Towards a Unifying View on Monotone Constructive Definitions *

Linde Vanbesien[1 [0000-0003-0700-7099]], Samuele Pollaci[2 [0000-0002-2914-787X]], Bart Bogaerts[2 [0000-0003-3460-4251]], and Marc Denecker[1 [0000-0002-0422-7339]]

[1] Department Of Computer Science, KU Leuven, Belgium
{linde.vanbesien,marc.denecker}@kuleuven.be
[2] Department Of Computer Science, Vrije Universiteit Brussel, Belgium
{samuele.pollaci,bart.bogaerts}@vub.be

Abstract. Constructive definitions, including inductive and recursive definitions, are ubiquitous in mathematical texts and occur in a wide variety of computer science fields and Knowledge Representation applications. While in different areas there is a high level of familiarity with certain types of constructive definitions, fairly little interaction between different areas seems to exist, resulting in a lack of deep understanding of principles and their applications. This paper aims to fill this void by laying the foundations for a single unifying framework, bringing together a wide variety of definitions. First, we recall the principle of (monotone) inductive definition and its formalization in fixpoint theory. We discuss the constructive and the non-constructive interpretation of inductive definitions and the induction process. We then analyze examples, including but not limited to (co)inductive and (co)recursive definitions, found in a wide range of areas through the lens of our proposed framework.

Keywords: Constructive definitions · Construction space · Induction process.

Introduction

In mathematics, there are perhaps few concepts so enigmatic as that of inductive or recursive definitions.[3] Students become familiar with them through examples such as the transitive closure of a binary relation, the Fibonacci function or the satisfaction relation of propositional or predicate logic.

Common to such definitions is that they define a concept by describing how to construct it through iterated application of rules, starting from the empty set. This construction process is often called the *induction process*. For definitions of sets, the defined set is often explained non-constructively as the least set satisfying the rules; the constructive and non-constructive ways are known to be equivalent. While usually, it is not formally explained what inductive definitions mean, students apparently learn to

* This work was supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G0B2221N).

[3] Is there a difference between *inductive* and *recursive* definition? According to some there is, according to others not. In this paper, we propose a way to distinguish *inductive* and *recursive* definitions that is sensible and seems to match with intuitions of some.

understand them and reason with them. Brouwer [6], the famous constructionist, observed that many fundamental objects in mathematics were defined by describing how to *construct* them and that in understanding these constructions, we rely on our basic cognitive skills for *temporal reasoning*.[4] It was later argued that also our skills for *causal reasoning* play a role here [10, 13, 15]: the hypothesis is that our understanding and reasoning capabilities for inductive definitions stem from our understanding of the induction process as a causal process, idealized and generalized to an (often infinite) universe of mathematical objects. This suggests a strong, but not well-known link between mathematics and common sense knowledge. While recently a lot of effort in the domain of large language models has resulted in surprisingly good commonsense reasoners, it is well-known that these are not reliable enough for sensitive applications where exactness and correctness are crucial. For these applications a logic-based approach that includes different constructive definitions is desired. Therefore the study of the principle of inductive definition is a worthy topic in Knowledge Representation (KR).

It is clear that the concept of inductive definition plays an important role in mathematics and foundations of computer science. We claim it also plays an important role in KR, at the meta-level (e.g., in the many inductive definitions used to define syntax and semantics for logics in KR), and at the object-level, since definitions constitute an important, common and precise form of human knowledge. In an important class of applications, the definition is inductive, in which case it is often not expressible in first-order logic (FO), yielding a second reason for studying inductive definitions [10]. A third reason is the intuition of some researchers that inductive and recursive definitions form the declarative understanding of two well-known declarative programming paradigms, logic and functional programming [11, 19]. Finally, due to the close connection between inductive definitions and causal information, studying inductive definitions is useful for expressing common sense causal knowledge [10, 13].

There exists extensive research on inductive definitions [33, 18, 23, 24, 2]. Also (co-) recursive definitions have received a lot of attention in relation to functional programming languages [26, 32, 27], as well as in *domain theory* where the functions, and the domain they are defined on are defined simultaneously [29, 1]. While there is a high level of familiarity with certain types of constructive definitions, in the current state of the art, fairly little interaction between research on different types of definitions seems to exist, resulting in a lack of deep understanding of common principles and applications. Many researchers seem aware that their theories only cover part of the topic. Already a long time ago, Moschovakis [24] explained how Kleene [21] in early papers had consciously studied constructive definitions[5] but *explicitly had drawn back from studying all of them*. Another complicating factor has been that inductive/recursive definitions have often been studied from a recursion-theoretic point of view, as programs to compute truth or function values, rather than as plain definitions of a concept.

[4] While we follow Brouwer in his views on the nature and importance of constructive definitions, we use standard mathematics and set theory (also in this paper) whenever suitable.

[5] In his work, Kleene used the term *inductive definitions* to denote the overarching class which we call constructive definitions.

This paper contributes to the study of monotone constructive definitions by introducing some key concepts as the foundations for a unifying set-theoretical framework. This offers the key insight that all these different types are instances of the same basic constructive principles. This has important practical implications. Firstly, it entails that research on a particular type of definition might transcend its class and actually be applicable to all constructive definitions. E.g., *non-monotone* inductive definitions have been researched algebraically [14], but non-monotone recursive definitions remain uncharted territory. Our correspondence suggests a way to generalise the study of non-monotonic definitions to other types of constructive definitions. Secondly, we believe this framework will be instrumental to integrate different types of definitions in a single knowledge representation language. The main contribution of this paper is to show how a whole range of examples from different areas can be reduced to instantiations of the same fundamental principles, using standard set-theoretic constructions. First, we recall the principle of (monotone) inductive definition and its formalization in fixpoint theory, which will involve a *semantic operator* on a so-called *construction space*, which is often richer than the *exact space*, in which the *defined object* naturally lives. We then analyze examples found in a wide range of areas. In each example, we describe the exact space, the construction space, the monotone semantic operator and the defined entity. We will see how the construction space can be used as the key factor to distinguish between classes of definitions from different research areas. We focus mostly on (co)inductive definitions of sets and (co)recursive definitions of functions, but also briefly discuss some more complex types of constructive definitions.

Algebraic Formalisation

We now introduce the algebraic formalism needed for an in-depth presentation of various types of constructive definitions and illustrate them with a first detailed example.

A *partially ordered set (poset)* $\langle C, \leq \rangle$ is a set C equipped with a partial order $\leq$, i.e., a reflexive, antisymmetric, transitive relation. When $\leq$ is clear from the context, we sometimes just write C to refer to $\langle C, \leq \rangle$. As usual, we write $x < y$ for $x \leq y \wedge x \neq y$. If S is a subset of C, then x is an *upper bound* of S if $s \leq x$ for each $s \in S$; it is a *least upper bound* ($\mathrm{lub}(S)$) of S if moreover it is smaller than every other upper bound. We call a poset $\langle C, \leq \rangle$ a *chain-complete partial order (cpo)* if every chain of C (i.e., every subset of C for which $\leq$ is total) has a least upper bound. Each cpo has a least element $\bot$, which is the least upper bound of $\emptyset$.

