A Semantics for Hybrid Iteration

Sergey Goncharov
Lehrstuhl für Theoretische Informatik, Friedrich-Alexander Universität Erlangen-Nürnberg, Germany
sergey.goncharov@fau.de

Julian Jakob
Lehrstuhl für Theoretische Informatik, Friedrich-Alexander Universität Erlangen-Nürnberg, Germany
julian.jakob@fau.de

Renato Neves
INESC TEC (HASLab) & University of Minho, Portugal
nevrenato@di.uminho.pt

Abstract
The recently introduced notions of guarded traced (monoidal) category and guarded (pre-)iterative monad aim at unifying different instances of partial iteration whilst keeping in touch with the established theory of total iteration and preserving its merits. In this paper we use these notions and the corresponding stock of results to examine different types of iteration for hybrid computations. As a starting point we use an available notion of hybrid monad restricted to the category of sets, and modify it in order to obtain a suitable notion of guarded iteration with guardedness interpreted as progressiveness in time – we motivate this modification by our intention to capture Zeno behaviour in an arguably general and feasible way. We illustrate our results with a simple programming language for hybrid computations and interpret it over the developed semantic foundations.

2012 ACM Subject Classification Theory of computation → Timed and hybrid models

Keywords and phrases Elgot iteration, guarded iteration, hybrid monad, Zeno behaviour

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.22

Related Version A full version of the paper is available at [13], https://arxiv.org/abs/1807.01053.

1 Introduction
Iteration is a basic concept of computer science that takes different forms across numerous strands, from formal languages, to process algebras and denotational semantics. From a categorical point of view, using the definite perspective of Elgot [10], iteration is an operator

\[ f : X \rightarrow Y + X \]

\[ f^1 : X \rightarrow Y \]  

1 Research supported by Deutsche Forschungsgemeinschaft (DFG) under project GO 2161/1-2.
2 Research supported by ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within projects POCI-01-0145-FEDER-016692 and 02/SIACT/2017.
that runs the function $f$ and terminates if the result is in $Y$, otherwise it proceeds with the result repetitively. One significant difficulty in the unification of various forms of iteration is that the latter need not be total, but can be defined only for a certain class of morphisms whose definition depends on the nature of the specific example at hand. In process algebra, for example, one typically considers recursive solutions of guarded process definitions, in complete metric spaces only fixpoints of contractive maps (which can then be found uniquely thanks to Banach’s fixpoint theorem), and in domain theory only least fixpoints over pointed predomains (i.e. domains). These examples have recently been shown as instances of the unifying notion of guarded traced category \[16, 15\].

In this work we aim to extend the stock of examples of this notion by including iteration on hybrid computations, which are encoded in the recently introduced hybrid monad \[26, 25\]. We argue that in the hybrid context guardedness corresponds to progressiveness – the property of trajectories to progressively extend over time during the iteration process (possibly converging to a finite trajectory in the limit) – we illustrate and examine the corresponding iteration operator and use it to develop while-loops for hybrid denotational semantics.

Hybrid computations are inherent to systems that combine discrete and continuous, physical behaviour \[31, 28, 1\]. Traditionally qualified as hybrid and born in the context of control theory \[32\], they range from computational devices interacting with their physical, external environment to chemical/biological reactions and physical processes that are subjected to discrete changes, such as combustions and impacts. Typical examples include pacemakers, cellular division processes, cruise control systems, and electric/water grids. Let us consider, for example, the following hybrid program, written in an algebraic programming style, and with \(\dot{x} = t \& r\) denoting “let variable $x$ evolve according to $t$ during $r$ milliseconds”.

\[
\begin{align*}
(\dot{v} = 1 \& 1) +_{v<120} (\dot{v} = -1 \& 1)
\end{align*}
\]

It represents a (simplistic) cruise controller that either accelerates ($\dot{v} = 1 \& 1$) or brakes ($\dot{v} = -1 \& 1$) during one millisecond depending if the car’s velocity $v$ is lower or greater than 120km/h. This program naturally fits in a slightly more sophisticated scenario obtained by wrapping a non-terminating while-loop around it:

\[
\text{while true } \{(\dot{v} = 1 \& 1) +_{v<120} (\dot{v} = -1 \& 1)\}
\]

Now the resulting program runs ad infinitum, measuring the car’s velocity every millisecond and changing it as specified by the if-then-else condition. How should we systematically interpret such while-loops?

Iteration on hybrid computations is notoriously difficult to handle due to the so called Zeno behaviour \[18, 2, 33\], a phenomenon of unfolding an iteration loop infinitely often in finite time, akin to the scenarios famously described by the greek philosopher Zeno, further
analyzed by Aristotle [3, Physics, 231a–241b], and since then by many others. To illustrate this, consider a bouncing ball dropped at a positive height and with no initial velocity. Due to the gravitational acceleration \( g \), it falls into the ground and bounces back up, losing a portion of its kinetic energy. In order to model this system, one can start by writing the program,

\[
(p = v, \dot{v} = g \land p \leq 0 \land v \leq 0; (v := v \times -0.5))
\]

(3) to specify the (continuous) change of height \( p \), and also the (discrete) change of velocity \( v \) when the ball touches the ground; the expression \( p \leq 0 \land v \leq 0 \) provides the termination condition: the ball stops when both its height and velocity do not exceed zero. Then, abbreviating program (3) to \( b \), one writes,

\[
(p := 1, v := 0; b; \ldots; b)_{n \text{ times}}
\]

as the act of dropping the ball and letting it bounce exactly \( n \) times. One may also wish to drop the ball and let it bounce until it stops (see Fig. 1), using some form of infinite iteration on \( b \) and thus giving rise to Zeno behaviour. Only a few existing approaches aim to systematically work with Zeno behaviour, e.g. in [18, 19] this is done by relying on non-determinism, although the results seem to introduce undesirable behaviour in some occasions (see details in the following subsection). Here, we do regard Zeno behaviour as an important phenomenon to be covered and as such helping to design and classify notions of iteration for hybrid semantics in a systematic and compelling way.

