Domains for polymorphism (Isle of Thorns, August 88)

# Introduction

The aim of these notes is to explain one possible interpretation of polymorphism in domain theory. There exist other models of polymorphism (term models and models built from a partial combinatory algebra), but we will limit ourselves to the field of denotational models. Some knowledge of domain theory is needed, and since references on this topic are not always easily available, these notes will consist first in a general introduction to domain theory, needed to motivate the interpretation of polymorphism. Warning: this will be more a survey of heuristic remarks than a formalised presentation ... We've tried to present simple and basic ideas in this field without going too deep into technical disgressions.

In the first lecture, we give "computer science" motivations for domains, continuity, stability... We show some connections between operational and denotational semantics and how domain theory may be used to give nice proofs of purely syntactical results. The second lecture is a study of the collection of domains as a whole and the notion of embeddings-projection pairs (where we see that this collection shares a lot of properties shared by domains, fact crucial in the modelisation of polymorphism). The third lecture is a study of the finitary projection model of $Type : Type$. In the last lecture finally (not included into these notes), we give an equational presentation of the system $F$ of variable types, and we show how to use this in order to build models of polymorphism.

# 1 An introduction to domain theory

## 1.1 Some questions

One may be surprised by the technicality of domain theory, to such an extent that one may wonder if so complicated notions are really needed. Furthermore, there is now a profusion of notions of domains (complete algebraic lattices, Scott domains, dI-domains, qualitative domains, coherent domains,...). It is thus important to try to motivate all these technical objects, and to see if they are really needed. That's what we'll try to do in this section.

## 1.2 What are domains?

The first thing to say is that elements of a domain are not representations of objects of a data type in themselves, but rather abstract representations of partial knowledge we have about them. Thus, what we want to formalise is really the knowledge that we have about the input-output behaviour of our programs. These elements are naturally ordered (an element may be "more informative" than another one). The problem is to give a plausible axiomatisation of these intuitive notions.

Let us look at the "canonical example" (cf. Kleene, IM52): we take as posets the set $N_\perp$ of subsets of the set of natural numbers $N$ with at most one element, ordered by inclusion. The empty sets represent the absence of information, whereas a singleton $\{n\}$ represents the maximal information that one may have about an integer. Furthermore, in a classical world, there is nothing in between (integers are supposed given as primitive token, and cannot be "partial").

An algorithm that takes integers as inputs and produce integers (and that may diverge) will be represented as a function form $N_\perp$ to $N_\perp$. If we think to the interpretation of elements as partial information, it is natural to restrict our attention to increasing maps (if you know more about the input, you know more about the output). Furthermore, there is a natural ordering between elements in this domain $N_1$, which is the extensional ordering: $f \leq g$ iff for all $x$, $f(x) \leq g(x)$.

In a way, what we are considering appeared already in Kleene: we consider partial functions from integers to integers, and the order is simply the inclusion ordering. It is important to note that the consideration of *partial* recursive functions was an important and non-trivial mathematical step, due to Kleene (Church, as well as Turing were considering only total recursive functions), and motivated by two examples: the problem of denotation of constructive ordinals, and the development of realisablity (it is impossible to realise the implication in a satisfactory way with only total functions, see IM52). The paper "Origins of Recursive Function Theory" (Kleene, in Annals of the History of Computing Volume 3) is a clear and lively account of how the concept of recursivity was found.

Notice however that there is no direct equivalence between this notion of increasing map and the consideration of partial functions between integers. For instance, there are two notions of "constant" functions, and it is instructive to compare them. One is the true constant function $x \longmapsto c$ that sends everything to $c$ and the other sends $\perp$ to $\perp$ and everything else to $c$. The first one says that the algorithm answers $c$ without even looking at its input, and the other says that the algorithm waits for its argument before answering.

Let us emphasize this simple, but important point. What does that mean for $f$ to satisfy $f(\perp) = 3$? This says the following information about the algorithm represented by $f$: in the course of the computation of this algorithm to a given input, this algorithm will never look at its input and will produce eventually 3 (since we consider deterministic algorithms, one can even say that the computation will take the same time regardless to its input). Notice that it is justified to say that this information is finitary: to get this information, it was enough to run, in a finite amount of time, the algorithm on a given input, represented by a program of type $N$ that we know to diverge. It seems intuitive also to say that we have more information when we say $f(\perp) = 3$, than when we say only $f(1) = 3$ (since $f(\perp) = 3$ implies $f(1) = 3$, by the fact that $f$ is increasing). This is a justification of the extensional ordering (see exercice 8).

This simple example is thus fundamental, since it shows one of the main advantages of domain theory. We are able to capture formally some purely intensional and dynamic notions about an algorithm (does it need its argument or not). This is the key for applying domain theory to syntactical results.

## 1.3 Continuity

We now try to deal with functionals. Here again, the examples will be with functionals that send partial functions, from integers to integers, to integers. Let us consider such a functional $F(f)$ coming from an ML program. Since a program is a finite text, it seems intuitive that $f$ will be used only a finite number of times, and so we expect the following property: if $F(f) = p_0$, then there exists $n_1, \ldots, n_p$ such that $f(n_j)$ converges for all $j$, and such that, if $g(n_j)$ converges, and $f(n_j) = g(n_j)$ for all $j$, then $F(g) = p_0$ (remark for people believing in the Curry-Howard analogy between proofs and algorithms: it seems worth looking at the meaning of this statement for proofs. It says roughly that if we have an hypothesis $\forall x.\varphi(x)$ then we use a finite number of instances in a proof, cf Bishop "Mathematics as a numerical language" in Intuitionism and Proof Theory North Holland). Even if there are recurive call of the function $f$ in the computation $F(f)$, it seems that if the computation of $F(f)$ finishes it will do it in a finite amount of time, and so there will be only a finite number of call to $f$. The surprising fact is that it is actually possible to give a purely mathematical explanation of this.

We need some basic definitions in poset theory.

**Definition:** Let $(D, \leq)$ be a poset. We say that $A \subset D$ is directed iff any finite subset of $A$ has an upper bound in $A$ (in particular, the empty set has an upper bound in $A$, and thus $A$ is not empty).

**Definition:** Let $(D, \leq)$ be a poset. We say that $U \subset D$ is Scott-open iff

- $U$ is upward-closed, if $x \in U$ and $x \leq y$, then $y \in U$
- if $A \subset D$ is directed, and $A$ has a least upper bound which is in $U$, then $A \cap U$ is not empty.