A function $f\colon C_1 \to C_2$ between cpo's is *monotone* if for all $x, y \in C_1$ such that $x \leq_1 y$, it holds that $f(x) \leq_2 f(y)$. We refer to functions $O\colon C \to C$ with domain equal to the codomain as *operators*. An element $x \in C$ is a *prefixpoint*, resp. a *fixpoint* of O if $O(x) \leq x$, resp. $O(x) = x$ [31]. By Tarski's least fixpoint theorem [34], every monotone operator O on a cpo has a least fixpoint, that we denote $\mathrm{lfp}(O)$. It is also the least prefixpoint of O and it can be *constructed* as the limit of the possibly transfinite sequence $(O^i)_{i \geq 0}$, where $O^{i+1} = O(O^i)$ and $O^\lambda = \mathrm{lub}(\{O^j \mid j < \lambda\})$ for limit ordinals λ (in particular, this means $O^0 = \bot$). This allows for a first, algebraic formalization of constructive definitions [2]. A constructive definition for a concept $\mathcal{D}$ is (formalized as) an operator $O : \mathcal{C} \to \mathcal{C}$ on a cpo $\mathcal{C}$. It defines the object D representing $\mathcal{D}$ by describing how to construct it. The construction, normally called the *induction process*, is the sequence $(O^i)_{i\geq 0}$. The defined object D is the limit of this sequence. This limit can be obtained by construction but it can also be characterized non-constructively, as the least (pre)fixpoint of O, yielding the duality between the constructive and non-constructive view on inductive definitions.

Let us illustrate this abstract formalization of constructive definitions on a prototypical example. To streamline the presentation of various examples, we initially present a constructive definition as a set $\mathcal{R}$ of rules[6] which resemble the style used in logic programming, as well as in functional programming. We believe this will lead to an improved understanding of our examples. Moreover, it gives an idea of how constructive definitions in natural language can be formalised, which is essential when developing knowledge representation languages that include them.

Example 1 (Transitive closure). Let $\mathcal{G} = (V, E)$ be a directed graph. The set F of edges of the *transitive closure* $\mathcal{T} = (V, F)$ of $\mathcal{G}$ is defined inductively:
- $(x, y) \in F$ if $(x, y) \in E$;
- $(x, y) \in F$ if there exists a vertex z such that $(x, z) \in F$ and $(z, y) \in F$.

The set of rules $\mathcal{R}_F$ defining $\mathcal{T} = (V, F)$ is as follows.

$$\left\lfloor \begin{array}{l} \forall x \forall y : F(x, y) \leftarrow E(x, y). \\ \forall x \forall y : F(x, y) \leftarrow \exists z : F(x, z) \land F(z, y). \end{array} \right\rfloor$$

We expect this definition to construct a set $F \subseteq V^2$ of edges. Hence, we consider the cpo $\mathcal{C}_F = \langle 2^{V^2}, \subseteq \rangle$, with the power set of V^2 as underlying set with subsetorder. Moreover, the rules in $\mathcal{R}_F$ suggest an operator $O_F : \mathcal{C}_F \rightarrow \mathcal{C}_F$ showcasing rule application, by mapping a set $S \in \mathcal{C}_F$ to

$$O_F(S) = E \cup \{(x, y) \mid (x, z), (z, y) \in S \text{ for some } z \in V\}.$$

It is not hard to prove that O_F is monotoneand that its least fixpoint is the set F of edges of the transitive closure of $\mathcal{G}$.

Proposition 1 *O_F is a monotone operator.*

Proof. Let $S_1 \subseteq S_2$ be two subsets of V^2. We have to show that $O_F(S_1) \subseteq O_F(S_2)$. By definition of O_F, we have that $O_F(S_1) = E \cup \{(x, y) \mid \exists z : (x, z) \in S_1 \land (z, y) \in S_1\}$. Let $(x, y) \in O_F(S_1)$. If $(x, y) \in E$, then $(x, y) \in O_F(S_2)$. If $(x, y) \notin E$, then there exists $z \in V$ such that $(x, z), (z, y) \in S_1$. Since $S_1 \subseteq S_2$, $(x, y) \in O_F(S_2)$, as desired.

In other words, O_F formalizes the constructive definition of F. The defined set F can be characterised non-constructively as the least fixpoint of O_F, or constructively as the limit of the induction process, i.e., the sequence built by iterative application of O_F starting from the empty set.

Denecker, et al. [15] remarked that given the informal rules of Example 1, we most likely picture the induction process as a sequence of applications of rule instances,

[6] We use different brackets to indicate the kind of definition: inductive and recursive definitions will be enclosed in floor-brackets $\lfloor \mathcal{R} \rfloor$, coinductive and corecursive definitions in ceil-brackets $\lceil \mathcal{R} \rceil$, and any other kind of constructive definition in curly brackets $\{\mathcal{R}\}$.

rather than iterations of O_F. In this view of the induction process, the elementary step is the application of a rule instance (or perhaps more generally, the application of a set of rule instances). This natural view of the induction process raises two issues. First, it identifies the *rule* as the modular unit of the definition and its induction process. This modularity is abstracted away when formalizing the definition as an operator O. Second, it leads to a highly non-deterministic notion of induction process, since rules can be applied in different orders. This non-determinism is of great pragmatical use when reasoning on the definition, since it allows us to *steer* the induction process towards a particular goal, e.g., towards computing whether a specific pair (a, b) belongs to F. On the other hand, the non-determinism raises the question whether all these different induction processes are *confluent* (i.e., have the same limit). This should be the case, otherwise the definition would be ambiguous!

In Fig. 1, the start of two such induction processes for Example 1 are visualized. In the top sequence $(F_0, \ldots, F_3)$, all applicable rules are applied at every step of the construction, making it the fastest process. This corresponds to $(O_F^0, \ldots, O_F^3)$, the first four iterations of the operator O_F. In the bottom sequence, a slower induction process $(F'_0, \ldots, F'_3)$ is shown, one that first applies all instances of the base rule, then a single instance of the transitivity rule per iteration.

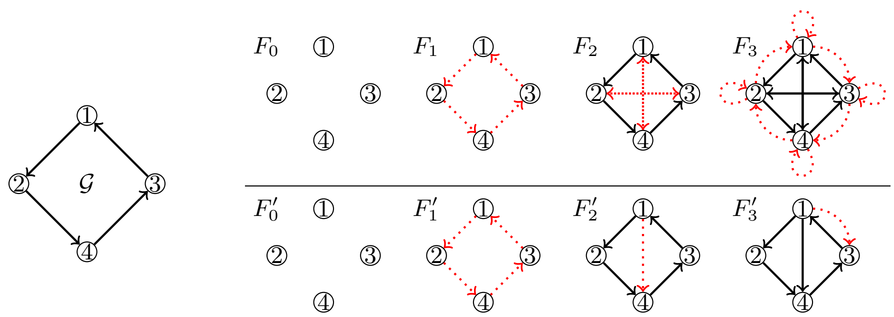

Fig. 1. A graph $\mathcal{G}$ (left) and the start of two monotone inductions of the definition of its transitive closure (right). Dotted red arrows indicate newly derived edges at each state.

This more natural approach is formalized as a *monotone induction* of O: an increasing sequence $(x_i)_{i \leq \beta}$ satisfying

- $x_i \leq x_{i+1} \leq O(x_i)$, for successor ordinals $i + 1 \leq \beta$,
- $x_\lambda = \text{lub}(\{x_i \mid i < \lambda\})$, for limit ordinals $\lambda \leq \beta$ (in particular, $O^0 = \bot$).

