CHAPTER 2

Computability

At this point we leave the general setting of logic and aim to get closer to mathematics. We introduce free algebras (for example, the natural numbers) as basic data structures and consider function spaces based on them. The functional objects are viewed as limits of their finite approximations. We call a functional computable if it is the limit of a recursively enumerable set of finite approximations. To work with such objects in a formal theory we need to have a language to denote them. Again lambda calculus is the appropriate tool, this time extended by constants for particular functionals defined by equations.

It is a fundamental property of computation that evaluation must be finite. So in any evaluation of $\Phi(\varphi)$ the argument φ can be called upon only finitely many times, and hence the value – if defined – must be determined by some finite subfunction of φ. This is the principle of finite support.

Let us carry this discussion somewhat further and look at the situation one type higher up. Let $\mathcal{H}$ be a partial functional of type-3, mapping type-2 functionals Φ to natural numbers. Suppose Φ is given and $\mathcal{H}(\Phi)$ evaluates to a defined value. Again, evaluation must be finite. Hence the argument Φ can only be called on finitely many functions φ. Furthermore each such φ must be presented to Φ in a finite form (explicitly say, as a set of ordered pairs). In other words, $\mathcal{H}$ and also any type-2 argument Φ supplied to it must satisfy the finite support principle, and this must continue to apply as we move up through the types.

To describe this principle more precisely, we need to introduce the notion of a "finite approximation" Φ_0 of a functional Φ. By this we mean a finite set X of pairs (φ_0, n) such that (i) φ_0 is a finite function, (ii) $\Phi(\varphi_0)$ is defined with value n, and (iii) if (φ_0, n) and (φ_0', n') belong to X where φ_0 and φ_0' are "consistent", then $n = n'$. The essential idea here is that Φ should be viewed as the union of all its finite approximations. Using this notion of a finite approximation we can now formulate the

> *Principle of finite support.* If $\mathcal{H}(\Phi)$ is defined with value n, then there is a finite approximation Φ_0 of Φ such that $\mathcal{H}(\Phi_0)$ is defined with value n.

The monotonicity principle formalizes the simple idea that once $\mathcal{H}(\Phi)$ is evaluated, then the same value will be obtained no matter how the argument Φ is extended. This requires the notion of "extension". Φ' extends Φ if for any piece of data (φ_0, n) in Φ there exists another (φ_0', n) in Φ' such that φ_0 extends φ_0' (note the contravariance!). The second basic principle is then

> *Monotonicity principle*. If $\mathcal{H}(\Phi)$ is defined with value n and Φ' extends Φ, then $\mathcal{H}(\Phi')$ is defined with value n.

An immediate consequence of finite support and monotonicity is that the behaviour of any functional is indeed determined by its set of finite approximations. For if Φ, Φ' have the same finite approximations and $\mathcal{H}(\Phi)$ is defined with value n, then by finite support, $\mathcal{H}(\Phi_0)$ is defined with value n for some finite approximation Φ_0, and then by monotonicity $\mathcal{H}(\Phi')$ is defined with value n. Thus $\mathcal{H}(\Phi) = \mathcal{H}(\Phi')$, for all $\mathcal{H}$.

This observation now allows us to formulate a notion of abstract computability:

> *Effectivity principle*. An object is computable just in case its set of finite approximations is (primitive) recursively enumerable (or equivalently, Σ^0_1-definable).

The general theory of computability concerns partial functions and partial operations on them. However, we might be interested in total objects only, so once the theory of partial objects is developed, we can look for ways to extract the total ones. We will define the total objects of base type inductively, and then by induction on types the notion of totality for arbitrary types.

2.1. Abstract computability via information systems

We need to define appropriate domains for our to-be-defined computable functionals, viewed as limits of their finite approximations. Information systems are a convenient setting to introduce and study the latter.

2.1.1. Information systems. The basic idea of information systems is to provide an axiomatic setting to describe approximations of abstract objects (like functions or functionals) by concrete, finite ones. We do not attempt to analyze the notion of "concreteness" or finiteness here, but rather take an arbitrary countable set A of "bits of data" or "tokens" as a basic notion to be explained axiomatically. In order to use such data to build approximations of abstract objects, we need a notion of "consistency", which determines when the elements of a finite set of tokens are consistent with each other. We also need an "entailment relation" between consistent sets U of data and single tokens a, which intuitively expresses the fact that the information contained in U is sufficient to compute the bit of information a.

The axioms below are a minor modification of Scott's (1982), due to Larsen and Winskel (1991).

DEFINITION. An *information system* is a structure $(A, \mathrm{Con}, \vdash)$ where A is a countable set (the *tokens*), Con is a non-empty set of finite subsets of A (the *consistent* sets) and $\vdash$ is a subset of $\mathrm{Con} \times A$ (the *entailment* relation), which satisfy

$$\begin{aligned}
&U \subseteq V \in \mathrm{Con} \to U \in \mathrm{Con},\\
&\{a\} \in \mathrm{Con},\\
&U \vdash a \to U \cup \{a\} \in \mathrm{Con},\\
&a \in U \in \mathrm{Con} \to U \vdash a,\\
&U \in \mathrm{Con} \to \forall_{a \in V}(U \vdash a) \to V \vdash b \to U \vdash b.
\end{aligned}$$

The elements of Con are called *formal neighborhoods*. We use U, V, W to denote *finite* sets, and write

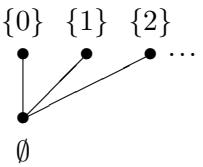
$$\begin{aligned}
&U \vdash V \quad \text{for} \quad U \in \mathrm{Con} \wedge \forall_{a \in V}(U \vdash a),\\
&a \uparrow b \quad \text{for} \quad \{a, b\} \in \mathrm{Con} \qquad (a, b \text{ are } \textit{consistent}),\\
&U \uparrow V \quad \text{for} \quad \forall_{a \in U, b \in V}(a \uparrow b).
\end{aligned}$$

DEFINITION. The *ideals* (also called *objects*) of an information system $\boldsymbol{A} = (A, \mathrm{Con}, \vdash)$ are defined to be those subsets x of A which satisfy

$$\begin{aligned}
&U \subseteq x \to U \in \mathrm{Con} \quad (x \text{ is } \textit{consistent}),\\
&U \vdash a \to U \subseteq x \to a \in x \quad (x \text{ is } \textit{deductively closed}).
\end{aligned}$$

For example the *deductive closure* $\overline{U} := \{\, a \in A \mid U \vdash a \,\}$ of $U \in \mathrm{Con}$ is an ideal. The set of all ideals of $\boldsymbol{A}$ is denoted by $|\boldsymbol{A}|$.

EXAMPLES. Every countable set A can be turned into a *flat* information system by letting the set of tokens be A, $\mathrm{Con} := \{\emptyset\} \cup \{\, \{a\} \mid a \in A \,\}$ and $U \vdash a$ mean $a \in U$. In this case the ideals are just the elements of Con. For $A = \mathbb{N}$ we have the following picture of the Con-sets.



A rather important example is the following, which concerns approximations of functions from a countable set A into a countable set B. The tokens are the pairs (a, b) with $a \in A$ and $b \in B$, and

$$\mathrm{Con} := \{\, \{\, (a_i, b_i) \mid i < k \,\} \mid \forall_{i,j<k}(a_i = a_j \to b_i = b_j) \,\},$$

$$U \vdash (a,b) := (a,b) \in U.$$

It is easy to verify that this defines an information system whose ideals are (the graphs of) all partial functions from A to B.

One can show that for every information system $\boldsymbol{A} = (A, \mathrm{Con}, \vdash)$ the structure $(|\boldsymbol{A}|, \subseteq, \overline{\emptyset})$ is a "domain" (also called Scott-Ershov domain, or "bounded complete algebraic cpo"), whose set of "compact elements" can be represented as $|\boldsymbol{A}|_c = \{\, \overline{U} \mid U \in \mathrm{Con} \,\}$. The converse holds as well: every contable domain can be represented as an information system. We will not need this relation to standard (non-constructive) domain theory, and hence not even define these notions here.

2.1.2. Function spaces. We define the "function space" $\boldsymbol{A} \to \boldsymbol{B}$ between two information systems $\boldsymbol{A}$ and $\boldsymbol{B}$.