Remark that the Scott-opens define a topology over any poset (hence their name). This topology is far from being Haussdorf in general...

Let $(\varphi_n)$ be a recursive enumeration of all partial recursive functions, and let $\mathcal{P}$ be the poset of all partial recursive functions from $N$ to $N$, with the inclusion ordering.

**Theorem:** (Rice-Shapiro) if $\mathcal{C} \subset \mathcal{P}$ and $\{n \mid \varphi_n \in \mathcal{C}\}$ is recursively enumerable, then $\mathcal{C}$ is Scott-open.

**Corollary:** (Myhill-Shepherdson) let $F$ be an "effective functional", i.e. such that there exists $n_0$ such that $F(\varphi_n) \simeq \varphi_{n_0}(n)$ for all $n$ (where $\simeq$ is Kleene's equality, that is, if one converges, both converge and they are equal). Then $F$ is Scott-continuous, that is, if $F(\varphi) = n$, then there exists $\varphi_0 \subset \varphi$ such that $\varphi_0$ is finite (as a subset of $N^2$) and, if $\psi \in \mathcal{P}$ and $\varphi_0 \subset \psi$, then $F(\psi)$ converges and is equal to $n$.

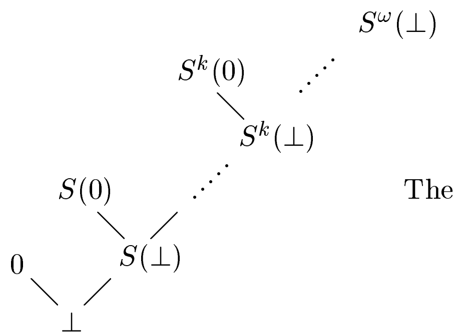
This corollary is the formal counterpart of our first intuitive remark. It strongly suggests that "computable functions are continuous". This principle is known as "Scott's thesis", and it is thus a weakening of Church's thesis.

We'll make use of this result in our first modelisation of higher-order algorithms. If $D_1$ and $D_2$ are two posets, we consider $[D_1 \to D_2]$, the poset of Scott-continuous functions, with the pointwise ordering. Then, the poset we use in order to represent the type of functionals is $[[N_\perp \to N_\perp] \to N_\perp]$. We can then represent as well third-order functionals...

Notice that this model lays the emphasis only upon one aspect of effective computations (Scott-continuity, this emphasis is connected to the importance of continuity in intuitionism). It is thus not surprising that this model presents a lot of non-standard features, that is, functions, functionals that are in the model, but don't come from real programs (for instance, any function that is not computable, but even a computable function may not be representable, the parallel-or is in the model, but not in the syntax, see exercices). But this makes it particularly "smooth" and simple (the main point is that there is no heavy coding, like in recursion theory). It seems to be a good level of abstraction, where one doesn't want to see really the syntax, but only the "continuous aspect of it". Besides, this model presents a (weak) form of finitary content: any information is the directed sups of the "finite" informations that are below it (where, see below, "finite" has somewhat a non-standard flavour).

### 1.4 Scott induction principle

Before getting into the mathematical subtleties of the various notions of domains, let us examine what we've done so far, and present a principle which, though mathematically very simple, gives an elegant way of reasoning about infinite objects, and was the basic motivation for LCF. Each type (even of higher-order functionals) is represented by a poset closed under directed sups, and programs are represented by Scott-continuous functions between these posets. So far, we have only represented the ground type of (flat) integers, which are given in one piece at a time, but it is possible to represent nicer ground types. For instance, the type of lazy integers, in unary notations (or stream over a type with one object in it), and what we can say about informations on a given element: either we know nothing about it, and it will be represented by $\perp$, or we know it is 0, or we know it is the successor of an element, for which we may apply the same reasoning. Hence the following representation:




The domain **D** of lazy integers.

Notice the presence of the "non-standard and infinite" element $S^\omega(\perp)$, which like the "ideal elements" in algebra or geometry, will make all the theory smoother. The remark that the idea of continuity which applies to effective functionals (like the first recursion theorem Kleene IM52), is also relevant in the representation of lazy data types, streams... is fondamental in domain theory.

The principle used in reasoning about infinite objects is the following remark. Let say that we want to prove a property $P$ of the least fixed-point of a functional $Y(F)$. Then it is enough to prove $P(\perp)$ and that $P$ is closed under $F$ (that is $P(x)$ implies $P(F(x))$), provided $P$ is *admissible*, that is $P$ is closed under directed sups. This is clear, since $Y(F)$ is nothing but the $\omega$-sups of the sequence $(F^n(\perp))$. For the domain $D$ of lazy integers, if we want to prove that $P$ holds for any $x \in D$, then it is enough to prove that $P(0)$, $P(\perp)$ hold, and that $P(x)$ implies $P(S(x))$, provided $P$ is closed under directed sups. Here again, $P(S^\omega(\perp))$ will hold since $S^\omega(\perp)$ is the $\omega$-sup of the sequence $(S^n(\perp))$, and $P(S^n(\perp))$ holds by ordinary induction on $n$.

This simple remark is called "Scott-induction principle" and is much more useful than it seems. The fact is that almost all usual predicates are admissible. The idea of R. Milner (and D. Scott) was then to mechanise such a system [1] in the following way: the user can apply the induction principle, without worrying about admissibility conditions, left to a special tactic of the interactive search program (in LCF, this tactic is even part of the system, but it seems more attractive and general to translate LCF in a more general formalism, for instance higher-order logic, and to check admissibility with a special tactic). This leads to very elegant proof about infinite objects (stream, recursion), and about programs which we don't know about the termination, that are furthermore completely formalised (whenever such a proof is possible, this is to be prefered to a messy syntactical approach). LCF is thus particularly well-suited for proving facts about programs that we don't know if they converge or not, or reasoning about infinite streams, for instance a correctness proof of Erathostenes' sieve (see S. Abramsky "Reasoning about Concurrent Systems" Distributed Computing, Ed. Chambers & Duce, Academic Press 1985 and L. Paulson's book "Logic and computation", Cambridge University Press). Another simple example, is the proof that $maps(g \circ f, l) = maps(g, maps(f, l))$ where $maps$ is the generalisation of the function $map$ to streams.

One of the nicest application of denotational semantics is certainly Kahn-MacQueen representation of data-flow network as a set of recursive equations over streams (see Abramsky's paper, cited above). Even the monotonicity property may be a powerful tool in order to show that some primitives over data flow (as fair merge) cannot be implemented by other given primitives (see the paper of P. Panangaden and V. Shanbhogue, Cornell University "On The Expressive Power of Indeterminate Network Primitives", ICALP 88).

