Implementing Hybrid Semantics: From Functional to Imperative

Sergey Goncharov[1], Renato Neves[2] and José Proença[3]

[1] Dept. of Comp. Sci., FAU Erlangen-Nürnberg, Germany
[2] University of Minho & INESC-TEC, Portugal
[3] CISTER/ISEP, Portugal

Abstract. Hybrid programs combine digital control with differential equations, and naturally appear in a wide range of application domains, from biology and control theory to real-time software engineering. The entanglement of discrete and continuous behaviour inherent to such programs goes beyond the established computer science foundations, producing challenges related to e.g. infinite iteration and combination of hybrid behaviour with other effects. A systematic treatment of *hybridness* as a dedicated computational effect has emerged recently. In particular, a generic idealized functional language HybCore with a sound and adequate operational semantics has been proposed. The latter semantics however did not provide hints to implementing HybCore as a runnable language, suitable for hybrid system simulation (e.g. the semantics features rules with uncountably many premises). We introduce an imperative counterpart of HybCore, whose semantics is simpler and runnable, and yet intimately related with the semantics of HybCore at the level of *hybrid monads*. We then establish a corresponding soundness and adequacy theorem. To attest that the resulting semantics can serve as a firm basis for the implementation of typical tools of programming oriented to the hybrid domain, we present a web-based prototype implementation to evaluate and inspect hybrid programs, in the spirit of GHCi for Haskell and UTop for OCaml. The major asset of our implementation is that it formally follows the operational semantic rules.

1 Introduction

The core idea of hybrid programming. Hybrid programming is a rapidly emerging computational paradigm [26,29] that aims at using principles and techniques from programming theory (e.g. compositionality [12,26], Hoare calculi [29,34], theory of iteration [2,8]) to provide formal foundations for developing computational systems that interact with physical processes. Cruise controllers are a typical example of this pattern; a very simple case is given by the hybrid program below.

```
while true do {
    if v ≤ 10 then (v' = 1 for 1) else (v' = −1 for 1)        (cruise controller)
}
```

In a nutshell, the program specifies a digital controller that periodically measures and regulates a vehicle's velocity (`v`): if the latter is less or equal than `10` the controller accelerates during `1` time unit, as dictated by the program statement `v' = 1 for 1` (`v' = 1` is a differential equation representing the velocity's rate of change over time. The value `1` on the right-hand side of `for` is the duration during which the program statement runs). Otherwise, it decelerates during the same amount of time (`v' = −1 for 1`). Figure 1 shows the output respective to this hybrid program for an initial velocity of 5.

Note that in contrast to standard programming, the cruise controller involves not only classical constructs (while-loops and conditional statements) but also differential ones (which are used for describing physical processes). This cross-disciplinary combination is the core feature of hybrid programming and has a notably wide range of application domains (see [29,30]). However, it also hinders the use of classical techniques of programming, and thus calls for a principled extension of programming theory to the hybrid setting.

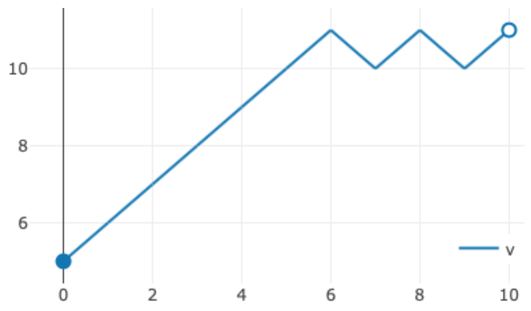


Fig. 1: Vehicle's velocity

As is already apparent from the (cruise controller) example, we stick to an *imperative* programming style, in particular, in order to keep in touch with the established denotational models of physical time and computation. A popular alternative to this for modelling real-time and hybrid systems is to use a *declarative* programming style, which is done e.g. in real-time Maude [27] or Modelica [10]. A well-known benefit of declarative programming is that programs are very easy to write, however on the flip side, it is considerably more difficult to define what they exactly mean.