Definition. Let $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$ and $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$ be information systems. Define $\boldsymbol{A} \to \boldsymbol{B} = (C, \mathrm{Con}, \vdash)$ by

$$C := \mathrm{Con}_A \times B,$$

$$\{\, (U_i, b_i) \mid i \in I \,\} \in \mathrm{Con} := \forall_{J \subseteq I}\Big(\bigcup_{j \in J} U_j \in \mathrm{Con}_A \to \{\, b_j \mid j \in J \,\} \in \mathrm{Con}_B\Big).$$

For the definition of the entailment relation $\vdash$ it is helpful to first define the notion of an *application* of $W := \{\, (U_i, b_i) \mid i \in I \,\} \in \mathrm{Con}$ to $U \in \mathrm{Con}_A$:

$$\{\, (U_i, b_i) \mid i \in I \,\}U := \{\, b_i \mid U \vdash_A U_i \,\}.$$

From the definition of Con we know that this set is in Con_B. Now define $W \vdash (U, b)$ by $WU \vdash_B b$.

Clearly application is *monotone in the second argument*, in the sense that $U \vdash_A U'$ implies ($WU' \subseteq WU$, hence also) $WU \vdash_B WU'$. In fact, application is also *monotone in the first argument*, i.e.,

$$W \vdash W' \quad \text{implies} \quad WU \vdash_B W'U.$$

To see this let $W = \{\, (U_i, b_i) \mid i \in I \,\}$ and $W' = \{\, (U'_j, b'_j) \mid j \in J \,\}$. By definition $W'U = \{\, b'_j \mid U \vdash_A U'_j \,\}$. Now fix j such that $U \vdash_A U'_j$; we must show $WU \vdash_B b'_j$. By assumption $W \vdash (U'_j, b'_j)$, hence $WU'_j \vdash_B b'_j$. Because of $WU \supseteq WU'_j$ the claim follows.

Lemma. *If $\boldsymbol{A}$ and $\boldsymbol{B}$ are information systems, then so is $\boldsymbol{A} \to \boldsymbol{B}$ defined as above.*

Proof. Let $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$ and $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$. The first, second and fourth property of the definition are clearly satisfied. For the third, suppose

$$\{(U_1, b_1), \ldots, (U_n, b_n)\} \vdash (U, b), \quad \text{i.e.,} \quad \{\, b_j \mid U \vdash_A U_j \,\} \vdash_B b.$$

We have to show that $\{(U_1, b_1), \ldots, (U_n, b_n), (U, b)\} \in \mathrm{Con}$. So let $I \subseteq \{1, \ldots, n\}$ and suppose

$$U \cup \bigcup_{i \in I} U_i \in \mathrm{Con}_A.$$

We must show that $\{b\} \cup \{\, b_i \mid i \in I \,\} \in \mathrm{Con}_B$. Let $J \subseteq \{1, \ldots, n\}$ consist of those j with $U \vdash_A U_j$. Then also

$$U \cup \bigcup_{i \in I} U_i \cup \bigcup_{j \in J} U_j \in \mathrm{Con}_A.$$

Since

$$\bigcup_{i \in I} U_i \cup \bigcup_{j \in J} U_j \in \mathrm{Con}_A,$$

from the consistency of $\{(U_1, b_1), \ldots, (U_n, b_n)\}$ we can conclude that

$$\{\, b_i \mid i \in I \,\} \cup \{\, b_j \mid j \in J \,\} \in \mathrm{Con}_B.$$

But $\{\, b_j \mid j \in J \,\} \vdash_B b$ by assumption. Hence

$$\{\, b_i \mid i \in I \,\} \cup \{\, b_j \mid j \in J \,\} \cup \{b\} \in \mathrm{Con}_B.$$

For the final property, suppose

$$W \vdash W' \quad \text{and} \quad W' \vdash (U, b).$$

We have to show $W \vdash (U, b)$, i.e., $WU \vdash_B b$. We obtain $WU \vdash_B W'U$ by monotonicity in the first argument, and $W'U \vdash b$ by definition. □

We shall now give an alternative characterization of the ideals in $\boldsymbol{A} \to \boldsymbol{B}$, as "approximable maps". The basic idea for approximable maps is the desire to study "information respecting" maps from $\boldsymbol{A}$ into $\boldsymbol{B}$. Such a map is given by a relation r between Con_A and B, where $(U, b) \in r$ intuitively means that whenever we are given the information $U \in \mathrm{Con}_A$, then we know that at least the token b appears in the value.

Definition. Let $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$ and $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$ be information systems. A relation $r \subseteq \mathrm{Con}_A \times B$ is an *approximable map* if it satisfies the following:

(a) if $(U, b_1), \ldots, (U, b_n) \in r$, then $\{b_1, \ldots, b_n\} \in \mathrm{Con}_B$;
(b) if $(U, b_1), \ldots, (U, b_n) \in r$ and $\{b_1, \ldots, b_n\} \vdash_B b$, then $(U, b) \in r$;
(c) if $(U', b) \in r$ and $U \vdash_A U'$, then $(U, b) \in r$.

Theorem. *Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be information systems. Then the ideals of $\boldsymbol{A} \to \boldsymbol{B}$ are exactly the approximable maps from $\boldsymbol{A}$ to $\boldsymbol{B}$.*

PROOF. Let $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$ and $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$. If $r \in |\boldsymbol{A} \to \boldsymbol{B}|$ then $r \subseteq \mathrm{Con}_A \times B$ is consistent and deductively closed. We have to show that r satisfies the axioms for approximable maps.

(a) Let $(U, b_1), \ldots, (U, b_n) \in r$. We must show that $\{b_1, \ldots, b_n\} \in \mathrm{Con}_B$. But this clearly follows from the consistency of r.

(b) Let $(U, b_1), \ldots, (U, b_n) \in r$ and $\{b_1, \ldots, b_n\} \vdash_B b$. We must show that $(U, b) \in r$. But

$$\{(U, b_1), \ldots, (U, b_n)\} \vdash (U, b)$$

by the definition of the entailment relation $\vdash$ in $\boldsymbol{A} \to \boldsymbol{B}$, hence $(U, b) \in r$ since r is deductively closed.

(c) Let $U \vdash_A U'$ and $(U', b) \in r$. We must show that $(U, b) \in r$. But

$$\{(U', b)\} \vdash (U, b)$$

since $\{(U', b)\}U = \{b\}$ (which follows from $U \vdash_A U'$), hence $(U, b) \in r$, again since r is deductively closed.

For the other direction suppose that $r \subseteq \mathrm{Con}_A \times B$ is an approximable map. We must show that $r \in |\boldsymbol{A} \to \boldsymbol{B}|$.

Consistency of r. Suppose $(U_1, b_1), \ldots, (U_n, b_n) \in r$ and $U = \bigcup\{ U_i \mid i \in I \} \in \mathrm{Con}_A$ for some $I \subseteq \{1, \ldots, n\}$. We must show that $\{ b_i \mid i \in I \} \in \mathrm{Con}_B$. Now from $(U_i, b_i) \in r$ and $U \vdash_A U_i$ we obtain $(U, b_i) \in r$ by axiom (c) for all $i \in I$, and hence $\{ b_i \mid i \in I \} \in \mathrm{Con}_B$ by axiom (a).

Deductive closure of r. Suppose $(U_1, b_1), \ldots, (U_n, b_n) \in r$ and

$$W := \{(U_1, b_1), \ldots, (U_n, b_n)\} \vdash (U, b).$$

We must show $(U, b) \in r$. By definition of $\vdash$ for $\boldsymbol{A} \to \boldsymbol{B}$ we have $WU \vdash_B b$, which is $\{ b_i \mid U \vdash_A U_i \} \vdash_B b$. Further by our assumption $(U_i, b_i) \in r$ we know $(U, b_i) \in r$ by axiom (c) for all i with $U \vdash_A U_i$. Hence $(U, b) \in r$ by axiom (b). □

2.1.3. Continuous functions. We can also characterize approximable maps in a different way, which is closer to usual characterizations of continuity[1]:

LEMMA. *Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be information systems and $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ monotone (i.e., $x \subseteq y$ implies $f(x) \subseteq f(y)$). Then the following are equivalent.*

(a) *f satisfies the "principle of finite support" PFS: If $b \in f(x)$, then $b \in f(\overline{U})$ for some $U \subseteq x$.*

[1]In fact, approximable maps are exactly the continuous functions w.r.t. the so-called Scott topology. However, we will not enter this subject here.

(b) *f commutes with directed unions: for every directed $D \subseteq |\boldsymbol{A}|$ (i.e., for any $x, y \in D$ there is a $z \in D$ such that $x, y \subseteq z$)*

$$f(\bigcup_{x\in D} x) = \bigcup_{x\in D} f(x).$$

Note that in (b) the set $\{ f(x) \mid x \in D \}$ is directed by monotonicity of f; hence its union is indeed an ideal in $|\boldsymbol{A}|$. Note also that from PFS and monotonicity of f it follows immediately that if $V \subseteq f(x)$, then $V \subseteq f(\overline{U})$ for some $U \subseteq x$.