### 1.5 Some Poset theory

We are now going in the purely mathematical development of domain theory. There exist several possible approaches (information system, neighbourhood systems,...). The one chosen here is a purely poset-theoretic description, followed by representation theorem. The advice is actually to

[1] Which is easily completely formalised, in contrast with a direct syntactical modelisation, since we are at a right level of abstraction

know all the possible definitions, since they bring quite different intuitions about the notion of domain.

The goal is first to axiomatise the shared properties of the posets $N_\perp$, $[N_\perp \to N_\perp]$, $[[N_\perp \to N_\perp] \to N_\perp]$,... We come then to the following definitions.

**Definition:** Let $(D, \leq)$ be a poset, we say that $a \in D$ is (weakly) finite or compact, iff $a \leq \bigvee A$ with $A \subset D$ directed, implies that there is $c \in A$ such that $a \leq c$.

**Definition:** A poset $(D, \leq)$ is algebraic iff for any $x \in D$, the set $\{a \in D \mid a \text{ is finite } \& \ a \leq x\}$ is directed and has $x$ for l.u.b.

**Definition:** A poset $(D, \leq)$ is a Scott domain iff

- $D$ has a least element $\perp$
- $D$ is closed under directed sups
- $D$ is boundly complete: a bounded subset of $D$ has a l.u.b.
- $D$ is algebraic

Notice that the first condition may be replaced by: $D$ is not empty, since then $\perp$ is the l.u.b. of the (bounded) empty set. Intuitively, the algebraicity says that any element is a directed sup of compact element. The word "finite" may be misleading for compact elements since $\{y \in D \mid y \leq x\}$ may be infinite even with $x$ "finite" (hint: consider the ordinal $\omega+2$). However, it is a good intuition to think of the elements of $B(D) = \{x \in D \mid x \text{ is compact}\}$ (the *basis* of the domain $D$) as "finitary informations".

For showing that indeed this is an axiomatisation of the continuous type hierachy built over $N_\perp$, it is enough to show that $N_\perp$ is indeed a Scott domain (all elements are compact), and that the notion of Scott-domain is closed by continuous exponentiation. The best way of stating this is to say that the category of Scott-domains, with continuous maps as morphisms is a cartesian closed category (and as such, can be used to model simple typed lambda-calculus). It is an alternative to the interpretation via environments: we don't need it once we have shown that it is possible to interpret the typed lambda-calculus in cartesian closed categories.

**Definition:** Let $(D, \leq)$ be a poset, a subset $A$ of $D$ is an ideal iff it is downward closed (i.e. if $x \in A$ and $y \leq x$ then $y \in A$) and directed.

It is remarkable that if $D$ is an algebraic poset with a basis $B(D)$, then $D$ is isomorphic to the "ideal completion of $B(D)$" (that is the set of ideals of $B(D)$, ordered by inclusion, or alternatively the poset we get by adding formally directed sups to $B(D)$). If $(P, \leq)$ is an arbitrary poset, let us write $Idl(P)$ for the poset of ideal of $P$, ordered by inclusion. If $D$ is an algebraic poset, we've just said that $D$ is (isomorphic to) $Idl(B(D))$. Conversely, one may state.

**Proposition:** Let $(P, \leq)$ be a poset, then $Idl(P)$ is an algebraic poset, and $B(Idl(P))$ is (isomorphic to) $P$.

Thus $B(D)$ and $D$ determine uniquely each other: one is the set of compact elements, and the other is the ideal completion.

In a way, only the elements of $B(D)$ "exist" and have a concrete meaning, the other elements in $D$ are only "ideal means" in order to make the theory smoother (so that we can formulate Scott-induction principle). It would thus have been possible to work only with posets of the form $B(D)$. Let us say that a poset is a sup-semi lattice iff any finite bounded subset has a l.u.b.

**Proposition:** Let $(P, \leq)$ be a poset, then $Idl(P)$ is a Scott domain iff $(P, \leq)$ is a sup-semi lattice.

From these remarks, it's not surprising that a "constructive" presentation of domains will put the emphasis upon the basis. Neighbourhood systems, filter models, information systems are nothing but a way of describing directly the basis of a domain (with the difference that it is convenient to work with preorders, i.e. relation reflexive and transtive, and not necessarily antisymetric, since a lot of constructions lead naturally to preorders, and we get the basis by quotienting).

## 1.6 The variety of domains

There are several possibilities if one wants a cartesian closed category of algebraic posets, even if we want to work with Scott-continuous functions. I simply list, without proofs (exercice), some of them.

- Scott domains, the basis being sup-semi lattices
- Complete algebraic lattices, the basis being sup-lattice, i.e. posets closed under finite l.u.b.
- L-domain, the basis being posets where $(P, \leq)$ such that, if $x \in P$, then $\{y \in P \mid y \leq x\}$ is a sup-lattice.
- SFP domains (see below).

The simpler, logically speaking, are complete algebraic lattice, that are simply posets which are algebraic and complete (any subset has a l.u.b).

## 1.7 Adequacy

We can use the idea behind the proof of the first recursion theorem (Kleene IM52) in order to interpret the fixed point functional $Y : (\sigma \to \sigma) \to \sigma$. There is then no problem to interpret a simple programming language: we have one ground type of integers, with primitive operations on them, the general recursion, higher-order functionals (a good way of doing this is via the translation of simply typed $\lambda$-calculus into cartesian closed category, with categorical combinators).

One fundamental result is the adequacy theorem, which shows that there are good relations between the syntax and the semantics for ground type.

**Theorem:** let $t$ be a closed term of type $N$, then $t$ is $\beta$-convertible to an integer $n$ iff its denotation $[\![t]\!]$ is equal to $n$.

