqsubseteq x$ and $w \not\sqsubseteq y$, then $f(w) = y \sqcap U = y$ and $(x \to z)(w) = z$;
- if $w \not\sqsubseteq x$ and $w \sqsubseteq y$, then $f(w) = U \sqcap z = z$ and $(x \to z)(w) = U$;
- if $w \not\sqsubseteq x$ and $w \not\sqsubseteq y$, then $f(w) = U \sqcap U = U$ and $(x \to z)(w) = U$.

Then, the only case that $f(w) \sqsubseteq (x \to z)(w)$ may fail is the second case, that there is some w with $w \sqsubseteq x$ but $w \not\sqsubseteq y$, and further $y \not\sqsubseteq z$. But by the assumption, if $y \not\sqsubseteq z$ then $x \sqsubseteq y$, and so no such w exists.

For (ii), by a similar case analysis, we note that if $w \not\sqsubseteq x$ or $w \not\sqsubseteq x'$ then $w \not\sqsubseteq x \sqcap x'$ and so $((x \sqcap x') \to (y \sqcap y'))(w) = U$ so the inequality holds. Now if $w \sqsubseteq x, x'$ then $w \sqsubseteq x \sqcap x'$ and

$$((x \to y) \sqcap (x' \to y'))(w) = y \sqcap y' = ((x \sqcap x') \to (y \sqcap y'))(w). \qquad \square$$

Thus $\sqcap$ acts like intersection of functions in a types-as-sets system. On the other hand, $\sqcup$ is very different from set union, and in fact redundant: for any x, x', y, y',

$$(x \to y) \sqcup (x' \to y') = (x \sqcap x') \to (y \sqcup y').$$

Indeed, if $f = \sup\{(x \to y), (x' \to y')\}$, then $f(z) = \sup\{(x \to y)(z), (x' \to y')(z)\}$ for $z \in \hat{D}$. Thus, if $z \not\sqsubseteq x \sqcap x'$ then either $z \not\sqsubseteq x$ or $z \not\sqsubseteq x'$, so either $(x \to y)(z) = U$ or $(x' \to y')(z) = U$, and $f(z) = U$. If $z \sqsubseteq x \sqcap x'$ then $f(z) = y \sqcup y'$.

Special cases of compound types can be used to perform *case-analysis*. The idea is that we want to perform a finer analysis of the functions of type $x \to y$ (i.e. elements "below" $x \to y$ in the pointwisely-ordered poset of functions).

Definition 5.6. We define a *case-analysis* of $x \to y$ to be a function of the form

$$(x_1 \to y) \sqcap (x_2 \to y) \sqcap \cdots \sqcap (x_n \to y),$$

such that the type x is the *set union* of types $x_1, \ldots, x_n$.

Informally, when we treat D as a subset of $\widehat{D}$, this means that for all $y \in D$,

$$y \sqsubseteq x \Leftrightarrow (\exists i) y \sqsubseteq x_i;$$

formally, we constructed $\widehat{D}$ as a subset of the powerset $\mathscr{P}(D)$ (and identified D with the family of sets of the form $\tilde{d}$), so the condition is to be interpreted literally. As D will in general not be a flat domain, we leave open the possibility that some of the x_i may actually represent elements of D. (Note, however, that the case that x represents an element of D is uninteresting, as then some x_i equals x.)

As an example, $(\textsc{Even} \to \mathbb{R}) \sqcap (\textsc{Odd} \to \mathbb{R})$ is a case analysis of $\mathbb{N} \to \mathbb{R}$. This example shows that a function and its case analysis may not be equal on $\widehat{D}$: applied to $\mathbb{N}$ the first one gives U, whereas the second one $\mathbb{R}$. However:

Lemma 5.7. For a case analysis $f \overset{\text{def}}{=} (x_1 \to y) \sqcap \cdots \sqcap (x_n \to y)$ of $x \to y$ and any z in D,

$$f(z) = (x \to y)(z).$$

Proof. That z is in D is an abuse of notation to say that $z = \tilde{d}$ for some $d \in D$; fix that (unique) d. There are two cases: $z \subseteq x$ or not. If not, then also $z \not\subseteq x_i$ for all i, and hence $f(z) = U = (x \to y)(z)$. If $z \subseteq x$, then $d \in z \subseteq x$ and thus for some i, $d \in x_i$, meaning (because types are downwards-closed) that $z = \tilde{d} \subseteq x_i$, and thus $(x \to y)(z) = y = f(z)$. (The last equality is true because every $(x_i \to y)(z)$ is either U or y.) □

The relation between f in the notation of the preceding lemma and $(x \to y)$ is also more profound:

Theorem 5.8. If f is a case analysis of $x \to y$ then $\bar{f} = (x \to y)$.

Proof. Recall that the tightening $\bar{f}$ of f satisfies

$$\bar{f}(z) = \sup_{d \in z} f\left(\tilde{d}\right).$$

As $f(\tilde{d}) = (x \to y)(\tilde{d})$ by Lemma 5.7, we have to show that

$$(x \to y)(z) = \sup_{d \in z} (x \to y)\left(\tilde{d}\right).$$

If $z \subseteq x$ then for all $d \in z$, $\tilde{d} \subseteq x$ so $(x \to y)(\tilde{d}) = y$, and the two sides are equal indeed. If $z \not\subseteq x$ then there is some $d \in z \setminus x$, and $\tilde{d} \not\subseteq x$ so $(x \to y)(\tilde{d}) = U$ and again the two sides are equal. □

At this point, we remind that while we constructed types as sets of objects (*objects* being a word for "elements of D"), this is only a formal distinction, to convince ourselves that the approach we propose is consistent, and that in fact one should think that types are (some kind of) generalized objects themselves. From this point of view, we have the ability to add types dynamically to a (PCF-like) programming language. In fact it is precisely this ability together with the distinction between a function and its case analysis that allows us to use case analysis to prove properties about recursive programs (as we will see in Chapter 7), and the inability to perform such analysis in a traditional set-based type system that inhibits this method of proof in such systems.

6 Recursion over Extended Domain

Before we procceed to study proofs by case analysis, we will explore the relation between the meaning (semantics/model) of recursive programs over D and over $\hat{D}$. Assume τ is a term (also called *functional*) in our PCF-like language, and consider the recursive program

$$F = \tau[F]. \tag{6.1}$$

The semantics of this has already been discussed in Chapter 1. Given an interpretation[(1)] b in the target domain D of the function-symbols, put $Y\tau_b \overset{\text{def}}{=} \mathit{lfp}(\tau_b)$, the least fixed point of τ_b. As an example, let

$$\tau_![F] = \lambda n.\text{if } n < 1 \text{ then } 1 \text{ else } n \times F(n-1)\ .$$

In this example, the primitive language symbols used are the "λ", "if-then-else", "$<$", "$+$", "$-$" and "$\times$", of which all except "λ" are function-symbols. If we take as our target domain $(\mathbb{N} \cup \{\mathtt{tt}, \mathtt{ff}\})_\perp$ (i.e. $\mathbb{N}$ together with truth-values flatly ordered) and interpret the function-symbols as the operations in $\mathbb{N}$ they usually stand for (with the usual definition that $a - b = 0$ if $a < b$), then the least fixed point of this recursive program is the factorial function.