Proof. Let f satisfy PFS, and $D \subseteq |\boldsymbol{A}|$ be directed. $f(\bigcup_{x\in D} x) \supseteq \bigcup_{x\in D} f(x)$ follows from monotonicity. For the reverse inclusion let $b \in f(\bigcup_{x\in D} x)$. Then by PFS $b \in f(\overline{U})$ for some $U \subseteq \bigcup_{x\in D} x$. From the directedness and the fact that U is finite we obtain $U \subseteq z$ for some $z \in D$. From $b \in f(\overline{U})$ and monotonicity infer $b \in f(z)$. Conversely, let f commute with directed unions, and assume $b \in f(x)$. Then

$$b \in f(x) = f(\bigcup_{U\subseteq x} \overline{U}) = \bigcup_{U\subseteq x} f(\overline{U}),$$

hence $b \in f(\overline{U})$ for some $U \subseteq x$. □

We call a function $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ continuous if it satifies the conditions in the lemma above. Hence continuous maps $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ are those that can be completely described from the point of view of finite approximations of the abstract objects $x \in |\boldsymbol{A}|$ and $f(x) \in |\boldsymbol{B}|$: whenever we are given a finite approximation V to the value $f(x)$, then there is a finite approximation U to the argument x such that already $f(\overline{U})$ contains the information in V; note that by monotonicity $f(\overline{U}) \subseteq f(x)$.

Clearly the identity and constant functions are continuous, and also the *composition* $g \circ f$ of continuous functions $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ and $g\colon |\boldsymbol{B}| \to |\boldsymbol{C}|$.

Theorem. *Let $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$, $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$ be information systems. Then the ideals of $\boldsymbol{A} \to \boldsymbol{B}$ are in a natural bijective correspondence with the continuous functions from $|\boldsymbol{A}|$ to $|\boldsymbol{B}|$, as follows.*

(a) *With any approximable map $r \subseteq \mathrm{Con}_A \times B$ we can associate a continuous function $|r|\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ by*

$$|r|(z) := \{ b \in B \mid (U, b) \in r \text{ for some } U \subseteq z \}.$$

We call $|r|(z)$ the application *of r to z.*

(b) *Conversely, with any continuous function $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ we can associate an approximable map $\hat{f} \subseteq \mathrm{Con}_A \times B$ by*

$$\hat{f} := \{ (U, b) \mid b \in f(\overline{U}) \}.$$

These assignments are inverse to each other, i.e., $f = |\hat{f}|$ *and* $r = \widehat{|r|}$.

PROOF. Let r be an ideal of $\boldsymbol{A} \to \boldsymbol{B}$; then by the theorem just proved r is an approximable map. We first show that $|r|$ is well-defined. So let $z \in |\boldsymbol{A}|$.

$|r|(z)$ is consistent: let $b_1, \dots, b_n \in |r|(z)$. Then there are $U_1, \dots, U_n \subseteq z$ such that $(U_i, b_i) \in r$. Hence $U := U_1 \cup \dots \cup U_n \subseteq z$ and $(U, b_i) \in r$ by axiom (c) of approximable maps. Now from axiom (a) we can conclude that $\{b_1, \dots, b_n\} \in \mathrm{Con}_B$.

$|r|(z)$ is deductively closed: let $b_1, \dots, b_n \in |r|(z)$ and $\{b_1, \dots, b_n\} \vdash_B b$. We must show $b \in |r|(z)$. As before we find $U \subseteq z$ such that $(U, b_i) \in r$. Now from axiom (b) we can conclude $(U, b) \in r$ and hence $b \in |r|(z)$.

Continuity of $|r|$ follows immediately from part (a) of the lemma above, since by definition $|r|$ is monotone and satisfies PFS.

Now let $f\colon |\boldsymbol{A}| \to |\boldsymbol{B}|$ be continuous. It is easy to verify that $\hat{f}$ is indeed an approximable map. Furthermore

$$\begin{aligned} b \in |\hat{f}|(z) &\leftrightarrow (U, b) \in \hat{f} && \text{for some } U \subseteq z \\ &\leftrightarrow b \in f(\overline{U}) && \text{for some } U \subseteq z \\ &\leftrightarrow b \in f(z) && \text{by monotonicity and PFS.} \end{aligned}$$

Finally, for any approximable map $r \subseteq \mathrm{Con}_A \times B$ we have

$$\begin{aligned} (U, b) \in r &\leftrightarrow \exists_{V \subseteq \overline{U}} (V, b) \in r \quad \text{by axiom (c) for approximable maps} \\ &\leftrightarrow b \in |r|(\overline{U}) \\ &\leftrightarrow (U, b) \in \widehat{|r|}, \end{aligned}$$

hence $r = \widehat{|r|}$. □

Consequently we can (and will) view approximable maps $r \subseteq \mathrm{Con}_A \times B$ as continuous functions from $|\boldsymbol{A}|$ to $|\boldsymbol{B}|$. Equality of two subsets $r, s \subseteq \mathrm{Con}_A \times B$ means that they consist of the same tokens (U, b). We can characterize equality $r = s$ by extensional equality of the associated functions $|r|, |s|$. It even suffices that $|r|$ and $|s|$ coincide on all compact elements $\overline{U}$ for $U \in \mathrm{Con}_A$.

LEMMA (Extensionality). *Let* $\boldsymbol{A} = (A, \mathrm{Con}_A, \vdash_A)$, $\boldsymbol{B} = (B, \mathrm{Con}_B, \vdash_B)$ *be information systems and* $r, s \subseteq \mathrm{Con}_A \times B$ *approximable maps. Then the following are equivalent.*

(a) $r = s$,
(b) $|r|(z) = |s|(z)$ *for all* $z \in |\boldsymbol{A}|$,
(c) $|r|(\overline{U}) = |s|(\overline{U})$ *for all* $U \in \mathrm{Con}_A$.

PROOF. It suffices to prove (c) → (a). As above this follows from

$$(U,b) \in r \leftrightarrow \exists_{V \subseteq \overline{U}}(V,b) \in r \quad \text{by axiom (c) for approximable maps}$$
$$\leftrightarrow b \in |r|(\overline{U}).$$

□

Moreover, one can easily check that

$$r \circ s := \{\, (U,c) \mid \exists_V((U,V) \subseteq s \wedge (V,c) \in r) \,\}$$

is an approximable map (where $(U,V) := \{\, (U,b) \mid b \in V \,\}$), and

$$|r \circ s| = |r| \circ |s|, \quad \widehat{f \circ g} = \hat{f} \circ \hat{g}.$$

We usually write $r(z)$ for $|r|(z)$, and similarly $(U,b) \in f$ for $(U,b) \in \hat{f}$. It should always be clear from the context where the mods and hats should be inserted.

2.1.4. Algebras and types. We now consider concrete information systems, our basis for continuous functionals.

Types will be built from base types by the formation of function types, $\rho \to \sigma$. As domains for the base types we choose non-flat and possibly infinitary free algebras, given by their constructors. The main reason for taking non-flat base domains is that we want the constructors to be injective and with disjoint ranges. This generally is not the case for flat domains.

DEFINITION (Types). We inductively define *type forms*

$$\rho, \sigma ::= \alpha \mid \rho \to \sigma \mid \mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k}$$

with α, ξ type variables and $k \geq 1$. Note that $(\rho_\nu)_{\nu<n} \to \sigma$ means $\rho_0 \to \ldots \to \rho_{n-1} \to \sigma$, associated to the right. Let $\mathrm{TV}(\rho)$ denote the set of (free) type variables in ρ. We define $\mathrm{SP}(\alpha,\rho)$ "α occurs at most *strictly positive* in ρ" by induction on ρ.

$$\mathrm{SP}(\alpha,\beta) \qquad \frac{\alpha \notin \mathrm{TV}(\rho) \quad \mathrm{SP}(\alpha,\sigma)}{\mathrm{SP}(\alpha,\rho \to \sigma)} \qquad \frac{\mathrm{SP}(\alpha,\rho_{i\nu}) \text{ for all } i<k,\ \nu<n_i}{\mathrm{SP}(\alpha,\mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k})}$$

Now we can define $\mathrm{Ty}(\rho)$ "ρ is a *type*", again by induction on ρ.

$$\mathrm{Ty}(\alpha) \qquad \frac{\mathrm{Ty}(\rho) \quad \mathrm{Ty}(\sigma)}{\mathrm{Ty}(\rho \to \sigma)} \qquad \frac{\mathrm{Ty}(\rho_{i\nu}) \text{ and } \mathrm{SP}(\xi,\rho_{i\nu}) \text{ for all } i<k,\ \nu<n_i}{\mathrm{Ty}(\mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k})}$$

We call

$$\iota := \mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k}$$

an *algebra*. Sometimes it is helpful to display the type parameters and write $\iota(\vec{\alpha},\vec{\beta}\,)$, where $\vec{\alpha},\vec{\beta}$ are all type variables except ξ free in some $\rho_{i\nu}$, and $\vec{\alpha}$ are the ones occuring only strictly positive. If we write the i-th component of ι in the form $(\rho_{i\nu}(\xi))_{\nu<n_i} \to \xi$, then we call

$$(\rho_{i\nu}(\iota))_{\nu<n_i} \to \iota$$

the i-th *constructor type* of ι.