One direction is the consistency of the semantics: it preserves conversion, and so if $t = n$, then $[\![t]\!] = n$ (we have $[\![n]\!] = n$ by definition of the model). The other direction is subtler: a priori, it may well be that the semantics "lies", it says that $[\![t]\!] = n$ though $t$ diverges (i.e. is not convertible to an integer). The proof is an application of the reducibility method. Define for each type $\sigma$ a relation $R_\sigma$ between the syntax and the semantics, i.e. between the set $T_\sigma$ of closed terms of type $\sigma$, and the set $D_\sigma$ which is the domain associated to $\sigma$. $R_{N_\perp}(t, v)$ is: $v = \perp$ or else there exists $n$ such that $v = n$ and $t$ is convertible to $n$ [2]. $R_{\sigma\to\tau}(t, f)$ means that $R_\sigma(u, v)$ implies $R_\tau(t(u), f(v))$. Then we claim that for all closed term $t$, we have $R_\sigma(t, [\![t]\!])$, where $\sigma$ is the type of $t$. This implies the theorem when $\sigma$ if $N$. The most interesting case is to check that $R(Y, [\![Y]\!])$. For this, prove first the following

**Lemma:** For each type $\sigma$, and closed term $M$ of type $\sigma$, we have

- $R(M, \perp)$
- if $M_1$ and $M_2$ are convertible of type $\sigma$, then $R(M_1, v)$ iff $R(M_2, v)$
- if $v$ is the directed sup of $(v_i)$, and $R(M, v_i)$ for each $i$, then $R(M, v)$.

What we want to show is that, if $R(t, v)$, then $R(Y(t), \bigvee(v^n(\perp)))$. By the lemma, it's enough to show $R(Y(t), v^n(\perp))$ for all $n$, and we show this by induction on $n$. The lemma shows the case $n = 0$. If it is true for $n$, since $R(t, v)$ we deduce $R(t(Y(t)), v^{n+1}(\perp))$. But $Y(t)$ is convertible to $t(Y(t))$, hence the result by the lemma.

This result is important in order to prove operational equivalence: let say that $t_1$ and $t_2$ programs of type $\sigma$ (that may be functional) are *operationally equivalent* iff for all context $C[\_]$ of type $N$, $C[t_1] \simeq C[t_2]$ (i.e. $C[t_1]$ is convertible to an integer iff $C[t_2]$ is convertible to an integer, and in that case, these integers are equal). The intuition is that we can "test" programs only by embedding them into programs that are "observable", and functions, functionals... are not really observable (they are really ideal elements), and we can print only value of type $N$. A direct corollary of the theorem is that equality of the denotation implies operational equivalence. This is not so trivial: try to prove directly that $Y(id)$ and $Y(id, 3)$ (of type $N$) are operationaly equivalent...

One natural question is whether or not the operational equivalence implies the identity of denotation. From the fact that there are functions like the parallel-or in the semantics, but not in the syntax (see exercices) it is possible to build two operationaly equivalent algorithms (of type $(\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}$) that have distinct semantics: the observations that we can make in the syntax are not refined enough to distinguish the two algorithms (see exercices).

[2] Of course, the reader has noticed the overload of the syntax and semantics of integers

We can refine our notion of operational equivalence: let us say that for $t_1$ and $t_2$ programs of type $\sigma$ (that may be functional) $t_1 \leq t_2$ for the *operational ordering* iff for all context $C[\_]$ of type $N$, if $C[t_1]$ is convertible to an integer, then $C[t_2]$ is convertible to an integer, and in that case these integers are equals. It follows then from the adequacy theorem that the denotational ordering implies the operational ordering (exercice). A semantics which satisfies the adequacy theorem and such that the reverse implication holds, that is, the operational ordering implies the denotational ordering, is called *fully abstract*. In that case, the identity of denotation implies the operational equivalence, and it follows from the above paragraph that Scott-domain semantics is *not* fully abstract.

As we'll see, it's not enough to remove the parallel-or from the semantics to get a fully-abstract model. The problem (a little vague) to get a notion of domain, and a refinement of Scott-continuity, that leads to a fully abstract model is still open, and it is closely related to find the right notion of sequentiality at higher type. At the proof level, a solution to this problem will lead to a system where it is possible to show that two programs are not operationaly equivalent (the adequacy theorem allows us only to prove the operational equivalence of programs: if the semantics differs, we can only deduce the non equivalence by full abstraction).

We will examine now a class of domains, that is not cartesian closed for Scott-continuous maps, but for another notion of maps: the stable continuous maps. These notions were developped first in the study of the notion of sequentiality.

## 1.8 Stability

The notion of stability may be motivated as an attempt to keep our model closer to the syntax, and typically to try to remove the parallel-or algorithm. We want to express that we consider only sequential algorithm. Unfortunately, this is not so easy in poset theoretic term. Vuillemin's definition (with sequentiality index, see exercices), makes sense only for first-order types. It is however possible to formulate a weaker version: let $D_1$ and $D_2$ be two algebraic posets, we say that $f : D_1 \to D_2$ is stable iff whenever $y_0 \leq f(x)$ and $y_0$ is compact, then there exists $x_0 \leq x$ compact such that for $u \leq x$, $y_0 \leq f(u)$ is equivalent to $x_0 \leq u$. Such a $x_0$ is then uniquely determined, and we write it $m(f; x, y_0)$. It is possible then to define $m(f; x, y)$ for $y \leq f(x)$ not necessarily finite, so that $m(f; x, y) \leq x$, and, for $u \leq x$, we have $y \leq f(u)$ iff $m(f; x, y) \leq u$ (exercice). Notice that the continuity of $f$ says that there exists $x_0 \leq x$ finite such that $y_0 \leq f(x_0)$, and stability that there exists a *minimum* such $x_0$ (if we restrict ourselves to elements below $x$). In an intuitive sense, stability is weaker than sequentiality, but it is "almost sequential: once the computation has been done, it was possible to have done it in a sequential way".

The next notion is almost more important than stability: it is the notion of *stable ordering*. Heuristically, if $f \leq g$ for the stable ordering, the computation represented by $f$ "looks like" the computation represented by $g$. In particular, we don't have in general $\lambda x.x \leq \lambda x.\top$ (if there is a "top" element $\top$), in contrast with the extensional ordering. The formal definition is: first $f(x) \leq g(x)$ for all $x$, next if $y \leq f(x) \leq g(x)$, then $m(f; x, y) = m(g; x, y)$ (a priori, we have only that $m(g;x,y) \leq m(f;x,y)$ for $y \leq f(m(f;x,y)) \leq g(m(f;x,y))$ and $m(f;x,y) \leq x$. This remark may provide help for a categorical generalisation of this notion.).

For the domains we will study, it turns out that stability and the stable ordering are equivalent to other conditions, less intuitive, but technically more tractable. The stability becomes preservation of pull-backs, and the stable ordering cartesian natural transformation. N.B.: the categorical language is not used here only to look smart. The definitions are really more "visual" in this way: preservation of pull-backs means the preservation of meet of *compatible* elements (which is not standard in poset theory), and cartesian natural transformation means that a cartesian (=pull-back)square is sent to a cartesian square, or alternatively that the usual commutative squares associated to a natural transformation are cartesian (make a picture). In poset terms, it gives the (not very natural) definition: $f$ is less than $g$ for the stable ordering iff $x \leq y$ implies that $f(x)$ is the meet of $f(y)$ and $g(x)$.