Here, $x_i \leq x_{i+1} \leq O(x_i)$ formalizes the idea of applying *some* rule instances in x_i, but not necessarily all. We say a monotone induction of O is *terminal* if there does not exist a strictly greater refinement of its limit, i.e., if $x_\beta \not< O(x_\beta)$. It is straightforward to prove that all terminal monotone inductions are confluent (see, e.g., [5, Corollary 3.7]).

In the next section, we present several examples of constructive definitions. While they originate from very different fields, they can be presented in a uniform way using the following mathematical objects:

- A mathematical object D corresponding to the concept defined by the constructive definition. We call D the *defined object*.
- A set $\mathcal{E}$ where D lives. This should be naturally identified by the specifications in the constructive definition. We call $\mathcal{E}$ the *exact space*.

- A cpo $\mathcal{C} = \langle C, \leq \rangle$ with an injection $\theta : \mathcal{E} \hookrightarrow 2^C \setminus \{\emptyset\}$ such that for all $e_1, e_2 \in \mathcal{E}$ with $e_1 \neq e_2$, $\theta(e_1) \cap \theta(e_2) = \emptyset$, i.e., different elements of the exact space are mapped to disjoint subsets of C. The elements of $\theta(e)$ are (potentially different) representations of $e \in \mathcal{E}$ in $\mathcal{C}$. We call $\mathcal{C}$ the *construction space*.
- An operator $O\colon \mathcal{C} \to \mathcal{C}$ on the construction space, of which the least fixpoint coincides with the defined object D: $\mathrm{lfp}(O) \in \theta(D)$. We call O the *semantic operator*.

The elements of the construction space are meant to approximate the elements of the exact space. Some, or all, elements $c \in \mathcal{C}$ are representations of exact elements $e \in \mathcal{E}$, namely those for which $c \in \theta(e)$. Inversely, θ determines a surjective partial function $\pi\colon C \to \mathcal{E}$ such that for $c \in \mathcal{C}$, $\pi(c)$ exists and is equal to e iff $c \in \theta(e)$. In practice, we will use π to project away any additional information that was needed for construction and to derive the associated value in the exact space from the least fixpoint of the operator, i.e., $D = \pi(\mathrm{lfp}(O))$. Often but not always, the exact and the construction space are the same and π is the identity function. E.g., in Example 1, the defined object is the set F of edges of the transitive closure, the exact space $\mathcal{E}_F$ is the set 2^{V^2} of all sets of possible edges; the construction space is $\mathcal{C}_F = \langle 2^{V^2}, \subseteq \rangle$; the semantic operator is O_F.

Different Flavours of Constructive Definitions

In this section, we instantiate the earlier introduced framework for a range of constructive definitions coming from different areas. We bring them together to show that indeed, in different fields, the design of the exact and construction space is the key point. Once this choice is made explicit, typically the definition of the operator follows straightforwardly, and the defined object is constructed by the fixpoint theory. The final step may be to project the fixpoint from the construction space to an element of the exact space, i.e., the defined object, using π. In the majority of the proposed examples, this is not needed, since the injection θ just sends an exact element $e \in \mathcal{E}$ to the singleton $\{e\} \in 2^C$. However, Example 7 illustrates where the projection π plays a role.

(Co)inductive Definitions of Sets

Inductive definitions are ubiquitous in mathematical texts. Concepts such as the transitive closure, the natural numbers, ordinals, and formulas in logic, are usually defined inductively [3, 2]. On the other hand, many common infinite datatypes such as infinite streams, infinite trees and coterms, are typically defined coinductively [22]. In general, these definitions define sets of elements of a certain type $\mathcal{T}$, given by the context. Naturally, the exact space then consists of all sets of elements of $\mathcal{T}$, i.e., it is $2^{\mathcal{T}}$. Intuitively, the construction process associated with inductive definitions gradually grows the defined set, starting from the empty set. In contrast, the construction process for coinductive definitions puts stronger restrictions on the defined set in every step, resulting in a gradually shrinking set. In both cases the power set contains all elements necessary for the construction, since we are only adding or removing elements from a subset of $\mathcal{T}$. By endowing the exact space with the subset order $\subseteq$ and the superset order $\supseteq$, we capture the respective behaviours of growing and shrinking associated with induction and coinduction. For inductive definitions we obtain the power set lattice $\langle 2^{\mathcal{T}}, \subseteq \rangle$ as a construction space. This is a complete lattice and thus a cpo. The same holds for the construction space $\langle 2^{\mathcal{T}}, \supseteq\rangle$.

Let us consider the domain of (finite or infinite) lists of natural numbers. The set of all such lists is denoted by $List$. We use a well-known notation for lists where Nil represents the empty list and $[x \mid y]$ represents the list starting with $x \in \mathbb{N}$ (often referred to as the *head*) followed by the list y (often referred to as the *tail* of the list).

Example 2 (Prime array). The set PA of all *prime arrays* is defined inductively:
- $Nil \in PA$.
- If x is a prime number and $y \in PA$, then $[x \mid y] \in PA$.

This is a monotone inductive definition, formally represented by the set of rules

$$\left\lfloor \begin{array}{l} \forall y \in List : PA(y) \leftarrow y = Nil. \\ \forall x \in \mathbb{N}, \forall y \in List : PA([x \mid y]) \leftarrow P(x) \wedge PA(y). \end{array} \right\rfloor$$

with P the set of prime numbers. The exact space is the power set 2^{List}. As construction space we then have $\mathcal{C}_{PA} = \langle 2^{List}, \subseteq\rangle$. The semantic operator for this example is $O_{PA} : \mathcal{C}_{PA} \to \mathcal{C}_{PA}$, defined by mapping a set of lists $S \subseteq \mathcal{C}_{PA}$ to

$$O_{PA}(S) = \{l \mid l = Nil \text{ or } l = [x \mid y] \text{ for some } x \in P, y \in S\}$$

Proposition 2 *The operator O_{PA} is monotone.*

Proof. Let $S_1 \subseteq S_2$ be two subsets of $List$, and let $l \in O_{PA}(S_1)$. Either $l = Nil$, in which case $x \in O_{PA}(S_2)$, or there exist $x \in P$ and $y \in S_1$ such that $l = [x \mid y]$. Since $S_1 \subseteq S_2$, we have $y \in S_2$, which implies that $x \in O_{PA}(S_2)$ also for the latter case.

The fastest induction process corresponds to the sequence $\emptyset = PA_0 \subseteq PA_1 \subseteq \ldots \subseteq PA$, with $PA_i = \bigcup_{m<i}\{[n_0, \ldots, n_m] \mid n_0, \ldots, n_m \in P\}$, the set of lists of primes with length at most i. Thus, the defined set of prime arrays consists of all finite lists containing only prime numbers. Interestingly, the same set of rules gives rise to a sensible *coinductive* definition.

Example 3 (Prime lists). The set PL of all *prime lists* is defined coinductively:
- $Nil \in PL$.
- $[x \mid y] \in PL$, if x is a prime number and $y \in PL$.

As suggested before, this definition corresponds to exactly the same formal set of rules as the previous example after replacing PA by PL

$$\left\lceil \begin{array}{l} \forall y \in List : PL(y) \leftarrow y = Nil. \\ \forall x \in \mathbb{N}, \forall y \in List : PL([x \mid y]) \leftarrow PL(y) \wedge P(x). \end{array} \right\rceil$$

Unsurprisingly, we consider the same exact space 2^{List} as in Example 2, and the construction space with inverted order, namely $\langle 2^{List}, \supseteq\rangle$. Except for its signature, the inverted order does not influence the semantic operator O_{PL}, which equals O_{PA}.

