Inductive types in constructive languages

Bruin, Peter Johan de

Chapter 9

Partiality

Up till now, all objects were fully defined. Functional programming languages that employ so-called *lazy* (non-strict) evaluation require quite different recursive types. In these languages objects can be defined, some parts of which are undefined. "Undefined" means here that the part is given by a program whose computation proceeds indefinitely, without producing any output. (Note that a non-terminating program may still produce fully defined infinite objects.)

Classically, such types are modeled by adding a special value $\bot$ to represent an undefined (sub-)value. A recursive type T with one constructor $\tau\colon F.T \to T$ may then be represented by an initial algebra $(T\colon \mathbf{Type};\ [\tau, \mathsf{K}\,\bot]\colon F.T + 1 \to T)$. Constructively, this does not work, as it is in general undecidable whether a program will produce anything or not. We shall give an alternative representation in 9.3.

First, in section 9.1, we give a brief overview of the standard theory of complete partial orders (*cpo*'s). This theory can be used to interpret recursive object definitions, classically as well as constructively.

In 9.2, we treat the simplified case of optional objects, which are either undefined or fully defined. These may be used to model partial functions or procedures in a language without lazy evaluation.

In 9.3, we give a new constructive representation of the cpo of recursive types with lazy parts, using final coalgebras in the category of strict types.

Finally, section 9.4 shows how recursive object definitions can be interpreted in this representation without using the cpo structure.

9.1 Domain theory

There are many constructions of categories of domains that model recursive data types, see e.g. Scott [77]. Smyth and Plotkin set up [79] a categorical framework that generalizes the construction of recursive domains in most categories occurring in computational semantics.

One of these categories is the category of *complete partial orders* (cpo's). We define this category, and give three fixed-point theorems: 9.1 shows how to interpret recursive object definitions, 9.3 shows how to interpret recursive type definitions, and theorem 9.4 shows how to prove properties of recursively defined objects.

We define first the category of *partial orders*, then ω*-chains* over a partial order, then the category of cpo's. The object and arrow parts of these categories are given by means of structure definitions (paragraph 2.6.3); but the arrow part might in fact be obtained from the definition of the object part by means of a straightforward procedure.

Define **PoSet**: **Cat** by
X: **PoSet** :=: (X: **TYPE**;
$(\leq)$: $\subseteq X^2$;
$(\leq) \cdot (\leq) \subseteq (\leq)$;
$(\leq) \cap (\leq)^{\cup} = (=_X)$
)
f: $X \to Y$ in **PoSet** :=: (f: $X \to Y$ in **TYPE**;
$(f, f) \in (\leq) \to (\leq)$
);

ω_chain(X: **PoSet**) := $\{ s: X^\omega \mid: \forall i: \omega :: s_i \leq s_{i+1} \}$

Define **CPO**: **Cat** by
$\mathcal{D}$: **CPO** :=: ($(\mathcal{D}; (\sqsubseteq_{\mathcal{D}}))$: **PoSet**;
$\bot_{\mathcal{D}}: \mathcal{D}$; $\forall (x: \mathcal{D} :: \bot \sqsubseteq x)$;
$s: \omega\_\mathsf{chain}\, \mathcal{D} \vdash \bigsqcup s: \mathcal{D}$; $\forall x: \mathcal{D} :: (\bigsqcup s \sqsubseteq x \Leftrightarrow \forall i: \omega :: s_i \sqsubseteq x)$
);
$f: \mathcal{D} \to \mathcal{E}$ in **CPO** :=: ($f: \mathcal{D} \to \mathcal{E}$ in **PoSet**;
$f.\bot_{\mathcal{D}} = \bot_{\mathcal{E}}$;
$s: \omega\_\mathsf{chain}\, \mathcal{D} \vdash f.\bigsqcup s = \bigsqcup (f^\omega.s)$
)

Relation '$x \sqsubseteq y$' may be understood as 'Partial object x is an approximation of y', and $\bot$ is the undefined object, which approximates everything.

A *continuous function* $f: \mathcal{D} \to_{\mathsf{c}} \mathcal{E}$ between two cpo's is a function that is monotonic with respect to $\sqsubseteq$ and that preserves limits of ω-chains. Note that arrows $f: \mathcal{D} \to \mathcal{E}$ in **CPO** are continuous functions that preserve $\bot$ too.