In order to get cartesian closed category, the definition of domains is more complicated seen "poset theoretically", but may become very simple in term of representations (coherent space). I give the definition without much of motivations ("it works", see, if possible, Berry's thesis Paris VII, or G. Winskel's thesis, Edimburgh, or Girard "Fifteen Years Later" TCS 45).

**Definition:** A poset $D$ is a dI-domain iff it is a Scott domain such that

- it is distributive, i.e. if $y, z$ are compatible, then $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$,
- if $x$ is compact, then $\{y \in D \mid y \leq x\}$ is finite.

**Definition:** Let $D$ be a poset, $p \in D$ is prime iff for any $A \subset D$ with a l.u.b. $M$, if $p \leq M$, then there exists $x \in A$ such that $p \leq x$. If $D$ has a least element $\perp$, we say that $p \in D$ is an atom iff $q \leq p$ implies $q = p$ or $q = \perp$.

Remark that in a dI-domain, any element is the l.u.b. of the primes below it (exercice).

**Definition:** A dI-domain is a qualitative domain iff all primes are atoms.

Qualitative domains get thus a very simple representation theorem. If $X$ is a set and $C$ a predicate of compatibity on $\mathcal{P}(X)$ (i.e. such that $C$ is downward closed, and closed under directed union), then $C$ defines a qualitative domain for the inclusion ordering. And conversely, a qualitative domain $D$ defines a set $|D|$ (the sets of atoms of $D$, called also the web of $D$), and compatible sets are precisely the set of the form $\{a \in |D| \mid a \leq x\}$ for one $x \in D$.

It is possible to interpret typed $\lambda$-calculus in a stable framework.

**Theorem:** The category of dI-domains (resp. qualitative domains, resp. coherent space) and stable maps is cartesian closed.

The notion of "finite informations" becomes in this case rather the set of prime elements than the finite elements. In the qualitative and coherent case, it is even stronger: all finite, concrete informations are atomic things.

### 1.9 Why stable maps are not enough

The function parallel-or is not stable, and so it may seem that stable models are closer to the syntax. A first result of G. Berry shows however that they are not close enough: Gustave's function of type $\mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$ defined by $f(x) = 0$ iff $(0, 1, \perp) \leq x$ or $(\perp, 0, 1) \leq x$ or $(1, \perp, 0) \leq x$ is stable but not representable in the syntax since it is not sequential (exercice, use the spirit of Plotkin's activity lemma described in "LCF used as a programming language"). But there is more: stable semantics allow non-standard functionals "worse than the parallel-or" in that they are not increasing for the extensional order (see exercices), and so there is no real way in which the stable model is "better" than the continuous model. In an attempt to solve this problem, Berry, Curien Winskel have introduced the notion of bi-domains (later simplified by Winskel), which mixes the extensional and stable ordering, but this notion is far from being "as clean" as Scott-domains, or dI-domains, and there is clearly the need here of a new and simple idea to get a still better denotational model of sequential functional language.

### 1.10 Exercices

1. Prove the Rice-Shapiro theorem. Hint: for showing that $\mathcal{C}$ is upward closed, prove the following lemma. If $g$ is more defined than $f$, then there exists a computable function $F(x)(y)$ such that $F(x) = f$ if $\varphi_x(x)$ diverges, and $F(x) = g$ if $\varphi_x(x)$ converges. Apply a similar reasoning for showing that the complement of $\mathcal{C}$ is closed by directed unions.

2. Give the denotational semantics of your favorite functional language. Use it to show that you cannot program a functional $F$ such that $F(f)$ is 0 if $f(n) = 0$ for all $n$, and 1 if there exists $n$ such that $f(n) \neq 0$.

3. Show that it is not possible in (lazy) ML to program a functions $f : \mathit{bool\ stream} \to \mathit{int\ stream}$ which sends infinite streams to infinite streams, and is onto when seen as a map on infinite streams to infinite streams.

4. Define a first-order language with recursion, with an operational semantics (let say lazy evaluation) and a denotational semantics (you can look at P. Dybjer "Inverse Image Analysis", Chalmers Report 32). If $f : D_1 \times \ldots D_n \to D$, and $v_i \in D_i$, say that $i_0$ is a sequentiality index for $(f, v)$, with $v = (v_1, \ldots, v_n)$ iff $v \leq w$ (for the product ordering) and $v_{i_0} = w_{i_0}$ implies $f(v) = f(w)$. Say that $f$ is Vuillemin-sequential iff for each $v$, $(f, v)$ has at least one sequentiality index. Show that a denotation of an algorithm may have several sequentiality index at a point. Show that denotations of algorithm are all Vuillemin-sequential.

5. Algebraicity is not preserved by continuous exponentiation: take $D$ the poset with $\perp$, $n$ and $n'$ (for $n \in N$) and $\omega$, with $\perp \leq x$, $x \leq \omega$, and $n \leq m'$, $n' \leq m$, $n' \leq m'$ iff $n \leq m$. Show that $Id_D$ is not compact, and that $\{x \in B([D \to D]) | x \leq Id_D\}$ is not directed.

6. Give a stable semantics of a functional language. Use it to show that you cannot program

parallel-or in ML (for a syntactical proof, see Plotkin's article "LCF as a programming language.").

7. Show that Scott-domains are closed by continuous exponentiation. Show that this is still true for L-domains, where we replace the condition of being boundly complete by: the poset of element below any point is a complete lattice (explain why this doesn't entail that the domain is boundly complete).

8. Show that in $[N_\perp \to N_\perp]$ there are compact elements with infinitely many elements below them.

9. Show that the l.u.b of two compact elements is compact, but, in Scott-domains, the g.l.b. of two compact elements may be not compact.

10. Show that in the domain $[[N_\perp \to N_\perp] \to N_\perp]$ there are compact elements with an infinitely decreasing chain of elements below it. More generally, show that this is true for $[D_1 \to D_2]$ whenever $D_1$ has an infinite element, and $D_2$ is not $\{\perp\}$.

