Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 5–31
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Interactive Computation: Stepping Stone in the Pathway From Classical to Developmental Computation[1]

Antônio Carlos da Rocha Costa[a,b,2] Graçaliz Pereira Dimuro[a,3]

[a] *Escola de Informática, Universidade Católica de Pelotas, Pelotas, Brazil*

[b] *PPGC, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*

Abstract

This paper reviews and extends previous work on the domain-theoretic notion of Machine Development. It summarizes the concept of Developmental Computation and shows how Interactive Computation can be understood as a stepping stone in the pathway from Classical to Developmental Computation. A critical appraisal is given of Classical Computation, showing in which ways its shortcomings tend to restrict the possible evolution of real computers, and how Interactive and Developmental Computation overcome such shortcomings. The idea that Developmental Computation is more encompassing than Interactive Computation is stressed. A formal framework for Developmental Computation is sketched, and the current frontier of the work on Developmental Computation is briefly exposed.

Keywords: Interactive computation, developmental computation, domain theory, classical theory of computation

1 Introduction

In [5], the first author introduced a domain-theoretic approach to the conceptual analysis of Interactive and Developmental Computation. That thesis consisted of an epistemological analysis of the principles of Artificial Intelligence and the Theory of Computation, aiming:

[1] Work partially supported by CNPq and FAPERGS.

[2] Email: rocha@atlas.ucpel.tche.br

[3] Email: liz@atlas.ucpel.tche.br

doi:10.1016/j.entcs.2005.05.014

(i) to establish a sound, constructivist foundation for the notion of *machine intelligence* as a regulation structure for the functional interactions between computing machines and their environments;

(ii) to outline a naturalistic approach to Artificial Intelligence, so that the central purpose of AI becomes the study of machine intelligence as a physical symbol systems-based structure that naturally occurs in computing machines, not the attempt to simulate the human mind [4];

(iii) to show that such structure can only be apprehended according to a constructivist approach, where the intelligence of machines arises as a limit structure in a developmental process occurring in a suitable domain;

(iv) to make clear that such developmental processes can only happen in the context of interactive computing, but also that interaction, although necessary, is not sufficient for such purpose, *internal developmental operations* being the indispensable complementary components of such processes.

In connection to the third and fourth goals mentioned above, the work in [5] aimed:

(i) to make clear, by means of a historical review of Computer Science, that the notions of Interaction and Development were present in the area since the very beginning (and even before, in areas such as Cybernetics, and beyond); but that, for various reasons (mainly the too restrictive notion of *computational effectiveness* that was adopted in Classical Computation), they were always kept latent, and never fully explored;

(ii) to show that interactive and developmental machines can go beyond the models of Classical Computation (CC), in the sense of introducing a shift in the scope of the notion of computation, bringing it from the strictly algorithmic computational processes to the non-algorithmic computational processes;

(iii) to introduce, in a tentative way, some elementary developmental mechanisms capable of supporting processes of machine development.

Considering that [5] was elaborated in the late 1980's and early 1990's, before the seminal papers by Peter Wegner [29,30,31] on Interactive Computation (IC), and also before his immediately subsequent papers with Dina Golding [33,34,12,11] – so, being unable to benefit from such papers –, it is remarkable how close the results in [5] concerning the third and fourth goals match the general results of the work by Wegner & Golding.

[4] We take the expression *physical symbol system* right in the sense introduced by A. Newell [17].

In particular, it is notable that both works coincided in the identification of the three main shortcomings of Classical Computation, namely, the requirements that:

(i) the machine operates as a closed system, during the computation, thus forbidding interaction;

(ii) only finitely many resources be used during the computation, thus dismissing infinite computations;

(iii) the structure of the machine remains fixed during the computation, thus forbidding machine evolution and development.

On the other hand, [5] being based on different epistemological principles, namely those of Jean Piaget's *Genetic Epistemology* [18,19] (for the epistemological foundation of P. Wegner's work, see [32]), it is not surprising that some differences in purposes and results have appeared between that thesis and the work by Wegner & Goldin.

Also, having been put to sleep for ten years, it is not surprising that the work started in [5] did not achieve the degree of formalization achieved by Goldin and co-authors in their later works [11].

The present paper concerns the computation-theoretic aspects of [5]. It's goals are, in the first place, to define Developmental Computation and to relate it to Interactive Computation. It also aims to highlight the domain-theoretic basis that [5] adopted. Finally, it aims to show that Developmental Computation is more encompassing than Interactive Computation.

The paper is organized as follows. Section 2 summarizes Domain Theory. Section 3 shows that Interaction was already embedded in the well known *von Neumann's computer architecture*. Section 4 introduces Developmental Computation.

Section 5 sketches a domain-theoretic framework for Developmental Computation. Section 6 presents a sample model of machine development. Section 7 concerns the Conclusion, related work, and a brief overview of the current frontier of Developmental Computation.

2 A Domain-based Appraisal of CC

Classical Computation (CC) was settled by the foundational works of Turing, Church, Kleene, Post, Curry and others, and was consolidated in widely used text-books, such as the ones by Kleene [15] and Rogers [22].

Domains [1,14] were introduced by Dana Scott as mathematical structures that allow for a model of the λ-calculus, and support the denotational semantics of programming languages. Scott himself gave various presentations of

the structure of domains, e.g., [23,25,26], and specifically [24], in which the general ideas of Domain Theory are presented in an informal way that is well compatible with the way they are used here.

Domain Theory officially introduced in Computer Science the idea of *partial object*, that is, the result of a *partial* (*unfinished*) computation. Using partial objects, Domain Theory was able to give *infinite computations* the status of first order citizens. Each infinite computation can be assigned a non-trivial meaning, thus allowing infinite computations to be distinguished from each other, so that they may not be simply dismissed as *divergent*.

2.1 Domains

A domain is an ordered structure whose elements are called *objects*. Objects of a domain are considered to be *results* of a computation. Computations are seen as processes that construct objects of a domain.

Objects of domains are ordered according to the way each object participates in the construction of other objects. That is, if x and y are objects of a domain D and x is a part of y, one denotes this by $x \sqsubseteq y$. The relation $\sqsubseteq$ is called *approximation relation*, and x is said to be an approximation of y. Objects are said to be *partial* objects, since – in general – it is possible to aggregate new (partial) objects to a given object, to make it become a more complete object. Objects from which it is not possible to construct other objects, because nothing can be added to them, are said to be *total* (*complete*) objects. Total objects are the maximal elements of the ordering $\sqsubseteq$.

The concept of domain involves the concept of *chain*: given a domain D ordered by the relation $\sqsubseteq$, a chain in D is any finite or infinite sequence $X = \{x_i\}_{i\in\mathbb{N}}$ of elements of D, such that $x_i \sqsubseteq x_j$ whenever $i \leq j$. Chains are the simplest ways of modelling constructions in domains.