1.1 Related Work, Contributions, Roadmap, and Notation

There exist two well-established program semantics for hybrid systems: Höfner’s “Algebraic calculi for hybrid systems” [18] where programs are interpreted as sets of trajectories, and Platzer’s Kleene algebra [28] interpreting programs as maps \( X \to \mathcal{P}X \) for the powerset functor \( \mathcal{P} \). Both approaches are inherently non-deterministic and the corresponding iteration operators crucially rely on non-determinism. In [28], the iteration operator is modelled by the Kleene star \((-)^*\), i.e. essentially by the non-deterministic choice between all possible finite iterates of a given program \( p \); more formally, \( p^* \) is the least fixpoint of

\[
x \mapsto p; x + \text{skip}
\]

Semantics based on Kleene star deviates from the (arguably more natural) intuition given above for the non-terminating while-loop (2). It is also possible to extend the non-deterministic perspective summarized above to a more abstract setting via a monad that combines hybrid computations and nondeterminism [9], but in the present work we restrict ourselves to a purely hybrid setting, in order to study genuinely hybrid computations in isolation, without being interfered with other computational effects such as non-determinism.

One peculiarity of the Kleene star in [28] is that it is rather difficult to use for modelling programs with Zeno behaviour, the problem the authors are confronted with in [18, 19]. The authors of op. cit. extend the Kleene star setting with an infinite iteration operator \((-)^\omega\) that for a given program \( p \) it returns the largest fixpoint of the function on programs,

\[
x \mapsto p; x
\]

As argued in [18, 19], this operator still does not adequately capture the semantics of hybrid iteration, as it yields “too much behaviour”, e.g. if \( p = \text{skip} \), \( p^\omega \) is the program containing
Contributions. We propose semantic foundations for (Elgot) iteration in a hybrid setting: we identify two new monads for hybrid computations, one of which supports a partial guarded iteration operator, characterized as a least solution of the corresponding fixpoint equation, and another one extending the first and carrying a total iteration operator, although not generally being characterized in an analogous way. We show that both operators do satisfy the standard equational principles of iteration theories [5, 10] together with uniformity [29]. Moreover, we develop a language for hybrid computations with full-fledged while-loops as a prominent feature and interpret it using the underlying monad-based semantics. We discuss various use case scenarios and demonstrate various aspects of the iterative behaviour.

Plan of the paper. We proceed by defining a simple programming language for hybrid computations in Section 2, in order to present and discuss challenges related to defining a desirable semantics for it. In Section 3 we provide a summary of guarded (Elgot) iteration theory. In Sections 4 and 5 we present our main technical developments, including two new monads $\mathbf{H}_x$ and $\mathbf{H}$ for hybrid computations and the corresponding iteration operators. In Section 6 we provide a semantics for the while-loops of our programming language and then conclude in Section 7.

All omitted proofs can be found in an extended version of the paper [13] (available on arXiv).

Notation. We assume basic familiarity with the language of category theory [21], monads [21, 4], and topology [11]. Some conventions regarding notation are in order. By $|\mathcal{C}|$ we denote the class of objects of a category $\mathcal{C}$ and by $\text{Hom}_{\mathcal{C}}(A, B)$ ($\text{Hom}(A, B)$, if no confusion arises) the set of morphisms $f : A \to B$ from $A \in |\mathcal{C}|$ to $B \in |\mathcal{C}|$. We denote the set of Kleisli endomorphisms $\text{Hom}_{\mathcal{C}}(X,TX)$ by $\text{End}_{\mathcal{C}}(X)$. We agree to omit indices at natural transformations. We identify monads with the corresponding Kleisli triples, and use blackboard characters to refer to a monad and the corresponding roman letter to the monad’s functorial part, e.g. $T = (T, \eta, (-)^*)$ denotes a monad over a functor $T$ with $\eta : \text{Id} \to T$ being the unit and $(-)^* : \text{Hom}(X, TY) \to \text{Hom}(TX, TY)$ being the corresponding Kleisli lifting. Most of the time we work in the category $\text{Set}$ of sets and functions. We write $\mathbb{R}_+$ and $\mathbb{R}_\infty$ for the sets of non-negative reals, and non-negative reals extended with infinity $\infty$ respectively. Given $e : \mathbb{R}_+ \to X$ and $t \in \mathbb{R}_+$, we denote by $e^t$ the application $e(t)$. Given $x \in X$, $\mathbb{R}_+ : Y \to X$ is the function constantly equal to $x$. We use if-then-else constructs of the form $p < b > q$ returning $p$ if $b$ evaluates to true and $q$ otherwise.
2 A Simple Hybrid Programming Language

Let us build a simple hybrid programming language to illustrate some of our challenges and results. Intuitively, this language adds differential equation constructs to the standard imperative features, namely assignments, sequencing, and conditional branching. It was first presented in [25, Chapter 3] and we will use this paper’s results to extend it with a notion of iteration. We start by recalling the definition of the hybrid monad [26] here denoted by \( H_0 \), as a candidate semantic domain for this language. In the following sections, we will extend \( H_0 \) in order to obtain additional facilities for interpreting progressive and hybrid iteration.

\[ \text{Definition 1 ([26])}. \] The monad \( H_0 \) on \( \text{Set} \) is defined in the following manner.

- The set \( H_0 \text{X} \) has as elements the pairs \((d, e)\) with \( d \in \mathbb{R}_+ \) and \( e : \mathbb{R}_+ \to \text{X} \) a function satisfying the flattening condition: for every \( x \geq d \), \( e(x) = e(d) \). We call the elements of \((d, e)\) duration and evolution, respectively, and use the subscripts \( d \) and \( e \) to access the corresponding fields, i.e. given \( f = (d, e) \in H_0 \text{X} \), we mean \( f_d \) and \( f_e \) to denote \( d \) and \( e \) respectively.
- This convention extends to Kleisli morphisms as follows: given \( f : \text{X} \to H_0 \text{Y} \), \( f_d(x) = (f(x))_d \), \( e(x) = (f(x))_e \).
- The unit is defined by \( \eta(x) = (0, \underline{x}) \), where \( \underline{x} \) denotes the constant trajectory on \( x \);
- For every Kleisli morphism \( f : \text{X} \to H_0 \text{Y} \) and every value \((d, e) \in H_0 \text{X} \),

\[
(f^*(d, e))_d = d + f_d(e^t) < d \in \mathbb{R}_+ \lor \infty \quad (f^*(d, e))_e^t = f_{e}^t(e^t) < t < d \lor f_{e}^t - d(e^t)
\]

(recall that, according to our conventions, \((f^0)\) refers to \( f_0 \)); here we additionally simplify \((f^0)_{e}^t \) for the sake of readability).