Our aim in this chapter is to establish a compatibility result for the solution of recursive programs over D and $\hat{D}$. We fix some notation which is used in this and the following chapters: let

- D be a domain with a type extension $\hat{D}$;
- τ be a term (functional) of our programming language;
- b be an interpretation in D of function-symbols appearing in τ;
- $\hat{b}$ be an interpretation in $\hat{D}$ extending b (i.e. for every function-symbol $\mathtt{f}$, $\mathtt{f}_{\hat{b}}$ extends $\mathtt{f}_b$);
- $\bar{b}$ be the interpretation in $\hat{D}$ consisting of the *tight extensions* of the functions interpreted by b.

Lemma 6.2. With the notational convention made above, for any g in $[D \to D]$ with extension $\hat{g}$ in $[\hat{D} \to \hat{D}]$,

$$\tau_{\hat{b}}[\hat{g}] \text{ is an extension of } \tau_b[g]. \tag{6.3}$$

Proof. We want to show that for any $d \in D$,

$$\tau_{\hat{b}}[\hat{g}]\left(\tilde{d}\right) = \widetilde{\tau_b[g](d)}. \tag{6.4}$$

[The outline of the proof is as follows: The term $\tau_{\hat{b}}[\hat{g}]$ depends only on functional constants which are interpreted by b and $\hat{b}$, and on the function $\hat{g}$. Of these two interpretations, one is an extension of the other, and $\hat{g}$ also extends g. Thus in the process of evaluating[(2)] $\tau_{\hat{b}}[\hat{g}](\tilde{d})$, we will only deal with subterms of the form $\mathtt{f}_{\hat{b}}(\tilde{e})$ and $\hat{g}(\tilde{e})$, where $\mathtt{f}_{\hat{b}}$ stands for the interpretation of functional symbol $\mathtt{f}$ by $\hat{b}$. Since $\mathtt{f}_{\hat{b}}(\tilde{e}) = \widetilde{\mathtt{f}_b(e)}$ and $\hat{g}(\tilde{e}) = \widetilde{g(e)}$ for $e \in D$ and all functions involved ($g, \hat{g}$ and those interpreted by $\hat{b}$) are increasing, we conclude that the computations of $\tau_b[g](d)$ and $\tau_{\hat{b}}[\hat{g}](\tilde{d})$ are parallel in the sense that

(1) Recall that this means that, like in first-order logic, we assign a function f in $[D^n \to D]$ to every n-ary function-symbol $\mathtt{f}$ in the programming language.

in every step where the first computes a the second computes $\tilde{a}$, and also the second one always computes terms of this form (i.e. elements of the embedding of D into $\hat{D}$).]

For the formal proof, note that τ was assumed to be a functional of our programming language; this means that it is expressible in our programming language, and therefore we can use structural induction on subterms of $\tau[\mathrm{g}](\mathtt{d})$ to prove the result.

To be precise, fix $d \in D$, and incorporate to our programming language a new function-symbol g and a new constant d, and expand interpretations $b, \hat{b}$ to interpret g as $g, \hat{g}$ and d as $d, \tilde{d}$ respectively. Now set $T = \tau[\mathrm{g}](\mathrm{d})$, which is a closed term of our programming language. We prove by structural induction on T that for every subterm t of T,

$$[\![t]\!] = \widetilde{\langle t \rangle},$$

where $\langle t \rangle$, $[\![t]\!]$ are the meaning (interpreted under b,$\hat{b}$) of t in D, $\hat{D}$ respectively.

Every subterm of $\tau[\mathrm{g}](\mathrm{d})$ is of the form:

- c, for constant c – evidently the result holds for constants;
- $\mathtt{f}(t_1, \ldots, t_n)$ for n-ary function symbol f and terms $t_1, \ldots, t_n$ – as the interpretation of f by $\hat{b}$ is an extension of its interpretation by b, and the result holds for subterms t_i (by induction), we conclude that the result holds for this case as well;
- $\mathrm{g}(t)$ – by assumption, the meaning of g in $\hat{D}$, namely $\hat{g}$, is an extension of its meaning in D, namely g; thus the result holds for this case as well. □

Corollary 6.5. With the notation of Lemma 6.2, if $\bar{b}$ is a *tight* interpretation of functions in b, we have:

$$\overline{\tau_{\hat{b}}[\hat{g}]} \overset{\text{(i)}}{=} \overline{\tau_b[g]} \overset{\text{(ii)}}{\sqsubseteq} \tau_{\bar{b}}[\bar{g}] \overset{\text{(iii)}}{\sqsubseteq} \tau_{\bar{b}}[\hat{g}] \overset{\text{(iv)}}{\sqsubseteq} \tau_{\hat{b}}[\hat{g}].$$

Proof. (iii) and (iv) are similar to the preceding proof. We prove (i). First note that the bars mean different things in the two sides of the equation: in the left, it means the tightening; in the right, the tight extension.

Let x be in $\hat{D}$. Then by definition

$$\overline{\tau_{\hat{b}}[\hat{g}]}(x) = \sup_{d \in x} \tau_{\hat{b}}[\hat{g}]\left(\tilde{d}\right).$$

Similarly, by definition,

$$\overline{\tau_b[g]}(x) = \sup_{d \in x} \widetilde{\tau_b[g](d)}.$$

But by Lemma 6.2 (i.e. eq. (6.4)), the right sides of the two equations are equal, so (i) is proven. Finally, apply (i) to the special case $\hat{b} = \bar{b}$ and $\hat{g} = \bar{g}$ to get $\overline{\tau_{\bar{b}}[\bar{g}]} = \overline{\tau_b[g]}$. Since $\overline{\tau_{\bar{b}}[\bar{g}]} \sqsubseteq \tau_{\bar{b}}[\bar{g}]$, we get (ii). □

Theorem 6.6. With the notation of Lemma 6.2, the least fixed point $Y\tau_{\hat{b}}$ is an extension of the least fixed point $Y\tau_b$ of τ_b:

$$Y\tau_{\hat{b}}\left(\tilde{d}\right) = \widetilde{Y\tau_b(d)} \quad \text{for every} \quad d \in D. \tag{6.7}$$

Proof. To avoid notational clashes, we will write $\bigsqcup_{i \in I} A_i$ to denote the least upper bound (supremum) of ideals A_i in $\hat{D}$ – recall that $\hat{D}$ is a complete lattice by Theorem 3.5, and hence this supremum exists; we will reserve "sup" for the supremum of directed sets in A.