2.2 Examples of domains

For the examples below, we have chosen to use the simplest form of domains, namely, the CPOs. A CPO is a partially ordered set that has a least element and where every chain has a least upper bound.

Example 2.1 Let $\Sigma = \{a, b\}$ be an alphabet, and ω the cardinality of $\mathbb{N}$. The domain of strings over Σ, ordered by the prefix order, is the structure $D_\Sigma = (\Sigma^\infty; \bot, \sqsubseteq)$ where:

(i) $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$
where $\Sigma^* = \{w : \{0, 1, \ldots, n\} \to \Sigma \mid n \in \mathbb{N}\} \cup \{\bot\}$ is the set of finite strings over Σ, and $\Sigma^\omega = \{w \mid w : \mathbb{N} \to \Sigma\}$ is the set of infinite strings

over Σ

(ii) for all $s, t \in \Sigma^\infty$, $s \sqsubseteq t$ iff:
  (a) $s, t \in \Sigma^*$ and $length(s) \leq length(t)$, and for all $0 \leq i \leq lenght(s)$ it happens that $s(i) = t(i)$; or
  (b) $s \in \Sigma^*$, $t \in \Sigma^\omega$ and $s(i) = t(i)$ for all $0 \leq i \leq lenght(s)$; or else
  (c) $s, t \in \Sigma^\omega$ and $s = t$

(iii) $\perp$ is the empty string, and $\perp \sqsubseteq s$ for all $s \in \Sigma^\infty$

D_∞ is a CPO because every finite or infinite chain $X \subseteq D_\infty$ has a least upper bound $\sqcup X \in D_\infty$: either X is finite and $\sqcup X$ is its last element $x_n \in \Sigma^*$, or X is infinite and $\sqcup X \in \Sigma^\omega$ (see, e.g., [1]).

Example 2.2 Let $\mathbb{N} = \{0, 1, \ldots, n, \ldots\}$ be the set of natural numbers. The domain of *partial natural numbers* is the domain $\mathbb{P} = (\mathbb{P}; \underline{0}, \sqsubseteq)$ where

(i) $\mathbb{P} = \mathbb{N} \cup \{\underline{0}, \underline{1}, \ldots, \underline{n}, \ldots\} \cup \omega$, where every $\underline{n}$ is called a *partial natural number* and every n is called a *total natural number*

(ii) $\underline{0} \sqsubseteq p$ for all $p \in \mathbb{P}$, is the least element of the domain

(iii) $\underline{n} \sqsubseteq n$, for all $n \in \mathbb{N}$

(iv) $n \not\sqsubseteq m$ for all $n, m \in \mathbb{N}$

(v) $\underline{n} \sqsubseteq \omega$, for all $n \in \mathbb{N}$

(vi) $n \not\sqsubseteq \omega$, for all $n \in \mathbb{N}$

In $\mathbb{P}$, the most interesting chains are of one of the forms:

(i) a finite chain ending in a partial natural number: $\underline{0}, \underline{1}, \ldots, \underline{n}$

(ii) a finite chain ending in a total natural number: $\underline{0}, \underline{1}, \ldots, \underline{n}, n$

(iii) the sole infinite chain of partial numbers, including no total number: $\underline{0}, \underline{1}, \ldots, \underline{n}, \ldots$

One should notice that w is the limit (lub) of the infinite chain of partial numbers.

2.3 Computations as constructions in domains.

A computation in a domain D may be seen as any finite (or, infinite) chain of elements $x_0, x_1, \ldots, x_n$ (respectively, $x_0, x_0, \ldots, x_k, \ldots$).

The state of a computation at a given moment t is given by the (partial) objects that have been constructed up to that moment. If, at time t, a computation has constructed a sequence of partial objects $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_t$, then x_t is the state of the computation at time t (assuming that x_0 was the

first partial object constructed by the computation at time 0).

The objects constructed by a finished computation are said to be its *products*, or *results*. If a computation has ended at time t, and up to that time it has constructed a sequence of partial objects $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_t$, then x_t is the *final result* of the computation.

If an infinite computation constructs a chain $x_0 \sqsubseteq x_1 \ldots \sqsubseteq x_k \sqsubseteq \ldots$ of objects, the result of that computation is the *limit* of such chain, given by the *least upper bound* of that chain in the domain: $\sqcup\{x_0, x_1, x_2, \ldots\}$. Results of infinite computations are, thus, *ideal* objects, *limits* of the computations that construct them.

Often, the initial element in a computation is $x_0 = \perp$, meaning the absence of any available result before the computation starts.

Example 2.3 Assume a program P that prints a string of characters of $\Sigma = \{a, b\}$ on a paper tape. The sequence $\perp, s_1, s_2, \ldots, s_t$ of partial strings that it has printed up to time t may be read as the construction of the partial result s_t that it has produced up to time t.

Whenever P stops printing, the last string in the sequence of strings that it produced is the final result of the computation of P. If P never stops printing characters on the paper tape, the sequence of strings it produces is infinte, its computation is infinite, and the result of such infinite computation is an infinite string belonging to D_Σ.

Notice that whenever we have times t, t' such that $t \leq t'$ we have $s_t \sqsubseteq s_{t'}$.

Example 2.4 Let P be a program that counts the sizes of lists. Given a list L, we have the following situations for the possible results of P acting on L:

(i) before P starts, it has counted no element in L, but P can tell us that there are at least 0 elements in L. So, P prints $\underline{0}$.

(ii) after counting for a while, but before reaching the end of L, P can tell us that there are at least k elements in L. So, P prints $\underline{k}$.

(iii) in case L is finite, P will eventually reach the end of the list and will be able to tell us that there are exactly $n = length(L)$ elements in the list. So, P prints n.

(iv) in case P never ends counting the elements in L, the best that we can get from P is an infinite sequence $\underline{0}, \underline{1}, \ldots, \underline{k}, \ldots$ where P informs us at each time the least number of elements it can assures us there is in L. We conclude that there are ω (infinitely many) elements in L

We notice that, strictly speaking, counting is a process that cannot be adequately modelled using just the set $\mathbb{N}$ of total natural numbers, because thus

one is not able to represent the intermediate situations where the counting is not finished yet, but some information is already available, namely, that at least a certain quantity of elements is present in the set being counted. Counting can only be faithfully modelled as a computational process in a domain containing "partial countings" (see [8,9] for further theoretical implications of computing with $\mathbb{P}$).

2.4 Constraints on computations in domains.

Domains and operations on domains are required to satisfy a set of constraints that keep them within the acceptable limits of what are, intuitively, "computable operations". The main such constraints are the following:

1) The computation of a result object from a given object should not reduce the structure of the produced object if additional parts are added to the initial object. That is: better inputs do not excuse worse outputs.
2) The computation of a finite result should not depend on operations acting on infinite objects. That is, finite outputs can only depend on finite parts of input objects.

Such requirements, that are only natural about a computing machine, are called *monotonicity* and *continuity*, respectively. Formally: computable functions $f : D_1 \to D_2$ should be *monotonic* ($x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$) and *continuous* ($f(\sqcup X) = \sqcup f(X)$, where X is any chain of objects in D).