Motivation and related work. Most of the previous research on formal hybrid system modelling has been inspired by automata theory and Kleene algebra (as the corresponding algebraic counterpart). These approaches led to the well-known notion of hybrid automaton [17] and Kleene algebra based languages for hybrid systems [28,18,19]. From the purely semantic perspective, these formalizations are rather close and share such characteristic features as *nondeterminism* and what can be called *non-refined divergence*. The former is standardly justified by the focus on formal verification of safety-critical systems: in such contexts overabstraction is usually desirable and useful. However, coalescing *purely hybrid* behaviour with nondeterminism detaches semantic models from their prototypes as they exist in the wild. This brings up several issues. Most obviously, a nondeterministic semantics, especially not given in an operational form, cannot directly serve as a basis for languages and tools for hybrid system testing and simulation. Moreover, models with nondeterminism baked in do not provide a clear indication of how to combine hybrid behaviour with effects other than nondeterminism (e.g. probability), or to combine it with nondeterminism in a different way (*van Glabbeek's spectrum* [36] gives an idea about the diversity of potentially arising options). Finally, the Kleene algebra paradigm strongly suggests a relational semantics for programs, with the underlying relations connecting a state on which the program is run with the states that the program can reach. As previously indicated by Höfner and Möller [18], this view is too coarse-grained and contrasts to the trajectory-based one where a program is associated with a trajectory of states (recall Figure 1). The trajectory-based approach provides an appropriate abstraction for such aspects as notions of convergence, periodic orbits, and duration-based predicates [5]. This potentially enables analysis of properties such as *how fast* our (cruise controller) example reaches the target velocity or for *how long* it exceeds it.

The issue of *non-refined divergence* mentioned earlier arises from the Kleene algebra law $\mathtt{p} \mathbin{;} \mathtt{0} = \mathtt{0}$ in conjunction with Fischer-Ladner's encoding of while-loops $\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}$ as $(\mathtt{b} \mathbin{;} \mathtt{p})^* ; \neg \mathtt{b}$. This creates a havoc with all divergent programs $\mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\ \mathtt{p}\ \}$ as they become identified with divergence $\mathtt{0}$, thus making the above example of a (cruise controller) meaningless. This issue is extensively discussed in Höfner and Möller's work [18] on a *nondeterministic* algebra of trajectories, which tackles the problem by disabling the law $\mathtt{p} \mathbin{;} \mathtt{0} = \mathtt{0}$ and by introducing a special operator for infinite iteration that inherently relies on nondeterminism. This iteration operator inflates trajectories at so-called 'Zeno points' with arbitrary values, which in our case would entail e.g. the program

$$\mathtt{x} \coloneqq \mathtt{1} \mathbin{;} \mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\ \mathtt{wait}\ \mathtt{x} \mathbin{;} \mathtt{x} \coloneqq \mathtt{x}/2\ \} \qquad \text{(zeno)}$$

to output at time instant 2 all possible values in the valuation space (the expression $\mathtt{wait}\ \mathtt{t}$ represents a wait call of $\mathtt{t}$ time units). More details about Zeno points can be consulted in [18,14].

In previous work [12,14], we pursued a *purely hybrid* semantics via a simple *deterministic functional* language HybCore, with while-loops for which we used Elgot's notion of iteration [8] as the underlying semantic structure. That resulted in a semantics of finite and infinite iteration, corresponding to a refined view of divergence. Specifically, we developed an operational semantics and also a denotational counterpart for HybCore. An important problem of that semantics, however, is that it involves infinitely many premisses and requires calculating total duration of programs, which precludes using such semantics directly in implementations. Both the above examples (cruise controller) and (zeno) are affected by this issue. In the present paper we propose an *imperative* language with a denotational semantics similar to HybCore's one, but now provide a clear recipe for executing the semantics in a constructive manner.

Overview and contributions. Building on our previous work [14], we devise operational and denotational semantics suitable for implementation purposes, and provide a soundness and adequacy theorem relating both these styles of semantics. Results of this kind are well-established yardsticks in the programming language theory [37], and beneficial from a practical perspective. For example, small-step operational semantics naturally guides the implementation of compilers for programming languages, whilst denotational semantics is more abstract, syntax-independent, and guides the study of program equivalence, of the underlying computational paradigm, and its combination with other computational effects.

As mentioned before, in our previous work [14] we introduced a simple functional hybrid language HybCore with operational and denotational monad-based semantics. Here, we work with a similar imperative while-language, whose semantics is given in terms of a global state space of trajectories over $\mathbb{R}^n$, which is a commonly used carrier when working with solutions of systems of differential equations. A key principle we have taken as a basis for our new semantics is the capacity to determine behaviours of a program `p` by being able to examine only some subterms of it. In order to illustrate this aspect, first note that our semantics does not reduce program terms `p` and initial states σ (corresponding to valuation functions $\sigma\colon \mathcal{X} \to \mathbb{R}$ on program variables $\mathcal{X}$) to states σ', as usual in classical programming. Instead it reduces *triples* $\mathtt{p}, \sigma, \mathtt{t}$ of programs `p`, initial states σ and time instants `t` to a state σ'; such a reduction can be read as "given σ as the initial state, program `p` produces a state σ' at time instant `t`". Then, the reduction process of $\mathtt{p}, \sigma, \mathtt{t}$ to a state only examines fragments of `p` or unfolds it when strictly necessary, depending of the time instant `t`. For example, the reduction of the (cruise controller) unfolds the underlying loop only twice for the time instant $1 + 1/2$ (the time instant $1 + 1/2$ occurred in the second iteration of the loop). This is directly reflected in our prototype implementation of an interactive evaluator of hybrid programs Lince. It is available online and comes with a series of examples for the reader to explore (`http://arcatools.org/lince`). The plot in Figure 1 was automatically obtained from Lince, by calling on the previously described reduction process for a predetermined sequence of time instants `t`.