Proposition 3 *The operator O_{PL} is monotone.*

Proof. Clear by the defintion of O_{PL} and Proposition 2.

Here, the fastest induction process results in the sequence $List = PL_0 \supseteq PL_1 \supseteq \ldots \supseteq PL$, with $PL_i = \bigcup_{m<i}\{[n_0,\ldots,n_m] \mid n_0,\ldots,n_m \in P\} \cup \{[n_0,\ldots,n_i,\ldots] \mid n_0,\ldots,n_i \in P\}$ where $[n_0,\ldots,n_i,\ldots]$ denotes a list with length greater than $i-1$. Intuitively, this set corresponds to all lists l of natural numbers such that no non-primes occur within the first i elements of the list. Clearly, this sequence converges to the set of all finite and infinite lists of prime numbers. A final adaptation of the list-example restricts the defined object to only the infinite lists of prime numbers.

Example 4 (Prime streams). The set PS of all *prime streams* is defined coinductively:
– $[x \mid y] \in PS$ if x is a prime number and $y \in PS$.

By excluding the case for the empty list Nil, we obtain only the infinite lists, i.e., the streams. The definition is formalised by the following coinductive rule:

$$\lceil \forall x \in \mathbb{N}, \forall y \in List : PS([x \mid y]) \leftarrow PS(y) \land P(x). \rceil$$

We keep the same exact space and construction space as in Example 3. Here, the difference lies with the semantic operator O_{PS} which maps a set of lists S to

$$O_{PS}(S) = \{[y \mid z] \mid z \in S, y \in P\}$$

Proposition 4 *The operator O_{PS} is monotone.*

Proof. Let $S_1 \subseteq S_2$ be two subsets of $List$, and let $l \in O_{PS}(S_1)$, i.e. $l = [y, z]$ for some $y \in P$ and $z \in S_1$. Since $S_1 \subseteq S_2$, we have $y \in S_2$, which implies that $x \in O_{PS}(S_2)$, as desired.

The fastest induction process for this definition starts from $List$, since by default everything belongs to the set. During the first step it will delete the empty list and all lists with a head a such that $a \notin P$. At each subsequent step i it will remove all lists for which the ith element either does not exist, or it is not a prime number, giving us the sequence The fastest induction process then gives us the sequence $List = PS_0 \supseteq PS_1 \supseteq \ldots \supseteq PS$, with $PS_i = \{[n_0,\ldots,n_i,\ldots] \mid \forall j < i, n_j \in P\}$, i.e., the set of all lists of length at least i of which the first i elements are primes. Note that interpreting this set of rules inductively rather than coinductively will not be able to derive the inclusion of a single element, i.e., the defined object would be the empty set.

Now, let us turn our attention to a different type of examples that uses an aggregate expression, known as the "company controls" problem [20].

Example 5 (Company control-relation). Given a set C of companies, each of which owns a percentage of the shares of the other companies, the control-relation is defined inductively as follows: a company x controls another company y, if the sum of the shares of y owned by x or by companies controlled by x, is strictly more than half.

In formal rule notation:

$$\left\lfloor \forall x, y \in C : Cont(x,y) \leftarrow \left(\sum_{z \in Cont^x} Sh(z,y) \right) > 0.5. \right\rfloor$$

where $Sh\colon C^2 \to [0,1]$ is a function that maps a pair of companies (x,y) to the fraction of shares of y owned by x and $Cont^x = \{x\} \cup \{u \mid Cont(x,u)\}$. Under the (natural) assumption that $Sh(x,y) \geq 0$, this definition is monotone. The more companies that are determined to be under control of a company x, the higher the fraction of shares controlled by x in any (other) company y. The exact space is now given by the set of binary relations over C, i.e., 2^{C^2}, as the construction space we choose $\mathcal{C}_{Cont} = \langle 2^{C^2}, \subseteq \rangle$. Once again, the semantic operator $O_{Cont} : \mathcal{C}_{Cont} \to \mathcal{C}_{Cont}$ results from rule application, i.e., it maps a binary relation R to

$$O_{Cont}(R) = \left\{ (x,y) \;\middle|\; \sum_{z \in \{x\} \cup \{u \mid (x,u) \in R\}} Sh(z,y) > 0.5 \right\}.$$

Proposition 5 *The operator O_{Cont} is monotone.*

Proof. Let $S_1 \subseteq S_2$ be two subsets of C^2, and let $(x,y) \in O_{Cont}(S_1)$, i.e.

$$\sum_{z \in \{x\} \cup \{u \mid (x,u) \in S_1\}} Sh(z,y) > 0.5.$$

Since $S_1 \subseteq S_2$, we also have the inclusion $\{x\} \cup \{u \mid (x,u) \in S_1\} \subseteq \{x\} \cup \{u \mid (x,u) \in S_2\}$. Since $Sh(z,w) \geq 0$ for all $(z,w) \in C^2$, we have

$$\sum_{z \in \{x\} \cup \{u \mid (x,u) \in S_2\}} Sh(z,y) > 0.5,$$

i.e. $(x,y) \in O_{Cont}(S_2)$.

Fig. 2 visualizes the induction process for an example share-function Sh represented by a labeled directed graph. Coincidentally, the depicted induction is the only possible induction with strict increments since at every step exactly one rule is applicable.

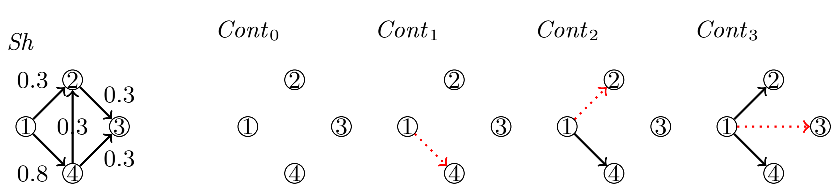

Fig. 2. An edge (a,b) in the leftmost graph indicates that $Sh(a,b) > 0$ and its label shows the exact value of $Sh(a,b)$. The other graphs show an induction process. Newly derived company pairs (represented by edges) are indicated with dotted red lines. At first only the base case is added. Later, combined ownership is derived.

(Co)recursive Definitions of Functions

We now present another set of examples, this time regarding the definition of functions. Recursion and its dual corecursion are extensively used as methods to define functions in a wide variety of mathematical and computer scientific fields [26, 32, 27]. Some well-known mathematical functions, like the factorial or the greatest common divisor, can be defined recursively, and (co)recursive definitions of functions are supported in most functional programming languages [16, 28, 17].

For our formalization, the exact space of (co)recursive definitions of functions is obtained in a natural way: if we want to define a function $f\colon X \to Y$, then the exact space is the set of functions from X to Y, denoted by Y^X. Contrary to (co)inductive definitions, here we cannot just choose the construction space to equal the exact space. The main reason for this is that in intermediate steps of the construction process only partial functions have been constructed.