Steps of computations of a program on a domain may be seen as functions $f : D \to D$. Thus, steps of computations of any program are required to satisfy the above two constraints, and programs should be developed so as to generate only computation steps with such characteristics.

Example 2.5 One sees that any process whose effect is the printing of strings of characters satisfy the constraints. The same apply to processes that generate strings of any other kind of objects, like partial results of counting processes.

2.5 Domains and the shift they induce in the notion of computation

The conceptual importance of domains is in that they equip the Theory of Computation with the notion of result of an infinite computation. As mentioned before, this notion is by itself a departure from the framework of CC.

To grasp the size of the conceptual shift induced on the Theory of Computation by the availability of the notion of *result* of an infinite (non-terminating) computation, confront that notion with Rogers' statement, in his discussion

of the mathematical notion of "effective procedure", in his classical book [22] (section 1.1, p.5):

> [...] should we require that, given any input and given any set of instructions P, we have some idea, "ahead of time," of how long the computation will take? We propose to make no such affirmative answer to the question [...]. We thus require only that a computation terminate after *some* finite number of steps [...].

Effectiveness of computations is taken, in CC, at the lowest possible level of conceptual richness: effectiveness is taken as deliverability of a finite result in finite time. It is interesting to contrast this requirement with Turing's original ideas: [27] was concerned with the computation of real numbers, and a successful computation was one that lasted forever, computing correctly all the digits of the infinite representation of its result.

The restriction of the Theory of Computation to the study of the recursive functions on integers, and the consequent possibility of considering that only finite computations are useful, seems to us to be connected to the effort of turning that theory into a regular academic subject, effort in which Kleene, Rogers and others were strongly engaged in the 1950's and 1960's. The consequence of that restriction, however, was that the study of recursive functions on real numbers (Turing's original aim) was put off the mainstream of the theory, along with the necessity of taking infinite computations into account.

As will be shown below, Domain Theory makes clear that *object construction* is a much richer notion than the strictly finite deliverability of results required by CC, because the *effectiveness of a construction* need not be constrained by finiteness, neither in the temporal, nor in the spatial sense. [5]

2.6 Domains and the shortcomings of the Classical Theory of Computation

The sole contemplation of some computational notions inherent to the Theory of Domains is enough to expose serious shortcomings in the classical notion of computation. Shortcomings that the contemplation of the notion of *interaction* makes even more overt. The first such shortcoming is:

> CC-1: *In CC, non-terminating computations are meaningless.*

The lack of a notion of limit, based on the notion of continuity that the

[5] Finiteness and finitarity are not synonyms. *Finiteness* concerns the total resources available to a computation: a finite computation can only involve a finite amount of resources. *Finitarity* concerns the finiteness of the resources used in each step of a computation: the sum total of the resources involved in a finitary computation may well be infinite. Developmental computations, as interactive computations, are conceived as finitary, not finite.

approximation order supports, prevents CC from being able to handle partial objects, even when the operational models that constitute its hallmarks (Turing machines, λ-calculus reductions, etc.) show wide open to any attentive eye the internal handling of such partial objects.

To see that, it's just a matter of looking at, for instance, the tape of a TM during a computation: the partial output objects that are being produced are there! What happens is that such objects cannot be output from the machine before the computation terminates, that is, the user of the machine is not *officially* allowed to make use of the partial results of the computation, before the computation terminates.

The second shortcoming of CC exposed by Domain Theory is:

CC-2: *CC performs input-output mappings, not constructions.*

The limitation to the computation of input-output mappings is the most serious shortcoming of CC exposed by the notion of interaction, as is well known from the research on *reactive systems* (e.g., [21]), and repeatedly stressed by Wegner & Goldin. This criticism is reinforced by Domain Theory. There is nothing in the notion of construction that requires that that sequence of aggregation steps be restricted so that just one single interaction step happens, with one initial complete object being given and one final object being (possibly) received back, and with everything that happens in-between concealed inside the constructing machine, inaccessible from outside.

One may say that construction steps are *input-output mappings* [11], thus Turing-computable if the construction is effective in the classical sense, and that a construction is nothing more than a succession of such Turing machine computation steps. This would picture domain constructions as *interactive processes* in the sense of Wegner & Goldin.

But the problem is that nothing in Domain Theory requires constructions steps to be Turing-computable. That is, nothing in the structure of domains limits constructions to be classical computations: Domain Theory points to a conceptual framework where CC appears as the lower bound of a wide range of possible notions of computability, compatible with domains as universes of object constructions.

In other terms, Domain Theory is compatible with a notion of *non-algorithmic computation*. That is, the third shortcoming of CC is:

CC-3: *In CC,* effective *is synonym of* algorithmic.

The iterated repetition of Turing-computable macro-steps, adopted in the

Interactive Turing Machine model of [34], and in its special case called Persistent Turing Machine [11], construes computations as iterated sequences of Turing-machine computations. In a sense, that model places itself in the framework of a *Turing-machine controlled* iterated repetition of Turing machine computations.

One could even conceive an interleaving of macro-steps performed by different machines, allowing for the *controlled* repetition of various Turing-machine programs, realized under the surveillance of a *supervisor Turing machine*, able to test a shared working tape to decide what particular Turing machine to activate in the next macro-step.

The notion of computation as construction in domains goes further than the interactive Turing machine models in the sense that it opens the possibility that *non-Turing controlled* sequences of *non-Turing computable construction steps* be considered *effective* in a concrete, physical symbol systems-based way. This possibility lies at the core of the notion of Developmental Computation.

3 A View of IC

In this section, we summarize the arguments presented in [5] on the centrality of the concept of Interaction to the notion of Developmental Computation.

3.1 Interaction and Church's Thesis

Since it's appearance, the so-called Church's Thesis has often been subject to dispute, usually by people unsatisfied with its normative status, implying the possibility of finding interesting things that computers cannot do.

The thesis in [5] has not been written as another attempt to show that Church was wrong. The intention is quite another, namely, to make explicit the limits of the validity of Chuch's Thesis, the limits within which Church is right.

Church's Thesis was not built in the air [6]: there are strong conceptual and epistemological foundations sustaining it, the most important of which is the following:

> *Computations are computations of mathematical functions, that is, of operations acting on completely defined input objects.*

The hallmark of Classical Computation, as captured by Church's Thesis, is that a computation is the execution of a series of operations on an input object, taken from a well determined domain where all objects are completely defined,

[6] See [13] for another account for the origins of Church' Thesis.

producing as a result an object in a well determined output domain, where all objects are also completely defined. Nothing coming from the computation process can alter the nature and structure of the objects in both the input and output domains, nothing can alter those domains, and nothing can alter the very input object, while it is being processed.