Theorem 9.1 (fixed points in a cpo) *Any equation $x = f.x$ where $f: \mathcal{D} \to_{\mathsf{c}} \mathcal{D}$, has a least solution in $\mathcal{D}$, called* $\mathsf{fix}\, f$. *That is, one has*

$$f.(\mathsf{fix}\, f) = \mathsf{fix}\, f$$
$$f.x \sqsubseteq x \;\Rightarrow\; \mathsf{fix}\, f \sqsubseteq x$$

Proof. Take $s_0 := \bot$, $s_{i+1} := f.s_i$. By induction one has $\forall i: \omega :: s_i \sqsubseteq s_{i+1}$, so we can define $\mathsf{fix}\, f := \bigsqcup s$. Then

$$f.(\bigsqcup s) \;=\; \bigsqcup (i :: f.s_i) \;=\; \bigsqcup (i :: s_{i+1}) \;=\; \bigsqcup (i :: s_i)\ ,$$

and if $f.x \sqsubseteq x$, then by a simple induction $\forall i: \omega :: s_i \sqsubseteq x$, so $\bigsqcup s \sqsubseteq x$. ■

Category **CPO** is closed under products, sums, continuous function space, and taking fixed points (modulo isomorphism) of suitable endofunctors, by Scott's inverse limit (colimit) construction, as follows.

A *cochain* in any category $\mathcal{C}$ with ω-products is an ω-tuple, $T{:}\,\mathcal{C}^\omega$, with arrows $\phi_i{:}\,T_{i+1} \to T_i$. A *cocone* is a structure $(T;\phi;S;\psi)$ where $(T;\phi)$ is a cochain, S an object, and where arrows $\psi_{i:\omega}{:}\,S \to T_i$ commute with ϕ_i:

$$\psi_i \;=\; \psi_{i+1} \mathbin{\bar{\circ}} \phi_i \ .$$

Given a cocone $(T;\phi;S;\psi)$, we call $(S;\psi)$ a *colimit* of $(T;\phi)$ iff for any cocone $(T;\phi;S';\psi')$ there is a unique homomorphism $(S';\psi') \to (S;\psi)$.

Theorem 9.2 *Every cochain $(T;\phi)$) has a colimit.*

Proof. Take

$$S := \{\, t{:}\,\Pi(\omega;T) \mid{:}\, \forall i{:}\,\omega :: t_i = \phi_i.t_{i+1} \} \ ,$$

then $(T;\phi;S;\pi)$ is a cocone, and for any cocone $(T;\phi;S';\psi')$ one has $\langle\psi'\rangle{:}\,(S';\psi') \to (S;\pi)$.
Assuming $\chi{:}\,(S';\psi') \to (S;\pi)$ too, one has $\langle\psi'\rangle = \chi$ because $\psi'_i = \chi \mathbin{\bar{\circ}} \pi_i$. ■

Theorem 9.3 (fixed points in CPO) *(Scott) Any functor $F{:}\,\mathbf{CPO} \to \mathbf{CPO}$ that preserves colimits of cochains has a unique fixed point (modulo isomorphism) μF, $F(\mu F) \cong \mu F$, yielding both an initial F-algebra and a final F-coalgebra.*

Proof. (sketch) Given functor F, we define a cochain $(T;\phi)$ by:

$$\begin{array}{rcl} T_0 & := & \{\bot\} \\ T_{i+1} & := & F.T_i \\ \phi_0 & := & x \mapsto \bot \\ \phi_{i+1} & := & F.\phi_i \end{array}$$

Let $(S;\psi)$ be its colimit, take $\mu F := S$. Now we define a constructor:

$$\tau{:}\,F.(\mu F) \to \mu F \;:=\; x \mapsto (0 :: \bot \mid i+1 :: F.\psi_i.x)$$

To construct $\tau^\cup$, note that $(F.\mu F; F^\omega.\psi)$ is a colimit of

$$(F^\omega.T; F^\omega.\phi) \;=\; ((i :: F.T_i); (i :: F.\phi_i)) \;=\; ((i :: T_{i+1}); (i :: \phi_{i+1})) \ .$$

But as $((i :: T_{i+1}); (i :: \phi_{i+1}); \mu_F; (i :: \psi_{i+1}))$ is a cocone as well, there must be a unique homomorphism

$$\tau^\cup{:}\,(\mu F;\ (i :: \psi_{i+1})) \to (F.\mu F;\ (i :: F.\psi_i)) \ .$$