For the denotational model, we build on our previous work [12,14] where hybrid programs are interpreted via a suitable monad $\mathbf{H}$, called the *hybrid monad* and capturing the computational effect of *hybridness*, following the seminal approach of Moggi [24,25]. Our present semantics is more lightweight and is naturally couched in terms of another monad $\mathbf{H}_S$, parametrized by a set S. In our case, as mentioned above, S is the set of trajectories over $\mathbb{R}^n$ where n is the number of available program variables $\mathcal{X}$. The latter monad is in fact parametrized in a formal sense [35] and comes out as an instance of a recently emerged generic construction [7]. A remarkable salient feature of that construction is that it can be instantiated in a constructive setting (without using any choice principles) – although we do not touch upon this aspect here, in our view this reinforces the fundamental nature of our semantics. Among various benefits of $\mathbf{H}_S$ over $\mathbf{H}$, the former monad enjoys a construction of an iteration operator (in the sense of Elgot [8]) as a *least fixpoint*, calculated as a limit of an ω-chain of approximations, while for $\mathbf{H}$ the construction of the iteration operator is rather intricate and no similar characterization is available. A natural question that arises is: how are $\mathbf{H}$ and $\mathbf{H}_S$ related? We do answer it by providing an instructive connection, which sheds light on the construction of $\mathbf{H}$, by explicitly identifying semantic ingredients which have to be added to $\mathbf{H}_S$ to obtain $\mathbf{H}$. Additionally, this results in "backward compatibility" with our previous work.

Document structure. After short preliminaries (Section 2), in Section 3 we introduce our while-language and its operational semantics. In Sections 4 and 5, we develop the denotational model for our language and connect it formally to the existing hybrid monad [12,14]. In Section 6, we prove a soundness and adequacy result for our operational semantics w.r.t. the developed model. Section 7 describes LINCE's architecture. Finally, Section 8 concludes and briefly discusses future work. Omitted proofs and examples are found in the extended version of the current paper [15].

2 Preliminaries

We assume familiarity with category theory [1]. By $\mathbb{R}$, $\mathbb{R}_+$ and $\bar{\mathbb{R}}_+$ we respectively denote the sets of reals, non-negative reals, and extended non-negative reals (i.e. $\mathbb{R}_+$ extended with the infinity value ∞). Let $[0, \bar{\mathbb{R}}_+)\!]$ denote the set of downsets of $\bar{\mathbb{R}}_+$ having the form $[0, d]$ ($d \in \mathbb{R}_+$) or the form $[0, d)$ ($d \in \bar{\mathbb{R}}_+$). We call the elements of the dependent sum $\sum_{I \in [0, \bar{\mathbb{R}}_+)\!]} X^I$ *trajectories* (over X). By $[0, \mathbb{R}_+]$, $[0, \mathbb{R}_+)$ and $[0, \bar{\mathbb{R}}_+)$ we denote the following corresponding subsets of $[0, \bar{\mathbb{R}}_+)\!]$: $\{[0, d] \mid d \in \mathbb{R}_+\}$, $\{[0, d) \mid d \in \mathbb{R}_+\}$ and $\{[0, d) \mid d \in \bar{\mathbb{R}}_+\}$. By $X \uplus Y$ we denote the *disjoint union*, which is the categorical coproduct in the category of sets with the corresponding left and right injections $\mathsf{inl}\colon X \to X \uplus Y$, $\mathsf{inr}\colon Y \to X \uplus Y$. To reduce clutter, we often use plain union $X \cup Y$ in place of $X \uplus Y$ if X and Y are disjoint by construction.

By $a \lhd b \rhd c$ we denote the case distinction construct: a if b is true and c otherwise. By $!$ we denote the *empty function*, i.e. a function with the empty domain. For the sake of succinctness, we use the notation e^t for the function application $e(t)$ with real-value t.