The fourth shortcoming of CC is:

CC-4: *A classical computing machine operates as a closed system, while it is computing, and thus is unable to alter its input objects, as such alterations require the active participation of the machine environment.*

No classical Turing machine can model a computational process where the input object is altered while the computation is going on, because all classical Turing machines require that the input object be completely defined before the machine can start its operation. No algorithm, in the sense of Classical Computation, can be applied to input data, before such input data is completely defined.

In classical Turing machines, input-output operations are not interactive: they are respectively supposed to happen *before the computation starts* and *after the computation finishes*. Computations where input objects are subject to modifications dependent on the construction process of output objects are computations where *interactive input-output operations* happen.

CC models can only be extended with interactive input-output operations at the expense of dismissing the essential commitment that Church, Turing, Kleene, Post and others had to the solution of Hilbert's *Entsheidungsproblem* (Decision Problem). Hilbert's problem raised the question if every mathematical problem could to be solved by a *mechanical*, *non-creative* procedure, performed by a mathematician thinking alone, in complete isolation from everyone else, doing calculations only with the help of paper and pencil, as vividly pictured by A. Turing [27]. And, Turing and all the others were strongly committed to keeping their models of computation within the bounds suggested by the procedural model proposed by Hilbert.

Interactive Computation means more than just the possibility of having input-output objects being exchanged during the operation of the machine. It means also that the input objects *need not be completely defined* before the computation starts, so the the output objects produced by the computation need not be pre-determined by a strictly functional dependence (in the mathematical sense) of the input objects. Both input and output objects may become *dynamically, and incrementally, defined* as the computation goes on.

The essential feature allowing for interaction is the integration of the *en-*

vironment as a true participant of the computation process, playing an *active role* in the process. If the environment is an active participant in the computation process, the computation is no more mechanical, in Hilbert's sense. That is, it is no more effective in the restricted sense that CC assigns to such term. But, sure, it may still be effective, and mechanical, in a wider, physical symbol system-based sense.

3.2 Interactive Computation and the von Neumann computer

The case, however, is not that some (possibly remote) idea of a (possibly hard to conceive) domain structure, modelling the computations of a (possibly futuristic) very special kind of computer, may someday reveal a (possibly weird) example of object construction that can not be performed by classical computations.

The case is that even everyday computers – the so-called von Neumann computers [3] – demonstrate, through interaction, that CC is a very restricted notion of computation. The input-output behavior of von Neumann computers, allowing for *interactive computations*, shifts the domain of computation of such computers to areas that are very far from that contemplated by Church's Thesis. The simple fact is that:

Turing machines are perfect operational models of von Neumann computers only when von Neumann computers are operating in a non-interactive way, that is, when they are computing mathematical functions. Only when operating in such special, restricted modes of operation, von Neumann computers are subsumed by Church's Thesis.

Besides introducing the environment as an active participant in the computation, and giving it the power to influence the construction process of output objects by affecting the structure of the input objects, the architecture that von Neumann designed for the programmable computer [3] supports other important features not present in CC machine models.

von Neumann computers stretch Turing's notion of stored program to an extent that could not be anticipated from it. Turing's notion of stored program (in a universal Turing machine) lies on the possibility of interpreting objects stored in memory (tape) either as data or as program instructions, depending on the context in which the computer's control unit accesses such objects.

By incorporating the notion of *stored program* (and the associated feature of the *duality of data and program*) in his model of computers, and by combining such feature with the possibility of *dynamically entering input objects* during a computation, von Neumann introduced a possibility that profoundly

departs from the very essence of the computational possibilities allowed by Turing machines, namely, the possibility of *a program being dynamically modified* by the environment, during its computation.

That is, by supporting the *duality* between data and program instruction, concerning objects stored in memory (so that objects that are input while a computation is being carried on are allowed to be interpreted not only as data, but also as program instructions), and by combining this possibility with the integration of the *environment as an active participant* in the computation process, von Neumann computers allow computations where not only *input and output objects are not pre-determined*, when the computation starts, but also where *the very program that will control the computation* is not pre-determined.

Any running program can aggregate new instructions, delete existing instructions (substituting them by null instructions), or have any of its instructions substituted by others, by transforming data interactively input to the machine into such instructions. In fact, it is precisely this feature of the interactive modification of the structure and behavior of the program running on the computer that allows for the notion of *operating system*, that only von Neumann computers can support and that makes them really *general purpose* computers. [7]

So, in von Neumann computers, computers and environments (users) are allowed to *cooperate*, and the possibility of their cooperation amounts to the introduction of the notion of *joint computation*: to understand what a computer will do in a given situation, to be able to predict what will happen after it starts a computation, it is indispensable to look at both the computer and the environment, because what will happen in the computation (and, in general, during the whole life-time of the machine) depends on the articulation of the behavior of both such elements. In other words:

The possibility of the interactive modification of running programs makes of von Neumann computers situated machines*, that is, interactive machines whose behaviors can only be fully understood in connection with the behaviors of the environments where they are situated.*

Interaction and situatedness, embodied in von Neumann computers, show that even if machine structures are fixed, programs and computations need not be so: machines with fixed structured can compute with programs that

[7] It seems that, in spite of all efforts to develop innovative non-von Neumann computers, practically no one of them turned out to be able to support full operating systems, thus turning out to become just special purpose computers.

can be modified interactively, that is, whose structure may change according to the interaction between computer and environment, which thus determine computational processes that can be modified, and thus controlled in their structures and goals, interactively.

We note, on the other hand, that Wegner & Goldin's examples of programs for interactive machines have not yet contemplated that feature of interactive modification of running programs, although the feature can easily be introduced in their programs, since they have universal interactive machines [11].

Building on such double interactive computational power, already introduced in the field of Computation by von Neumann in the early 1950's, one can easily grasp the possibility of developmental computational processes establishing themselves in *developmental computer models* that can possibly emerge from physical symbol systems able to overcome structural limitations of the von Neumann architecture, as indicated below.

This central result of [5] shows that computer technology has always been based on the notions of IC, and that Interaction is not a late novelty introduced by the development of computer technology (as suggested by Wegner [30]).

A current problem, then, is to characterize this wider notion of effectiveness, that surpasses the narrow sense of effectiveness of Classical Computation, and is able to encompass at least the undeniable effectiveness of von Neumann computers.

The work of Wegner & Goldin is, of course, a fundamental stepping stone in this direction.

4 Developmental Computation

The rationale behind Developmental Computation (DC) is that the principles that regulate complex organized dynamical systems are the same, independently of the nature of the elements that compose such systems, so that there should be no formal difference between the general principles that regulate the internal dynamics of biological organisms and those that regulate complex, interactive, developmental physical symbol system-based computational systems.

Thus, the essential step towards DC is not a technological step, because the historical analysis of Computer Science in [5] was able to show that the main technological ingredients for that step have all been there, for decades. The essential step towards DC is a conceptual, epistemological step. It consists in the superseding of the basic notions of the CC (like machines, programs and algorithmic computations), by the basic notions of a general theory of complex dynamical systems (like organization, development and adaptation).