Example 6 (Fibonacci sequence). The Fibonacci sequence is viewed here as the function $Fib\colon \mathbb{N} \to \mathbb{N}$, where $Fib(n)$ is the n^{th} Fibonacci number. Its recursive definition, in the formal notation, is the following:

$$\left\lfloor \begin{array}{ll} Fib(0) := 0. & Fib(1) := 1. \\ \forall n \in \mathbb{N} : Fib(n+2) := & Fib(n) + Fib(n+1). \end{array} \right\rfloor$$

Clearly, the exact space is $\mathbb{N}^{\mathbb{N}}$. Moreover, we can get some insight into the construction process from the rules above. Note that the image of $n+2$ under Fib depends on the image of n and $n+1$. As long as the latter are not derived, it is impossible to determine $Fib(n{+}2)$. Hence, it is natural to think of Fib at an intermediate step of the construction as a partial function, defined on a subset S of $\mathbb{N}$. Equivalently, we can view such a partial function as a function from $\mathbb{N}$ to $\mathbb{N}_\perp := \mathbb{N} \cup \{\perp\}$, where $\perp$ denotes “undefined”. The set $\mathbb{N}_\perp$ is naturally equipped with the definedness order $\leq_d$, given for all $n, m \in \mathbb{N}$ by $n \leq_d m$ iff $n = m$ or $n = \perp$. We take the construction space $\mathcal{C}_{Fib} = \langle (\mathbb{N}_\perp)^{\mathbb{N}}, \leq_d \rangle$ to be the set of functions $(\mathbb{N}_\perp)^{\mathbb{N}}$, equipped with the pointwise extension of $\leq_d$. This indeed forms a cpo.

Lemma 1. *$\langle \mathbb{N}_\perp, \leq \rangle$ is a cpo.*

Proof. Let $S \subseteq \mathbb{N}_\perp$ be a chain. By the definition of the order $\leq$, S has at most two elements. Hence, the $lub(S) = n$ where $\{n\} = S \cap \mathbb{N}$ if $S \cap \mathbb{N} \neq \emptyset$, or $\perp$ otherwise.

Proposition 6 *$\mathcal{C}_{Fib}$ is a cpo.*

Proof. Let $S \subseteq C_{Fib}$ be a chain. Since the order $\leq_d$ is defined pointwise, for all $n \in \mathbb{N}$, $S_n := \{f(n) \mid f \in S\} \subseteq \mathbb{N}_\perp$ is a chain. By Lemma 1, for all $n \in \mathbb{N}$, there exists $lub(S_n)$. It is easy to see that the function $F\colon \mathbb{N} \to \mathbb{N}_\perp$ defined by $F(n) := lub(S_n)$ is the least upper bound of S.

It remains to define a monotone operator on $\mathcal{C}_{Fib}$. First, the sum $+\colon \mathbb{N}^2 \to \mathbb{N}$ can be extended to $\mathbb{N}_\perp \times \mathbb{N}_\perp$ by defining for each $n \in \mathbb{N}_\perp$, $\perp + n = n + \perp = \perp$. Then, the operator $O_{Fib}\colon \mathcal{C}_{Fib} \to \mathcal{C}_{Fib}$ is defined to map a function $f \in \mathcal{C}_{Fib}$ to

$$O_{Fib}(f) := \begin{cases} n & \mapsto n \quad (\text{for } n \in \{0,1\}) \\ n+2 & \mapsto f(n+1) + f(n) \end{cases}$$

Proposition 7 *The operator O_{Fib} is monotone.*

Proof. Let $f, g \in \mathcal{C}_{Fib}$ such that $f \leq_d g$, i.e. for all $n \in \mathbb{N}$, $f(n) \leq g(n)$. Notice that we have

$$\begin{aligned} O_{Fib}(f)(0) &= 0 = O_{Fib}(g)(0) \\ O_{Fib}(f)(1) &= 1 = O_{Fib}(g)(1) \\ \forall n \in \mathbb{N} \setminus \{0,1\},\ O_{Fib}(f)(n) &= f(n-1) + f(n-2) \\ &\leq g(n-1) + g(n-2) \\ &= O_{Fib}(g)(n). \end{aligned}$$

Hence, $O_{Fib}(f) \leq_d O_{Fib}(g)$, as desired.

By the definition of O_{Fib}, it is easy to see that the desired function Fib is the least fixpoint of O_{Fib}. The element $\mathrm{lfp}(O_{Fib})$ can also be constructed as the limit of the increasing sequence of functions $f_0, f_1, \dots$ in $\mathcal{C}_{Fib}$ obtained by iterating O_{Fib} on the bottom element of $\mathcal{C}_{Fib}$. We write here the functions in the first iterations of the process:

$$f_0(n) := \bot \qquad f_1(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \bot & \text{otherwise} \end{cases} \qquad f_2(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n \in \{1,2\} \\ \bot & \text{otherwise} \end{cases}$$

The enrichment of the exact space with $\bot$ allows us to deal with partially defined concepts, providing us with a suitable choice for the construction space. Nevertheless, such choice may be even more subtle, as we show next.

Example 7 (Ackermann function). $Ack\colon \mathbb{N}^2 \to \mathbb{N}$ is defined recursively as follows:

$$\left[\begin{array}{ll} \forall y \in \mathbb{N} : Ack(0, y) & := y + 1. \\ \forall x \in \mathbb{N} : Ack(x+1, 0) & := Ack(x, 1). \\ \forall x, y \in \mathbb{N} : Ack(x+1, y+1) & := Ack(x, Ack(x+1, y)). \end{array}\right]$$

The exact space is again the set of functions with the right signature, namely $\mathbb{N}^2 \to \mathbb{N}$. Analogously to Example 6, the function Ack is defined on every element of its domain only after infinitely many steps. Hence, it may seem natural to consider as construction space the functions from $\mathbb{N}^2$ to $\mathbb{N}_\bot$. However, this enlargement of the construction space is not sufficient: due to the third rule of the definition, during the construction, the Ackermann function might be invoked on an output of a partially constructed object, which can possibly be $\bot$. This prompts us to add $\bot$ to the *domain* of the functions in the construction space.

Just as before, we can order this expanded space ${\mathbb{N}_\bot}^{\mathbb{N}_\bot \times \mathbb{N}_\bot}$ by the pointwise extension of the definedness order $\leq_d$ on $\mathbb{N}_\bot$. However, the operator induced by the definition of Ack is not monotone on the full set of functions. Fortunately, it was shown that this operator is monotone on a sufficiently large subset defined next. We expand the definedness order $\leq_d$ to $\mathbb{N}_\bot \times \mathbb{N}_\bot$ as the product order of $\leq_d$ on $\mathbb{N}_\bot$, and consider the subset C_{Ack} of monotone functions of $\mathbb{N}_\bot^{\mathbb{N}_\bot \times \mathbb{N}_\bot}$. It turns out that the operator of the Ackerman definition, and the limit operation for increasing sequences of monotone functions both preserve monotoniticy of functions, making $C_{Ack} \subseteq \mathbb{N}_\perp^{\mathbb{N}_\perp \times \mathbb{N}_\perp}$ a suitable space to perform the induction process. In other words, we choose the construction space to be the cpo $\mathcal{C}_{Ack} := \langle C_{Ack}, \leq_d \rangle$. The proof of the fact that the construction space $\mathcal{C}_{Ack}$ is a cpo analogous to the proof of Proposition 6, given that $\langle \mathbb{N}_\perp \times \mathbb{N}_\perp, \leq_d \rangle$ is a cpo, which follows easily from Lemma 1.