In $(\rho_{i\nu}(\xi))_{\nu<n_i} \to \xi$ we call $\rho_{i\nu}(\xi)$ a *parameter* argument type if ξ does not occur in it, and a *recursive* argument type otherwise. A recursive argument type $\rho_{i\nu}(\xi)$ is *nested* if it has an occurrence of ξ in a strictly positive parameter position of another (previously defined) algebra, and unnested otherwise. An algebra ι is called *nested* if it has a constructor with at least one nested recursive argument type, and *unnested* otherwise.

Examples.

$$\begin{aligned}
\mathbf{U} &:= \mu_\xi \xi \qquad \text{(unit)},\\
\mathbf{B} &:= \mu_\xi(\xi,\xi) \qquad \text{(booleans)},\\
\mathbf{N} &:= \mu_\xi(\xi,\xi\to\xi) \qquad \text{(natural numbers, unary)},\\
\mathbf{P} &:= \mu_\xi(\xi,\xi\to\xi,\xi\to\xi) \qquad \text{(positive numbers, binary)},\\
\mathbf{D} &:= \mu_\xi(\xi,\xi\to\xi\to\xi) \qquad \text{(binary trees, or derivations)},\\
\mathbf{O} &:= \mu_\xi(\xi,\xi\to\xi,(\mathbf{N}\to\xi)\to\xi) \qquad \text{(ordinals)},\\
\mathbf{T}_0 &:= \mathbf{N}, \quad \mathbf{T}_{n+1} := \mu_\xi(\xi,(\mathbf{T}_n\to\xi)\to\xi) \qquad \text{(trees)}.
\end{aligned}$$

Examples of algebras strictly positive in their type parameters are

$$\begin{aligned}
\mathbf{I}(\alpha) &:= \mu_\xi(\alpha\to\xi) \qquad \text{(identity)},\\
\mathbf{L}(\alpha) &:= \mu_\xi(\xi,\alpha\to\xi\to\xi) \qquad \text{(lists)},\\
\alpha\times\beta &:= \mu_\xi(\alpha\to\beta\to\xi) \qquad \text{(product)},\\
\alpha+\beta &:= \mu_\xi(\alpha\to\xi,\beta\to\xi) \qquad \text{(sum)}.
\end{aligned}$$

An example of a nested algebra is

$$\mathbf{T} := \mu_\xi(\mathbf{L}(\xi)\to\xi) \quad \text{(finitely branching trees)}.$$

However, for simplicity we one deal with unnested algebras here.

Let ρ be a type; we write $\rho(\vec{\alpha}\,)$ for ρ to indicate its dependence on the type parameters $\vec{\alpha}$. We can substitute types $\vec{\sigma}$ for $\vec{\alpha}$, to obtain $\rho(\vec{\sigma}\,)$. Examples are $\mathbf{L}(\mathbf{B})$, the type of lists of booleans, and $\mathbf{N}\times\mathbf{N}$, the type of pairs of natural numbers.

Note that often there are many equivalent ways to define a particular type. For instance, we could take $\mathbf{U}+\mathbf{U}$ to be the type of booleans, $\mathbf{L}(\mathbf{U})$ to be the type of natural numbers, and $\mathbf{L}(\mathbf{B})$ to be the type of positive binary numbers.

For every constructor type of an algebra we provide a (typed) *constructor symbol* C_i. In some cases they have standard names, for instance

$\mathtt{tt}^{\mathbf{B}}, \mathtt{ff}^{\mathbf{B}}$ for the two constructors of the type $\mathbf{B}$ of booleans,

$0^{\mathbf{N}}, S^{\mathbf{N}\to\mathbf{N}}$ for the type $\mathbf{N}$ of (unary) natural numbers,

$1^{\mathbf{P}}, S_0^{\mathbf{P}\to\mathbf{P}}, S_1^{\mathbf{P}\to\mathbf{P}}$ for the type $\mathbf{P}$ of (binary) positive numbers,

$0^{\mathbf{O}}, S^{\mathbf{O}\to\mathbf{O}}, \mathrm{Sup}^{(\mathbf{N}\to\mathbf{O})\to\mathbf{O}}$ for the type $\mathbf{O}$ of ordinals,

$[]^{\mathbf{L}(\rho)}, \mathrm{cons}^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)}$ for the type $\mathbf{L}(\rho)$ of lists,

$(\mathrm{Inl}_{\rho\sigma})^{\rho\to\rho+\sigma}, (\mathrm{Inr}_{\rho\sigma})^{\sigma\to\rho+\sigma}$ for the sum type $\rho+\sigma$.

An algebra form ι is *structure-finitary* if all its argument types $\rho_{i\nu}$ are not of arrow form. It is *finitary* if in addition it has no type variables. In the examples above $\mathbf{U}$, $\mathbf{B}$, $\mathbf{N}$, $\mathbf{P}$ and $\mathbf{D}$ are all finitary, but $\mathbf{O}$ and $\mathbf{T}_{n+1}$ are not. $\mathbf{L}(\rho)$, $\rho\times\sigma$ and $\rho+\sigma$ are structure-finitary, and finitary if their parameter types are.

An algebra is *explicit* if all its constructor types have parameter argument types only (i.e., no recursive argument types). In the examples above $\mathbf{U}$, $\mathbf{B}$, $\rho\times\sigma$ and $\rho+\sigma$ are explicit, but $\mathbf{N}$, $\mathbf{P}$, $\mathbf{L}(\rho)$, $\mathbf{D}$, $\mathbf{O}$ and $\mathbf{T}_{n+1}$ are not.

We will also need the notion of the *level* of a type, which is defined by

$$\mathrm{lev}(\iota) := 0, \qquad \mathrm{lev}(\rho\to\sigma) := \max\{\mathrm{lev}(\sigma), 1+\mathrm{lev}(\rho)\}.$$

Base types are types of level 0, and a *higher* type has level at least 1.

2.1.5. Partial continuous functionals. For every type ρ we define the information system $\boldsymbol{C}_\rho = (C_\rho, \mathrm{Con}_\rho, \vdash_\rho)$. The ideals $x \in |\boldsymbol{C}_\rho|$ are the *partial continuous functionals* of type ρ. Since we will have $\boldsymbol{C}_{\rho\to\sigma} = \boldsymbol{C}_\rho \to \boldsymbol{C}_\sigma$, the partial continuous functionals of type $\rho\to\sigma$ will correspond to the continuous functions from $|\boldsymbol{C}_\rho|$ to $|\boldsymbol{C}_\sigma|$. It will not be possible to define $\boldsymbol{C}_\rho$ by recursion on the type ρ, since we allow algebras with constructors having function arguments (like $\mathbf{O}$ and Sup). Instead, we shall use recursion on the "height" of the notions involved, defined below.

Definition (Information system of type ρ). We simultaneously define C_ι, $C_{\rho\to\sigma}$, Con_ι and $\mathrm{Con}_{\rho\to\sigma}$.

(a) The *tokens* $a \in C_\iota$ are the type correct constructor expressions $\mathrm{C}a_1^*\dots a_n^*$ where a_i^* is an *extended token*, i.e., a token or the special symbol $*$ which carries no information.

(b) The tokens in $C_{\rho\to\sigma}$ are the pairs (U,b) with $U \in \mathrm{Con}_\rho$ and $b \in C_\sigma$.

(c) A finite set U of tokens in C_ι is *consistent* (i.e., $\in \mathrm{Con}_\iota$) if all its elements start with the same constructor C, say of arity $\tau_1\to\dots\to\tau_n\to\iota$, and all $U_i \in \mathrm{Con}_{\tau_i}$ for $i=1,\dots,n$, where U_i consists of all (proper) tokens at the i-th argument position of some token in $U = \{\mathrm{C}\vec{a_1^*}, \dots, \mathrm{C}\vec{a_m^*}\}$.

(d) $\{\,(U_i,b_i) \mid i\in I\,\} \in \mathrm{Con}_{\rho\to\sigma}$ is defined to mean $\forall_{J\subseteq I}(\bigcup_{j\in J} U_j \in \mathrm{Con}_\rho \to \{\,b_j \mid j\in J\,\} \in \mathrm{Con}_\sigma)$.