To be able to perform such step, one such general theory of complex organized dynamical systems should be adopted. In [5], the general theory of biological organisms that Jean Piaget exposed in [19] was adopted. That theory, more than a general biological theory, is in fact a *general cybernetic theory* of complex, interactive and developmental dynamical systems, and that is the real reason for its adoption.

This section introduces developmental machines and explains two of the *general developmental notions* that are critical to DC, namely, *equilibration* (development) and *adaptation*. It also gives an example of developmental computation, showing that DC is more encompassing than IC.

4.1 Developmental machines

By exploring the possibility of interactive input-output, a situated computer can exchange objects with the environment, objects that can be interpreted as both data and program, depending on the interpretation rules determined by the structure and functioning of the computer's control unit.

But the consideration of this very possibility of having non-predetermined programs controlling non-predetermined constructions of output objects from non-predetermined input objects, according to a computation jointly governed by the computer and the environment, exposes an assumption implicit in Classical Computation, that still permeates the von Neumann computers: the computer operates under a fixed control structure, with a fixed set of primitive operations.

No matter how computer and environment interact, no matter how input and output objects evolve under this interaction, no matter how the environment changes due to its open nature, the control structure of the computer is fixed, immune to the vagaries of the computation it controls.

Interactivity and situatedness in von Neumann computers are not enough to allow such machines to explore a further possibility, that could leverage the power of computation to a yet higher level, namely, the possibility of having the very structure of the computer change as the computation goes on, thus extending IC with developmental features that go beyond interaction.

The consideration of von Neumann computers as situated computers immediately exposes a fifth shortcoming of CC:

CC-5: *Classical computing machines have fixed control structures.*

DC concerns systems where the control structure of computers can be modified dynamically. In particular, it concerns computational systems where

the modification of the control structure of machines can be understood as *development*.

The question that immediately poses itself is, then, how the fixity of the computer structure can be overcome? One possible answer is to mimic the solution found by biological organisms: to arrange that elements of the system structure – material elements – be exchanged with the environment while the system is operating. Programming models inspired by molecular biology (e.g., [4]) will certainly serve as sources of answers for such question.

That is, the full exploitation of the possibilities of the notion of computation requires that one envisages a computing machine that is able to exchange not only *information objects* with its environment, but also *material objects*, so that such *material exchange* can support processes of *machine development*.

In a situation of *joint computation* involving computer and environment, and with the possibility of *material exchanges* between them, both the computer and the environment are not pre-determined in their structures, with the consequence that even the control rules of the computer's control unit need not rest fixed during the computation. We call *developmental computation* any computation where the structure of the computer is able to develop as the computation goes on, and we state the following requirement for DC:

DC-1: *Developmental computers may vary their control structures while computing, by exchanging material objects with their environments.*

With the help of domains, DC can be seen as involving a special kind of construction, namely, the construction of the computing machine itself. This allows the discrimination of two aspects of computations, when seen as constructions: on the one hand, a computation constructs the objects handled by the computing machine; on the other hand, a computation can construct the machine itself (if it is a developmental machine).

4.2 Machine development and machine autonomy

In the context of DC, the notion of *purpose of computation* has to be rethought. For if the construction of objects by machines can be seen as an attempt to satisfy needs or requests from the environment (the users of the machine), what could be the purpose of the construction of the machine itself?

The latter question seems to accept two kinds of answers. First, one can see that the construction of the machine may serve some purposes of the environment (users), since more developed machines may be expected to perform better services. The second, somewhat unexpected answer, is that the

construction of the machine may serve some purpose of the machine itself.

The latter answer is surely an epistemological divisor, separating two different notions of machines: *autonomous machines*, that is, machines endowed with goals that are of their own; and *heteronomous machines*, that is, machines that have no goals of their own, its working being dedicated essentially for the fulfilment of goals of the environment (see [6] for an attempt to give a functional definition of autonomy in the context of multi-agent systems).

The question if a computing machine is possible, which is autonomous and yet is not a living being existing on its own is yet unsolved. We thus have to proceed noting that everything that follows is based on an unverified assumption, namely, the assumption that there may exist computing machines which are autonomous and yet are not living beings existing by their own. [8]

For such computing machines, the process of machine construction should be a *development*, guided by internal principles, devoted to modify the machine in order to make it function in a better way, for the sake of the machine functioning itself.

4.3 Why development is a material, not an informational concept

The question may be posed why development implies material exchanges with the environment. The answer is that, since it is not possible to have computations performed by non material structures (even when performed in human minds, it seems that computations need the support of a physical system, namely, the brain) it is clear that the power of every computational process is limited by the computational power of the physical system performing it.

Thus, if development is to happen in a computing system, implying an ever growing capability of performing new computational processes, such development implies the correlative development of the underlying physical system.

Simulation is no solution to the problem: the limits of the simulated processes are determined by the limits of the simulator. There is no way, of course, for a simulator to produce a simulated system more powerful than the simulator itself.

Even metalevel processes in metalevel architectures are no solution either: hierarchies of metalevel languages are doomed to be supported by a definite computing system, whose computational power sets the limits to any possible development process that a meta-level program may intend to induce on a given object-level program.

[8] We adopt such strong distinction between computing machines and living beings because otherwise, if one assumes no essential difference between those two kinds of entities, one would end by embedding Computer Science in Biology, which is not our intention.

Only if the computer itself is able to physically develop, to really create operational novelties in its control structure, is development a concept that can be really introduced in computational processes, in the form of creation of novel operational and storage structures that were not possible before.

As Biology teaches, development of a physical system requires that the system exchanges material elements with its environment. The exchange of information is insufficient for the purpose of developing the computational processes, because information can govern the realization of operations and occupation of storage with information, but cannot itself create neither the physical operators needed for the operations, nor the physical structures needed for the storage of data.

4.4 Why information does not exhaust the possible contents of interactions

Classical Computation sacred the conception that computation is a logical, informational process, that is independent of the physical nature of the system performing it. Such view, adequate as it is to face the *Entscheidungsproblem* posed by Hilbert, does not conform to the engineering view of computers as concrete machines whose performances dissipate energy and weaken the physical quality of the materials used to build them.

If information is conceived as a *pattern* on a physical substrate, the exchange of patterns between two systems does not require the exchange of material elements between them. On the other hand, material exchanges can happen between two systems without necessarily information being exchanged between them: it is enough for that that such exchanges occur at a level where no disturbance in the informational interface between the systems can be noticed by any of them.

However, systems that exchange material elements between them, besides exchanging information, are systems that can potentially interact at much richer level them systems that only exchange information: they can directly influence each other development.

Of course, if a system is a developmental one, its development can be influenced by information exchanged with its environment, because it is well possible that the system's developmental mechanisms be regulated by information contained in the system. That is, the material development of a system may clearly be influenced by its informational exchanges. But it should be clear that information exchanges do not support development by their own.

Thus, a better picture of a developmental computing machine is one that, besides exchanging information (both programs and data) exchanges material elements with its environment.