Since D is chain-complete (being a domain) and $\hat{D}$ is a complete lattice, $[D \to D]$ and $[\hat{D} \to \hat{D}]$ are also chain-complete, and hence the least fixed points of $\tau_{\hat{b}}$, τ_b are given by the construction in Theorem 2.3. Define the ordinal-sequences f_α, $\hat{f}_\alpha$ by recursion: $f_0 = \lambda d.\bot$, $\hat{f}_0 = \lambda x.\tilde{\bot}$, and for $\alpha > 0$,

$$\begin{aligned} f_\alpha &= \tau_b[f_\beta] \quad \text{and} \quad \hat{f}_\alpha = \tau_{\hat{b}}[\hat{f}_\beta] \quad \text{if } \alpha = \beta + 1, \\ f_\alpha &= \sup_{\beta < \alpha} f_\beta \quad \text{and} \quad \hat{f}_\alpha = \bigsqcup_{\beta < \alpha} \hat{f}_\beta \quad \text{if } \alpha \text{ is a limit,} \end{aligned}$$

(2) We haven't specified an *evaluation strategy* for our language, but this does not affect the result as long as we choose one consistently for both D and $\hat{D}$.

where the operation of supremum is defined pointwise. Recall that each of the sequences (f_α), $(\hat{f_\alpha})$ is pointwise increasing (Theorem 2.3). We will prove by induction on α that

$$\hat{f_\alpha} \text{ is an extension of } f_\alpha.$$

There are three cases to examine.

- *Zero*: $\hat{f_0}(\tilde{d}) = \tilde{\bot} = \widetilde{f_0(d)}$, so $\hat{f_0}$ is an extension of f_0.
- *Successor*: Suppose that $\alpha = \beta + 1$ and that $\hat{f_\beta}$ is an extension of f_β. Then

  $$\hat{f}_{\beta+1} = \tau_{\hat{b}}[\hat{f_\beta}] \text{ is an extension of } \tau_b[f_\beta] = f_{\beta+1}$$

  by Lemma 6.2. Thus $\hat{f_\alpha}$ is an extension of f_α.
- *Limit*: Finally suppose that α is a limit ordinal, and that for every $\beta < \alpha$, $\hat{f_\beta}$ is an extension of f_β. This means that for every $\beta < \alpha$ and every $d \in D$, $\hat{f_\beta}(\tilde{d}) = \widetilde{f_\beta(d)}$. Recall that

  $$f_\alpha(d) = \sup_{\beta<\alpha} f_\beta(d) \text{ and } \hat{f_\alpha}\left(\tilde{d}\right) = \bigsqcup_{\beta<\alpha} \hat{f_\beta}\left(\tilde{d}\right),$$

  and note that the inductive hypothesis yields

  $$\hat{f_\alpha}\left(\tilde{d}\right) = \bigsqcup_{\beta<\alpha} \hat{f_\beta}\left(\tilde{d}\right) = \bigsqcup_{\beta<\alpha} \widetilde{f_\beta(d)}. \tag{6.8}$$

  Now $f_\beta(d) \sqsubseteq f_\alpha(d) \Rightarrow \widetilde{f_\beta(d)} \subseteq \widetilde{f_\alpha(d)}$ for $\beta < \alpha$, i.e. $\widetilde{f_\alpha(d)}$ is an upper bound of $\left\{\widetilde{f_\beta(d)} : \beta < \alpha\right\}$, and hence it is greater ($\supseteq$) than the *least* upper bound, which is $\hat{f_\alpha}(\tilde{d})$ by (6.8). This establishes that

  $$\hat{f_\alpha}\left(\tilde{d}\right) \subseteq \widetilde{f_\alpha(d)}.$$

  Conversely, we show that $\widetilde{f_\alpha(d)} \subseteq \hat{f_\alpha}(\tilde{d})$; this is equivalent to showing

  $$f_\alpha(d) \in \hat{f_\alpha}\left(\tilde{d}\right) \overset{(6.8)}{=} \bigsqcup_{\beta<\alpha} \widetilde{f_\beta(d)}.$$

  But for every $\beta < \alpha$, $f_\beta(d) \in \widetilde{f_\beta(d)} \subseteq \hat{f_\alpha}(\tilde{d})$ which is an ideal (being an element of $\hat{D}$); thus $\{f_\beta(d) : \beta < \alpha\}$ is a directed subset (in fact a chain) of $\hat{f_\alpha}(\tilde{d})$, so its supremum $f_\alpha(d)$ is also in $\hat{f_\alpha}(\tilde{d})$ by Definition 3.2. This concludes the induction.

Finally, for any $d \in D$,

$$\begin{aligned} Y\tau_{\hat{b}}\left(\tilde{d}\right) &= \bigsqcup_{\alpha\in\mathbf{Ord}} \hat{f_\alpha}\left(\tilde{d}\right) \\ &= \bigsqcup_{\alpha\in\mathbf{Ord}} \widetilde{f_\alpha(d)} \\ &= \widetilde{\sup_{\alpha\in\mathbf{Ord}} f_\alpha(d)} && \text{because the increasing sequence } f_\alpha(d) \text{ stabilises to a greatest element} \\ &= \widetilde{Y\tau_b(d)}. \end{aligned}$$

This concludes the proof of the theorem. □

Remark 6.9. Generally, all functionals one encounetrs in practice are continuous (cf. Chapter 2), and hence have closure ordinal ω. In the proof we used Lemma 6.2 which assumed τ to be a programming functional, so the theorem needs to make this assumption as well. Thus, if the proof causes confusion, it may be read to terminate at ω; however we still need the limit case. However, we chose to write the prooff this theorem holds more generally even for non-continuous functionals. (Of course, if for a non-continuous functional the result of Lemma 6.2 holds, the conclusion of the theorem also holds.)

Corollary 6.10. With the notation of Theorem 6.6, and for types x, y in $\hat{D}$, and $f, \hat{f}$ are the least fixed points of $\tau_b, \tau_{\hat{b}}$, if $\bar{f} \sqsubseteq (x \to y)$ or $\hat{f} \sqsubseteq (x \to y)$ then $f(d) \in y$ for all $d \in x$.