We now fix a finite set of real-valued variables \( \text{X} = \{x_1, \ldots, x_n\} \) and denote by \( \text{At}(\text{X}) \) the set of atomic programs given by the grammar,

\[
\varphi \equiv (x_1 := t, \ldots, x_n := t) \mid (x_1 = t, \ldots, x_n = t \& r) \mid (\dot{x}_1 = t, \ldots, \dot{x}_n = t \& \psi),
\]

\( t \equiv r \mid r \cdot x \mid t + t, \quad \psi \equiv t \leq t \mid t > t \mid \psi \land \psi \mid \psi \lor \psi \)

where \( x \in \text{X} \) and \( r \in \mathbb{R}_+ \). The next step is to construct an interpretation map,

\[
[ - ] : \text{At}(\text{X}) \to \text{End}_{\text{H}_0}(\mathbb{R}^n)
\]

that sends atomic programs \( \text{a} \) to endomorphisms \([a] : \mathbb{R}^n \to H_0(\mathbb{R}^n) \) in the Kleisli category of \( \text{H}_0 \). This map extends to terms and predicates as \([t](v_1, \ldots, v_n) \in \mathbb{R}^n \) and \([\psi] \subseteq \mathbb{R}^n \times \mathbb{R}^n \) in the standard way by structural induction. We interpret each assignment \((x_1 := t, \ldots, x_n := t)\) as the map,

\[
(v_1, \ldots, v_n) \mapsto \eta_{\mathbb{R}^n}([t_1](v_1, \ldots, v_n), \ldots, [t_n](v_1, \ldots, v_n))
\]

Recall that linear systems of ordinary differential equations \( \dot{x}_1 = t, \ldots, \dot{x}_n = t \) always have unique solutions \( \phi : \mathbb{R}^n \to (\mathbb{R}^n)^{\mathbb{R}^n} \). We use this property to interpret each program \((\dot{x}_1 = t, \ldots, \dot{x}_n = t \& r)\) as the respective solution \( \mathbb{R}^n \to (\mathbb{R}^n)^{\mathbb{R}^n} \) but restricted to \( \mathbb{R}^n \to (\mathbb{R}^n)^{[0, r]} \). In order to interpret programs of the type \((\dot{x}_1 = t, \ldots, \dot{x}_n = t \& \psi)\) we can call on the following result.

\[ \text{Theorem 2 ([9])}. \] Consider a program \((\dot{x}_1 = t, \ldots, \dot{x}_n = t \& \psi)\), the solution \( \phi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \) of the system \( \dot{x}_1 = t, \ldots, \dot{x}_n = t \), and a valuation \((v_1, \ldots, v_n) \in \mathbb{R}^n \). If there exists a time instant \( r \in \mathbb{R}_+ \), such that \( \phi(v_1, \ldots, v_n, r) \in [\psi] \) then there exists a smallest time instant that also satisfies this condition.
Using this theorem, we interpret each program \((\dot{x}_1 = t, \ldots, \dot{x}_n = t \& \psi)\) as the function defined by,

\[(v_1, \ldots, v_n) \mapsto (d, \phi(v_1, \ldots, v_n, -))\]

where \(d\) is the smallest time instant that intersects \([\psi]\) if \((\text{Img}(\phi(v_1, \ldots, v_n, -))) \cap [\psi] \neq \emptyset\) and \(\infty\) otherwise. This final step provides the desired interpretation map of atomic programs (4).

We can now systematically build the hybrid programming language using standard algebraic results, as observed in [9, 25]. The set \(\text{End}_H_0(\mathbb{R}^n)\) of endomorphisms \(\mathbb{R}^n \to H_0(\mathbb{R}^n)\) together with Kleisli composition \(\bullet\) and the unit \(\eta : \text{Id} \to H_0\) form a monoid \((\text{End}_H_0(\mathbb{R}^n), \bullet, \eta)\). Therefore, the free monoidal extension of \([-\cdot] : \text{At}(X) \to (\text{End}_H_0(\mathbb{R}^n), \bullet, \eta)\) is well-defined and induces a semantics for program terms,

\[p = a \in \text{At}(X) | \text{skip} | p \cdot p\]

**Example 3.** Let us consider some programs written in this language.

1. We can have classic, discrete assignments, such as \(x := x + 1\) or \(x := 2 \cdot x\), and their sequential composition.

2. We can also write a \(\text{wait}(r)\) call, frequently used in the context of embedded systems for making the system halt its execution during \(r\) time units. This is achieved with the program \((\dot{x}_1 = 0, \ldots, \dot{x}_n = 0 \& r)\).

3. It is also possible to consider oscillators using histeresis [12], in particular via the sequential composition \((\dot{x} = 1 \& 1) ; (\dot{x} = -1 \& 1)\).

4. The bouncing ball system that was examined in the introduction is another program of this language.

We next extend our language with if-then-else clauses. This can be achieved in the following manner. Denote by \(B\) the free Boolean algebra generated by the expressions \(t = t\) and \(t < t\). Each \(b \in B\) induces an obvious predicate map \([b] : \mathbb{R}^n \to 2\).

Any \(b\) induces a binary function \(+_b : \text{End}_H_0(\mathbb{R}^n) \times \text{End}_H_0(\mathbb{R}^n) \to \text{End}_H_0(\mathbb{R}^n)\) defined as follows: \((f +_b g)(x) = f(x) < b(x) \triangleright g(x)\). This allows us to freely extend the interpretation map,

\[[-\cdot] : \text{At}(X) \to (\text{End}_H_0(\mathbb{R}^n), \bullet, \eta, (+)_b B)\]

into a hybrid programming language with if-then-else clauses \(p +_b B\).

**Example 4.** Let us consider some programs of this language with control decision features.

1. Aside from while-loops, our language carries the basic features of classic programs with discrete assignments, sequential composition, and if-then-else constructs.

2. The (simplistic) cruise controller, \((\dot{v} = 1 \& 1) +_{v<120} (\dot{v} = -1 \& 1)\) discussed in the introduction is also a program of this language.