11. State, and prove, a representation theorem for coherent space.

12. (G. Berry) Let $B$ be the flat domain of booleans, and $O$ the domain with only two elements $\perp$ and $\top$. Show that $\lambda x.x$ and $\lambda x.\top$ in $[O \to O]$ are not comparable for the stable ordering. Hence, there is a stable functional $h \in [O \to O] \to B$ such that $h(\lambda x.x) = T$ and $h(\lambda x.T) = F$. Show that such a functional cannot come from the syntax (hint: use the adequacy theorem). Ponder about the affirmation of G. Berry that such functional "has even less computer science reality" than the parallel-or. This fact lead G. Berry to consider bi-domains, that mix the extensional and the stable ordering, in the quest of a fully-abstract model.

13. Prove that if $f \leq id$ for the stable ordering, then $f^2 = f$, so that the definition of projector becomes particularly simple in the stable case. Show furthermore that if $p \leq Id$, $q \leq Id$, then $p \circ q = q \circ p = p \sqcap q$.

14. use the fixed point induction principle to show that if $f, g : \sigma \to \sigma$ commute and $f(\perp) = g(\perp)$, then $Y(f) = Y(g)$. Show that this result is true over a complete lattice with monotone (non necessarily continuous) functions.

15. (Manna) Let $G : \sigma \to \sigma$ and $P : \sigma \to \mathsf{Bool}$ be arbitrary functions, and $F : \tau \to \tau$ a strict function. Define the function $H = Y(K)$, where $K = \lambda h, x, y.\text{if } P(x) \text{ then } y \text{ else } F(h(G(x), y))$. Show that $F(H(x, y)) = H(x, F(y))$ for all $x, y$ by fixed-point induction. This illustrates how to use LCF to prove equivalence of programs, and ultimately compiler correctness.

16. Prove that if $t_1$, $t_2$ are closed terms of type $\sigma \to N_\perp$, $t_1$ and $t_2$ are operationally equivalent if for all closed term $u$ of type $\sigma$, we have that $t_1(u) \simeq t_2(u)$. Hint: define by induction on the type $\sigma$ the relation $\simeq_\sigma$. For $N$, it is Kleene's equality. Show that if $M(x_1, \ldots, x_n)$ is a term

and $t_i \simeq u_i$, then $M[t_1, \ldots, t_n] \simeq M[u_1, \ldots, u_n]$. Deduce from that two terms operationaly equivalent with distinct denotations.

17. One may try to simplify coherent space, and to work only with complete atomic boolean algebras (posets of the form $\mathcal{P}(X)$, $X$ arbitrary set). This would be the analog of complete algebraic lattice. Show however that the poset of stable functions$\mathcal{P}(X_1) \to \mathcal{P}(X_2)$, for the stable ordering, has no greatest element. This shows that there cannot be anything like complete algebraic lattices in a stable framework.

18. Let $D, E$ two Scott domains, show that $f \in [D \to E]$ is stable iff it preserves all pull-backs, finite and infinite (that is, if $X \subset D$ is bounded then $f(\cap X) = \cap f(X)$).

19. (G. Berry) Give an example of a function which preserves pull-back, but is not stable. Show that if $D_1$, $D_2$ are Scott-domains, $f \in [D_1 \to D_2]$ is stable if it preserves pull-back as soon as $B(D_1)$ is well-founded.

directed colimits. Then there exists $X$ such that $X$ and $F(X)$ are isomorphic.

Simply take $X$ to be the limit of the diagram $\perp \to F(\perp) \to F(F(\perp))$... (where $\perp$ denotes the one-point domain). Insteadof $\perp$ you can start with any map $D \to F(D)$ (for instance if $F(\perp)$ is already isomorphic to $\perp$ and you want a non-trivial solution). Actually, we get in this way an isomorphism $\epsilon : F(X) \to X$ which is a "least fixed point", in the sense that if there is a map $g$ from $F(Y)$ into $Y$, then there is a unique map $f$ from $X$ to $Y$ such that $\epsilon \circ f = F(f) \circ g$. In the case of poset, this result says that the least fixed point is th g.l.b. of points $y$ such that $F(y) \leq y$. You lift this in the categorical framework, by adding, as usual, maps and unicity property. A map $F(D) \to D$ is called a $F$-algebra, so that we are really building initial $F$-algebra here. One "nice" feature of working with a concrete representation, like information systems, is that you can forget about categorical features: simply take a directed union of the information systems.

This proposition is very useful due to the fact that all syntactic type expression with variables defines a functor (exercice), even if one has negative occurences, or mixed occurence: for instance $X \mapsto [X \to X]$ is the object part of a functor.

The intuition of set theory may be misleading. If the least solution of $X \simeq 1+X$ is, as expected, the lazy domain of integers, the least solution of $X \simeq N \times X$ is not trivial as in the category of sets (indeed, the empty set is a solution in set theory), but it is the domain of infinite streams of integers (surprisingly, $N\times \perp$ is not isomorphic to $\perp$).

### 1.11 Application: model of the lambda-calculus

In a way, this shows that domain theory is not only a model of simple typed lambda-calculus: it models also "transfinite types" that are "infinite limits of simple types". For instance, provided you can find $D_0$ non trivial and a map $D_0 \to [D_0 \to D_0]$, you will find a non trivial domain $D$ such that $D$ is isomorphic to $[D \to D]$. But to find such a $D_0$ is not so hard: for any domain $X$, we have the embedding projection pair $(K, \lambda f.f(\perp))$ between $X$ and $[X \to X]$ (with $K(x, y) = y$).

Two remarks on this example. The first one is that from an isomorphism $D \simeq [D \to D]$, you get a model of untyped $\lambda$-calculus with $\beta$ and $\eta$-conversion. The second is that, though $X = [X \to X]$ is not a positive inductive definition, since the variable $X$ occurs both positively and negatively in the functor $F(X) = [X \to X]$, the compact elements of the colimit of $F^n(D_0)$ *are defined inductively.* Indeed, if one thinks at the form of the step function, a compact element of $D = \mathrm{colim}(F^n(D_0))$ is either a compact element of $D_0$ or else a finite union of pairs $(a, b)$ where $a$ and $b$ are finite elements of $D_0$ (this remark is important for the extension of the adequacy theorem to recursive types).

This model of untyped $\lambda$-calculus is called $D_\infty$ (a lot of its property are independent of the non-trivial domain $D_0$ we start with, and it doesn't figure explicitely as a parameter). Note finally the nice following result.