It is important to note that for the first time, the injection θ is nontrivial, since we have enlarged the domain of the considered functions to $\mathbb{N}_\perp \times \mathbb{N}_\perp$. In particular, $\theta \colon \mathbb{N}^{\mathbb{N}^2} \to 2^{C_{Ack}}$ sends a function $f \colon \mathbb{N}^2 \to \mathbb{N}$ to the set of functions

$$\theta(f) := \{g : (\mathbb{N}_\perp)^2 \to \mathbb{N}_\perp \mid \forall (x,y) \in \mathbb{N}^2, f(x,y) = g(x,y)\}.$$

Hence, it is easy to see that the surjective partial function $\pi \colon C_{Ack} \to \mathbb{N}^{\mathbb{N}^2}$ associated to θ is defined only on the set of of functions whose restriction to $\mathbb{N}^2$ maps into $\mathbb{N}$ and maps each such function to its restriction to $\mathbb{N}^2$.

By the above recursive definition of Ack, the choice for the operator $O_{Ack} \colon \mathcal{C}_{Ack} \to \mathcal{C}_{Ack}$ is clear: for all $f \in L$,

$$O_{Ack}(f) := \begin{cases} (0,y) & \mapsto y+1 \\ (x+1,0) & \mapsto f(x,1) \\ (x+1,y+1) & \mapsto f(x, f(x+1,y)) \end{cases}$$

where $+$ is extended to $\mathbb{N}_\perp \times \mathbb{N}_\perp$ as in Example 6. Note that $O_{Ack}(f)$ is indeed an element of $\mathcal{C}_{Ack}$, since the composition of monotone functions is monotone.

Proposition 8 *The operator O_{Ack} is monotone.*

Proof. Let $f, g \in C_{Ack}$ such that $f \leq_d g$. Then we have

$$\begin{aligned} &\forall y \in \mathbb{N}_\perp,\ O_{Ack}(f)(0,y) = y+1 = O_{Ack}(g)(0,y), \\ &O_{Ack}(f)(\perp,0) = f(\perp,0) \leq_d g(\perp,0) = O_{Ack}(g)(\perp,0), \\ &\forall x \in \mathbb{N} \setminus \{\perp, 0\},\ O_{Ack}(f)(x,0) = f(x-1,0) \\ &\qquad\qquad\qquad\qquad \leq_d g(x-1,0) = O_{Ack}(g)(x,0). \end{aligned}$$

Moreover, for all $x, y \in \mathbb{N}_\perp$,

$$\begin{aligned} O_{Ack}(f)(x+1,y+1) &= f(x, f(x+1,y)) \\ &\leq_d f(x, g(x+1,y)) \\ &\leq_d g(x, g(x+1,y)) \\ &= O_{Ack}(g)(x+1,y+1), \end{aligned}$$

where the first inequality holds by the monotonicity of f. Hence, $O_{Ack}(f) \leq_d O_{Ack}(g)$, as desired.

The construction process starts from the bottom element $\perp_{Ack}$ of $\mathcal{C}_{Ack}$, namely the function $\perp_{Ack} \colon \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$ sending every pair to $\perp$. By iteratively applying the operator O_{Ack}, we obtain an increasing sequence of monotone functions $f_0, f_1, \ldots$ in

$\mathcal{C}_{Ack}$, representing the partial functions of the intermediate steps of the recursion. At the first steps of the process we get the functions defined on $(x, y) \in \mathbb{N}_\perp \times \mathbb{N}_\perp$ as follows:

$$f_0(n) := \perp \qquad f_1(x,y) := \begin{cases} y+1 & \text{if } x = 0 \\ \perp & \text{otherwise} \end{cases} \qquad f_2(x,y) := \begin{cases} y+1 & \text{if } x = 0 \\ 2 & \text{if } (x,y) = (1,0) \\ \perp & \text{otherwise} \end{cases}$$

Only after transfinitely many steps, we reach the least fixpoint of O_{Ack}. Finally, it is not hard to see that applying the projection π on the least fixpoint yields the defined object, i.e., $\pi(\mathrm{lfp}(O_{Ack})) = \mathrm{lfp}(O_{Ack})\big|_{\mathbb{N}^2} = Ack$.

We now move to examples of co-recursive definitions of functions. Once again, the choice of a suitable construction space turns out to be non-trivial.

Example 8 (Co-Fibonacci). The co-Fibonacci function $co\_Fib\colon \mathbb{N}^2 \to List$, which maps a pair (x, y) of natural numbers to the Fibonacci sequence starting with x, y, is defined co-recursively to send (x, y) to $[x \mid co\_Fib(y, x + y)]$.

We can present the corecursive definition of co_Fib by

$$\lceil \forall x, y \in \mathbb{N} : co\_Fib(x, y) := [x \mid co\_Fib(y, x + y)]. \rceil$$

In particular, $co\_Fib(0, 1)$ is the list corresponding to the Fibonacci sequence, defined recursively in Example 6. The exact space is again clear from the signature of the function we want to define: it is the set of functions from $\mathbb{N}^2$ to $List$.

As in Example 6, we need to enlarge the codomain of the considered functions in order to represent intermediate steps of the process. Thus, we define a set $List^o$, containing lists of natural numbers and finite lists of natural numbers ending with o. A list $[x_1, \ldots, x_n \mid o]$ of the latter type represents a list of natural numbers with overdefined o as tail. In other words, the list $[x_1, \ldots, x_n \mid o]$ represents the set $\{[x_1, \ldots, x_n \mid l] \colon l \in List\}$ of lists of natural numbers. In particular, the list o represents the overdefined list, i.e., the set of all lists of natural numbers.[7] Accordingly, on $List^o$, definedness order $\leq_d$ is defined inductively as follows:

- for all $t \in List^o$: $t \leq_d o$
- for all $x \in \mathbb{N}, t_1, t_2 \in List^o$: $[x \mid t_1] \leq_d [x \mid t_2]$ if $t_1 \leq_d t_2$

In this order, o is indeed "more defined" than any list. The set $List^o$ with the order $\leq_d$ is not a cpo since it has no least element, however, with the inverted order $\geq_d$ it indeed is a cpo, with "least" element o.

Lemma 2. *$\langle List^o, \geq_d \rangle$ is a cpo.*

Proof. Let $S \subseteq List^o$ be a chain. We have to show that S has a least upper bound. If S has finite cardinality, the claim is trivial. Suppose S has an infinite number of elements. By the definition of $\geq_d$ and since S is a chain, if S contains an infinite list l, then l is the least upper bound of S. Suppose otherwise, i.e. all elements in S are finite lists of natural numbers or finite lists of natural numbers ending with o. We denote the length

[7] Note that the earlier introduced notation $[x \mid y]$ is used now to denote a list of $List^o$ with head a finite list x of natural numbers, and tail a list y of $List^o$.

of a list $l \in S$ by $length(l) \in \mathbb{N}$. We can order the lists in S following the total order, and we denote by l^i the i-th list in S with such ordering, i.e. $i_1 \leq i_2$ if and only if $l^{i_1} \leq l^{i_2}$. Notice that if $length(l^i) = length(l^{i+1})$, then l^i must end with o and l^{i+1} with a natural number. In particular, $length(l^{i+2})$ is strictly greater than $length(l^i)$. Let $l \in S$ be a list and $n \leq length(l)$, we denote by l_n the n-th element of l We define an infinite list $L := [L_j]_{j \in \mathbb{N}}$ where

$$\forall j \in \mathbb{N} : L_j := l^{2j+2}_{length(l^{2j+2})-1}.$$

Notice that, for all $i \in \mathbb{N}$, the first $\min(length(l^i), length(l^{i+1})) - 1$ elements of l^i and l^{i+1} coincide. Hence, it is not hard to see that for any $l \in S$ we have $l \geq_d L$ by construction. Let U be an upper bound for S, i.e. U is an infinite sequence of natural numbers such that $l \geq_d U$ for all $l \in S$. By the definition of the order, for all $l \in S$, the first $length(l) - 1$ elements of l and U coincide. Hence, it is easy to see that $L = S$. In particular, L is the least upper bound of S.