Building on this definition, we define $U \vdash_\rho a$ for $U \in \mathrm{Con}_\rho$ and $a \in C_\rho$.

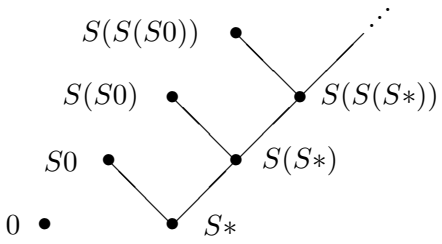


FIGURE 1. Tokens and entailment for **N**

(e) $\{\mathrm{C}\vec{a_1^*}, \ldots, \mathrm{C}\vec{a_m^*}\} \vdash_\iota \mathrm{C}'\vec{a^*}$ is defined to mean $\mathrm{C} = \mathrm{C}'$, $m \geq 1$ and $U_i \vdash a_i^*$, with U_i as in (c) above (and $U \vdash *$ taken to be true).
(f) $W \vdash_{\rho\to\sigma} (U, b)$ is defined to mean $WU \vdash_\sigma b$, where application WU of $W = \{\, (U_i, b_i) \mid i \in I \,\} \in \mathrm{Con}_{\rho\to\sigma}$ to $U \in \mathrm{Con}_\rho$ is defined to be $\{\, b_i \mid U \vdash_\rho U_i \,\}$; recall that $U \vdash V$ abbreviates $\forall_{a\in V}(U \vdash a)$.

If we define the *height* of the syntactic expressions above by

$$|\mathrm{C}a_1^* \ldots a_n^*| := 1 + \max\{\, |a_i^*| \mid i = 1, \ldots, n \,\}, \qquad |*| := 0,$$
$$|(U, b)| := \max\{1 + |U|, 1 + |b|\},$$
$$|\{\, a_i \mid i \in I \,\}| := \max\{\, 1 + |a_i| \mid i \in I \,\},$$
$$|U \vdash a| := \max\{1 + |U|, 1 + |a|\},$$

then these are definitions by recursion on the height.

It is left as an exercise to show that $(C_\rho, \mathrm{Con}_\rho, \vdash_\rho)$ is an information system (by induction on the height, with case distinctions on the type). Observe that all the notions involved are computable: $a \in C_\rho$, $U \in \mathrm{Con}_\rho$ and $U \vdash_\rho a$.

DEFINITION (Partial continuous functionals). For every type ρ let $\boldsymbol{C}_\rho$ be the information system $(C_\rho, \mathrm{Con}_\rho, \vdash_\rho)$. The set $|\boldsymbol{C}_\rho|$ of ideals in $\boldsymbol{C}_\rho$ is the set of *partial continuous functionals* of type ρ. A partial continuous functional $x \in |\boldsymbol{C}_\rho|$ is *computable* if it is recursively enumerable when viewed as a set of tokens.

Notice that we have $\boldsymbol{C}_{\rho\to\sigma} = \boldsymbol{C}_\rho \to \boldsymbol{C}_\sigma$, as defined generally for information systems.

For example, the tokens for the algebra **N** are shown in Figure 1. For tokens a, b we have $\{a\} \vdash b$ if and only if there is a path from a (up) to b (down). As another (more typical) example, consider the algebra **D** of binary trees with a nullary constructor 0 and a binary C. Then $\{\mathrm{C}0*, \mathrm{C}*0\}$ is consistent, and $\{\mathrm{C}0*, \mathrm{C}*0\} \vdash \mathrm{C}00$.

2.1.6. Constructors as continuous functions. Let ι be an algebra. Every constructor C generates the following ideal in the function space:

$$r_{\mathrm{C}} := \{\, (\vec{U}, \mathrm{C}\vec{a^*}\,) \mid \vec{U} \vdash \vec{a^*}\,\}.$$

Here $(\vec{U}, a)$ abbreviates $(U_1, (U_2, \dots (U_n, a) \dots))$.

According to the general definition of a continuous function associated to an ideal in a function space the continuous map $|r_{\mathrm{C}}|$ satisfies

$$|r_{\mathrm{C}}|(\vec{x}\,) = \{\, \mathrm{C}\vec{a^*} \mid \exists_{\vec{U} \subseteq \vec{x}}(\vec{U} \vdash \vec{a^*})\,\}.$$

An immediate consequence is that the (continuous maps corresponding to) constructors are injective and their ranges are disjoint, which is what we wanted to achieve by associating non-flat rather than flat information systems with algebras.

Lemma (Constructors are injective and have disjoint ranges). *Let ι be an algebra and* C *be a constructor of ι. Then*

$$|r_{\mathrm{C}}|(\vec{x}\,) \subseteq |r_{\mathrm{C}}|(\vec{y}\,) \leftrightarrow \vec{x} \subseteq \vec{y}.$$

If $\mathrm{C}_1, \mathrm{C}_2$ are distinct constructors of ι, then $|r_{\mathrm{C}_1}|(\vec{x}\,) \neq |r_{\mathrm{C}_2}|(\vec{y}\,)$, since the two ideals are non-empty and disjoint.

Proof. Immediate from the definitions. □

Remark. Notice that neither property holds for flat information systems, since for them, by monotonicity, constructors need to be *strict* (i.e., if one argument is the empty ideal, then the value is as well). But then we have

$$|r_{\mathrm{C}}|(\overline{\emptyset}, y) = \overline{\emptyset} = |r_{\mathrm{C}}|(x, \overline{\emptyset}),$$
$$|r_{\mathrm{C}_1}|(\overline{\emptyset}) = \overline{\emptyset} = |r_{\mathrm{C}_2}|(\overline{\emptyset})$$

where in the first case we have one binary and, in the second, two unary constructors.

2.1.7. Total and cototal ideals. In the information system $\boldsymbol{C}_\iota$ associated with an algebra ι, the "total" and "cototal" ideals are of special interest.

Recall that a token in ι is a constructor tree P possibly containing the special symbol $*$. Because of the possibility of parameter arguments we need to distinguish between "structure-" and "fully" total and cototal ideals. For the definition it is easiest to refer to a constructor tree $P(*)$ with a distinguished occurrence of $*$. This occurrence is called *non-parametric* if the path from it to the root does not pass through a parameter argument of a constructor. For a constructor tree $P(*)$, an arbitrary $P(\mathrm{C}\vec{a^*})$ is called *one-step extension* of $P(*)$, written $P(\mathrm{C}\vec{a^*}) \succ_1 P(*)$.

DEFINITION. Let ι be an algebra, and $\boldsymbol{C}_\iota$ its associated information system. An ideal $x \in |\boldsymbol{C}_\iota|$ is *cototal* if every constructor tree $P(*) \in x$ has a $\succ_1$-predecessor $P(\mathrm{C}\vec{*}) \in x$; it is called *total* if it is cototal and the relation $\succ_1$ on x is well-founded. It is called *structure-cototal* (*structure-total*) if the same holds with $\succ_1$ defined w.r.t. $P(*)$ with a non-parametric distinguished occurrence of $*$.

If there are no parameter arguments, we shall simply speak of total and cototal ideals. For example, for the algebra $\mathbf{N}$ every total ideal is the deductive closure of a token $S(S\dots(S0)\dots)$, and the set of all tokens $S(S\dots(S*)\dots)$ is a cototal ideal. For the algebra $\mathbf{L}(\mathbf{N})$ of lists of natural numbers the structure-total ideals are the finite lists and the structure-cototal ones the finite or infinite lists. For the algebra $\mathbf{D}$ of derivations the total ideals can be viewed as the finite derivations, and the cototal ones as the finite or infinite "locally correct" derivations of Mints (1978); arbitrary ideals can be viewed as "partial" or "incomplete" derivations, with "holes".

A partial continuous function of a higher type is *total* if it maps total arguments into total values. The definition of cototality for higher types is more involved and will be given only later.

2.2. A term language for computable functionals

To work with computable functionals in a formal theory we need to have a language to denote them. Again lambda calculus is the appropriate tool, this time extended by constants for computable functionals.

Recall that a partial continuous functional is defined to be computable if it is the limit of a recursively enumerable set of finite approximations. We introduce a convenient way to define computable functionals, by means of defining equations or more precisely, computation rules. Therefore we extend the term language by constants D defined by certain "computation rules". The resulting term system can be seen as a common extension of Gödel's T (1958) and Plotkin's PCF; we call it T^+.