To be able to address more complex behaviours we need some means for forming iterative computations, such as *while-loops*

\[
\text{while } b \{ p \}
\]

(5)

This poses the main challenge of our present work, which is to give a semantics of such constructs w.r.t. to a suitably designed *hybrid monad*. As a starting point, we refer to [26, 25] where \(H_0\) and an iteration operator \((-)\) : \(\text{Hom}(X, H_0X) \to \text{Hom}(X, H_0X)\), which we call basic iteration, were introduced. One limitation of this approach can already be read from the type profile: \((-)\) can only interpret non-terminating loops of the form \(\text{while true} \{ p \}\). The semantics of \((-)\) in \(H_0\) is given by virtue of metric spaces and Cauchy sequences, making
We drop the adjective “guarded” for guarded Elgot monads for which guardedness is total, however, are those properly between these two extreme situations, e.g. completely iterative monads [22] for which the notion of guardedness is defined via monad modules and the iteration operator is partial, but uniquely satisfies the fixpoint law.

3 Guarded Monads and Elgot Iteration

We proceed to give the necessary definitions related to guardedness for monads [16]. A monad \( T \) (on Set) is (abstractly) guarded if it is equipped with a notion of guardedness, which is a relation between Kleisli morphisms \( f : X \to TY \) and injections \( \sigma : Y \to Y \) closed under the rules in Fig. 2 where \( f : X \to \sigma Y \) denotes the fact that \( f \) and \( \sigma \) are in the relation in question. In the sequel, we also write \( f : X \to TY \) for \( f : X \to_{inl} TY \). More generally, we use the notation \( f : X \to_{p,q,...} TY \) to indicate guardedness in the union of injections \( in_p, in_q, \ldots \) where \( p, q, \ldots \) are sequences over \( \{1, 2\} \) identifying the corresponding coproduct summand in \( Y \). For example, we write \( f : X \to_{12,2} T((Y + Z) + Z) \) to mean that \( f \) is \([in_1 \ in_2]\)-guarded.

Definition 5 (Guarded Elgot monads). A monad \( T \) is a guarded Elgot monad if it is equipped with a guarded iteration operator,

\[
(f : X \to_2 T(Y + X)) \to (f^1 : X \to TY)
\]

satisfying the following laws:

- fixpoint law: \( f^1 = [\eta, f^!]*f \); 
- naturality: \( g^*f^! = ([T inl] g \ inl f )^! \) for \( f : X \to_2 T(Y + X), g : Y \to TZ \); 
- codiagonal: \( (T id, inr f )^! = f^! \) for \( f : X \to_{12,2} T((Y + X) + X) \); 
- uniformity: \( f h = T (id + h) g \) implies \( f^1 h = g^1 \) for \( f : X \to_2 T(Y + X), g : Z \to_2 T(Y + Z) \) and \( h : Z \to X \).

We drop the adjective “guarded” for guarded Elgot monads for which guardedness is total, i.e. \( f : X \to_\sigma TY \) for any \( f : X \to TY \) and \( \sigma \).

The notion of guarded monad is a common generalization of various cases occurring in practice. Every monad can be equipped with a least notion of guardedness, called vacuous guardedness and defined as follows: \( f : X \to_2 T(Y + Z) \) iff \( f \) factors through \( T inl : TY \to T(Y + Z) \). Every vacuously guarded monad is guarded iterative, for every fixpoint \( f^1 \) unfolds precisely once. On the other hand, the greatest notion of guardedness is total guardedness and is defined as follows: \( f : X \to_2 T(Y + Z) \) for every \( f : X \to T(Y + Z) \). This addresses total iteration operators on \( T \) (e.g. for \( T \) being Elgot), whose existence depends on special properties of \( T \), such as being enriched over complete partial orders. Motivating examples, however, are those properly between these two extreme situations, e.g. completely iterative monads [22] for which the notion of guardedness is defined via monad modules and the iteration operator is partial, but uniquely satisfies the fixpoint law.
Example 6. We illustrate the above concepts with the following simplistic examples.

1. The powerset monad \( P \) is Elgot, with the iteration operator sending \( f : X \to P(Y + X) \) to \( f^* : X \to PY \) calculated as the least solution of the fixpoint law \( f^* = [\eta, f^*] \ast f \).

2. An example of partial guarded iteration can be obtained from the previous clause by replacing \( P \) with the non-empty powerset monad \( P^* \). The total iteration operator from the previous clause does not restrict to a total iteration operator on this monad, because empty sets can arise from solving systems not involving empty sets, e.g. \( \eta \text{inr} : 1 \to P^*(1 + 1) \) would not have a solution in this sense. However, it is easy to see that the iteration operator from the previous clause restricts to a guarded one for \( P \) with the notion of guardedness defined as follows: \( f : X \to P^*(Y + X) \) iff for every element \( x \in X \), \( f(x) \) contains at least one element from \( Y \).

The axioms of guarded Elgot monads are given in Fig. 3 in an intuitive pictorial form. The shaded boxes indicate the scopes of the corresponding iteration loops and bullets attached to output wires express the corresponding guardedness predicates. As shown in [16], other standard principles such as dinaturality and the Bekić law follow from this axiomatization.

A Fistful of Hybrid Monads

According to Moggi [23], Kleisli morphisms can be viewed as generalized functions carrying a computational effect, e.g. non-determinism, process algebra actions, or their combination. In this context, hybrid computations can be seen as computations extended in time.
By definition, the pairs \((d, e) \in H_0X\) fall into two classes: closed trajectories with \(d \neq \infty\) and open trajectories with \(d = \infty\). Due to the flattening condition (see Definition 1), closed trajectories are completely characterized by their restrictions to \([0, d]\). We proceed by extending \(H_0\) to a larger monad that brings open trajectories over arbitrary intervals \([0, d]\) with \(d > 0\) into play, and call the resulting monad \(H_+\). It is instrumental in our study to cope with open trajectories, as in the presence of Zeno behaviour iteration might produce open trajectories \([0, d] \to X\) which we cannot sensibly extend into \([0, d] \to X\) without assuming some structure on \(X\) [18, 19, 24]. Furthermore, we introduce a variant of \(H_+\), which we call \(H\) and that extends the facilities of \(H_+\) even further by including the empty trajectory \([0, 0] \to X\) which will be used to accommodate divergent computations (see Remark 11).

The notation for \(H\) and \(H_+\) is selected to be suggestive, and is a reminiscent of \(P\) and \(P_+\) for the powerset and the non-empty powerset monads as in Example 6. Indeed, the analogy goes further, as in the next section we show that \(H_+\) supports guarded (progressive) iteration, \(H\) supports total iteration, and the former is a restriction of the latter.