Proof. Recall that for f in $[D \to D]$ and $h \in [\hat{D} \to \hat{D}]$, h is an extension of f iff $\bar{f} \sqsubseteq h$. Since $\hat{f}$ is an extension of f, it suffices to consider only the case for $\bar{f}$. Let $d \in x$. Then $\tilde{d} \subseteq x$ and $\bar{f}(\tilde{d}) \subseteq y$, so $\widetilde{f(d)} = \bar{f}(\tilde{d}) \subseteq y$. Consequently, $f(d) \in y$. ☐

Corollary 6.11. With the notation of Theorem 6.6,

$$\overline{Y\tau_{\hat{b}}} = \overline{Y\tau_b} \sqsubseteq Y\tau_{\bar{b}} \sqsubseteq Y\tau_{\hat{b}}.$$

Proof. This is similar to Corollary 6.5. For $x \in \hat{D}$,

$$\overline{Y\tau_{\hat{b}}}(x) = \bigsqcup_{d \in x} Y\tau_{\hat{b}}\left(\tilde{d}\right) = \bigsqcup_{d \in x} \widetilde{Y\tau_b(d)} = \overline{Y\tau_b}(x).$$

The inequality $\overline{Y\tau_b} \sqsubseteq Y\tau_{\bar{b}}$ is proved as in Corollary 6.5, by considering the special case $\hat{b} = \bar{b}$. For the final inequality, let f be in $[\hat{D} \to \hat{D}]$ and observe that $\tau_{\bar{b}}[f] \sqsubseteq \tau_{\hat{b}}[f]$. The result is obtained by iterating over the ordinals the application of $\tau_{\bar{b}}, \tau_{\hat{b}}$ on $f = \lambda x.\bot$. ☐

7 Type Checking

In this chapter, our aim is to show how we can use the theory developed so far to perform basic type-checking, and also property proving. For our purposes, to type check a program $F = \tau[F]$ is to prove an inequality of the form

$$\overline{Y\tau_b} \sqsubseteq (x_1 \to y_1) \sqcap (x_2 \to y_2) \sqcap \cdots \sqcap (x_n \to y_n).$$

The results in Chapter 6, especially Corollary 6.11, may be useful.

Example. Suppose our program is

$$F(n) = \tau[F](n) \equiv \text{if } (n = 0) \text{ then } 0 \text{ else } (3 \times F(n-1) + 1) \ , \tag{7.1}$$

and we want to show that the least fixed point (with the standard interpretations of operations on $\mathbb{N}$ and the *tight* if-then-else defined in Chapter 1) maps even integers to even integers, and odd to odd. By Corollary 6.11 it suffices to show that

$$Y\tau_{\hat{b}} \sqsubseteq (\text{Even} \to \text{Even}) \sqcap (\text{Odd} \to \text{Odd}),$$

for some suitable interpretation $\hat{b}$ (which we define below).

Now proving a property of the program is reduced to proving an assertion about least fixed points, which is a well-studied problem.

In our particular problem, we go for brute-force direct computation. Fist we will rewrite(1) the program:

$$F(n) = \text{if Equals}(n, 0) \text{ then } 0 \text{ else Add1}(\text{MultBy3}[F(\text{Pred}(n))]) \ , \tag{7.2}$$

where $\text{Equals}(a, b) \equiv (a = b)$, $\text{Add1}(n) \equiv (n + 1)$, $\text{MultBy3}(b) \equiv (3 \times b)$, $\text{Pred}(n) = (n - 1)$ (the latter is 0 if $n = 0$). We rewrote the equation in this form to avoid misunderstandings.

Now consider the interpretations b and $\hat{b}$ defined by

$$b = \langle (\text{if } \cdot \text{ then } \cdot \text{ else} \cdot); \text{Equals}_{\mathbb{N}}; \text{Add1}_{\mathbb{N}}; \text{MultBy3}_{\mathbb{N}}; \text{Pred}_{\mathbb{N}} \rangle$$

and

$$\hat{b} = \langle (\text{if } \cdot \text{ then } \cdot \text{ else} \cdot)_{\text{tight}}; \overline{\text{Equals}_{\mathbb{N}}}; \overline{\text{Add1}_{\mathbb{N}}}; \overline{\text{MultBy3}_{\mathbb{N}}}; \overline{\text{Pred}_{\mathbb{N}}} \rangle,$$

where $\text{Equals}_{\mathbb{N}}$ (&c.) stands for the standard equality (&c.) in $\mathbb{N}$, $\overline{\text{Equals}_{\mathbb{N}}}$ (&c.) stands for its tight extension to $\hat{D}$, and (as we defined it in the introduction)

$$\begin{aligned} \text{if } \texttt{tt} \text{ then } a \text{ else } b &= a; \\ \text{if } \texttt{ff} \text{ then } a \text{ else } b &= b; \\ \text{if } \mathbb{B} \text{ then } a \text{ else } b &= a \sqcup b = \sup\{a, b\}. \end{aligned}$$

In the following we will use the more familiar $=, +, \times, -$, understanding them to be shorthands for Equals &c. interpreted by $\hat{b}$. For any function F in $[\hat{D} \to \hat{D}]$,

$$\tau[F](\text{Even}) = \text{if } (\text{Even} = 0) \text{ then } 0 \text{ else } (3 \times F(\text{Even} - 1) + 1) \ .$$

(1) This step is not essential and the reader may omit it without loss of understanding.

But $\textsc{Equals}$ is tight, so $\textsc{Equals}(\textsc{Even}, 0) = \sup_{n \sqsubseteq \textsc{Even}} \textsc{Equals}(n, 0) = \mathbb{B}$, because the expression $\textsc{Equals}(n, 0)$ evaluates to `tt` for $n = 0 \sqsubseteq \textsc{Even}$, and `ff` for $0 \neq n \sqsubseteq \textsc{Even}$. Similarly, because $\textsc{Pred}$ is tight, $\textsc{Pred}(\textsc{Even}) = \sup_{n \sqsubseteq \textsc{Even}} \textsc{Pred}(n) = \textsc{Odd}$, and thus

$$\tau[F](\textsc{Even}) = [\text{if } \mathbb{B} \text{ then } 0 \text{ else } (3 \times F(\textsc{Odd}) + 1)\ ] = 0 \sqcup (3 \times F(\textsc{Odd}) + 1). \tag{7.3}$$

By similar but simpler arguments we can show that

$$\tau[F](\textsc{Odd}) = 3 \times F(\textsc{Even}) + 1. \tag{7.4}$$

In particular, (7.3), (7.4) hold when F is $Y\tau_{\hat{b}}$, which is given by

$$F = \sup_{\alpha \in \mathbf{Ord}} \tau^{(\alpha)}[\text{constantly}_\perp],$$

where constantly$_\perp$ is the constantly $\perp$ ("constantly undefined") function; this equation is an instance of Theorem 2.3 for the functional τ.[(2)] Thus we may compute the values of $F(\textsc{Even})$, $F(\textsc{Odd})$ for $F = Y\tau_{\hat{b}}$ by successive approximations, i.e. by successively applying the map

$$\begin{pmatrix} A \\ B \end{pmatrix} \mapsto \begin{pmatrix} 0 \sqcup [3 \cdot B + 1] \\ [3 \cdot A + 1] \end{pmatrix}$$

beginning with $(\perp, \perp)$. We get:

$$\begin{aligned}
(\perp, \perp) &\mapsto (0 \sqcup (3\perp + 1);\ (3\perp + 1)) && = (0; \perp) \\
&\mapsto (0 \sqcup (3\perp + 1);\ (3 \times 0 + 1)) && = (0; 1) \\
&\mapsto (0 \sqcup (3 \times 1 + 1);\ (3 \times 0 + 1)) && = (0 \sqcup 4;\ 1) = (\textsc{Even}; 1) \\
&\mapsto (0 \sqcup (3 \times 1 + 1);\ 3 \times \textsc{Even} + 1) && = (0 \sqcup 4;\ \textsc{Odd}) = (\textsc{Even}, \textsc{Odd}),
\end{aligned}$$

and we are done (the process stabilises at this point: $(\textsc{Even}, \textsc{Odd}) \mapsto (\textsc{Even}, \textsc{Odd})$).