4.5 Equilibration and Adaptation: the foundation of DC.

Classical Computation is based on three fundamental concepts, namely, *machines, programs and computations*, which are adequate to support *heteronomous* object construction processes. DC is meant to surpass Classical Computation in the sense that machines are promoted from being *subordinate objects* in the environment to being *actors* in it, from being *instruments* of the users to being their *collaborators*. DC, thus, concerns object construction processes that are oriented by purposes that belong to the machine itself. This implies that the fundamental concepts of DC should support such internal purposes, and allow for object construction processes that are able to articulate internal and external purposes.

Analyzing the general biological and psychological models presented by Piaget [18,19,20], including his models of development of biological and cognitive structures, we think that two fundamental concepts should be incorporated into computing machines to leverage their developmental processes, namely, a process of internally regulated object construction, called *equilibration*, and a dynamical concept of *adaptation* of the machine to the environment.

4.6 Equilibration.

A straightforward, informal definition is:

DC-2: *Equilibration is the process of self-regulated construction of objects.*

We note, first, that self-regulated constructions are not a new idea in the Theory of Computation. John von Neumann himself explored them, in order to define computing machines with reliability features that approximate that of the human brains [28]. Following Piaget, we construe *equilibration* as a process operating through a set of *development stages* of the computing machine. At each development stage, the machine is able to construct particular kinds of internal and external objects, in certain ways, determined by the set of operations it has available for such purpose, at that stage.

Development stages are ordered according to the degree of their development, determined by some measure of the richness of the set of operations for object constructions available at that stage. When development is seen as a construction in a domain, the ordering of the stages of development is given by the approximation relation of the domain.

The equilibration process has two dimensions, namely, a *diachronic dimension* and a *synchronic dimension* [20]. The *diachronic dimension* is the one that regulates the development process as such. That is, it regulates the

way the machine changes from one development stage to the next development stage. *Major equilibration* is the name applied to denote the diachronic process of equilibration. The *synchronic dimension* is the one responsible for regulating the construction of the internal and external objects, at each stage. *Minor equilibration* is the name applied to such process.

We further notice that the notion of *object constructed in a development* includes the operators that realize the internal dynamics of the machine, and thus includes the object constructors themselves. That is, Piaget's notion of equilibration encompasses the notion of *autopoiesis*, by Maturana & Varela [16].

4.7 Adaptation.

Adaptation is correlative to equilibration, in the sense that the equilibration process produces better adaptation resources to the computing machine, while dysadaptation acts as an indicator of the need of new steps in the equilibration process. As the machine develops through its set of development stages, under the supervision of the adaptation process, it gets more and more adapted to the environment, as richer construction processes of internal and external objects become possible at each new stage, due to the richer set of object constructors that become available at each new stage.

Adaptation is defined in terms of two ancillary notions:

(i) *Assimilation*: the process by which the machine is able to apply to internal and external objects the set of its currently available operations, in order to achieve its current goals.

(ii) *Accommodation*: the process by which the machine is able to adjust its current set of operations, in order to make them better applicable to internal and external objects, in order to achieve its current goals.

Adaptation is thus defined as:

DC-3: *Adaptation is the situation where every required assimilation is possible, because every required operation on a given environment can be performed, and every required accommodation is possible, because every required adjustment in the internal and external operations can also be performed.*

Major equilibration furthers the stages of adaptation, because more internal and external objects can be handled with more sophisticated operations. Thus, major equilibration is the *central factor of development* [20]. On the other hand, the progress of adaptation requires ever more sophisticated stages of development, that can only be achieved through major equilibration.

The usual alternative notion of adaptation, conceived as a process and not as a situation as defined here, is clearly due to the usual way of taking the term "adaptation" to mean what we have called here "accommodation".

5 Sketch of a Formal Framework for the Theory of DC

The distinction between *development* and *evolution* is based on the idea that development concerns *individuals* while evolution concerns *sets of individuals*, and can be formalized by requiring that development happens in a domain, so that the sequence of stages of a development (construction) guarantees the increasing richness of the operational structures of those stages, while evolution may be defined without that requirement.

The main purpose of the following preliminary formal framework is just to indicate the basis on which we think it is possible to formally prove that DC is a more encompassing notion than IC.

Let M be a developmental computing machine, T be a discrete-time temporal structure. Then, define:

• *Development stages:*

(1) D_M is the set of possible stages of development of M

(2) D_M^t the set of possible stages of development of M at time $t \in T$

(3) $D_M^\tau = \bigcup_{t\in\tau} D_M^t$ the set of possible stages of development of M for $\tau \subseteq T$

(4) $D_M^0 \in D_M$ the set of possible initial stages of development of M

• *Machine operations:*

(5) $op(d)$ the *operational structure* of stage d, that is, set of (internal and external) machine operations available at development stage d

• *Approximation relation:*

(6) $\sqsubseteq \subseteq D_M \times D_M$ the approximation relation between development stages of M, so that $d \sqsubseteq d'$ iff $d \in D_M^t$, $d' \in D_M^{t'}$, with $t \leq t'$, and $op(d) \subseteq op(d')$ (development increases the richness of the operational structure).

• *Development steps:*

(7) $\Delta_t^{t+1} \subseteq D_M^t \rightarrow D_M^{t+1}$ the set of possible development steps at time t, defined so that every $\delta_t^{t+1} \in \Delta_t^{t+1}$ guarantees the inclusion relation between the operational structures of development steps, that is, $op(d) \subseteq op(\delta_t^{t+1}(d))$, for every $d \in D_M^t$

• *Development relation:*

(8) $\Delta_t^{t+n} = \Delta_{t+n-1}^{t+n} \circ \Delta_{t+n-2}^{t+n-1} \circ \ldots \circ \Delta_{t+1}^{t+2} \circ \Delta_t^{t+1}$ the development relation (possibly, a function) operating from t to $t+n$, so that for all $d, d' \in D_M$ it happens that $d \sqsubseteq d'$ iff $d \in D_M^t$ and $d' \in D_M^{t'}$, for $t, t' \in T$ with $t < t'$, and $(d, d') \in \delta_t^{t'}$ for $\delta_t^{t'} \in \Delta_t^{t'}$; so that if $(d, d') \in \delta_t^{t'}$ then $op(d) \subseteq op(d')$.