3 An imperative hybrid while-language and its semantics

This section introduces the syntax and operational semantics of our language. We first fix a stock of n-variables $\mathcal{X} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ over which we build atomic programs, according to the grammar

$$\begin{aligned} \mathtt{At}(\mathcal{X}) &\ni \mathtt{x} := \mathtt{t} \mid \mathtt{x}_1' = \mathtt{t}_1, \ldots, \mathtt{x}_n' = \mathtt{t}_n \ \mathtt{for}\ \mathtt{t} \\ \mathtt{LTerm}(\mathcal{X}) &\ni \mathtt{r} \mid \mathtt{r} \cdot \mathtt{x} \mid \mathtt{t} + \mathtt{s} \end{aligned}$$

where $\mathtt{x} \in \mathcal{X}$, $\mathtt{r} \in \mathbb{R}$, $\mathtt{t}_i, \mathtt{t}, \mathtt{s} \in \mathtt{LTerm}(\mathcal{X})$. An atomic program is thus either a classical assignment $\mathtt{x} := \mathtt{t}$ or a differential statement $\mathtt{x}_1' = \mathtt{t}_1, \ldots, \mathtt{x}_n' = \mathtt{t}_n \ \mathtt{for}\ \mathtt{t}$. The latter reads as "*run the system of differential equations* $\mathtt{x}_1' = \mathtt{t}_1, \ldots, \mathtt{x}_n' = \mathtt{t}_n$ *for* $\mathtt{t}$ *time units*". We then define the while-language via the grammar

$$\mathtt{Prog}(\mathcal{X}) \ni \mathtt{a} \mid \mathtt{p}\,;\mathtt{q} \mid \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q} \mid \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}$$

where $\mathtt{p}, \mathtt{q} \in \mathtt{Prog}(\mathcal{X})$, $\mathtt{a} \in \mathtt{At}(\mathcal{X})$ and $\mathtt{b}$ is an element of the free Boolean algebra generated by the terms $\mathtt{t} \leqslant \mathtt{s}$ and $\mathtt{t} \geqslant \mathtt{s}$. The expression $\mathtt{wait}\ \mathtt{t}$ (from the previous section) is encoded as the differential statement $\mathtt{x}_1' = 0, \ldots, \mathtt{x}_n' = 0\ \mathtt{for}\ \mathtt{t}$.

Remark 1. The systems of differential equations that our language allows are always linear. This is not to say that we could not consider more expressive systems; in fact we could straightfowardly extend the language in this direction, for its semantics (presented below) is not impacted by specific choices of solvable systems of differential equations. But here we do not focus on such choices regarding the expressivity of continuous dynamics and concentrate on a core hybrid semantics instead on which to study the fundamentals of hybrid programming.

In the sequel we abbreviate differential statements $\mathtt{x}_1' = \mathtt{t}_1, \ldots, \mathtt{x}_n' = \mathtt{t}_n \mathtt{for} \mathtt{t}$ to the expression $\bar{\mathtt{x}}' = \bar{\mathtt{t}}$ $\mathtt{for}$ $\mathtt{t}$, where $\bar{\mathtt{x}}'$ and $\bar{\mathtt{t}}$ abbreviate the corresponding vectors of variables $\mathtt{x}_1' \ldots \mathtt{x}_n'$ and linear-combination terms $\mathtt{t}_1 \ldots \mathtt{t}_n$. We call functions of type $\sigma \colon \mathcal{X} \to \mathbb{R}$ *environments*; they map variables to the respective valuations. We use the notation $\sigma \nabla [\bar{\mathtt{v}}/\bar{\mathtt{x}}]$ to denote the environment that maps each $\mathtt{x_i}$ in $\bar{\mathtt{x}}$ to $\mathtt{v_i}$ in $\bar{\mathtt{v}}$ and the rest of variables in the same way as σ. Finally, we denote by $\phi_\sigma^{\bar{\mathtt{x}}'=\bar{\mathtt{t}}} \colon [0, \infty) \to \mathbb{R}^n$ the solution of a system of differential equations $\bar{\mathtt{x}}' = \bar{\mathtt{t}}$ with σ determining the initial condition. When clear from context, we omit the superscript in $\phi_\sigma^{\bar{\mathtt{x}}=\bar{\mathtt{t}}}$. For a linear-combination term $\mathtt{t}$ the expression $\mathtt{t}\sigma$ denotes the corresponding interpretation according to σ and analogously for $\mathtt{b}\sigma$ where $\mathtt{b}$ is a Boolean expression.

We now introduce a small-step operational semantics for our language. Intuitively, the semantics establishes a set of rules for reducing a triple ⟨program statement, environment, time instant⟩ to an environment, via a *finite* sequence of reduction steps. The rules are presented in Figure 2. The terminal configuration $\langle \mathit{skip}, \sigma, \mathtt{t} \rangle$ represents a successful end of a computation, which can then be fed into another computation (via rule **(seq-skip$^\to$)**). Contrastingly, $\langle \mathit{stop}, \sigma, \mathtt{t} \rangle$ is a terminating configuration that inhibits the execution of subsequent computations. The latter is reflected in rules **(diff-stop$^\to$)** and **(seq-stop$^\to$)** which entail that, depending on the chosen time instant, we do not need to evaluate the whole program, but merely a part of it – consequently, infinite while-loops need not yield infinite reduction sequences (as explained in Remark 2). Note that time $\mathtt{t}$ is consumed when applying the rules **(diff-stop$^\to$)** and **(diff-seq$^\to$)** in correspondence to the duration of the differential statement at hand. The rules **(seq)** and **(seq-skip$^\to$)** correspond to the standard rules of operational semantics for while languages over an imperative store [37].