Another approach to obtain the result would be to show that $(\textsc{Even} \to \textsc{Even}) \sqcap (\textsc{Odd} \to \textsc{Odd})$ is a prefixed point of $\tau_{\hat{b}}$; as $Y\tau_{\hat{b}} = \mathit{lfp}(\tau_{\hat{b}})$ is the least prefixed point, this implies the result.

Let g is the function $(\textsc{Even} \to \textsc{Even}) \sqcap (\textsc{Odd} \to \textsc{Odd})$. We want to show the inequality $\tau_{\hat{b}}(g) \sqsubseteq g$. The calculations now are even more straightforward:

$$\begin{aligned}
\tau_{\hat{b}}(g)(\textsc{Even}) &= \text{if } (\textsc{Even} = 0) \text{ then } 0 \text{ else } 3 \times g(\textsc{Even} - 1) + 1 \\
&= \text{if } \mathbb{B} \text{ then } 0 \text{ else } 3 \times g(\textsc{Odd}) + 1 \\
&= 0 \sqcup (3 \times g(\textsc{Odd}) + 1) \\
&= 0 \sqcup (3 \times (U \sqcap \textsc{Odd}) + 1) \\
&= 0 \sqcup (3 \times \textsc{Odd} + 1) \\
&= 0 \sqcup (\textsc{Odd} + 1) \\
&= 0 \sqcup \textsc{Even} \\
&= \textsc{Even},
\end{aligned}$$

and similarly for $\tau_{\hat{b}}(g)(\textsc{Odd})$.

(2) As a side comment, we remark that almost all functionals arising in practice are continuous as they are defined using monotonic functions, so we need only consider the ordinals $\alpha < \omega$.

8 Case Analysis of Recursive Programs

The methods described in the previous chapter fail to produce results when some sort of analysis by cases is required. As a very simple example, consider the functional

$$\tau[F] = \lambda n.\text{if } (n = 0) \text{ then } n \text{ else } 0\ .$$

Then the solution f to the program $F = \tau[F]$ over D is the constantly zero function; but we cannot prove for its fixed point $\hat{f}$ in $\hat{D}$ that $\hat{f} \sqsubseteq (\mathbb{N} \to 0)$[1] ; indeed, while $(\mathbb{N} \to 0)(\mathbb{N}) = 0$, with the interpretation of "=" as $\overline{\text{Equals}}$ from the previous chapter one has

$$\hat{f}(\mathbb{N}) = \tau[\hat{f}](\mathbb{N}) = [\text{if } (\mathbb{N} = 0) \text{ then } \mathbb{N} \text{ else } 0\,] = [\text{if } \mathbb{B} \text{ then } \mathbb{N} \text{ else } 0\,] = [\mathbb{N} \sqcup 0] = \mathbb{N}.$$

In this chapter we show how we can exploit the fact that types can be added at will to perform a satisfactory case analysis of such programs. Suppose that we add a new type $\mathbb{N}_+$ for positive integers to our domain $\hat{D}$. Then (with the strict interpretation of equality), $(\mathbb{N}_+ = 0)$ is `ff`, and hence

$$\hat{f} \sqsubseteq (0 \to 0) \sqcap (\mathbb{N}_+ \to 0).$$

Since the right side is a case analysis of $(\mathbb{N} \to 0)$, by Theorem 5.8 we obtain

$$\bar{\hat{f}} = \bar{f} \sqsubseteq (\mathbb{N} \to 0),$$

and *from this* we conclude that f applied to every integer is 0.

Let us recapitulate: we wanted to show that a certain recursive program $F = \tau[F]$ satisfies a simple property (Π) which is equivalent to showing that (for a suitable interpretation b in our target domain D with extension $\hat{b}$ in $\hat{D}$) the least fixed point $\hat{f}$ of $\tau_{\hat{b}}$ satisfies $\hat{f} \sqsubseteq (a \to b)$ for some types a, b; and it may sometimes happen that this is not the case, although the program does indeed satisfy property (Π). This is a general problem, whose true extent does not appear in this simplistic example (although, as in the example above, it is genenrally the value of $\hat{f}(a)$ that breaks the proof; one might try to circumvent this by adding more types to the system: sometimes this solves the problem, and sometimes not).

The solution suggested above is simple: *split the proof into cases*, i.e. prove instead that $\bar{f} \sqsubseteq g$ for a *suitable case-analysis* g of $(a \to b)$; that way we avoid difficulties about $\hat{f}(a)$. Now there are in principle many ways to prove this inequality; one that was hinted in the last chapter (chapter 7) is to prove that g is a *prefixed point* of $\tau_{\hat{b}}$. The advantage of this method is that one does not need to compute $\overline{Y\tau_b}$, or in fact know anything about it.

Theorem 8.1. With the notation of Lemma 6.2, if g is any element of $[\hat{D} \to \hat{D}]$,

$$\tau_{\hat{b}}[\bar{g}] \sqsubseteq g \quad \text{implies} \quad \overline{Y\tau_b} \sqsubseteq g.$$

Proof. Define the sequence f_α as in the proof of Theorem 6.6. For every ordinal α we shall show for the tight extension $\bar{f}_\alpha$ of f_α that $\bar{f}_\alpha \sqsubseteq g$. The base of the induction $(\alpha = 0)$ is trivial. Now assume that it holds for α. Then:

$$\begin{aligned} \bar{f}_\alpha \sqsubseteq g \Rightarrow\ & \bar{f}_\alpha = \bar{\bar{f}}_\alpha \sqsubseteq \bar{g} \\ \Rightarrow\ & \tau_{\hat{b}}[\bar{f}_\alpha] \sqsubseteq \tau_{\hat{b}}[\bar{g}] \\ \Rightarrow\ & \tau_{\hat{b}}[\bar{f}_\alpha] \sqsubseteq g && \text{by assumption} \\ \Rightarrow\ & \overline{\tau_b[f_\alpha]} \sqsubseteq g && \text{by Corollary 6.5} \\ \Rightarrow\ & \bar{f}_{\alpha+1} \sqsubseteq g && \text{by definition of } f_{\alpha+1}. \end{aligned}$$

[1] For convenience, we shall write 0 instead of the –technically correct– $\tilde{0}$, and shall do likewise for other elements of D, keeping only distinct notations for functions on D and their extensions on $\hat{D}$: the latter's names will usually have "hats" ($\hat{}$).