2.2.1. A common extension T^+ of Gödel's T and Plotkin's PCF. *Terms* of T^+ are built from (typed) variables and (typed) constants (constructors C or defined constants D) by application and abstraction:

$$M, N ::= x^\rho \mid \mathrm{C}^\rho \mid D^\rho \mid (\lambda_{x^\rho} M^\sigma)^{\rho\to\sigma} \mid (M^{\rho\to\sigma} N^\rho)^\sigma.$$

The set $\mathrm{FV}(M)$ of free variables of a term M is defined by

$$\begin{aligned} \mathrm{FV}(x) &:= \{x\}, & \mathrm{FV}(C) &:= \mathrm{FV}(D) := \emptyset, \\ \mathrm{FV}(\lambda_x M) &:= \mathrm{FV}(M) \setminus \{x\}, & \mathrm{FV}(MN) &:= \mathrm{FV}(M) \cup \mathrm{FV}(N). \end{aligned}$$

We define a conversion relation $\mapsto_\beta$ for terms similarly to what we did for derivation terms:

$$(\lambda_x M(x))^{\rho\to\sigma} N^\rho \mapsto_\beta M(N)^\sigma.$$

In addition we will employ another conversion relation $\mapsto_\eta$ defined by

$$\lambda_x(Mx) \mapsto_\eta M \quad \text{if } x \notin \mathrm{FV}(M) \ (M \text{ not an abstraction}).$$

Definition (Computation rule). Every defined constant D comes with a system of *computation rules*, consisting of finitely many equations

$$D\vec{P}_i(\vec{y}_i) = M_i \qquad (i = 1, \dots, n) \tag{7}$$

with free variables of $\vec{P}_i(\vec{y}_i)$ and M_i among $\vec{y}_i$, where the arguments on the left hand side must be "constructor patterns", i.e., lists of applicative terms built from constructors and distinct variables. To ensure consistency of the defining equations, we require that for $i \neq j$ $\vec{P}_i$ and $\vec{P}_j$ have disjoint free variables, and either $\vec{P}_i$ and $\vec{P}_j$ are non-unifiable (i.e., there is no substitution which identifies them), or else for the most general unifier ϑ of $\vec{P}_i$ and $\vec{P}_j$ we have $M_i\vartheta = M_j\vartheta$. Notice that the substitution ϑ assigns to the variables $\vec{y}_i$ in M_i constructor patterns $\vec{R}_k(\vec{z})$ $(k = i, j)$. A further requirement on a system of computation rules $D\vec{P}_i(\vec{y}_i) = M_i$ is that the lengths of all $\vec{P}_i(\vec{y}_i)$ are the same; this number is called the *arity* of D, denoted by $\mathrm{ar}(D)$. A substitution instance of a left hand side of (7) is called a *D-redex*.

More formally, constructor patterns are defined inductively by (we write $\vec{P}(\vec{x}\,)$ to indicate all variables in $\vec{P}$):

(a) x is a constructor pattern.
(b) The empty list is a constructor pattern.
(c) If $\vec{P}(\vec{x}\,)$ and $Q(\vec{y}\,)$ are constructor patterns whose variables $\vec{x}$ and $\vec{y}$ are disjoint, then $(\vec{P}, Q)(\vec{x}, \vec{y}\,)$ is a constructor pattern.
(d) If C is a constructor and $\vec{P}$ a constructor pattern, then so is $\mathrm{C}\vec{P}$, provided it is of ground type.

Remark. The requirement of disjoint variables in constructor patterns $\vec{P}_i$ and $\vec{P}_j$ used in computation rules of a defined constant D is needed to ensure that applying the most general unifier produces constructor patterns again. However, for readability we take this as an implicit convention, and write computation rules with possibly non-disjoint variables.

Examples of constants D defined by computation rules are abundant. In particular, the (structural) recursion and corecursion operators will be defined by computation rules.

The boolean connectives andb, impb and orb are defined by

$$\begin{array}{lll} \mathtt{tt}\ \mathrm{andb}\ y = y, & \mathtt{ff}\ \mathrm{impb}\ y = \mathtt{tt}, & \mathtt{tt}\ \mathrm{orb}\ y = \mathtt{tt}, \\ x\ \mathrm{andb}\ \mathtt{tt} = x, & \mathtt{tt}\ \mathrm{impb}\ y = y, & x\ \mathrm{orb}\ \mathtt{tt} = \mathtt{tt}, \\ \mathtt{ff}\ \mathrm{andb}\ y = \mathtt{ff}, & x\ \mathrm{impb}\ \mathtt{tt} = \mathtt{tt}, & \mathtt{ff}\ \mathrm{orb}\ y = y, \\ x\ \mathrm{andb}\ \mathtt{ff} = \mathtt{ff}, & & x\ \mathrm{orb}\ \mathtt{ff} = x. \end{array}$$

Notice that when two such rules overlap, their right hand sides are equal under any unifier of the left hand sides.

Decidable *equality* $=_\iota \colon \iota \to \iota \to \mathbf{B}$ for a finitary algebra ι can be defined easily by computation rules. For example,

$$\begin{array}{ll} (0 =_{\mathbf{N}} 0) = \mathtt{tt}, & (Sn =_{\mathbf{N}} 0) = \mathtt{ff}, \\ (0 =_{\mathbf{N}} Sm) = \mathtt{ff}, & (Sn =_{\mathbf{N}} Sm) = (n =_{\mathbf{N}} m). \end{array}$$

For the algebra $\mathbf{D}$ of binary trees with constructors 0 (leaf) and C (construct a new tree from two given ones) we have

$$\begin{array}{ll} (0 =_{\mathbf{D}} 0) = \mathtt{tt}, & (\mathrm{C}ts =_{\mathbf{D}} 0) = \mathtt{ff}, \\ (0 =_{\mathbf{D}} \mathrm{C}ts) = \mathtt{ff}, & (\mathrm{C}ts =_{\mathbf{D}} \mathrm{C}t's') = (t =_{\mathbf{D}} t'\ \mathrm{andb}\ s =_{\mathbf{D}} s'). \end{array}$$

For the algebra $\mathbf{N}$ of natural numbers we have the doubling function

$$\begin{aligned} \mathrm{Double}(0) &:= 0, \\ \mathrm{Double}(S(n)) &:= S(S(\mathrm{Double}(n))). \end{aligned}$$

Addition (written infix) is defined similarly, this time with a parameter m:

$$\begin{aligned} m + 0 &:= m, \\ m + S(n) &:= S(m + n). \end{aligned}$$

Multiplication (again written infix) is defined using addition by

$$\begin{aligned} m \cdot 0 &:= 0, \\ m \cdot S(n) &:= (m \cdot n) + m. \end{aligned}$$

Similarly we can define all primitive recursive functions.

2.2.2. Structural recursion operators. Important examples of such constants D are the (structural) higher type *recursion operators* $\mathcal{R}^\tau_\iota$ introduced by Hilbert (1925) and Gödel (1958). They are used to construct maps from the algebra ι to the value type τ, by recursion on the structure of ι.

For instance, $\mathcal{R}^\tau_{\mathbf{N}}$ has type $\mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau$. It is defined by the computation rules

$$\begin{aligned} \mathcal{R}^\tau_{\mathbf{N}}(0, a, f) &= a, \\ \mathcal{R}^\tau_{\mathbf{N}}(S(n), a, f) &= f(n, \mathcal{R}^\tau_{\mathbf{N}}(n, a, f)). \end{aligned}$$

The first argument is the recursion argument, the second one gives the base value, and the third one gives the step function, mapping the recursion argument and the previous value to the next value. For example, $\mathcal{R}^{\mathbf{N}}_{\mathbf{N}} nm\lambda_{n,p}(Sp)$ defines addition $m+n$ by recursion on n. For $\lambda_{n,p}(Sp)$ we often write $\lambda_{\_,p}(Sp)$ since the bound variable n is not used.

Generally, we define the type of the recursion operator $\mathcal{R}^\tau_\iota$ for the algebra $\iota = \mu_\xi((\rho_{i\nu}(\xi))_{\nu<n_i} \to \xi)_{i<k}$ and result type τ to be

$$\iota \to ((\rho_{i\nu}(\iota \times \tau))_{\nu<n_i} \to \tau)_{i<k} \to \tau.$$

Here ι is the type of the recursion argument, and each $(\rho_{i\nu}(\iota\times\tau))_{\nu<n_i} \to \tau$ is called a *step type*. Usage of $\iota \times \tau$ rather than τ in the step types can be seen as a "strengthening", since then one has more data available to construct the value of type τ. Moreover, for (unnested) recursive argument types $\vec\sigma \to \tau$ we avoid the product type in $\vec\sigma \to \iota \times \tau$ and take the two argument types $\vec\sigma \to \iota$ and $\vec\sigma \to \tau$ instead ("duplication").