The unique homomorphisms required for initiality and finality are (see also Paterson [67], where they are called $\mathbf{reduce}_F\,\phi$ and $\mathbf{generate}_F\,\psi$)

$$\mathsf{fix}(g \mapsto \tau^\cup \mathbin{\bar{\circ}} F.g \mathbin{\bar{\circ}} \phi){:}\ \ (\mu F;\tau) \to (X;\phi)$$

and

$$\mathsf{fix}(g \mapsto \psi \mathbin{\bar{\circ}} F.g \mathbin{\bar{\circ}} \tau){:}\ \ (X;\psi) \to (\mu F;\tau^\cup) \ .$$

(end of proof sketch) ■

Theorem 9.4 (fixed point induction)

$$\frac{\begin{array}{l} f\colon \mathcal{D} \to_{\mathsf{c}} \mathcal{D} \\ P(x\colon \mathcal{D})\colon \mathbf{Prop} \\ s\colon \omega\_\mathsf{chain}\, \mathcal{D} \vdash \forall (i\colon \omega :: P(s_i)) \Rightarrow P(\bigsqcup s) \\ P(\bot) \\ \forall (x\colon \mathcal{D} :: P(x) \Rightarrow P(f.x)) \end{array}}{P(\mathsf{fix}\, f)}$$

Proof. Take $s_0 := \bot$, $s_{i+1} := f.s_i$ as in theorem 9.1, then one has $\forall i\colon \omega :: P(s_i)$ and $\mathsf{fix}\, f = \bigsqcup s$, so $P(\mathsf{fix}\, f)$. ■

One of the first computer verification systems was LCF [35, 70], Logic of Computable Functions. It has fixed point induction as a primitive rule, using a syntactic check for chain-completeness.

9.2 Optional objects

We wish to lift a type A to a type $\uparrow A$ with $A \subseteq_{\mathsf{t}} \uparrow A$ and a special value $\bot\colon \uparrow A$, called "undefined". (Such a type $\uparrow A$ is often named '$A_\bot$', but we do not want to use subscripts for this.) Partial functions from A to B may then be represented by total functions from A to $\uparrow B$.

The postfix predicate $x{\downarrow}$ means 'x is defined', and $\uparrow A$ is partially ordered, as follows:

$$\begin{array}{rrcl} x\colon \uparrow A \vdash & x{\downarrow}\colon \mathbf{Prop} & := & \exists a\colon A :: x =_{\uparrow A} a \\ x, y\colon \uparrow A \vdash & x \sqsubseteq y & := & (x{\downarrow} \Rightarrow x = y)\ , \end{array}$$

and this gives a cpo (classically), for

$$s\colon \omega\_\mathsf{chain}\, \uparrow A \vdash \quad \bigsqcup s \ := \ \begin{cases} s_i & \text{if } s_i{\downarrow}, \text{ for some } i \\ \bot & \text{if } \forall i :: s_i = \bot \end{cases} .$$

So $(\uparrow B)^A$ is a cpo too, and any continuous function $e\colon (\uparrow B)^A \to_{\mathsf{c}} (\uparrow B)^A$ must have a unique least fixed point $f = e.f$. This gives us the possibility of recursive definition of partial functions.

There are several ways to define such a type $\uparrow A$ within our calculus:

9.2.1 Explicit options. Classically, the idea is to simply add a singleton type to A, using a sum:

$$\begin{array}{rcl} \mathsf{Opt}\, A & := & A + 1\ , \\ \bot & := & \sigma_1.0 \\ \sigma_0\colon A & \subseteq_{\mathsf{t}} & \mathsf{Opt}\, A\ . \end{array}$$

Constructively, this does not give a cpo. For, to construct a value of type $\mathsf{Opt}\, A$ one must effectively decide whether it has to be $\bot$ or some $a\colon A$. But the limit of an ω-chain s should be $\bot$ if and only if all s_i equal $\bot$, and this cannot be effectively decided.

Similarly, a mapping $e\colon (\mathsf{Opt}\, B)^A \to (\mathsf{Opt}\, B)^A$ that is monotonic (which means that a more defined argument gives a more defined result) can classically be shown to have a unique least fixed point $f = e.f$, but this fixed point is not constructively definable, in general. For, given an argument a, it cannot be effectively decided whether computation of fa will terminate.