In order to develop \(H_+\), we first introduce a partial version of \(H_0\) that will greatly facilitate obtaining some of our results. Essentially, this partial version amounts to the combination of \(H_0\) with the maybe monad \(M\). Recall that \(MX = X + 1\), that the unit of \(M\) is given by the left coproduct injection \(\text{inl} : X \to X + 1\), and that the Kleisli lifting sends \(f : X \to Y + 1\) to \([f, \text{inr}] : X + 1 \to Y + 1\). We conventionally see Kleisli morphisms \(X \to MY\) as partial functions from \(X\) to \(Y\) and thus write \(f(x)\downarrow\) to indicate that \(f(x)\) is defined on \(x\), i.e. \(f(x) \neq \text{inr}\). Let \(\text{dom}(f) = \{x \in X \mid f(x) \downarrow\}\). We proceed by \(\uparrow\) and the totally undefined function \(x \mapsto \bot\). Finally, write \(f(x)\uparrow\) as a shorthand notation to \(f(x) = \bot\).

We will also need the following result.

\textbf{Proposition 1.} Every monad \(T = (T, \eta, (-)\ast)\) induces a monad \(TM\) whose functor is defined by \(X \mapsto TMX\), the unit by \(\eta \text{inl} : X \to TMX\), and the Kleisli lifting by \([f, \eta \text{inr}]\ast : TMX \to TMY\). for every \(f : X \to TMY\).

\textbf{Proof.} This is a consequence of the standard fact that every monad distributes over the maybe monad [20].

\textbf{Definition 7.} Let \(H_0M\) be the monad identified in Proposition 1 with \(T = H_0\). Then let \(H_+\) be the subfunctor of \(H_0M\) that is defined by,

\[(d, e) \in H_+X\quad\text{iff}\quad e \neq \bot\quad\text{and}\quad e^t \downarrow\quad\text{for all}\quad t \in [0, d).\]

This yields a monad \(H_+\), by restricting the monad structure of \(H_0M\). Explicitly, \(\eta(x) = (0, \bot)\) and for every \(f : X \to H_+Y\) and every \((d, e) \in H_+X\),

\[
\begin{align*}
(f^\ast(d, e))_d &= d & (f^\ast(d, e))_d &= f_d^\ast(e^t) < t < d \uparrow \bot & \text{(if } e^d \uparrow) \\
(f^\ast(d, e))_d &= d + f_d(e^d) & (f^\ast(d, e))_d &= f_d^\ast(e^t) < t < d \uparrow f_d^{-d}(e^d) & \text{(if } e^d \downarrow)
\end{align*}
\]

Note that the set \(H_+X\) consists precisely of elements \((d, e)\) for which either \(\text{dom}(e) = [0, d]\) or \(\text{dom}(e) = [0, d]\) and \(d > 0\). Of course, we need to verify that Definition 7 correctly introduces a monad.
We therefore need to extend which is a straightforward consequence of the fact that condition (6) entails condition (7).

The condition \( c \neq \perp \) in (6) is essential for the construction above, for otherwise we cannot ensure that computations with totally undefined trajectories are compatible with Kleisli liftings, as detailed in Remark 9 below. Such computations can be thought of as representing unproductive divergence, and are required for the semantics of programs like

\[
\text{while true } \{ x := x + 1 \}
\]

We therefore need to extend \( \mathcal{H}_* \) to a larger monad \( \mathcal{H} \) in which such divergent computations exist. Technically, this will amount to quotienting the monad \( \mathcal{H}_0 \mathcal{M} \) in a suitable manner.

**Definition 8.** Let \( \mathcal{H}_0 \mathcal{M} \) be the monad identified in Proposition 1 with \( T = \mathcal{H}_0 \) and let \( H \) be the subfunctor of \( \mathcal{H}_0 \mathcal{M} \) formed as follows:

\[
(d, e) \in HX \text{ iff } e^d \downarrow \text{ for all } t \in [0, d).
\]

Let \( v \) be the inclusion of \( H \) into \( \mathcal{H}_0 \mathcal{M} \) and let \( \rho : \mathcal{H}_0 \mathcal{M} \to H \) be the natural transformation whose components are defined by,

\[
(\rho_X(d, e))_d = d, \quad (\rho_X(d, e))^e = e^t \iff t < d \\
\downarrow (e^d \iff d \downarrow \perp)
\]

where \( d_* = \sup \{ t < d \mid [0, t) \subseteq \text{dom}(e) \} \).

We extend \( \mathcal{H} \) to a monad by defining \( x \mapsto \rho(\eta(x)) \) to be the unit and the Kleisli lifting the one that sends \( f : X \to HY \) to \( \rho(\nu f)^* v \). Explicitly, the monad structure of \( \mathcal{H} \) is as follows: \( \eta(x) = (0, \perp) \) and for every \( f : X \to HY \), and every \( (d, e) \in HX \),

\[
\begin{align*}
(f^*(d, e))_d &= \sup \{ t < d \mid f^0_e(e^t) \downarrow \} \quad \text{(if } \exists t < d, f^0_e(e^t) \uparrow \text{ or } e^d \downarrow) \\
(f^*(d, e))^e &= f^0_e(e^t) \iff t < \sup \{ t < d \mid f^0_e(e^t) \downarrow \} \downarrow & \text{ (if } \forall t < d, f^0_e(e^t) \downarrow \text{ and } e^d \downarrow) \\
(f^*(d, e))_d &= d + f_0(d^e) \quad \text{(if } e^d \downarrow) \\
(f^*(d, e))^e &= f^0_e(e^t) \iff t < d \downarrow & \text{ (if } \exists t < d, f^0_e(e^t) \downarrow \text{ or } e^d \downarrow)
\end{align*}
\]

Like in the case of \( \mathcal{H}_* \), we need to verify that \( \mathcal{H} \) is a monad; this is shown in the paper’s extended version [13].

**Remark 9.** As indicated above, \( \mathcal{H} \) is a quotient of \( \mathcal{H}_0 \mathcal{M} \) and not a submonad, specifically \( v \) is not a monad morphism. Indeed, given \( (0, \perp) : \mathbb{R}_* \to H\mathbb{R}_* \) and \( (1, \text{id}) \in H\mathbb{R}_* \), by definition, \( (0, \perp)^*(1, \text{id}) = (0, \perp) \), but in \( \mathcal{H}_0 \mathcal{M} \), \( (0, \perp)^*(1, \text{id}) = (1, \perp) \).