To define computation rules of $\mathcal{R}^\tau_\iota$ we use the following notation. Let

$$\rho_0 \to \ldots \to \rho_{m-1} \to (\vec\sigma_0 \to \xi) \to \ldots \to (\vec\sigma_{n-1} \to \xi) \to \xi,$$

be the type of the i-th constructor C_i of ι and consider a term $\mathrm{C}_i\vec N$ of type ι. We write $\vec N^P = N^P_0, \ldots, N^P_{m-1}$ for the *parameter arguments* $N^{\rho_0}_0, \ldots, N^{\rho_{m-1}}_{m-1}$ and $\vec N^R = N^R_0, \ldots, N^R_{n-1}$ for the *recursive arguments* $N^{\vec\sigma_0\to\iota}_m, \ldots, N^{\vec\sigma_{n-1}\to\iota}_{m+n-1}$. Writing $\mathcal{R}$ for $\mathcal{R}^\tau_\iota$ we take as its computation rules

$$\mathcal{R}(\mathrm{C}_i\vec N)\vec M = M_i\vec N\lambda_{\vec x}(\mathcal{R}(N^R_0\vec x)\vec M)\ldots\lambda_{\vec x}(\mathcal{R}(N^R_{n-1}\vec x)\vec M)$$

For some algebras we spell out the type of their recursion operators:

$\mathcal{R}^\tau_{\mathbf{B}}\colon \mathbf{B} \to \tau \to \tau \to \tau,$
$\mathcal{R}^\tau_{\mathbf{N}}\colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\mathbf{P}}\colon \mathbf{P} \to \tau \to (\mathbf{P} \to \tau \to \tau) \to (\mathbf{P} \to \tau \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\mathbf{D}}\colon \mathbf{D} \to \tau \to (\mathbf{D} \to \tau \to \mathbf{D} \to \tau \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\mathbf{O}}\colon \mathbf{O} \to \tau \to (\mathbf{O} \to \tau \to \tau) \to ((\mathbf{N} \to \mathbf{O}) \to (\mathbf{N} \to \tau) \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\mathbf{L}(\rho)}\colon \mathbf{L}(\rho) \to \tau \to (\rho \to \mathbf{L}(\rho) \to \tau \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\rho+\sigma}\colon \rho + \sigma \to (\rho \to \tau) \to (\sigma \to \tau) \to \tau,$
$\mathcal{R}^\tau_{\rho\times\sigma}\colon \rho \times \sigma \to (\rho \to \sigma \to \tau) \to \tau.$

It is a helpful exercise to write out the computation rules for these particular recursion operators.

There is an important variant of recursion, where no recursive calls occur. This variant is called the *cases operator*; it distinguishes cases according

to the outer constructor form. For the algebra $\iota = \mu_\xi((\rho_{i\nu}(\xi))_{\nu<n_i} \to \xi)_{i<k}$ and result type τ the type of the cases operator $\mathcal{C}^\tau_\iota$ is

$$\iota \to ((\rho_{i\nu}(\iota))_{\nu<n_i} \to \tau)_{i<k} \to \tau.$$

The simplest example (for type $\mathbf{B}$) is *if-then-else*. Another example is

$$\mathcal{C}^\tau_{\mathbf{N}} \colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau) \to \tau.$$

It can be used to define the *predecessor* function on $\mathbf{N}$, i.e., $\mathrm{P}0 := 0$ and $\mathrm{P}(Sn) := n$, by the term

$$\mathrm{P}m := \mathcal{C}^{\mathbf{N}}_{\mathbf{N}} m 0(\lambda_n n).$$

REMARK. When computing the value of a cases term, we do not want to (eagerly) evaluate all arguments, but rather compute the test argument first and depending on the result (lazily) evaluate at most one of the other arguments. This phenomenon is well known in functional languages; for instance, in SCHEME the `if`-construct is called a *special form* (as opposed to an operator). Therefore instead of taking the cases operator applied to a full list of arguments, one rather uses a `case`-construct to build this term; it differs from the former only in that it employs lazy evaluation. Hence the predecessor function is written in the form

$$[\mathbf{case}\ m^{\mathbf{N}}\ \mathbf{of}\ (0 \mapsto 0 \mid Sn \mapsto n)].$$

General recursion with respect to a measure. In practice it often happens that one needs to recur to an argument which is not an immediate component of the present constructor object; this is not allowed in structural recursion. Of course, in order to ensure that the recursion terminates we have to assume that the recurrence is w.r.t. a given well-founded set; for simplicity we restrict ourselves to the algebra $\mathbf{N}$. However, we do allow that the recurrence is with respect to a measure function μ, with values in $\mathbf{N}$. The operator $\mathcal{F}$ of *general recursion* then is defined by

$$\mathcal{F}\mu x G = Gx(\lambda_y[\mathbf{if}\ \mu y < \mu x\ \mathbf{then}\ \mathcal{F}\mu y G\ \mathbf{else}\ \varepsilon]), \tag{8}$$

where ε denotes a canonical inhabitant of the range. We leave it as an exercise to prove that $\mathcal{F}$ is definable from an appropriate structural recursion operator.

As an example for the use of $\mathcal{F}$ we define a function NatToPos converting a natural number ≥ 1 written in unary (i.e., built from the constructors 0 and S) into the same natural number written in binary (i.e., built from the constructors 1, S_0 and S_1). This uses the auxiliary functions $\mathrm{Even} \colon \mathbf{N} \to \mathbf{B}$ defined by

$$\mathrm{Even}(0) = \mathtt{tt},$$
$$\mathrm{Even}(S(0)) = \mathtt{ff},$$

$$\mathrm{Even}(S(S(n))) = \mathrm{Even}(n)$$

and $\mathrm{Half}\colon \mathbf{N} \to \mathbf{N}$ defined by

$$\begin{aligned} \mathrm{Half}(0) &= 0, \\ \mathrm{Half}(S(0)) &= 0, \\ \mathrm{Half}(S(S(n))) &= S(\mathrm{Half}(n)). \end{aligned}$$

Then $\mathrm{NatToPos}\colon \mathbf{N} \to \mathbf{P}$ is defined by

$$\mathrm{NatToPos}(n) = \mathcal{F}(\mathrm{id}, n, G)$$

with $\mathrm{id}(n) = n$ and $G\colon \mathbf{N} \to (\mathbf{N} \to \mathbf{P}) \to \mathbf{P}$ defined by

$$G(n, f) = \begin{cases} S_0(f(\mathrm{Half}(n))) & \text{if } \mathrm{Even}(n), \\ 1 & \text{if } n = S(0), \\ S_1(f(\mathrm{Half}(n))) & \text{otherwise.} \end{cases}$$

2.2.3. Corecursion. Up to now we have only considered examples of total functions, in the sense of Section 2.1.7. But recall that in our setting functions need not be total. In fact, for functions operating on (cototal) "stream" representations of real numbers totality plays no role.

To consider an example we first need to define some additional algebras.

$$\begin{aligned} \mathbf{Sd} &:= \mu_\xi(\xi, \xi, \xi) \qquad \text{(signed digits } -1, 0, 1), \\ \mathbf{Sd}_2 &:= \mu_\xi(\xi, \xi, \xi, \xi, \xi) \qquad \text{(extended signed digits } -2, -1, 0, 1, 2), \\ \mathbf{I} &:= \mu_\xi(\mathbf{Sd} \to \xi \to \xi). \end{aligned}$$

$\mathbf{I}$ is a variant of $\mathbf{L}(\mathbf{Sd})$ with only its binary constructor C kept, but without a nullary constructor. Its cototal elements are viewed as representations of real numbers by streams of signed digits.

The average of two such stream-represented reals can be obtained as follows. Let $f_{\mathrm{init}}\colon \mathbf{I} \to \mathbf{I} \to \mathbf{Sd}_2 \times \mathbf{I} \times \mathbf{I}$ be defined by

$$f_{\mathrm{init}}(\mathrm{C}_d(u), \mathrm{C}_e(v)) = (d + e, u, v),$$

and $f\colon \mathbf{Sd}_2 \times \mathbf{I} \times \mathbf{I} \to \mathbf{I}$ by

$$f(i, \mathrm{C}_d(v), \mathrm{C}_e(w)) = \mathrm{C}_{K(d+e+2i)}(f(J(d + e + 2i), v, w)).$$

Here we have used functions $J, K\colon \mathbb{Z} \to \mathbb{Z}$ such that for all i

$$\begin{aligned} & i = J(i) + 4K(i), \\ & |J(i)| \le 2, \\ & |i| \le 6 \to |K(i)| \le 1. \end{aligned}$$

We view f_{init} as computing the first "carry" and the tails of the inputs. Then $f\colon \mathbf{Sd}_2 \times \mathbf{I} \times \mathbf{I} \to \mathbf{I}$ is called repeatedly, computing the average step by

step. Note that the defining equation for f when viewed as a rewrite rule does not terminate. It is however a correct computation rule for f.