The limit case is immediate. Thus we've proven that $\bar{f_\alpha} \sqsubseteq g$ for all ordinals α.

Using this, we will show that $\overline{Y\tau_b} \sqsubseteq g$ (remember that $Y\tau_b = \sup_\alpha f_\alpha$, the supremum being taken *over all ordinals*). Indeed, for every x in D we have:

$$\begin{aligned}
\overline{Y\tau_b}(x) &= \sup_{d\in x} \widetilde{Y\tau_b(d)} \\
&= \sup_{d\in x} \widetilde{\sup_\alpha f_\alpha(d)} \\
&= \sup_{d\in x} \sup_\alpha \widetilde{f_\alpha(d)} && \text{because } \sup_\alpha f_\alpha(d) \text{ stabilises} \\
&= \sup_\alpha \sup_{d\in x} \widetilde{f_\alpha(d)} \\
&= \sup_\alpha \bar{f_\alpha}(x) \\
&\sqsubseteq \sup_\alpha g(x) && \text{by induction above} \\
&= g(x). && \square
\end{aligned}$$

Example. We shall now illustrate the importance of this theorem. Suppose that our programming language supports both *primitive* (truth-values, natural numbers, the empty list, and the empty tree) and also two kinds of *composite* objects: binary trees and strings.[2]

Suppose, further, that it has six specialised operators to work with these composite objects: $\texttt{left}(t)$, $\texttt{right}(t)$, $\texttt{atom?}(t)$, $\texttt{grow}(p, t, t')$, $\texttt{list}(a)$, and $s_1 {*} s_2$. These stand (in order) for the left subtree of t, the right subtree of t, a check if t is an atom (i.e. single-node tree or, equivalently, single-object string), the construction of a new tree with primitive object p as root and t [resp. t'] as left [resp. right] subtree, the construction of a list consisting of object a, and the concatenation of strings s_1, s_2.

Now consider the recursive program

$$F(t) = \text{if } \texttt{atom?}(t) \text{ then } t \text{ else } [F(\texttt{left}(t)) {*} F(\texttt{right}(t))]$$

It should be easy to verify that F flattens a tree into a string. Let $\hat{D}$ be obtained by adjoining to D the types TREE and STRING and let the operations mentioned above be extended tightly.[3] The property we *would like* to prove is that the least fixed point of this program, call it $\hat{f}$, lies below (TREE → STRING). But this is not true:

$$\hat{f}(\textsc{Tree}) = \text{if } \mathbb{B} \text{ then } \textsc{Tree} \text{ else } [\hat{f}(\textsc{Tree}) {*} \hat{f}(\textsc{Tree})] \;= \textsc{Tree} \sqcup [\hat{f}(\textsc{Tree}) {*} \hat{f}(\textsc{Tree})] \sqsupseteq \textsc{Tree},$$ [4]

and certainly TREE $\not\sqsubseteq$ STRING.

Even if we add two more types, ATOM and NONATOM, below TREE, we still fail to prove that $\hat{f} \sqsubseteq g :=$ (ATOM → STRING) $\sqcap$ (NONATOM → STRING), for this would imply that $\hat{f}$(ATOM) $\sqsubseteq$ STRING and $\hat{f}$(NONATOM) $\sqsubseteq$ STRING; but simple calculations give $\hat{f}$(ATOM) = ATOM and $\hat{f}$(NONATOM) = $\hat{f}$(TREE)$*\hat{f}$(TREE), and as we saw above, $\hat{f}$(TREE) $\not\sqsubseteq$ STRING.

If, however, we use Theorem 8.1, then $\bar{g}$ = (TREE → STRING) and we have to show that

$$\tau_{\hat{b}}[\textsc{Tree} \to \textsc{String}] \sqsubseteq (\textsc{Atom} \to \textsc{String}) \sqcap (\textsc{NonAtom} \to \textsc{String}).$$

This is true, since

$$\tau_{\hat{b}}[\textsc{Tree} \to \textsc{String}](u) = \text{if } \texttt{atom?}(u) \text{ then } u \text{ else } (\textsc{Tree} \to \textsc{String})(\texttt{left}(u)) {*} (\textsc{Tree} \to \textsc{String})(\texttt{right}(u))$$

[2] Strings are plain lists of primitive objects. Although every string can be represented by a binary tree every node of which has empty left subtree, we only identify *single-object* strings with *single-node* trees, which we call *atoms*. (The LISP programming language is relevant, in that it represents both strings and trees as *lists*. In LISP, lists can be nested: we leave open the possibility that our language may be LISP-like, and strings and trees are special instances of lists as well.)

[3] Of the statements below, those that do not hold for the tight extension will not hold for any other extension as well.

[4] It is interesting to examine what is the least solution to $a = \textsc{Tree} \sqcup (a {*} a)$. The answer seems to be TREE.

and thus

$$\begin{aligned}
\tau_{\hat{b}}[\textsc{Tree} \to \textsc{String}](\textsc{Atom}) &= \textsc{Atom} \sqsubseteq \textsc{String} \\
\tau_{\hat{b}}[\textsc{Tree} \to \textsc{String}](\textsc{NonAtom}) &= (\textsc{Tree} \to \textsc{String})(\texttt{left}(\textsc{NonAtom}))*(\textsc{Tree} \to \textsc{String})(\texttt{right}(\textsc{NonAtom})) \\
&= (\textsc{Tree} \to \textsc{String})(\textsc{Tree})*(\textsc{Tree} \to \textsc{String})(\textsc{Tree}) \\
&= \textsc{String}*\textsc{String} \\
&= \textsc{String}.
\end{aligned}$$

Example (McCarthy's 91-function)**.** We will now study McCarthy's famous 91-function:

$$F(n) = \tau[F](n) \equiv \text{if } (n \le 100) \text{ then } F(F(n+11)) \text{ else } (n-10)\ . \tag{8.2}$$

Let D be the the domain in Figure 8.1, b be the standard interpretation of the function-symbols in τ and $\bar{b}$ the associated interpretation of tight extensions. Put $f = Y\tau_b$. It is not difficult (but takes some thought) to prove that $f(n) = 91$ for $n \le 100$. Here we prove the more modest result that $\bar{f} \sqsubseteq (\mathbb{N} \to \mathbb{N}_{\ge 91})$.