In summary, the monads \( \mathcal{H}_0 \mathcal{M}, \mathcal{H}_*, \mathcal{H} \) are connected as depicted in Fig. 4. Here, \( \iota \) and \( \rho \) are monad morphisms, and the induced composite morphism \( \rho \iota : \mathcal{H}_0 \mathcal{M} \to \mathcal{H} \) is pointwise injective, which is a straightforward consequence of the fact that condition (6) entails condition (7).
5 Progressive Iteration and Hybrid Iteration

We start off by equipping the monad $H_+$ from the previous section with a suitable notion of guardedness.

Definition 10 (Progressiveness). A Kleisli morphism $(d,e) : X \to H_+(Y + Z)$ is progressive in $Z$ (in $Y$) if $e^0 : X \to Y + Z$ factors through $\text{inl}$ (respectively, $\text{inr}$).

Given $(d,e) : X \to H_+(Y + X)$, progressiveness in $X$ means precisely that $e^0 = \text{inl} u : X \to Y + X$ for a suitable $u : X \to Y$, which is intuitively the candidate for $(d,e)^t$ at $0$. In other words, progressiveness rules out the situations in which the iteration operator needs to handle compositions of zero-length trajectories.

Remark 11. A simple example of a morphism $(d,e) : X \to H_+(Y + X)$ not progressive in $X$ is obtained by taking $X = \{0, 1\}$, $Y = \emptyset$, $d = \emptyset$ and $e^0 = \text{inr swap}$ where swap interchanges the elements of $\{0, 1\}$. In attempts of defining $(d,e)^t$ we would witness oscillation between 0 and 1 happening at time 0, i.e. not progressing over time, which is precisely the reason why there is no candidate semantic for $(d,e)^t$ in this case.

Lemma 12. $H_+$ is a guarded monad with $f : X \to H_+(Y + Z)$ iff $f$ is progressive in $Z$.

Instead of directly equipping $H_+$ with progressive iteration, we take the following route: we enrich the monad $H_0M$ over complete partial orders and devise a total iteration operator for it using the standard least-fixpoint argument. Then we restrict iteration from $H_0M$ to $H_+$ via $\iota$ and to $H$ via $\nu$ (see Fig. 4). The latter part is tricky, because $\nu$ is not a monad morphism (Remark 9), and thus we will call on the machinery of iteration-congruent retractions, developed in [16], to derive a (total) Elgot iteration on $H_+$.

Consider the following order on $H_0MX$. Given $(d,e), (d_*, e_*) \in H_0MX$, let $(d,e) \sqsubseteq (d_*, e_*)$ if $d \leq d_*$ and $e$ is smaller or equal $e_*$ as partial maps, i.e. $\text{dom}(e) \subseteq \text{dom}(e_*)$ and $e^t = e_*^t$ for all $t \in \text{dom}(e)$. This order extends to the hom-sets $\text{Hom}(X, H_0MY)$ pointwise.

Theorem 13. The following properties hold.
1. Every set $H_0MX$ is an $\omega$-complete partial order under $\sqsubseteq$ with $(0, \bot)$ as the bottom element;
2. Kleisli composition is monotone and continuous w.r.t. $\sqsubseteq$ on both sides;
3. Kleisli composition is right-stric, i.e. for every $f : X \to H_0MY$, $f^\ast(0, \bot) = (0, \bot)$.

Note that Kleisli composition is not left strict (c.f. Remark 9). Using the previous result and [14, Theorem 5.8], we immediately obtain

Corollary 14. $H_0M$ possesses a total iteration operator $(-)^t$ obtained as a least solution of equation $f^t = [\eta, f^t] \ast f$. This makes $H_0MX$ into an Elgot monad.

We readily obtain a progressive iteration on $H_+$ by restriction via $\iota$ (see Fig. 4).

Corollary 15. $H_+$ possesses a guarded iteration operator $(-)^t$ obtained by restriction from $H_0M$ with guardedness being progressiveness and $f^t$ being the least solution of equation $f^t = [\eta, f^t] \ast f$.

Explicitly, $f^t$ is calculated via the Kleene fixpoint theorem as follows. Consider $f : X \to H_+(Y + X)$, let $f^{(0)} = (0, \bot)$ and $f^{(i+1)} = [\eta, f^{(i)}] \ast f$. This yields an $\omega$-chain

$(0, \bot) \sqsubseteq f^{(1)} \sqsubseteq f^{(2)} \sqsubseteq \cdots$

and $f^t = \bigsqcup_i f^{(i)}$. We proceed to obtain an iteration operator for $H$. Remarkably, we cannot use the same technique of restricting the iteration operator from $H_0M$ to $H$, even though $H$ embeds into $H_0M$: the following example illustrates the problem.
Example 16. Let \( f = (\underline{1}, e) : \mathbb{R}_+ \rightarrow H_0M(\mathbb{R}_+ + \mathbb{R}_+) \) with \( e(x) = \text{inr} \cdot 1 \) for every \( x \in \mathbb{R}_+ \). The result of calculating \( f^1 \) is the trajectory \((\infty, \bot)\), which is not present in \( H \).

Note that every \( HX \) inherits the \( \omega \)-complete partial order from \( H_0MX \).

Theorem 17. Let \( \rho : H_0M \rightarrow H \) and \( v : H \rightarrow H_0M \) be the pair of natural transformations from Definition 8. Then for every \( f : X \rightarrow H_0M(Y + X) \), \( \rho f^1 = \rho(\nu f)^\tau \).

In the terminology of [16], Theorem 17 states that the pair \((\rho, v)\) is an iteration-congruent retraction. Therefore, per [16, Theorem 21], \( H \) inherits a total Elgot iteration from \( H_0M \).

Corollary 18. \( H \) is an Elgot monad with the iteration operator \((-)^\tau \) defined as follows: for every \( f : X \rightarrow H(Y + X) \), \( f^1 = \rho(v f)^\tau \) assuming that \((-)^\tau \) is the iteration operator on \( H_0M \).

Corollary 19. The progressive iteration operator \((-)^\tau \) of \( H_+ \) is the restriction of the total iteration operator \((-)^\tau \) of \( H \) along \( \mu : H_+ \rightarrow H \) (as in Fig. 4), i.e. for every \( f : X \rightarrow_{12} H_+(Y + X) \), \( \rho f^1 = (\rho v f)^\tau \).