Remark 2. Putatively infinite while-loops do not necessarily yield infinite reduction steps. Take for example the while-loop below whose iterations have always duration $\mathtt{1}$.

$$\mathtt{x} \coloneqq \mathtt{0}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\} \tag{1}$$

It yields a finite reduction sequence for the time instant ½, as shown below:

$\mathtt{x} \coloneqq \mathtt{0}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}, \sigma, 1/2 \to$

{by the rules **(asg$^\to$)** and **(seq-skip$^\to$)**}

$\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}, \sigma\nabla[\mathtt{0}/\mathtt{x}], 1/2 \to$

{by the rule **(wh-true$^\to$)**}

$$
\begin{array}{llr}
(\textbf{asg}^{\rightarrow}) & \mathtt{x} := \mathtt{t}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{skip}, \sigma\nabla[\mathtt{t}\sigma/\mathtt{x}], \mathtt{t} & \\
(\textbf{diff-stop}^{\rightarrow}) & \bar{\mathtt{x}}' = \bar{\mathtt{u}}\;\mathtt{for}\;\mathtt{t}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{stop}, \sigma\nabla[\phi_\sigma(\mathtt{t})/\bar{\mathtt{x}}], \mathtt{0} & (\mathit{if}\;\; t < \mathtt{t}\sigma) \\
(\textbf{diff-skip}^{\rightarrow}) & \bar{\mathtt{x}}' = \bar{\mathtt{u}}\;\mathtt{for}\;\mathtt{t}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{skip}, \sigma\nabla[\phi_\sigma(\mathtt{t}\sigma)/\bar{\mathtt{x}}], \mathtt{t} - (\mathtt{t}\sigma) & (\mathit{if}\;\; t \geqslant \mathtt{t}\sigma) \\
(\textbf{if-true}^{\rightarrow}) & \mathtt{if}\;\mathtt{b}\;\mathtt{then}\;\mathtt{p}\;\mathtt{else}\;\mathtt{q}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{p}, \sigma, \mathtt{t} & (\mathit{if}\;\; \mathtt{b}\sigma = \top) \\
(\textbf{if-false}^{\rightarrow}) & \mathtt{if}\;\mathtt{b}\;\mathtt{then}\;\mathtt{p}\;\mathtt{else}\;\mathtt{q}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{q}, \sigma, \mathtt{t} & (\mathit{if}\;\; \mathtt{b}\sigma = \bot) \\
(\textbf{wh-true}^{\rightarrow}) & \mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{p}\,;\mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}, \sigma, \mathtt{t} & (\mathit{if}\;\; \mathtt{b}\sigma = \top) \\
(\textbf{wh-false}^{\rightarrow}) & \mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{skip}, \sigma, \mathtt{t} & (\mathit{if}\;\; \mathtt{b}\sigma = \bot)
\end{array}
$$

$$
(\textbf{seq-stop}^{\rightarrow})\;\; \frac{\mathtt{p}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{stop}, \sigma', \mathtt{t}'}{\mathtt{p}\,;\mathtt{q}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{stop}, \sigma', \mathtt{t}'} \qquad (\textbf{seq-skip}^{\rightarrow})\;\; \frac{\mathtt{p}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{skip}, \sigma', \mathtt{t}'}{\mathtt{p}\,;\mathtt{q}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{q}, \sigma', \mathtt{t}'}
$$

$$
(\textbf{seq}^{\rightarrow})\;\; \frac{\mathtt{p}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{p}', \sigma', \mathtt{t}'}{\mathtt{p}\,;\mathtt{q}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{p}';\mathtt{q}, \sigma', \mathtt{t}'} \qquad (\mathit{if}\;\; \mathtt{p}' \neq \mathit{stop}\;\; \mathit{and}\;\; \mathtt{p}' \neq \mathit{skip})
$$

Fig. 2: Small-step Operational Semantics

$\mathtt{x} := \mathtt{x} + 1\;;\;\mathtt{wait}\;1\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\,\mathtt{x} := \mathtt{x} + 1\;;\;\mathtt{wait}\;1\,\}, \sigma\nabla[0/\mathtt{x}], 1/2 \rightarrow$

$\quad$ {by the rules $(\textbf{asg}^{\rightarrow})$ and $(\textbf{seq-skip}^{\rightarrow})$}

$\mathtt{wait}\;1\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\,\mathtt{x} := \mathtt{x} + 1\;;\;\mathtt{wait}\;1\,\}, \sigma\nabla[0 + 1/\mathtt{x}], 1/2 \rightarrow$

$\quad$ {by the rules $(\textbf{diff-stop}^{\rightarrow})$ and $(\textbf{seq-stop}^{\rightarrow})$}

$\mathit{stop}, \sigma\nabla[0 + 1/\mathtt{x}], 0$

The gist is that to evaluate program (1) at time instant 1/2, one only needs to unfold the underlying loop until surpassing 1/2 in terms of execution time. Note that if the wait statement is removed from the program then the reduction sequence would not terminate, intuitively because all iterations would be instantaneous and thus the total execution time of the program would never reach 1/2.