9.2.2 Propositional options. An alternative is to think of an optional object as a proposition that tells whether the object is defined or not, together with the actual value in case the proposition is true. Thus, one can only access the value if one has a proof of the proposition.

$$\begin{array}{rcl} \uparrow A & := & \Sigma(D\colon \mathbf{Prop} :: A^D) \\ \bot & := & (\mathsf{False};\, ()) \\ x \mapsto (\mathsf{True};\, (() :: x)) & : & A \subseteq_{\mathsf{t}} \uparrow A \end{array}$$

This $\uparrow A$ gives (using strong existential quantifier elimination) a constructive cpo, for we can define:

$$\bigsqcup(s\colon \omega\_\mathsf{chain}\uparrow A) \;:=\; (\,\exists(i\colon \omega :: \mathsf{fst}\, s_i);\, \exists\_\mathsf{elim}((i; d) :: \mathsf{snd}(s_i d))\,)$$

(Check that for $(i; d), (i'; d')\colon \Sigma(i\colon \omega :: \mathsf{fst}\, s_i)$, one has that $\mathsf{snd}(s_i d) = \mathsf{snd}(s_{i'} d')$.) So continuous mappings $e\colon (\uparrow B)^A \to_{\mathsf{c}} (\uparrow B)^A$ have (constructible) fixed points.

This construction is not possible in Nuprl [18], for its type theory does not have strong $\exists\_\mathsf{elim}$. However, Nuprl has recursive definition of partial functions as a primitive rule. It employs a (restrictive) syntactic test for continuity of recursive definitions.

9.2.3 Lazy options. If one has co-inductive types, there is another alternative. Given a type A, we define a co-inductive type T that represents computations of objects of A. It has constructors $\eta\colon A \to T$ and $\zeta\colon T \to T$, so that a nonterminating computation can be represented by $\zeta.(\zeta.(\ldots))$, repeating ζ indefinitely. Type $\uparrow A$ is then the quotient type (section C.4.2) of T modulo the relation given by ζ, so that $\zeta.x$ and x are identified.

$$\begin{array}{rcl} (T; \delta) & := & \nu(\mathsf{K}\, A + \mathsf{Id}), \\ [\eta, \zeta] & := & \delta^{\cup}; \\ \bot\colon T & := & \zeta.\bot \\ \uparrow A & := & T/\!/\zeta \\ \bot\colon \uparrow A & := & /\!/\_\mathsf{in}(\bot\colon T) \\ (x \mapsto /\!/\_\mathsf{in}(\eta.x))\colon & A & \subseteq_{\mathsf{t}} \uparrow A \end{array}$$

Note that ζ induces an equivalence $\equiv_\zeta$ on T, and that T is partially ordered by:

$$\begin{array}{rcl} (x\colon T)\!\downarrow & := & \exists a\colon A :: x \equiv_\zeta \eta.a \\ x \sqsubseteq y & := & (x\!\downarrow \Rightarrow x \equiv_\zeta y)\ . \end{array}$$

To get a constructive definition of $\bigsqcup(s\colon \omega\_\mathsf{chain}(\uparrow A))$, we have to do a simultaneous quotient elimination on all $s_i\colon \uparrow A$. This is possible by a construction similar to the one in C.4.5. It then suffices to construct $\bigsqcup(t\colon \omega\_\mathsf{chain}\, T)\colon T$.

Note that if there are some $i, k{:}\,\omega$ and $a{:}\,A$ such that $t_i = \zeta^{(k)}.(\eta.a)$, then for any j, one necessarily has $t_{i+j} \equiv_\zeta \eta.a$, and we should have $\bigsqcup t \equiv_\zeta \eta.a$.

Now, one cannot effectively decide whether such i and k exist. Rather, we define an anamorphism $f{:}\,(\mathbb{N}, \phi) \to (T; \delta)$ such that $f.n$ tries only i and k up to bound n, and yields $\zeta.(f.(n+1))$ if that does not succeed. So, making improper use of $\underline{\text{if}}$, we define informally:

$$f.n \;:=\; \underline{\text{if}}\ t_i = \zeta^{(k)}.(\eta.a) \text{ for some } i, k{:}\leq n,\ a{:}\,A\ \underline{\text{then}}\ \eta.a\ \underline{\text{else}}\ \zeta.(f.(n+1))\ .$$

The proper definition requires two local recursions over $\mathbb{N}$, and is left to the reader. Finally we define $\bigsqcup t := f.0$.