Proof. Let \((-)^\tau \) be the iteration operator of \( H_0M \). Then, by definition,

\[
(\rho v f)^\tau = \rho(\nu f)^\tau = \rho f^1.
\]

Using the fact that the iteration operator for \( H \) satisfies the codiagonal law (see Definition 5), we factor the former through progressive iteration as follows.

Theorem 20 (Decomposition Theorem). Given \( f : X \rightarrow H(Y + X) \), let \( \tilde{f} : X \rightarrow_{12} H((Y + X) + X) \) be defined as follows:

\[
\tilde{f}(x) = f_a(x) \quad \tilde{f}^0_e(x) = (\text{id} + \text{inl})(f^0_e(x)) \quad \tilde{f}^1_e(x) = (\text{id} + \text{inr})(f^1_e(x)) \quad (x \in X, t > 0)
\]

Then \( f^1 = (\tilde{f}^1)^\tau \).

Proof. Note that \( f = H[\text{id}, \text{inr}]\tilde{f} \). Hence, by the codiagonal law: \( f^1 = (H[\text{id}, \text{inr}]\tilde{f})^1 = (\tilde{f}^1)^\tau \), and the latter is \( (f^1)^\tau \) per Corollary 19, as \( f^1 \) happens to be progressive in the second argument.

Theorem 20 presents the iteration of \( H \) as a nested combination of progressive iteration and what can be called singular iteration, as it is precisely the restriction of \((-)^\tau \) responsible for iterating computations of zero duration. Finally, we recover basic iteration, discussed in Section 2, on \( H_+ \) (and hence on \( H \)) by turning a morphism \( X \rightarrow H_+(X + X) \).
Definition 21 (Basic iteration). We define basic iteration \((d,e)^* : X \to H_* X\) to be
\[
((d, \lambda x. \lambda t. \text{ln} e^0(x) < t = 0 > \text{ln} e^t(x)) : X \to (X + X))' : X \to H_* X.
\]

Example 22. We illustrate our design decisions behind \(H_*\) (and \(H\)) with the following two examples of Zeno behaviour, computing \(f^+ = (d^*, e^*)\) for specific morphisms \(f : X \to H_* X\).

1. Let \(f = (d, e) : [0, 1] \to H_* [0, 1]\) be defined as follows: for every \(x \in [0, 1]\), \(d(x) = (1-x)/2\) and \(e^t(x) = x + t\) for \(t \in [0, (1-x)/2]\). It is easy to see that \(d^0(0) = 1/2 + 1/4 + \ldots = 1\), however, by definition, \((e^* (0))^\uparrow\). This is indeed a prototypical example of Zeno behaviour (specifically, this is precisely Zeno’s “Dichotomy” paradox analyzed by Aristotle [3, Physics, 231a–241b]): Given a distance of total length 1 to be covered, suppose some portion \(x < 1\) of it has been covered already. Then the remaining distance has the length \(1-x\). As originally argued by Zeno, in order to cover this distance, one has to pass the middle, i.e. walk the initial interval of length \((1-x)/2\) and our function \(f\) precisely captures the dynamics of this motion. The resulting evolution \(e^* (0)\) together with the corresponding approximations are depicted on the left of Fig. 5. In this formalization, the traveler can not reach the end of the track, but only because we designed \((\cdot)^*\) to be so. We could also justifiably define \((e^* (0))^\uparrow\) to be 1, for this is what \((e^* (0))^\downarrow\) tends to as \(t\) tends to 1. This is indeed the case of the approach from [26, 25] developed for the original monad \(H_0\).

2. It is easy to obtain an example of an open trajectory produced by Zeno iteration that cannot be continuously extended to a closed one by adapting a standard example of essentially discontinuous function from analysis: let e.g. \(u : [0, 1) \to [0, 1)\) be as follows:
\[
u^t(x) = (t + x) \cos \left(\frac{\pi t}{1-x}(1-x-t)\right) \quad (t \in [0, 1-x))
\]
\[
u^t(x) = 1 \quad (t \in [1-x, 1))
\]
The graph of \(u(0)\) is depicted on the right of Fig. 5 where one can clearly see the discontinuity at \(t = 1\). It is easy to verify that \((1, u) \in H[0, 1]\) is obtained by applying basic iteration to \(f = (d, e) : [0, 1) \to H_* [0, 1)\) given as follows:
\[
d(x) = \frac{2(1-x)^2}{3-2x}
\]
e^t(x) = (t + x) \cos \left(\frac{\pi t}{1-x}(1-x-t)\right) \quad (t \in [0, d(x))
\]
e^t(x) = d(x) + x \quad (t \in [d(x), 1))

Even though we carried our developments in the category of sets, we designed \(H_*\) and \(H\) keeping in touch with a topological intuition. The following instructive example shows that the iteration operators developed in the previous section cannot be readily transferred to the category of topological spaces and continuous maps, for reasons of instability: small changes in the definition of a given system may cause drastic changes in its behaviour. In particular, even if a morphism \((d, e) : X \to H_0 X\) is continuous (for the topology described in [26]) the duration component \(d^* : X \to [0, \infty]\) of \((d^*, e^*) = (d, e)^*\) need not be continuous.

Example 23 (Hilbert Cube). Let \(X = [0, 1]^\omega\) be the Hilbert cube, i.e. the topological product of \(\omega\) copies of \([0, 1]\) and let \(\text{hd} : X \to [0, 1]\) and \(\text{tl} : X \to X\) be the obvious projections realizing the isomorphism \([0, 1]^\omega \cong [0, 1] \times [0, 1]^\omega\). Let \(f = (\text{hd}, e) : X \to H_* X\) with e : X X \ X X \ X be defined as follows:
- \(e^t(x) = x\) if \(\text{hd}(x) = 0, t \in \mathbb{R}_+\);
- \(e^t(x) = (\text{hd}(x) - t) \cdot x + t \cdot \text{tl}(x) / \text{hd}(x)\) if \(0 < \text{hd}(x)\) and \(t < \text{hd}(x)\);
- \(e^t(x) = \text{tl}(x)\) if \(\text{hd}(x) > 0\) and \(t \geq \text{hd}(x)\).
In the second clause we use a convex combination of $x$ and $t l(x)$ as vectors of $X$ seen as a vector space (indeed, even a Hilbert space) over the reals. It can now be checked that the cumulative duration $d^u$ in $(d^u, e^u)$ is not continuous. To see why, note that $d^u(x)$ is the (possibly infinite) sum of the components of $x$ from left to right up to the first zero element, and therefore each $U = (d^u)^{-1}([0, a])$ contains all such vectors $x \in [0, 1]^\omega$ for which this sum is properly smaller than $a$. Then recall that a basic open set of $[0, 1]^\omega$ must be a finite intersection of sets of the form $\pi_i^{-1}(V)$, $V \subseteq [0, 1]$ open, $i \in \mathbb{N}$. Therefore, if $U$ was open the definition of the product topology on $[0, 1]_c$ would imply that for every vector $x$ in $U$ there exists a position such that by altering the components of $x$ arbitrarily after this position, the result would still belong to $U$. This is obviously not true for $U$, because by replacing the elements of any infinite vector from $[0, 1]^\omega$ after any position with 1, would give a vector summing to infinity.