The following theorem entails that our semantics is deterministic, which is instrumental for our implementation.

Theorem 1. *For every program* $\mathtt{p}$*, environment* σ*, and time instant* t *there is* at most one *applicable reduction rule.*

Let $\rightarrow^\star$ be the transitive closure of the reduction relation $\rightarrow$ that was previously presented.

Corollary 1. *For every program term* $\mathtt{p}$*, environments* σ*,* σ'*,* σ''*, time instants* $\mathtt{t}$*,* $\mathtt{t}'$*,* $\mathtt{t}''$*, and termination flags* $\mathtt{s}, \mathtt{s}' \in \{\mathit{skip}, \mathit{stop}\}$*, if* $\mathtt{p}, \sigma, \mathtt{t} \rightarrow^\star \mathtt{s}, \sigma', \mathtt{t}'$ *and* $\mathtt{p}, \sigma, \mathtt{t} \rightarrow^\star \mathtt{s}', \sigma'', \mathtt{t}''$*, then the equations* $\mathtt{s} = \mathtt{s}'$*,* $\sigma' = \sigma''$ *and* $\mathtt{t}' = \mathtt{t}''$ *must hold.*

Proof. Follows by induction on the number of reduction steps and Theorem 1. □

As alluded above, the operational semantics treats time as a resource. This is formalised below.

Proposition 1. *For all program terms* $\mathtt{p}$ *and* $\mathtt{q}$*, environments* σ *and* σ'*, and time instants* $\mathtt{t}$*,* $\mathtt{t}'$ *and* $\mathtt{s}$*, if* $\mathtt{p}, \sigma, \mathtt{t} \;\rightarrow\; \mathtt{q}, \sigma', \mathtt{t}'$ *then* $\mathtt{p}, \sigma, \mathtt{t} + \mathtt{s} \;\rightarrow\; \mathtt{q}, \sigma', \mathtt{t}' + \mathtt{s}$*; and if* $\mathtt{p}, \sigma, \mathtt{t} \;\rightarrow\; \mathit{skip}, \sigma', \mathtt{t}'$ *then* $\mathtt{p}, \sigma, \mathtt{t} + \mathtt{s} \;\rightarrow\; \mathit{skip}, \sigma', \mathtt{t}' + \mathtt{s}$.

4 Towards Denotational Semantics: The Hybrid Monad

A mainstream subsuming paradigm in denotational semantics is due to Moggi [24,25], who proposed to identify a *computational effect* of interest as a monad, around which the denotational semantics is built using standard generic mechanisms, prominently provided by category theory. In this section we recall necessary notions and results, motivated by this approach, to prepare ground for our main constructions in the next section.

Definition 1 (Monad). *A monad* $\mathbf{T}$ *(on the category of sets and functions) is given by a triple* $(T, \eta, (\text{–})^\star)$*, consisting of an endomap* T *over the class of all sets, together with a set-indexed class of maps* $\eta_X \colon X \to TX$ *and a so-called* Kleisli lifting *sending each* $f \colon X \to TY$ *to* $f^\star \colon TX \to TY$ *and obeying* monad laws*:* $\eta^\star = \mathsf{id}$*,* $f^\star \cdot \eta = f$*,* $(f^\star \cdot g)^\star = f^\star \cdot g^\star$ *(it follows from this definition that* T *extends to a functor and* η *to a natural transformation).*

A monad morphism $\theta \colon \mathbf{T} \to \mathbf{S}$ *from* $(T, \eta^{\mathbf{T}}, (\text{–})^{\star\mathbf{T}})$ *to* $(S, \eta^{\mathbf{S}}, (\text{–})^{\star\mathbf{S}})$ *is a natural transformation* $\theta \colon T \to S$ *such that* $\theta \cdot \eta^{\mathbf{T}} = \eta^{\mathbf{S}}$ *and* $\theta \cdot f^{\star\mathbf{T}} = (\theta \cdot f)^{\star\mathbf{S}} \cdot \theta$*.*

We will continue to use bold capitals (e.g. $\mathbf{T}$) for monads over the corresponding endofunctors written as capital Romans (e.g. T).

In order to interpret while-loops one needs additional structure on the monad.