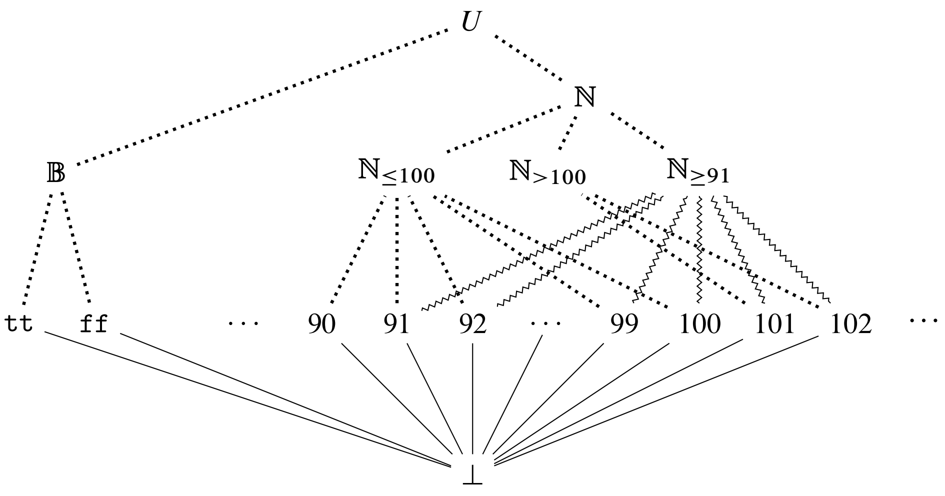

Figure 8.1: Domain D with extension $\hat{D}$. Solid lines show the ordering of elements of D, and dotted ones the object-type relations and type inclusions in the extension $\hat{D}$.

We use the case analysis $g \overset{\text{def}}{=} (\mathbb{N}_{\le 100} \to \mathbb{N}_{\ge 91}) \sqcap (\mathbb{N}_{>100} \to \mathbb{N}_{\ge 91})$ of $(\mathbb{N} \to \mathbb{N}_{\ge 91})$. By Theorem 8.1, it suffices to show that $\tau_{\bar{b}}[\bar{g}] \sqsubseteq g$.

To prove this, it suffices to show that $\tau_{\bar{b}}[\bar{g}] \sqsubseteq (\mathbb{N}_{\le 100} \to \mathbb{N}_{\ge 91})$ and $\tau_{\bar{b}}[\bar{g}] \sqsubseteq (\mathbb{N}_{>100} \to \mathbb{N}_{\ge 91})$. Both are easy. By Theorem 5.2, we have:

$$\begin{aligned}
\tau_{\bar{b}}[\bar{g}](\mathbb{N}_{>100}) &= \text{if } (\mathbb{N}_{>100} \le 100) \text{ then } \bar{g}(\bar{g}(\mathbb{N}_{>100} + 11)) \text{ else } (\mathbb{N}_{>100} - 10) \\
&= \text{if } \texttt{ff} \text{ then } \cdots \text{ else } (\mathbb{N}_{>100} - 10) \\
&= \mathbb{N}_{>100} - 10 \\
&= \{91, 92, 93, \ldots\} = \mathbb{N}_{\ge 91}.
\end{aligned}$$

$$\begin{aligned}
\tau_{\bar{b}}[\bar{g}](\mathbb{N}_{\le 100}) &= \text{if } (\mathbb{N}_{\le 100} \le 100) \text{ then } \bar{g}(\bar{g}(\mathbb{N}_{\le 100} + 11)) \text{ else } (\mathbb{N}_{\le 100} - 10) \\
&= \text{if } \texttt{tt} \text{ then } \bar{g}(\bar{g}(\mathbb{N}_{\le 100} + 11)) \text{ else } \cdots \\
&= \bar{g}(\bar{g}(\mathbb{N}_{\le 100} + 11)) \\
&= \bar{g}(\bar{g}(\mathbb{N})) \\
&= \bar{g}(\mathbb{N}_{\ge 91}) \\
&= \mathbb{N}_{\ge 91}.
\end{aligned}$$

Note that $\mathbb{N}_{\leq 100} + 11 = \sup_{n \in \mathbb{N}_{\leq 100}} \widetilde{n + 11} = \sup\{\tilde{m} \mid m \leq 111\} = \mathbb{N}$; this is because we don't have a type $\mathbb{N}_{\leq 111}$ in our system; even if we did, however, the result would still be the same, as $\mathbb{N}_{\leq 111} \subset \mathbb{N}$ and thus $\bar{g}(\mathbb{N}_{\leq 111}) = \mathbb{N}_{\geq 91}$.

There seems to be no (practical) way to prove that $\bar{f} \sqsubseteq \mathbb{N}_{\leq 100} \to \mathbb{N}_{\leq 91}$ (for a suitably defined type $\mathbb{N}_{\leq 91}$). Part of the problem with trying to prove this statement is that to compute with $\mathbb{N}_{\leq 100}$ we also need $\mathbb{N}_{>100}$, and there the statement is false. A possible workaround is to add, for every pair $m < n$ of non-negative integers, a data-type $[m, n] = \{m, m + 1, \ldots, n\} \cup \{\perp\}$. Then one can actually prove this statement by directly translating the normal mathematical proof into our type system (and in fact, one only needs to add data-types $[0, 1], [2, 12], \ldots, [68, 78], [79, 89], [90, 100]$; for every x in the k-th interval from the *right*, k is the least number such that $x + 11k > 100$). However this is highly inefficient, as it requires us to compute the values of all tight extensions used on these new data-types.

Another recursive equation that cannot be typed in our system is *Collatz's Conjecture*.

9 A Take on Negation Types

In this chapter we make an attempt to define a notion of negation for data-types. We will then demonstrate a few properties of our definition.

Definition 9.1. Let D be a domain and x be an ideal of D; we define the *core of the negation of* x to be the set

$$(\neg x)^\circ \overset{\text{def}}{=} \{d \in D : \tilde{d} \cap x = \tilde{\bot}\}. \tag{9.2}$$

For simplicity, we will use the notation $\neg x$ for $(\neg x)^\circ$.

This definition says that d is in the core of the negation of x if and only if the only member of x below d is $\bot$. Conceptually, when we interpret the domain's ordering $\sqsubseteq$ as an information ordering, the core of the negation of x is the set of elements that are information-independent from x.

As we will see, the core of the negation satisfies some intuitive properties one may expect of negation. Before we venture into that direction, however, we must first prove that it is an ideal if this concept is going to be of any value to us, as we have defined data-types to be ideals. As it happens, this is not always the case:

Example. The core of the negation of an ideal x may not be an ideal.