The order $\geq_d$ can be extended in the standard, pointwise way to $(List^o)^{\mathbb{N}^2}$. We define the construction space $\mathcal{C}_{co\_Fib} = \langle (List^o)^{\mathbb{N}^2}, \geq_d \rangle$. The inversion of the definedness order, often used for recursion, mimics the order inversion between inductive and coinductive definitions (hence the term *corecursion*).

Finally, we define the operator $O_{co\_Fib} \colon \mathcal{C}_{co\_Fib} \to \mathcal{C}_{co\_Fib}$ by sending a function $f \in \mathcal{C}_{co\_Fib}$ to

$$O_{co\_Fib}(f) \colon \mathbb{N}^2 \to List^o : (x, y) \mapsto [x \mid f(y, x + y)].$$

By the definition of $\geq_d$, it is easy to see that O_{co_Fib} is a monotone operator. Moreover, the desired function co_Fib is the least fixpoint of the operator O_{co_Fib}. This coincides with the limit of the increasing sequence $f_0, f_1, \ldots$ constructed by iterating O_{co_Fib} on the bottom element $\bot_{\mathcal{C}_{co\_Fib}}$ of $\mathcal{C}_{co\_Fib}$, i.e., the function $\bot_{\mathcal{C}_{co\_Fib}} : \mathbb{N}^2 \to List^o$ sending every tuple to o. We report here the images of the functions in the first iterations of the process, depending on $(x, y) \in \mathbb{N}^2$:

$$f_0(x,y) = \bot_{\mathcal{C}_{co\_Fib}}(x,y) = o \qquad f_2(x,y) = [x, y \mid o]$$
$$f_1(x,y) = [x \mid o] \qquad f_3(x,y) = [x, y, x+y \mid o]$$

Definitions with Custom-Designed Cpo's

In this third and last subsection, we present a final example of a constructive definition of a function. Even though this definition deviates from the standard (co)recursive account, it can indeed be formalized using our proposed framework. Just like Example 5, the definition illustrated here falls under the company controls domain: in this case, we want to define the number of shares of a company that another company controls.

Example 9 (Controlled shares). If x and y are two companies, we say that x controls n shares of y if n is the sum of the shares of y owned by x or any company z of which x controls more than half of the shares.

We can formally define the desired function $Csh\colon C^2 \to [0,1]$ as follows:

$$\left\{ \begin{array}{l} \forall x \forall y : Csh(x,y) := \sum_{z \in \{x\} \cup \{u \mid Csh(x,u) > 0.5\}} Sh(z,y). \end{array} \right\}$$

where $Sh\colon C^2 \to [0,1]$ is still the function mapping a pair of companies (x,y) to the fraction of shares of y owned by x. The exact space is the set of functions from S^2 to the interval $[0,1]$. The construction process is more complex than before: we now need to be able to decide whether $Csh(x,y) > 0,5$ is true before $Csh(x,y)$ is determined.

We can think about this construction process as a gradual refinement of each tuple's image. At the beginning of the process, we have no information about the image of Csh except that $Csh(x,y) \in [0,1]$ for all $x,y \in C$. At every rule application we get new information on the lower bounds of the images of elements of C^2. Since the upper bounds remain constant equal to 1, we may as well identify the interval in which an image is contained with its lower bound. Now, the choice for a construction space $\mathcal{C}_{Csh}$ becomes clear, namely we consider the cpo of functions from C^2 to $[0,1]$, with the pointwise extension of the standard order $\leq$ on real numbers.

Proposition 9 *$\mathcal{C}_{Csh}$ is a cpo.*

Proof. Since $([0,1], \leq)$ is a cpo, the proof is analogous to the proof of Proposition 6.

Finally, we can consider the monotone operator $O_{Csh} : \mathcal{C}_{Csh} \to \mathcal{C}_{Csh}$, which maps a function $f\colon C^2 \to [0,1]$ to $O_{Csh}(f)$, defined by

$$O_{Csh}(f)(x,y) := \sum_{z \in \{x\} \cup \{u \mid f(x,u) > 0,5\}} Sh(x,y).$$

Proposition 10 *The operator O_{Csh} is monotone.*

Proof. Let $f, g\colon C^2 \to [0,1]$ such that $f \leq_L g$. In particular, for all $x \in C$, we have $\{u \mid f(x,u) > 0,5\} \subseteq \{u \mid g(x,u) > 0,5\}$. Since $Sh(z,y) \geq 0$ for all $z,y \in C$, we have $O_{Csh}(f) \leq_L O_{Csh}(g)$, as desired.

As anticipated, we can start the recursion from the bottom element of $\mathcal{C}_{Csh}$, namely the function f_0 sending every pair of companies to 0. By iteratively applying the operator O_{Csh} we get an increasing sequence of functions $f_0 \leq_{\mathcal{C}_{Csh}} f_1 \leq_{\mathcal{C}_{Csh}} f_2 \leq_{\mathcal{C}_{Csh}} \cdots$, whose limit is the desired defined function Csh and coincides with the least fixpoint of O_{Csh}. Notice that at any step t of the construction process, for each pair $(x,y) \in C^2$, the image $f_t(x,y)$ may not be the correct value of $Csh(x,y)$. Only in the last step, when the fixpoint is reached, certainty is reached of the correct value $Csh(x,y)$, for all pairs (x,y) at once. This is much unlike previous examples. This type of construction, using increasingly more precise bounds, lies at the basis of bound-founded ASP [4, 8].

Conclusion and Future Work

We investigated a heterogeneous set of monotone constructive definitions, coming from different domains and never brought together before, in a uniform framework. Our analysis confirms the power of fixpoint theory for abstract formalization, but also points to a key distinguishing factor: the construction space, the set of objects that serve as approximations of the object being defined. We propose the general term *monotone constructive definitions* for a class of definitions that includes recursive and inductive definitions and developed a framework that clearly emphasises how different types of definitions can be classified according to different types of construction spaces. This is a crucial step towards the development of knowledge representation languages that include a variety of constructive definitions. Our framework suggests such language requires a formal syntax for definitions such that one can automatically and uniformly derive a suitable exact space, semantic operator and construction space. As shown by the examples, while the first two are straightforward, the latter may be non-trivial. We have illustrated different types of definitions by example, allowing us to handpick the most convenient, natural construction space. The challenge is that a uniformly derived construction space needs to be strong enough to handle all considered definitions, and the defined object should coincide with the one obtained with the handpicked construction space.[8] This paper offers an important first step towards solving this issue by classifying different types of definitions based on the kind of construction space they require. This means identifying the correct type of definition will be an essential part of the syntax of the considered knowledge representation language.