Definition 2 (Elgot Monad). *A monad* $\mathbf{T}$ *is called* Elgot *if it is equipped with an* iteration *operator* $(\text{–})^\dagger$ *that sends each* $f \colon X \to T(Y \uplus X)$ *to* $f^\dagger \colon X \to TY$ *in such a way that certain established axioms of iteration are satisfied [2,16].*

Monad morphisms between Elgot monads are additionally required to preserve iteration: $\theta \cdot f^{\dagger\mathbf{T}} = (\theta \cdot f)^{\dagger\mathbf{S}}$ *for* $\theta \colon \mathbf{T} \to \mathbf{S}$*,* $f \colon X \to T(Y \uplus X)$*.*

For a monad $\mathbf{T}$, a map $f \colon X \to TY$, called a *Kleisli map*, is roughly to be regarded as a semantics of a program $\mathtt{p}$, with X as the semantics of the input, and Y as the semantics of the output. For example, with T being the *maybe monad* $(\text{–}) \uplus \{\bot\}$, we obtain semantics of programs as partial functions. Let us record this example in more detail for further reference.

Example 1 (Maybe Monad $\mathbf{M}$*).* The maybe monad is determined by the following data: $MX = X \uplus \{\bot\}$, the unit is the left injection $\mathsf{inl} \colon X \to X \uplus \{\bot\}$ and given $f \colon X \to Y \uplus \{\bot\}$, $f^\star$ is equal to the copairing $[f, \mathsf{inr}] \colon X \uplus \{\bot\} \to Y \uplus \{\bot\}$.

It follows by general considerations (enrichment of the category of Kleisli maps over complete partial orders) that $\mathbf{M}$ is an Elgot monad with the following iteration operator $(\text{–})^\natural$: given $f \colon X \to (Y \uplus X) \uplus \{\bot\}$, and $x_0 \in X$, let $x_0, x_1, \ldots$ be the longest (finite or infinite) sequence over X constructed inductively in such a way that $f(x_i) = \mathsf{inl}\,\mathsf{inr}\, x_{i+1}$. Now, $f^\natural(x_0) = \mathsf{inr}\,\bot$ if the sequence is infinite or

$f(x_i) = \mathsf{inr}\,\bot$ for some i, and $f^\natural(x_0) = \mathsf{inl}\,y$ if for the last element of the sequence x_n, which must exist, $f(x_n) = \mathsf{inl}\,\mathsf{inl}\,y$.

Other examples of Elgot monad can be consulted e.g. in [16].

The computational effect of *hybridness* can also be captured by a monad, called *hybrid monad* [12,14], which we recall next (in a slightly different but equivalent form). To that end, we also need to recall *Minkowski addition* for subsets of the set $\bar{\mathbb{R}}_+$ of extended non-negative reals (see Section 2): $A + B = \{a + b \mid a \in A, b \in B\}$, e.g. $[a, b] + [c, d] = [a + c, b + d]$ and $[a, b] + [c, d) = [a + c, b + d)$.

Definition 3 (Hybrid Monad H). *The* hybrid monad H *is defined as follows.*

- $HX = \sum_{I\in[0,\mathbb{R}_+]} X^I \uplus \sum_{I\in[0,\bar{\mathbb{R}}_+\rangle} X^I$, *i.e. it is a set of trajectories valued on X and with the domain downclosed. For any $p = \mathsf{inj}\langle I, e\rangle \in HX$ with $\mathsf{inj} \in \{\mathsf{inl}, \mathsf{inr}\}$, let us use the notation $p_{\mathsf{d}} = I$, $p_{\mathsf{e}} = e$, the former being the duration of the trajectory and the latter the trajectory itself. Let also $\varepsilon = \langle\emptyset, !\rangle$.*
- $\eta(x) = \mathsf{inl}\langle[0, 0], \lambda t.\, x\rangle$, *i.e. $\eta(x)$ is a trajectory of duration 0 that returns x.*
- *given $f\colon X \to HY$, we define $f^\star\colon HX \to HY$ via the following clauses:*

$$\begin{aligned}
f^\star(\mathsf{inl}\langle I, e\rangle) &= \mathsf{inj}\langle I + J, \lambda t.\,(f(e^t))^0_{\mathsf{e}} \lhd t < d \rhd (f(e^d))^{t-d}_{\mathsf{e}}\rangle \\
&\qquad\qquad \textit{if } I' = I = [0, d] \textit{ for some } d, f(e^d) = \mathsf{inj}\,\langle J, e'\rangle \\
f^\star(\mathsf{inl}\langle I, e\rangle) &= \mathsf{inr}\langle I', \lambda t.\,(f(e^t))^0_{\mathsf{e}}\rangle \qquad\qquad \textit{if } I' \neq I \\
f^\star(\mathsf{inr}\langle I, e\rangle) &= \mathsf{inr}\langle I', \lambda t.\,(f(e^t))^0_{\mathsf{e}}\rangle
\end{aligned}$$

where $I' = \bigcup\{[0, t] \subseteq I \mid \forall s \in [0, t].\, f(e^s) \neq \mathsf{inr}\,\varepsilon\}$ *and* $\mathsf{inj} \in \{\mathsf{inl}, \mathsf{inr}\}$.