9.3 Building recursive cpo's by co-induction

We now generalize the construction in paragraph 9.2.3 of types with lazy optional objects to recursive type definitions. That is, we build a solution to the domain equation $T \cong \uparrow(F.T)$, where F is polynomial, using co-induction.

The value $\bot$ should represent an object that takes infinite time to compute. If we add a constructor $\zeta{:}\,T \to T$ (zeta) to represent a value that takes one step more than its argument, then $\bot$ can be represented by an infinite sequence of ζ's.

$$\begin{array}{rcl} (T;\delta) & := & \nu(F + \mathsf{Id}) \\ [\tau, \zeta] & := & \delta^\cup \\ \bot{:}\,T & := & \zeta.\bot \end{array}$$

Actually, T should be taken modulo a congruence $\simeq$ that identifies any $\zeta.x$ with x. To obtain this congruence, we first define the approximation relation $(\sqsubseteq){:}\subseteq T^2$ as the greatest relation such that, for all $x{:}\,F.T$; $y{:}\,T$:

$$\exists(n{:}\,\mathbb{N} :: \zeta^{(n)}.(\tau.x) \sqsubseteq y) \;\Rightarrow\; \exists(m{:}\,\mathbb{N};\ z{:}\,F.T :: y = \zeta^{(m)}.(\tau.z) \;\wedge\; x\ F.(\sqsubseteq)\ z)\ .$$

Put in relational calculus:

$$\tau \cdot \zeta^{(*)} \cdot (\sqsubseteq) \;\subseteq\; F.(\sqsubseteq) \cdot \tau \cdot \zeta^{(*)}\ . \tag{9.1}$$

Here, $F.(\sqsubseteq){:}\subseteq (F.T)^2$ stands for F lifted to relations as in section D.3, applied to $(\sqsubseteq)$, and $\zeta^{(*)}$ is the reflexive, transitive closure of ζ, i.e.

$$\zeta^{(*)} \;=\; \bigcup(n{:}\,\mathbb{N} :: \zeta^{(n)}) \;=\; \bigcap(Q{:}\,\mathcal{P}T^2 \mid{:}\, \zeta \cup (=) \cup Q \cdot Q \subseteq Q)\ .$$

Note that $\bot \sqsubseteq x$, for any $x{:}\,T$. Then we define $(\simeq) := (\sqsubseteq) \cap (\sqsubseteq)^\cup$.

Theorem 9.5 *1. Relation $\sqsubseteq$ is a preorder.*

2. *When we extend $\sqsubseteq$ to $T//\zeta$, then $(T//\zeta; (\sqsubseteq), \bot)$ is an ω-complete partial order, at least if F is a polynomial, $F.X = \Sigma(a{:}\,A :: X^{Ba})$.*

Proof 1. Relation $\sqsubseteq$ is reflexive, for $(=_T)$ satisfies (9.1) so $(=) \subseteq (\sqsubseteq)$. And $\sqsubseteq$ is transitive, i.e. $(\sqsubseteq) \cdot (\sqsubseteq) \subseteq (\sqsubseteq)$, follows again from $\sqsubseteq$ being maximal, for

$$\begin{array}{lll} & \tau \cdot \zeta^{(*)} \cdot ((\sqsubseteq) \cdot (\sqsubseteq)) & \\ \subseteq & F.(\sqsubseteq) \cdot \tau \cdot \zeta^{(*)} \cdot (\sqsubseteq) & \{\ (9.1)\ \} \\ \subseteq & F.(\sqsubseteq) \cdot F.(\sqsubseteq) \cdot \tau \cdot \zeta^{(*)} & \{\ (9.1)\ \} \\ = & F.((\sqsubseteq) \cdot (\sqsubseteq)) \cdot \tau \cdot \zeta^{(*)} & \end{array}$$

2. By definition of $\simeq$, preorder $\sqsubseteq$ gives a partial order on $T//(\simeq)$.

Proving that ω-chains over $\sqsubseteq$ have limits is rather complicated. Let $s\colon T^\omega$ be a chain, $s_i \sqsubseteq s_{i+1}$, we have to define $\bigsqcup s\colon T$. We follow the method of paragraph 9.2.3.