By no means do we claim our list of types of constructive definitions to be exhaustive. Other types of constructive definitions not considered here are nested inductive and coinductive definitions where multiple objects are defined in a hierarchy of inductive and coinductive definitions [30, 25] or non-monotone "iterated" inductive definitions which have been researched in mathematical logic [18, 23, 7]. In iterated inductive definitions, e.g., over a well-founded order, multiple objects are defined in terms of other defined objects on a lower or equal level. Once all objects on some level are well-defined, their values can be used to derive the value of any object on a higher level. This is the natural principle of *stratification*. It has been argued that this principle is implemented by the well-founded semantics of logic programming [11, 14, 15]. Thus, the declarative logic underlying logic programming can be seen as a logic of this type of constructive definition.

In this paper, we focused on *monotone* constructive definitions. Non-monotone *inductive* definitions have been studied intensively, including in a fixpoint-theoretic setting (known as *Approximation Fixpoint Theory (AFT)* [12]). In the terminology of the current paper, dealing with this non-monotonicity requires switching to a different construction space (a space of approximations). A natural next question we wish to tackle is whether this framework can also be of use for studying non-monotone *recursive* definitions.

As a final remark, we argued that constructive definitions are an important form of human knowledge. Of course, many other types of knowledge are important as well. Contrary to the languages of logic and functional programming, which support mainly definitions used as programs, expressive KR languages should offer language constructs for expressing a broad range of knowledge. In this respect, an example is the logic FO($\cdot$),

[8] In the supplementary material of the appendices, the suitable notion of isomorphism between construction spaces is introduced allowing to relate operators and defined objects in different construction spaces.

which extends FO with among others an expressive rule-based language construct for definitional knowledge, inspired by logic programming [10, 9].

References

1. Abramsky, S., Jung, A.: Handbook of logic in computer science (vol. 3). chap. Domain Theory, pp. 1–168. Oxford University Press, Inc., New York, NY, USA (1994), http://dl.acm.org/citation.cfm?id=218742.218744
2. Aczel, P.: An introduction to inductive definitions. In: Studies in Logic and the Foundations of Mathematics, vol. 90, pp. 739–782. Elsevier (1977)
3. Aczel, P.: The type theoretic interpretation of constructive set theory: inductive definitions. In: Studies in Logic and the Foundations of Mathematics, vol. 114, pp. 17–49. Elsevier (1986)
4. Aziz, R.A.: Bound founded answer set programming. CoRR **abs/1405.3367** (2014), http://arxiv.org/abs/1405.3367
5. Bogaerts, B., Vennekens, J., Denecker, M.: Safe inductions and their applications in knowledge representation. Artificial Intelligence **259**, 167 – 185 (2018). https://doi.org/10.1016/j.artint.2018.03.008, http://www.sciencedirect.com/science/article/pii/S000437021830122X
6. Brouwer, L.E.J.: Over de grondslagen der wiskunde. Maas & van Suchtelen (1907)
7. Buchholz, W., Feferman, S., Pohlers, W., Sieg, W.: Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies, Lecture Notes in Mathematics, vol. 897. Springer (1981)
8. Cabalar, P., Fandinno, J., Schaub, T., Schellhorn, S.: Lower bound founded logic of here-and-there. In: Calimeri, F., Leone, N., Manna, M. (eds.) Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11468, pp. 509–525. Springer (2019). https://doi.org/10.1007/978-3-030-19570-0"˙34, https://doi.org/10.1007/978-3-030-19570-0_34
9. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system. In: Declarative Logic Programming: Theory, Systems, and Applications, pp. 279–323 (2018). https://doi.org/10.1145/3191315.3191321, https://doi.org/10.1145/3191315.3191321
10. Denecker, M.: Extending classical logic with inductive definitions. In: Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) CL. LNCS, vol. 1861, pp. 703–717. Springer (2000)
11. Denecker, M., Bruynooghe, M., Marek, V.: Logic programming revisited: Logic programs as inductive definitions. ACM Trans. Comput. Log. **2**(4), 623–654 (2001)
12. Denecker, M., Marek, V., Truszczyński, M.: Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, The Springer International Series in Engineering and Computer Science, vol. 597, pp. 127–144. Springer US (2000). https://doi.org/10.1007/978-1-4615-1567-8˙6, http://dx.doi.org/10.1007/978-1-4615-1567-8_6
13. Denecker, M., Ternovska, E.: Inductive situation calculus. Artif. Intell. **171**(5-6), 332–360 (2007)
14. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Log. **9**(2), 14:1–14:52 (Apr 2008), http://dx.doi.org/10.1145/1342991.1342998
15. Denecker, M., Vennekens, J.: The well-founded semantics is the principle of inductive definition, revisited. In: Baral, C., De Giacomo, G., Eiter, T. (eds.) KR. pp.

1–10. AAAI Press (2014), http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7957
16. Doets, K., Eijck, J.: The Haskell road to logic, maths and programming. College Publications (2004)
17. Downen, P., Ariola, Z.M.: Classical (co)recursion: Programming. CoRR **abs/2103.06913** (2021), https://arxiv.org/abs/2103.06913
18. Feferman, S.: Formal theories for transfinite iterations of generalised inductive definitions and some subsystems of analysis. In: Kino, A., Myhill, J., Vesley, R. (eds.) Intuitionism and Proof theory, pp. 303–326. North Holland (1970)
19. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. **21**(3), 359–411 (1989). https://doi.org/10.1145/72551.72554, http://dx.doi.org/10.1145/72551.72554
20. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: Saraswat, V.A., Ueda, K. (eds.) ISLP. pp. 387–401. MIT Press (1991)
21. Kleene, S.C.: On the forms of the predicates in the theory of constructive ordinals (66), 41–58 (1944)
22. Kozen, D., Silva, A.: Practical coinduction. Mathematical Structures in Computer Science **27**(7), 1132–1152 (2017)
23. Martin-Löf, P.: Hauptsatz for the intuitionistic theory of iterated inductive definitions. In: Fenstad, J. (ed.) Second Scandinavian Logic Symposium. pp. 179–216 (1971)
24. Moschovakis, Y.N.: Elementary Induction on Abstract Structures. North-Holland Publishing Company, Amsterdam- New York (1974)
25. Paulson, L.C.: A fixedpoint approach to (co) inductive and (co) datatype definitions. In: Proof, Language, and Interaction. pp. 187–212 (2000)
26. Roberts, E.S.: Thinking recursively. Wiley (1986)
27. Rubio-Sanchez, M.: Introduction to recursive programming. CRC Press (2017)
28. Rusu, V., Nowak, D.: Defining corecursive functions in Coq using approximations. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 12:1–12:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.12, https://doi.org/10.4230/LIPIcs.ECOOP.2022.12
29. Scott, D.: Data types as lattices. In: Proceedings of the International Summer Institute and Logic Colloquium. Lecture Notes in Mathematics, vol. 499, p. 579–651. Springer-Verlag (1975)
30. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Logic Programming: 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 22. pp. 330–345. Springer (2006)
31. Smyth, M.B., Plotkin, G.D.: The category-theoretic solution of recursive domain equations. SIAM Journal on Computing **11**(4), 761–783 (1982)
32. Soare, R.I.: Recursively enumerable sets and degrees - a study of computable functions and computability generated sets. Perspectives in mathematical logic, Springer (1987)
33. Spector, C.: Inductively defined sets of natural numbers. In: Infinitistic Methods (Proc. 1959 Symposium on Foundation of Mathematis in Warsaw). pp. 97–102. Pergamon Press, Oxford (1961)
34. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics (1955)