Let E be the evolutive environment of a developmental computing machine M. Define:

• *Evolution stages*:

(1) E_M the set of possible evolution stages of the environment E of M

(2) E_M^t the set of possible stages of evolution of the environment at $t \in T$

(3) $E_M^\tau = \bigcup_{t\in\tau} E_M^t$ the set of possible stages of evolution of the environment during the period $\tau \subseteq T$

(4) $E_M^0 \in E_M$ the set of possible initial stages of evolution of the environment

• *Environments operations*:

(5) $op(e)$ the set of operations that the environment is able to apply on the machine at evolution stage $e \in E_M^t$

• *Approximation relation*:

(6) $\sqsubseteq \subseteq E_M \times E_M$ the approximation relation between stages of evolution of the environment, so that $e \sqsubseteq e'$ iff $e \in E_M^t$, $e' \in E_M^{t'}$, with $t \leq t'$ (with no requirement of enrichment of the environment's operational structure)

• *Evolution steps*:

(7) $\Upsilon_t^{t+1} \subseteq E_M^t \rightarrow E_M^{t+1}$ the set of possible environment evolution steps at t

• *Evolution relation*:

(8) $\Upsilon_t^{t+n} = \Upsilon_{t+n-1}^{t+n} \circ \Upsilon_{t+n-2}^{t+n-1} \circ \ldots \circ \Upsilon_{t+1}^{t+2} \circ \Upsilon_t^{t+1}$ the environment evolution relation operating from t to $t+n$.

Developmental machines and their evolutive environments must interact, if the machine is to operate in the environment. The idea of interaction that underlies the following formalization is that any *interaction step* is a coordination of two operations, one performed by the machine, the other performed by the environment. For such coordination to occur, the two operations are required to be *coherent* (or, *compatible*) in some sense.

Let M be a developmental computing machine, E its evolutive environment, $d \in D_M^t$, and $e \in E_M^t$. Then define:

• *Interaction coherence*:

(1) $\approx \subseteq op(d) \times op(e)$ the coherence relation between operations of the development stage $d \in D_M^t$ and operations of the evolution stage $e \in E_M^t$, so that the meaning of $o_d \approx o_e$ is that $o_d \in d$ and $o_e \in e$ are coherent (or, compatible) with each other

• *Adaptation of development stages*:

(2) $\bowtie \subseteq D_M^t \times E_M^t$ the adaptation relation between the set of development stages D_M^t and the set of evolution stages E_M^t, so that $d \bowtie e$ iff $\forall o_d \in d.\exists o_e \in e.o_d \approx o_e$ and $\forall o_e \in e.\exists o_d \in d.o_d \approx o_e$

• *Adaptation of machines*:

(3) $\bowtie^\tau \subseteq Mach \times Env$ the adaptation relation between developmental comput-

ing machines and environments, during the period $\tau \subseteq T$, so that $M \bowtie^{\tau} E$ iff $\forall t \in \tau . \forall d \in D_M^t . \forall e \in E_M^t . d \bowtie e$.

6 A tentative model of machine development

For the sake of illustration of the idea of machine development, we introduce here a tentative model of steps of machine development. The model is tentative because it is incomplete in many senses, as will be explained. However, it seems to be clear enough to allow the grasping of what we think is essential about the idea machine development.

6.1 A tentative model of development step

Our tentative model of development step is an attempt of formalization of the basic model of system development informally introduced by Piaget in [20].

Definition 6.1 *Let $D_M = (S; Op)$ be a stage of development of machine M, where S is the set of states of M at that stage of development, $Op = \cup_n Op_n \subseteq S^n \to S$ ($n \in \mathbb{N}$) is the set of operations that M can perform at that stage. A* basic development step *is a transformation $\delta_\rho(D_M) = D'_M = (S'; Op')$ with:*

(i) *$S' \subseteq \wp(S)$ such that $\rho(S) = \rho(S') = S'$, that is, S' contains exactly all subsets of S that satisfy ρ*

(ii) *$\rho : \wp(S) \to \wp(S)$ is a restriction on $\wp(S)$, accounting for the choice of the states for the new development stage, among all possible new states*

(iii) *$Op'_n \subseteq S'^n \to S'$ with $o'(S'^n) = \{o(\overline{x}) \mid \overline{x} \in S^n\}$ for all $o' \in Op'_n$*

The central issue in this definition of development step is the employment of the subset operator $\wp$ to achieve two developmental results that are essential to Piaget's model of development:

(i) the augmentation of the set of possible states, in the new stage

(ii) the preservation of the previous set of possible states in the new set, by their inclusion in the new development stage as singleton sets

(iii) the use of a set of criteria ρ to exclude state possibilities that do not conform to the way the new development stage is construed

Combining those three features gives us a simple model of development step that may be tentatively held for the initial studies in this area. The model should not be taken as final, precisely because it lacks the most essential features that we have been discussing in this paper: it is not a model of an *interactive* development step, and so it does not encompasses exchanges between the machine and the environment (with the possible exception of the

situations in which the restrictions ρ come from the environment).

6.2 A simple example of machine development step

We now introduce a simple example of step of machine develop, and then confront it with what would be expected from a more complete example .

Example 6.2 Let's consider a simple machine, with memory addressed by non-negative integers and able to perform arithmetic operations on integers:

Let $M = (M; +, -, \cdot, r, s)$ be such that

(i) $M = \mathbb{N} \to \mathbb{Z}$ is the set of states of the machine memory

(ii) $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is the integer addition operation

(iii) $- : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is the integer subtraction operation

(iv) $\cdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is the integer multiplication operation

(v) $r : M \times \mathbb{N} \to \mathbb{Z}$ is the read operation, that reads values from the memory

(vi) $s : M \times \mathbb{N} \times \mathbb{Z} \to M$ is the operation that stores values in the memory

Typical programs in $\mathcal{M}$ are:

(i) $s(M_0, 2, r(M_0, 0) + r(M_0, 1))$
that stores at position 2 the sum of the values at positions 0 and 1

(ii) $s(s(M_0, 2, r(M_0, 0) + r(M_0, 1)), 4, r(M_0, 2) \cdot r(M_0, 3))$
that extends the previous program by multiplying the result stored at position 2 with the value at position 3, storing the result at position 4.

We now let M develop by performing a series of developmental operations, and becoming a new machine $M' = (M'; \boxplus, \boxminus, \boxdot, r', s')$:

(i) $\mathcal{M}'$ is able to store intervals of integers:
$M' = \mathbb{N} \to \mathrm{I}(\mathbb{Z})$

(ii) $\mathcal{M}'$ is able to realize interval operations:
  (a) $\boxplus : \mathrm{I}(\mathbb{Z}) \times \mathrm{I}(\mathbb{Z}) \to \mathrm{I}(\mathbb{Z})$ is the integer interval addition operation
  (b) $\boxminus : \mathrm{I}(\mathbb{Z}) \times \mathrm{I}(\mathbb{Z}) \to \mathrm{I}(\mathbb{Z})$ is the integer interval subtraction operation
  (c) $\boxdot : \mathrm{I}(\mathbb{Z}) \times \mathrm{I}(\mathbb{Z}) \to \mathrm{I}(\mathbb{Z})$ is the integer interval multiplication operation
  (d) $r' : M' \times \mathbb{N} \to \mathrm{I}(\mathbb{Z})$ is the read operation, that reads integer interval values from the memory
  (e) $s' : M' \times \mathbb{N} \times \mathrm{I}(\mathbb{Z}) \to M'$ is the operation that stores integer interval values in the memory

Typical programs in $\mathcal{M}'$ would be:

(i) $s'(M'_0, 2, r'(M'_0, 0) \boxplus r'(M'_0, 1))$
that stores at position 2 the sum of the intervals at positions 0 and 1

(ii) $s'(s'(M'_0, 2, r'(M'_0, 0) \boxplus r'(M'_0, 1)), 4, r'(M'_0, 2) \boxdot r'(M'_0, 3))$
that extends the previous program by multiplying the interval stored at position 2 with the interval at position 3, storing the result at position 4.