**Theorem (Hyland-Wadsworth):** a $\lambda$-term has a head-normal form iff its denotation is different form $\perp$.

Actually, the result is much more precise, since it relates the Böhm-tree of a term to its denotation. To understand this result, it is strongly adviced to try first to prove that the denotation of $\Delta(\Delta)$ is $\perp$, with $\Delta = \lambda x.x(x)$ (see exercices).

## 1.12 A syntactic corollary

Let us see how to use domains to prove the consistency of an equational theory, that is to show that the equation $x = y$ is not derivable. For this, a semantical approach is to build a model of the equational theory with two distinct points. Recall that the consistency of the equational theory of untyped $\lambda$-calculus, with $\eta$ and $\beta$ conversion, follows from Church-Rosser theorem. A semantical proof was given above, by building the model $D_\infty$. For the untyped $\lambda$-calculus with pairing, if one adds the surjective pairing conversion, then there is a problem via the syntactical approach, since Church-Rosser fails for this calculus (Klop, see Barendregt's book on $\lambda$-calculus, page 403).

To find a nontrivial model in domain theory, it is enough to find a non-trivial domain $D$ such that $D \simeq [D \to D] \simeq D \times D$. Notice the difficulty: one wants to solve simultaneously two recusive domain equations.

There is a little trick for this. First solve in a non-trivial way the equation $D_0 \simeq D_0 \times D_0$ (exercice). You can then apply the general technique to solve $X \simeq [X \to D_0]$ (and a solution of this is necessarily not trivial since $[\perp \to D_0] \simeq D_0$. But then, we have $[X \to D_0] \simeq [X \to (D_0 \times D_0)]$, and $[X \to (D_0 \times D_0)] \simeq [X \to D_0] \times [X \to D_0]$ so that we also have $X \simeq X \times X$. Hence, $[X \to D_0] \simeq [X \times X \to D_0] \simeq [X \to [X \to D_0]]$, and so $X \simeq [X \to X]$.

## 1.13 Profinite domains

We've seen that algebraicity is not preserved by continuous exponentiation in general. Consider the category of posets that are directed colimits of finite posets with a least element (with as morphisms embedding-projection pairs). Any such poset is an algebraic poset (the basis is the directed colimit of the finite poset). Since continuous exponentiation (resp. product) is a preserving directed colimits functors, it is clear that this category has product and exponentiation, and by construction, this category is stable by directed colimits of embedding-projection pairs. Furthermore we have seen that any Scott domain is an object of this category, but it is more general, since there are finite posets with $\perp$ which are not boundly complete and so not Scott domain. We get thus a large cartesian closed category of algebraic domain (useful in the modelisation of parallelism). The object of this category are called SFP domains ($\perp$ was put in for solving domain equation. If we forget it, we get the so called profinite domains see Gunter "Universal Profinite Domains.", Information and Computation).

### 1.14 Dependant family of domains

From a type theoretic point of view, if one sees domains as types, it is natural to ask about the existence of dependent family of types, product and sums. We will limit here to some remarks.

Due to the fact that $\mathcal{D}^E$ looks like a "big domain", we take the following

**Definition:** A dependent family of domains over a domain $D$ is a Scott-continuous (i.e. preserving directed colimits) functor from the poset $D$, seen as a category, into $\mathcal{D}^E$.

The hypothesis of continuity looks like the definition of the fiber at a point of a sheaf over a topological space. A nice example is the functor over the domain of lazy integers $D$ such that $F(\perp)=\perp$, $F(0) = D$ and in general $F(S^k(\perp)) = D^k\times \perp$, $F(S^k(0)) = D^{k+1}$. Then $F(S^\omega(\perp))$ is nothing but the domain of streams over $D$. It may be interesting to look for other examples of "dependent family of structures over a domain", where one study the collection of dependent types as a whole (after all, it is a model of type theory), and where we get interesting results by looking at particular fibers.

The definition of dependent product and dependent sums are then natural: $\Sigma(F)$ is the poset of pairs $(a, b)$ with $b \in F(a)$, and $\Pi(F)$ the poset of continuous section of the first projection map $\Sigma(F) \to D$. It is straightforward to check that these posets are indeed domains (with any "reasonable" notion of domains). What is much more difficult curiously is to show rigorously that these constructions lead indeed to a model of a type calculus with dependent sums and products (see P. Taylor's thesis Cambridge University).

### 1.15 In the stable case

In the stable case, the situation is simpler, if one looks at the representation (set of prime elements or set of atoms). Let us take the case of coherent space, which is illuminating. An embedding is simply an inclusion of graphs (a first-order inclusion, that is two points are consistent iff their images are consistent). This notion of embedding corresponds to stable embedding-projection pairs $(f^E, f^R)$ such that $f^R \circ f^E = Id$ and $f^E \circ f^R \leq Id$ for the stable ordering (exercice). We can so concretise the solution of recursive domain equations as an inductive construction of a graph. The fact that any domain is a colimit of finite one becomes that a graph is the directed union of its finite subgraphs.

### 1.16 Exercices

1. Show that the opposite of the category of continuous projection has directed colimits.

2. Extend the adequacy theorem with recursive types (use the hint of the next exercice to extend the relation between syntax and semantics to recursive types).

3. Show directly that the interpretation of $(\lambda x.x(x))(\lambda x.x(x))$ in $D_\infty$ is $\perp$. Hint: show that a finite element below the denotation of $\Delta(\Delta)$ has to be $\perp$. For this, translates the fact that $a \leq [\![M]\!]$, where $a$ is a compact element of $D_\infty$, as "$a$ is a type of $M$" with a type system with intersection types, and where the empty intersection $\omega$ is such that $\omega = \alpha \to \omega$ and $\alpha = \omega \to \alpha$, where $\alpha$ is any type. $\omega$ corresponds to $\perp$, and $T_1 \cap T_2$ to the sup of the finite elements corresponding to $T_1$ and $T_2$. The finite elements of $D_0$ correspond to ground types, with given inclusions between them. This is an example which shows the use of a "syntactic" description of the compact elements of a domain.

4. Show that if $(f^L, f^R)$ is an adjoint pair for the stable ordering (that is $f^L \circ f^R \leq Id$ and $Id \leq f^R \circ f^L$), then $f^R \circ f^L = Id$ (hint: show that a stable map cannot be strictly greater than the identity for the stable ordering).

5. Show that if $M, M_1, ..., M_n$ are pure $\lambda$-terms such that $M(M_1, ..., M_n)$ has a hnf, then $M$ has a hnf.

6. Describe, first in term of information systems, then in term of graphs, the dependent product and dependent sums of domains.