The idea is that, if there are some $i, k_0\colon \omega$ and $u_0\colon F.T$ such that $s_i = \zeta^{(k0)}.(\tau.u_0)$, then all s_{i+j} must necessarily equal $\zeta^{(kj)}.(\tau.u_j)$ for certain k_j and u_j, and $u_j\ F.(\sqsubseteq)\ u_{j+1}$ (exercise). Hence all u_j equal $(a; v_j)$ for a fixed a and $v_j\colon T^{Ba}$, and $v_j y \sqsubseteq v_{j+1} y$ for $y\colon Ba$. So $(j :: v_j y)$ are chains again, and $\bigsqcup s$ should equal, for some n,

$$\zeta^{(n)}.(\tau.(a; (y :: \bigsqcup (j :: v_j y)))) \ .$$

As trying an unbounded number of i and k cannot be done constructively, we define a homomorphism $f\colon (\omega\_\mathsf{chain}\, T \times \mathbb{N}) \to T$ such that $f.(s, n)$ tries only values of i and k up to bound n, and yields $\zeta.(f.(s, n+1))$ if that does not succeed.

$$\begin{array}{lll} f.(s,n) & := & \underline{\mathsf{if}}\ s_i = \zeta^{(k)}.(\tau.(a; v_0)) \text{ for some } i, k\colon \leq n \text{ and } (a; v_0)\colon F.T \\ & & \underline{\mathsf{then}}\ \text{let } v_j \text{ be such that } s_{i+j} = \zeta^{(k')}.(\tau.(a; v_j)) \text{ in} \\ & & \quad \tau.(a; (y :: f.((j :: v_j y), 0))) \\ & & \underline{\mathsf{else}}\ \zeta.(f.(s, n+1)) \end{array}$$

We leave the constructive definition of i, k, a and the v_j to the reader. Finally we define $\bigsqcup s := f.(s, 0)$. ∎

9.4 Recursive object definitions

The use of a cpo in section 9.3 to define recursive objects in T is something of a detour. In this section we give a more direct construction of T-elements out of recursive object definitions, a construction which does not use the partial order at all.

Suppose that we have a system of mutually recursive tree expressions. We wish to construct the (infinite or partial) trees that are defined by these expressions. For this purpose we need, given a type V representing tree variables, a type E_V that represents tree expressions with variables from V. The family of types E_V with its operations is defined as a co-inductive algebra, and includes a constructor $\zeta\colon E_V \to E_V$ to accommodate nontermination.

Let F be polynomial, $F.X = \Sigma(x\colon A :: X^{Bx})$ and $(T; \delta) := \nu(F + \mathsf{Id})$ as in 9.3. Given a valuation, i.e. a binding of expressions to variables, $t\colon T^V$, any expression $e\colon E_V$ should define a tree $\mathsf{eval}_t.e\colon T$. Elements of E_V may have one of the following forms:

- $\tau.(x\colon F.E_V)$, representing a tree constructed from subtrees

- $\zeta.(e\colon E_V)$, equivalent to just e
- $\eta.(v\colon V)$, representing a variable occurrence
- $\gamma.(d\colon E_V,\ c\colon \Pi(x\colon A :: E_{V+Bx}))$, representing a case analysis on the result of tree expression d. If d evaluates to some tree $\tau.(a;u)$, then evaluation of $\gamma.(d,c)$ should boil down to evaluation of c_a under the valuation $(t,u)\colon T^{V+Ba}$. A suggestive program notation for $\gamma.(d,c)$ might be:

  $$\underline{\mathsf{case}}\ d\ \underline{\mathsf{is}}\ \tau.(a;u) \Longrightarrow c_a$$

  where expression c_a may contain variables referring to the tuple of trees $u\colon T^{Ba}$.

Thus, we define E as follows.

$$\begin{array}{rcl} F' & : & \mathbf{TYPE}^{\mathbf{Type}} \to \mathbf{TYPE}^{\mathbf{Type}} \\ (F'.X)_V & := & F.X_V + X_V + V + (X_V \times \Pi(x\colon A :: X_{V+Bx})) \\ (E;\delta') & := & \nu F' \\ [\tau,\zeta,\eta,\gamma] & := & \delta'^{\cup} \end{array}$$

We omit the index V of the operations. Note that there is an embedding $[\![V :: \delta \circ [\sigma_0,\sigma_1]\,]\!]_V\colon T \subseteq_{\mathsf{t}} E_V$.