It is not difficult to see that the development step from $\mathcal{M}$ to $\mathcal{M}'$ conforms to the definition of development step given previously.

7 Conclusion: related work and the current frontier of DC

The first results on DC were established even before the Ph.D. work in [5] was officially begun. They supported the M.Sc. dissertation by Martín Escardó [9], where the computability of recursive functions on partial (lazy) natural numbers were analyzed. Partiality (laziness) of natural numbers arises from the allowance of an interactive input of such numbers through successive approximations [2], so that recursive functions on them realize a model of IC. A short report about that work appeared in [8].

The second author made use in her thesis [7] of the idea of *construction independent processes* to define a structure where real numbers and intervals of real numbers are constructively obtained. The structure, called *bi-structured coherence spaces* is based on Girard's coherence spaces [10]. It is said to be a *bi-structure* because, besides the ordered-structure of the approximation relation, that regulates the construction of real numbers and intervals, it also supports the algebraic structure of the operations on real numbers and intervals, established on the basis of the usual ordering of numbers. The operations of such algebraic structure are defined so that they are all construction independent. An effort to work out the notion of computational systems with autonomous goals is going on [6].

In this paper we have presented a summary of our ideas about DC and how it can open new ways to further the evolution of computing machines. The full theory of DC is yet to be developed. We hope that the formal framework introduced above can give some indication on at least the main features that we expect the theory to have. Of course, the main issue that still has to be clarified is the notion of material exchange between machines and environments, so better models of developmental can be obtained. DC introduces also many other issues, the most central of them being:

• *Axiology*: the idea that computing machines have goals of their own implies the idea that computing machines have *values* of their own. The understanding of what such values may be, and how they should give rise to rules to which computing machines would adhere by their own, is a major

problem, that should be solved prior to the establishment of the Theory of Developmental Computation.

• *Teleonomy*: the idea that the development of a computing machine should proceed according to principles that are internal to the computing machine is connected to the axiological problem just mentioned, but concerns specifically *developmental goals and rules*. The problem of teleonomy is, thus, the central problem of DC.

Acknowledgement

To the anonymous referees, for very useful comments.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Claredon Press, 1994.

[2] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 1998.

[3] A. W. Burks, H. H. Goldstine, and J. v. Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Part I, vol.1, 1946. In A. H. Taube, editor, *John von Neumann - Collected Works*, pages 34–79. MacMillan, New York, 1963.

[4] L. Cardelli. Bioware languages. In A. Herbert and K. S. Jones, editors, *Computer Systems: Theory, Technology, and Applications – A Tribute to Roger Needham*, pages 59–65, Berlin, 2004. Springer.

[5] A. C. R. Costa. *Machine Intelligence: Sketch of a Constructivist Approach*. PhD thesis, Programa de Posgraduação em Computação, UFRGS, Porto Alegre, RS, Brazil, October 1993. In Portuguese.

[6] A. C. R. Costa and G. P. Dimuro. Agent drives and the functional foundation of agent autonomy. ESIN-UCPel, 2004. To be submitted.

[7] G. P. Dimuro. *Bi-structured Coherence Spaces and the Construction of Real Numbers and Intervals of Real Number*. PhD thesis, PPGC-UFRGS, Porto Alegre, Brazil, 1998. In Portuguese.

[8] M. H. Escardó. On lazy natural numbers with applications to computability theory and functional programming. *ACM SIGACT News*, February 1993.

[9] M. H. Escardó. Partial natural numbers. Master's thesis, CPGCC/UFRGS, Porto Alegre, 1993. In Portuguese.

[10] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 59:1–102, 1987.

[11] D. Goldin, S. Smolka, P. Attie, and E. Sonderegger. Turing machines, transition systems, and interaction. *Information and Computation*, 194(2):101–128, 2004.

[12] D. Goldin, S. Smolka, and P. Wegner. Turing machines, transition systems, and interaction. *Electronic Notes in Theoretical Computer Science*, 52(1), 2001.

[13] D. Goldin and P. Wegner. The Church-Turing Thesis: Breaking the Myth. In *CiE 2005 – Computability in Europe*, Amsterdam, June 2005.

[14] A. Jung. Domains and denotational semantics: History, accomplishments, and open problems. *Bulletin of ETAPS*, 1996.

[15] S. C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, N.Y., 1952.

[16] H. Maturana and F. Varela. *Autopoiesis and Cognition - the realization of the living*. D. Reidel, Dordrecht, 1980.

[17] A. Newell. Physical symbol systems. *Cognitive Science*, 4:135–183, 1980.

[18] J. Piaget. *Introduction à l'Épistémologie Génétique*. PUF, Paris, 1950.

[19] J. Piaget. *Biology and Knowledge: an essay on the relations between organic regulations and cognitive processes*. UCP, Chicago, 1971.

[20] J. Piaget. *The development of thought : equilibration of cognitive structures*. Viking Press, New York, 1977.

[21] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In W. Brauer, editor, *ICALP185 – 12th Int. Colloq. on Automata, Languages, and Programming*, pages 15–32. Springer-Verlag, 1985.

[22] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[23] D. S. Scott. The lattice of flow diagrams. In E. Engeler, editor, *Symp. on semantics of algorithmic languages*, pages 311–366. Springer-Verlag, 1971, 1971.

[24] D. S. Scott. Lattice theory, data types and semantics. In R. Rustin, editor, *NYU Symposium on Formal Semantics*, pages 64–106, New York, 1972. Prentice-Hall.

[25] D. S. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[26] D. S. Scott. Domains for denotational semantics. In *Proc. ICALP'82*, pages 577–613, Berlin, 1982. Springer-Verlag. LNCS, vol. 140.

[27] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936.

[28] J. von Neumann. *The Computer and the Brain*. Yale Univ. Press, New Haven, 1958. (2nd. ed., 1967).

[29] P. Wegner. Machine models and simulations. In Wegner Agha and Yonezawa, editors, *Research Directions in Concurent Object-Oriented Programming*. MIT Press, 1993.

[30] P. Wegner. Why interaction is more powerful then algorithms. *Comm. of the ACM*, May 1997.

[31] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, Feb. 1998.

[32] P. Wegner. Towards empirical computer science. *The Monist*, Spring 1999.

[33] P. Wegner and D. Goldin. Interaction, computability, and Church's Thesis, 1999.

[34] P. Wegner and D. Goldin. Mathematical models of interactive computing, 1999.