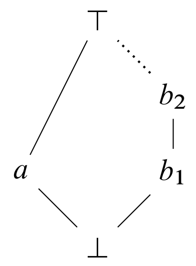

Figure 9.1: A domain with an ideal whose core of the negation is not an ideal. (The dotted line represents an infinite chain $b_1 \sqsubseteq b_2 \sqsubseteq b_3 \sqsubseteq \cdots$ with supremum $\top$)

Demonstration. Let the domain D be as shown in Figure 9.1, and let x be the ideal $\{\bot, a\}$ in D. Then

$$\neg x = \{\bot\} \cup \{b_1, b_2, b_3, \dots\}$$

is not an ideal (the supremum of the directed set $\{b_i \mid i \in \mathbb{N}\}$ is not in $\neg x$). □

The core of the negation, however, is an ideal under some conditions:

Definition 9.3. A chain-complete poset D is said to have *determinate infinite chains* if for every infinite chain C of D and every $a \in D$ it holds that

$$a \sqsubset \sup C \Leftrightarrow (\exists c \in C)(a \sqsubseteq c),$$

that is, every element below (and not equal to) $\sup C$ is below (or equal to) some element of C.

Lemma 9.4. Let D be a domain and x an ideal of D; then $\neg x$ is downward-closed. Furthermore, if $\neg x$ is countable and D has determinate infinite chains, then $\neg x$ is an ideal.

Proof. If $d_1 \sqsubseteq d_2$ and $d_2 \in \neg x$, then $\tilde{d_2} \cap x = \tilde{\bot}$; consequently also $\tilde{d_1} \cap x = \tilde{\bot}$, as $\tilde{d_1} \subseteq \tilde{d_2}$.

Now assume that D has determinate infinite chains. We prove that $\neg x$ is an ideal; by the previous part, it remains to show that $\neg x$ contains the suprema of its D-directed subsets. Suppose that A is a D-directed subset of $\neg x$, and put $s = \sup A$.

If A has a greatest element, that element is s, hence $s \in \neg x$.

Thus suppose that A does not have a greatest element. As $\neg x$ is countable, so is A, and thus one may enumerate its elements as $a_1, a_2, a_3, \dots$. Now we construct an increasing sequence (b_n) of elements of A with the same supremum as A. We do this recursively. Put $b_1 = a_1$, and for $n \geq 1$, let b_{n+1} be an element of A greater than b_n, a_n; to avoid arbitrary choises, we define

$$b_{n+1} = a_k \text{ for the least } k \text{ such that } b_n, a_n \sqsubseteq a_k \text{ — at least one such } k \text{ exists as } A \text{ is directed.}$$

It is plain to see that the set of values of (b_n) is infinite (because A doesn't have a greatest element).

Claim. $\sup_j b_j = \sup_i a_i$.
That $\sup_j b_j \sqsubseteq \sup_i a_i$ is trivial. Conversely, since $a_n \sqsubseteq b_{n+1}$ for every n, we deduce $\sup_i a_i \sqsubseteq \sup_j b_j$.

Now we conclude the proof that $s \in \neg x$. Pick $d \in \tilde{s} \cap x$; in other words, $d \sqsubseteq s$ and $d \in x$. Note that $d \neq s$ as $s \notin x$ (as then all b_n would be in x as well). Thus $d \sqsubset s$. Because D has determinate infinite chains and (b_n) is one with supremum s, we deduce that $d \sqsubseteq b_n$ for some n, and consequently $d \in \tilde{b_n} \cap x$. But $b_n \in A \subseteq \neg x$, so $d \in \tilde{b_n} \cap x$ implies $d = \bot$. As d was arbitrary, this means $\tilde{s} \cap x = \tilde{\bot}$, and $s \in \neg x$. □

Remark 9.5. Note that all we needed is that D has determinate infinite chains, and that for every directed subset A of D without a greatest element there is a chain with supremum $\sup A$.

The most important application of this lemma is the following special case:

Theorem 9.6. If D is a countable domain with determinate infinite chains, then for every ideal x its core of the negation $(\neg x)^\circ$ is also an ideal.

The next question one has to ask is whether this notion satisfies any of the familiar properties of negation. It turns out that it does not satisfy the double negation law — but it shouldn't be supposed to, by the Curry-Howard Isomorphism.

Example. The core of the negation does not satisfy the double negation law.

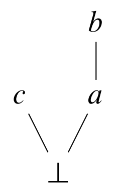

Figure 9.2: A domain and an ideal whose double negation is not itself.

Demonstration. Let D be the domain shown in Figure 9.2, and x be the ideal $\{\bot, a\}$. Then $\neg x = \{\bot, c\}$, $\neg\neg x = \{\bot, a, b\}$. Obviously $x \neq \neg\neg x$; note however that $x \subseteq \neg\neg x$. □

Lemma 9.7. For every ideal x of D, $x \subseteq \neg\neg x$.

Proof. Let $d \in x$. We show that $\tilde{d} \cap (\neg x) = \tilde{\bot}$; equivalently, we show that if $a \sqsubseteq d$ and $a \in (\neg x)$ then $a = \bot$. Indeed, let $a \sqsubseteq d$ be such that $\tilde{a} \cap x = \bot$. As x is an ideal and $d \in x$, also $a \in x$, or equivalently, $\tilde{a} \subseteq x$. This means that $a = \bot$. □

However, as predicted again by the Curry-Howard isomorphism, the core of the negation does satisfy the triple negation law.

Theorem 9.8. For any ideal x of D,

$$\neg\neg\neg x = \neg x.$$

Proof. We already have that $\neg x \subseteq \neg\neg\neg x$. We show the reverse inclusion. Let d be in $\neg\neg\neg x$; this means that $\tilde{d} \cap (\neg\neg x) = \tilde{\bot}$. As $x \subseteq \neg\neg x$ (Lemma 9.7), a fortiori we get $\tilde{d} \cap x = \tilde{\bot}$. Thus $d \in \neg x$. $\square$

Example. In terms of the domain in Figure 1.3, $\textsc{Even} = \mathbb{N} \cap (\neg\textsc{Odd})^{\circ}$. Similarly, if one allows data-types for lists of even and odd length, one has $\textsc{EvenList} = \textsc{List} \cap (\neg\textsc{OddList})^{\circ}$. Sometimes, the core of the negation fails to give any meaningful information: e.g. the core of the negation of the data-type $\tilde{\bot}$ is again $\tilde{\bot}$, but one shouldn't expect to be able to define a "defined elements" data-type, as that is generally uncomputable.

Bibliography

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[PWF12] A. Pitts, G. Winskel, and M. Fiore. Lecture Notes on Denotational Semantics for the Computer Science Tripos, Part II, 2012. `https://www.cl.cam.ac.uk/teaching/1112/DenotSem/dens-notes-bw.pdf`, Accessed: 2024, November 2.

[SW77] Adi Shamir and William W. Wadge. Data types as objects. In Arto Salomaa and Magnus Steinby, editors, *Automata, Languages and Programming*, pages 465–479, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

[Ten91] Robert Tennent. *Semantics of programming languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages, An Introduction*. MIT Press, January 1993.