## 2 The finitary projection model of Type:Type

### 2.1 Finitary projection of a domain

We work only with Scott domain, but it is possible to translate all this part in a stable framework (exercice). We have seen that any Scott-domain is a directed colimit (for embedding-projection pairs) of finite domain. In general, if $(f^E, f^R)$ is an embedding-projection pair from $D_1$ to $D_2$, the composition $f^E \circ f^R$ will be called a finitary projection of $D_2$. It must be thought of as the "isomorphic representation of $D_1$ as a subdomain of $D_2$", so that we get a concrete representation of the poset of subobject of $D_2$ in the category $\mathcal{D}^E$.

A direct caracterisation is possible.

**Definition:** Let $D$ be a Scott domain. A finitary projection of $D$ is a map $f \in [D \to D]$ such that $f \leq id_D$, $f \circ f = f$ and the range of $f$ is a Scott domain (for the induced ordering). The poset of finitary projection of $D$ is written $\mathsf{Fp(D)}$.

The goal of this section is to show the following surprising result, which will be the key lemma in the modelisation of $Type : Type$ in domain theory.

**Proposition:** If $D$ is a Scott domain, then $\mathsf{Fp(D)}$ is a Scott domain. Furthermore, the inclusion $\mathsf{Fp(D)} \to [\mathsf{D} \to \mathsf{D}]$ is the left part of an embedding-projection pair.

This says actually that there exists a finitary projection on $[D \to D]$ such that its range is $\mathsf{Fp(D)}$. Let us define directly this finitary projection $F \in \mathsf{Fp([D \to D])}$: $F(f)(x) = \bigvee\{y \in B(D) | y \leq x \ \& \ y \leq f(y)\}$. Then, it is easy to check that $F \leq Id$, and $F \circ F = F$. Remark then that finitary projections of $D$ correspond bijectively to subsets $A$ of $B(D)$ such that $\perp \in A$ and $A$ is stable by finite partial sups. Indeed, such a $A$ defines the finitary projection $p_A(x) = \bigvee\{a \in A | a \leq x\}$ and if $p$ is given, consider $A = \{x \in B(D) | p(x) = x\}$. The extensional ordering corresponds to the inclusion ordering. From this, it follows that $\mathsf{Fp(D)}$ is a Scott-domain: the compact elements correspond to the finite subset of $\mathsf{Fp(D)}$ with $\perp$ closed by finite partial sups. It follows also that $p(x) = \bigvee\{y \in B(D) | p(y) = y \ \& \ y \leq x\}$ if $p \in \mathsf{Fp(D)}$, and from this, we can conclude that $p \in \mathsf{Fp(D)}$ iff $F(p) = p$.

It is actually possible to show that $\mathsf{Fp}$ defines a functor from $\mathcal{D}^E$ into itself (exercice).

## 2.2 A type of all types in domain theory

From what is above, it is possible to build a non trivial domain $D$ with an embedding-projection pair $[D \to D] \to D$. Let us fix such a domain (notice that the choice of such a domain is not at all canonical, and the way in which the model of $Type : Type$ we will build, will depend on this choice is not so clear). That says that we have two maps $\mathsf{app} \in [\mathsf{D} \to [\mathsf{D} \to \mathsf{D}]]$ and $\mathsf{lam} \in [[\mathsf{D} \to \mathsf{D}] \to \mathsf{D}]$ such that $\mathsf{app(lam(f),x) = f(x)}$ and $\mathsf{lam(app(x)) \leq x}$.

From what is above, there is a finitary projection $F \in \mathsf{Fp([D \to D])}$ which "represents" $\mathsf{Fp(D)}$. For $x, y \in D$, defines $x \in y$ by $F(\mathsf{app(y)})(\mathsf{x}) = \mathsf{x}$, and say that $x \in D$ is a type iff there is a finitary projection $p$ such that $x = \mathsf{lam(p)}$ (so that we have then $y \in x$ iff $p(y) = y$). Notice that this is equivalent to $x = \mathsf{lam(F(app(x)))}$. But $\mathsf{lam} \circ \mathsf{F} \circ \mathsf{app}$ is itself a finitary projection of $D$, so that there exists a type $V$ in $D$ such that $x$ is a type iff $x \in V$. By construction, $V \in V$, and $V$ is like a type of all types.

Intuitively $D$ represents the collection of all terms, types and programs (we have "types as values" here, but it may be that some values are not types). Any $x \in D$ determines a type $\mathsf{app(V,x)}$, which defines a domain $T(x) = Im(F(\mathsf{app(x)}))$.

If $x \in D$, and $f \in [D \to D]$, we can define the typed abstraction $\lambda(x, f) = \mathsf{lam(f \circ F(app(x)))}$, so that now $\mathsf{app}(\lambda(\mathsf{x},\mathsf{f}),\mathsf{y}) = \mathsf{f(y)}$ if $y \in x$. Furthermore, we can define the type $\Pi(x, f)$ which is the type corresponding to the domain which is the dependent product of the family of types $T \circ f$ over the domain $T(x)$ (exercice: write explicitely $\Pi(x, f)$).

The important point is that we have all what is needed in order to give an interpretation of $Type : Type$ up to *equality*. Simply interpret inductively a derivation of a sequent $x_1 : A_1, ..., x_n : A_n \vdash M : A$, by two maps $[\![M]\!] \in [D^n \to D]$ and $[\![A]\!] \in [D^n \to D]$ such that $[\![M]\!](v_1, ..., v_n) \in [\![A]\!](v_1, ..., v_n)$ provided $v_j \in [\![A_j]\!](v_1, ..., v_{j-1})$ (exercice).

Let us show that the interpretation of $\Pi\alpha \in V.\alpha$ is not $\perp$. Simply take $x_0 \in D$ distinct from $\perp$, and then, the "continuous section" $\lambda\alpha : V.\mathsf{app}(\alpha, \mathsf{x_0})$ belongs to $\Pi\alpha \in V.\alpha$ (remember that

$\mathsf{app}(\alpha, \mathsf{x}) \in \alpha$ if $\alpha \in V$). In the syntax, it is known that there are closed terms in the type $\Pi\alpha \in V.\alpha$, and that such terms cannot have head normal form. The problem whether there exist such terms with a denotation $\neq \perp$, that is the problem of adequacy for this semantics w.r.t. the $Type : Type$ system, is open.

These semantical ideas give nice proofs for the equational consistency of types systems that extend $Type : Type$, proofs that may become tedious in a purely syntactical way (for instance, one may add recursive types, strong sums,...).