6 Bringing While-loops Into The Scene

In Section 2, we started building a simple hybrid programming language. We sketched a monad-based semantics for the expected programs constructs, except the while-loops. Here we extend it by taking $H$, which is a supermonad of $H_0$, as the underlying monad and interpret while-loops (5) via the iteration operator of $H$.

Recall that $b$ is an element of the free Boolean algebra generated by the expressions $t = t$ and $t < t$, and that there exists a predicate map $b : \mathbb{R}^n \rightarrow 2$. Now for each $b : \mathbb{R}^n \rightarrow 2$ and $f : \mathbb{R}^n \rightarrow H(\mathbb{R}^n)$ denote the function,

$$
\left( \mathbb{R}^n \xrightarrow{\text{dist}(id, b)} \mathbb{R}^n + \mathbb{R}^n \xrightarrow{[\eta \text{int}, (H \text{int}) \ f]} H(\mathbb{R}^n + \mathbb{R}^n) \xrightarrow{m} H(\mathbb{R}^n + \mathbb{R}^n) \right) ^t
$$

by $w(b, f)$ where $\text{dist} : X \times 2 \rightarrow X + X$ is the obvious distributivity transformation, and $m(d, e) = (d, e')$ with $e'(t) = \text{int}(x) \triangleq \text{int}(x) = e(t)$ and $t < d \triangleright e(t)$. Intuitively, the function $m$ makes the last point of the trajectory be the only one that is evaluated by the test condition of the while-loop. Then, we define $[\text{while } b \{p\}] = w(b, [p])$ and this gives a hybrid programming language,

$$
p = a \in \text{At}(X) \mid \text{skip} \mid p \mid p + b \ p \mid \text{while } b \{p\}
$$

with while-loops.

\textbf{Example 24.} Let us consider some programs written in this language.
1. We start again with a classic program, in this case \texttt{while true \{ x := x + 1 \}}. It yields the empty trajectory $\perp$.
2. Another example of a classic program is,

$$
\text{while } x \leq 10 \{ x := x + 1 ; \text{wait}(1) \}
$$

If for example the initial value is 0 the program takes eleven time units to terminate.
3. Let us consider now the program \texttt{while }x \geq 1 \{ (\dot{x} = -1 \& 1) \}. If the initial value is 0 the program outputs the trajectory with duration 0 and constant on 0, since it never enters in the loop. If we start e.g. with 3 as initial value then the program inside the while-loop will be executed precisely three times, continuously decreasing $x$ over time.
4. In contrast to classic programming languages, here infinite while-loops need not be undefined. The cruise controller discussed in the introduction,

$$
\text{while true \{ (\dot{y} = 1 \& 1) +_{_\nu \leq 120} (\dot{y} = -1 \& 1) \}}
$$

is a prime example of this.
5. Finally, the bouncing ball, \((p := 1, v := 0); \text{while true } \{ b \})\) which has Zeno behaviour, outputs a trajectory describing the ball’s movement over the time interval \([0,d]\) where \(d\) is the instant of time at which the ball stops.

7 Conclusions and Further Work

We developed a semantics for hybrid iteration by bringing together two abstraction devices introduced recently: guarded Elgot iteration [16] and the hybrid monad [26, 25]. Our analysis reveals that, on the one hand, the abstract notion of guardedness can be interpreted as a suitable form of progressiveness of hybrid trajectories, and on the other hand, the original hybrid monad from [26, 25] needs to be completed for the sake of a smooth treatment of iteration, specifically, iteration producing Zeno behaviour. In our study we rely on Zeno behaviour examples as important test cases helping to design the requisite feasible abstractions. As another kind of guidance, we rely on Elgot’s notion of iteration [10] and the corresponding laws of iteration theories [5]. In addition to the new hybrid monad \(\mathcal{H}_+\) equipped with (partial) progressive iteration, we introduced a larger monad \(\mathcal{H}\) with total hybrid iteration extending the progressive one. In showing the iteration laws we heavily relied on the previously developed machinery for unifying guarded and unguarded iteration [14, 16]. We illustrated the developed semantic foundations by introducing a simple language for hybrid iteration with while-loops interpreted over the Kleisli category of \(\mathcal{H}\).

We regard our present work as a stepping stone for further developments in various directions. After formalizing hybrid computations via (guarded) Elgot monads, one obtains access to further results involving (guarded) Elgot monads, e.g. it might be interesting to explore the results of applying the generalized coalgebraic resumption monad transformer [14] to \(\mathcal{H}\) and thus obtain in a principled way a semantic domain for hybrid processes in the style of CCS. As shown by Theorem 20, the iteration of \(\mathcal{H}\) is a combination of progressive iteration and “singular iteration”. An interesting question for further work is if this combination can be framed as a universal construction. We also would like to place \(\mathcal{H}\) in a category more suitable than \(\text{Set}\), but as Example 23 suggests, this is expected to be a very difficult problem.

Every monad on \(\text{Set}\) determines a corresponding Lawvere theory, whose presentation in terms of operations and equations is important for reasoning about the corresponding – in our case hybrid – programs. We set as a goal for further research the task of identifying the underlying Lawvere theories of hybrid monads and integrating them into generic diagrammatic reasoning in the style of Fig. 3. This should prospectively connect our work to the line of research by Bonchi, Sobociński, and Zanasi (see e.g. [6, 7]), who studied various axiomatizations of PROPs (i.e. monoidal generalizations of Lawvere theories) and their diagrammatic languages. For a proper treatment of guarded iteration (i.e. a specific instance of guarded monoidal trace in the sense of [15]), one would presumably need to develop the corresponding notions of guarded Lawvere theory and guarded PROP.

References

22:16  A Semantics for Hybrid Iteration