One can show that an arbitrary "reduction sequence" beginning with a term involving only constructors C and recursion and cases operators $\mathcal{R}$, $\mathcal{C}$ terminates. It follows that every such closed term denotes a total ideal.

The computation rules for $\mathcal{R}$ work from the leaves towards the root, and terminate because total ideals are well-founded. If, however, we deal with cototal ideals (infinitary derivations, for example), then a similar operator is available to define functions with cototal ideals as values, namely "corecursion".

To understand the type of a corecursion operator recall the constructor types $\kappa_i(\iota)$ of an algebra $\iota = \mu_\xi(\kappa_0, \dots, \kappa_{k-1})$:

$$(\rho_{i\nu}(\iota))_{\nu<n_i} \to \iota \quad (i < k).$$

The product of these k constructor types is isomorphic to

$$\sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota) \to \iota$$

and the type of the recursion operator $\mathcal{R}^\tau_\iota$ is isomorphic to

$$\iota \to (\sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota \times \tau) \to \tau) \to \tau.$$

Dually for the algebra ι the type of its *destructor* D_ι (disassembling a constructor-built object into its parts) is

$$\iota \to \sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota).$$

The corecursion operator ${}^{\mathrm{co}}\mathcal{R}^\tau_\iota$ is used to construct a mapping from τ to ι by "corecursion" on the structure of ι. Its type is

$$\tau \to (\tau \to \sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota + \tau)) \to \iota. \tag{9}$$

We list the types of the corecursion operators for some algebras:

$$\begin{aligned}
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{B}} \colon \tau \to (\tau \to \mathbf{U} + \mathbf{U}) \to \mathbf{B},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{N}} \colon \tau \to (\tau \to \mathbf{U} + (\mathbf{N} + \tau)) \to \mathbf{N},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{P}} \colon \tau \to (\tau \to \mathbf{U} + (\mathbf{P} + \tau) + (\mathbf{P} + \tau)) \to \mathbf{P},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{D}} \colon \tau \to (\tau \to \mathbf{U} + (\mathbf{D} + \tau) \times (\mathbf{D} + \tau)) \to \mathbf{D},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{L}(\rho)} \colon \tau \to (\tau \to \mathbf{U} + \rho \times (\mathbf{L}(\rho) + \tau)) \to \mathbf{L}(\rho).
\end{aligned}$$

The computation rule for each of these is defined below. For $f \colon \rho \to \tau$ and $g \colon \sigma \to \tau$ we denote $\lambda_x(\mathcal{R}^\tau_{\rho+\sigma} x f g)$ of type $\rho + \sigma \to \tau$ by $[f, g]$, and similary

for ternary sumtypes etcetera. The identity functions id below are of type $\iota \to \iota$ with ι the respective algebra.

$$\begin{aligned}
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{B}} NM &= [\lambda_\mathtt{tt}, \lambda_\mathtt{ff}](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{N}} NM &= [\lambda_0, \lambda_x(S([\mathrm{id}^{\mathbf{N}\to\mathbf{N}}, P_{\mathbf{N}}]))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{P}} NM &= [\lambda_1, \lambda_x(S_0([\mathrm{id}, P_{\mathbf{P}}]x)), \lambda_x(S_1([\mathrm{id}, P_{\mathbf{P}}]x))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{D}} NM &= [\lambda_0, \lambda_{x_0,x_1}(\mathrm{C}([\mathrm{id}, P_{\mathbf{D}}]x_0)([\mathrm{id}, P_{\mathbf{D}}]x_1))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{L}(\rho)} NM &= [\lambda_[], \lambda_{x_0,x_1}(x_0 :: [\mathrm{id}, P_{\mathbf{L}(\rho)}]x_1)](MN)
\end{aligned}$$

with $P_\alpha := \lambda_y({}^{\mathrm{co}}\mathcal{R}^\tau_\alpha yM)$ for $\alpha \in \{\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{L}(\rho)\}$.

Generally, we define the (single) computation rule for ${}^{\mathrm{co}}\mathcal{R}^\tau_\iota$ of type (9) by

$${}^{\mathrm{co}}\mathcal{R}^\tau_\iota NM = [L_0, \dots, L_{k-1}](MN)$$

where L_i of type $\prod_{\nu<n_i} \rho_{i\nu}(\iota+\tau) \to \iota$ is defined as

$$L_i := \lambda_{\vec{x}}(\mathrm{C}_i(N_\nu)_{\nu<n_i}) \qquad \text{with } x_\nu \colon \rho_{i\nu}(\iota+\tau),$$

$$N_\nu := \begin{cases} x_\nu & \text{if } \rho_{i\nu}(\xi) \text{ is a parameter arg. type,}\\ \lambda_{\vec{z}_\nu}([\mathrm{id}^{\iota\to\iota}, P^{\tau\to\iota}](x_\nu \vec{z}_\nu)^{\iota+\tau}) & \text{otherwise,}\end{cases}$$

where $P := \lambda_y({}^{\mathrm{co}}\mathcal{R}^\tau_\iota yM)$ contains the corecursive call. For simplicity we have assumed that recursive argument types $\rho_{i\nu}(\xi)$ are of the form $\vec{\sigma}_\nu \to \xi$.

2.3. Denotational semantics

How can we use computation rules to define an ideal z in a function space? The general idea is to inductively define the set of tokens (U, a) that make up z. It is convenient to define the value $[\![\lambda_{\vec{x}} M]\!]$, where M is a term with free variables among $\vec{x}$. Since this value is a token set, we can define inductively the relation $(\vec{U}, a) \in [\![\lambda_{\vec{x}} M]\!]$.

For a constructor pattern $\vec{P}(\vec{x})$ and a list $\vec{V}$ of the same length and types as $\vec{x}$ we define a list $\vec{P}(\vec{V})$ of formal neighborhoods of the same length and types as $\vec{P}(\vec{x})$, by induction on $\vec{P}(\vec{x})$. $x(V)$ is the singleton list V, and for $\langle\rangle$ we take the empty list. $(\vec{P}, Q)(\vec{V}, \vec{W})$ is covered by the induction hypothesis. Finally

$$(\mathrm{C}\vec{P})(\vec{V}) := \{\, \mathrm{C}\vec{a^*} \mid a^*_i \in P_i(\vec{V_i}) \text{ if } P_i(\vec{V_i}) \neq \emptyset, \text{ and } a^*_i = * \text{ otherwise} \,\}.$$

We use the following notation. $(\vec{U}, a)$ means $(U_1, (U_2, \dots (U_n, a)) \dots)$, and $(\vec{U}, V) \subseteq [\![\lambda_{\vec{x}} M]\!]$ means $(\vec{U}, a) \in [\![\lambda_{\vec{x}} M]\!]$ for all (finitely many) $a \in V$.

Definition (Inductive, of $(\vec{U}, a) \in [\![\lambda_{\vec{x}} M]\!]$).

$$\frac{U_i \vdash a}{(\vec{U}, a) \in [\![\lambda_{\vec{x}} x_i]\!]}(V), \qquad \frac{(\vec{U}, V, a) \in [\![\lambda_{\vec{x}} M]\!] \qquad (\vec{U}, V) \subseteq [\![\lambda_{\vec{x}} N]\!]}{(\vec{U}, a) \in [\![\lambda_{\vec{x}}(MN)]\!]}(A).$$

For every constructor C and defined constant D we have

$$\frac{\vec{V} \vdash \vec{a^*}}{(\vec{U}, \vec{V}, \mathrm{C}\vec{a^*}) \in [\![\lambda_{\vec{x}} \mathrm{C}]\!]}(\mathrm{C}), \qquad \frac{(\vec{U}, \vec{V}, a) \in [\![\lambda_{\vec{x},\vec{y}} M]\!] \qquad \vec{W} \vdash \vec{P}(\vec{V})}{(\vec{U}, \vec{W}, a) \in [\![\lambda_{\vec{x}} D]\!]}(D)$$

with one such rule (D) for every computation rule $D\vec{P}(\vec{y}) = M$.

This "denotational semantics" has good properties; however, we do not carry out the proofs here (cf. Appendix A or Schwichtenberg and Wainer (2012)). First of all, one can prove that $[\![\lambda_{\vec{x}} M]\!]$ is an ideal . Moreover, our definition above of the denotation of a term is reasonable in the sense that it is not changed by an application of the standard (β- and η-) conversions or a computation rule.