To define the evaluation function as a homomorphism, we need a substitution operation. For a fixed type W, we define the tuple of substitution functions $\mathsf{subst}_V\colon (E_{V+W} \times E_V^W) \to E_V$ by the equations:

$$\begin{array}{rcl} \mathsf{subst}_V.(\tau.x,t) & = & \tau.(F.(\mathsf{subst}_V \circ \langle \mathsf{I}, \mathsf{K}\,t\rangle).x) \\ \mathsf{subst}_V.(\zeta.d,t) & = & \zeta.(\mathsf{subst}_V.(d,t)) \\ \mathsf{subst}_V.(\eta.(0;v),t) & = & \eta.v \\ \mathsf{subst}_V.(\eta.(1;w),t) & = & \zeta.tw \\ \mathsf{subst}_V.(\gamma.(d,c),t) & = & \gamma.(\mathsf{subst}_V.(d,t),(x :: \mathsf{subst}_{V+Bx}.(cx^{\backslash}E_r, t^{\backslash}E^W_{\sigma_0}))) \end{array}$$

where we use that E_V is functorial in its subscript V, and

$$r\colon (V+W)+Bx \to (V+Bx)+W \ :=\ [\,[\sigma_0\circ\sigma_0,\sigma_1],\sigma_1\circ\sigma_0]\ .$$

To see the need for this, note that subst_{V+Bx} requires an argument of type $(E_{V+Bx+W} \times E^W_{V+Bx})$ whereas we have $cx\colon E_{V+W+Bx}$ and $t\colon E_V^W$. For well-definedness of subst, note that these equations can be given the shape of a dual recursion (paragraph 7.3.1)

$$\mathsf{subst} = \delta^{\cup} \circ F'.[\mathsf{I},\mathsf{subst}] \circ \phi$$

for some $\phi\colon D \to E + F'.D$ $\underline{\mathsf{in}}$ $\mathbf{TYPE}^{\mathbf{Type}}$ where $D_V := E_{V+W} \times E_V^W$.

Now suppose we have a tuple of recursive expressions,

$$t\colon E_V^V\ .$$

In context t we define, for any expression $e\colon E_V$, the tree $\mathsf{eval}_t.e$ that is denoted by e when parameters $v\colon V$ are bound to tv. This eval_t is defined as the unique homomorphism

$$\mathsf{eval}_t\colon (E_V;\phi) \to (T;\delta)$$

for some $\phi\colon E_V \to F.E_V + E_V$ by:

$$\begin{array}{rcl}
\mathsf{eval}_t.(\tau.x) & = & \tau.(F.\mathsf{eval}_t.x) \\
\mathsf{eval}_t.(\zeta.e) & = & \zeta.(\mathsf{eval}_t.e) \\
\mathsf{eval}_t.(\eta.v) & = & \zeta.(\mathsf{eval}_t.tv) \\
\mathsf{eval}_t.(\gamma.(\tau.(a;u),c)) & = & \zeta.(\mathsf{eval}_t.(\mathsf{subst}_V.(ca,u))) \\
\mathsf{eval}_t.(\gamma.(\zeta.e,c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(e,c))) \\
\mathsf{eval}_t.(\gamma.(\eta.v,c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(tv,c))) \\
\mathsf{eval}_t.(\gamma.(\gamma.(e,d),c)) & = & \zeta.(\mathsf{eval}_t.(\gamma.(e,(x :: \gamma.(dx,c))))))
\end{array}$$

One may prove that if $\mathsf{eval}_t.d \simeq \tau.(a;u)$, then $\mathsf{eval}_t.(\gamma.(d,c)) \simeq \mathsf{eval}_{(t,u)}.ca$.

9.5 Conclusion

We introduced the most basic notions of recursive domain theory, and defined simple classical and constructive domains for representing optional objects. We then developed in 9.2 a representation for lazy optional objects by means of co-induction within the strict type theory of *ADAM*, which we generalized in 9.3 to lazy recursive types. While this representation of types is quite elegant, the representation of actual recursive object definitions is not. If one wishes to use such objects in a constructive type theory, it seems preferable to include them in the theory as primitives.