The definition of the hybrid monad H is somewhat intricate, so let us complement it with some explanations (details and further intuitions about the hybrid monad can also be consulted in [12]). The domain HX constitutes three types of trajectories representing different kinds of hybrid computation:

- *(closed) convergent*: $\mathsf{inl}\langle[0, d], e\rangle \in HX$ (e.g. instant termination $\eta(x)$);
- *open divergent*: $\mathsf{inr}\langle[0, d), e\rangle \in HX$ (e.g. instant divergence $\mathsf{inr}\,\varepsilon$ or a trajectory $[0, \infty) \to X$ which represents a computation that runs *ad infinitum*);
- *closed divergent*: $\mathsf{inr}\langle[0, d], e\rangle \in HX$ (representing computations that start to diverge *precisely* after the time instant d).

The Kleisli lifting $f^\star$ works as follows: for a given trajectory $\mathsf{inj}\langle I, e\rangle$, we first calculate the largest interval $I' \subseteq I$ on which the trajectory $\lambda t \in I'. f(e^t)$ does not instantly diverge (i.e. $f(e^t) \neq \mathsf{inr}\,\varepsilon$) throughout, hence I' is either $[0, d']$ or $[0, d')$ for some d'. Now, the first clause in the definition of $f^\star$ corresponds to the successful composition scenario: the argument trajectory $\langle I, e\rangle$ is convergent, and composing f with e as described in the definition of I' does not yield divergence all over I. In that case, we essentially concatenate $\langle I, e\rangle$ with $f(e^d)$, the latter being the trajectory computed by f at the last point of e. The remaining two clauses correspond to various flavours of divergence, including divergence of the input ($\mathsf{inr}\langle I, e\rangle$) and divergences occurring along $f \cdot e$. Incidentally, this explains how closed divergent trajectories may arise: if $I' = [0, d']$ and d' is properly smaller than d, then we diverge precisely *after* d', which is possible e.g. if the program behind f continuously checks a condition which did not fail up until d'.

5 Deconstructing the Hybrid Monad

As mentioned in the introduction, in [14] we used $\mathbf{H}$ for giving semantics to a *functional* language HybCore whose programs are interpreted as morphisms of type $X \to HY$. Here, we are dealing with an *imperative* language, which from a semantic point of view amounts to fixing a type of *states* S, shared between all programs; the semantics of a program is thus restricted to morphisms of type $S \to HS$. As explained next, this allows us to make do with a simpler monad $\mathbf{H}_S$, globally parametrized by S. The new monad $\mathbf{H}_S$ has the property that $H_S S$ is naturally isomorphic to HS. Apart from (relative to $\mathbf{H}$) simplicity, the new monad enjoys further benefits, specifically $\mathbf{H}_S$ is mathematically a better behaved structure, e.g. in contrast to $\mathbf{H}$, Elgot iteration on $\mathbf{H}_S$ is constructed as a least fixed point. Factoring the denotational semantics through $\mathbf{H}_S$ thus allows us to bridge the gap to the operational semantics given in Section 3, and faciliates the soundness and adequacy proof in the forthcoming Section 6.

In order to define $\mathbf{H}_S$, it is convenient to take a slightly broader perspective. We will also need to make a detour through the topic of ordered monoid modules with certain completeness properties so that we can characterise iteration on $\mathbf{H}_S$ as a least fixed point.

Definition 4 (Monoid Module, Generalized Writer Monad [14]). *Given a (not necessarily commutative) monoid* $(\mathbb{M}, +, 0)$, *a* monoid module *is a set* $\mathbb{E}$ *equipped with a map* $\triangleright : \mathbb{M} \times \mathbb{E} \to \mathbb{E}$ *(monoid action), subject to the laws* $0 \triangleright e = e$, $(m + n) \triangleright e = m \triangleright (n \triangleright e)$.

Every monoid-module pair $(\mathbb{M}, \mathbb{E})$ *induces a* generalized writer monad $\mathbf{T} = (T, \eta, (-)^\star)$ *with* $T = \mathbb{M} \times (-) \cup \mathbb{E}$, $\eta_X(x) = \langle 0, x\rangle$, *and*

$$\begin{aligned} f^\star(m,x) &= (m+n,y) && \textit{where} && m \in \mathbb{M},\ x \in X,\ f(x) = \langle n, y\rangle \in \mathbb{M} \times Y \\ f^\star(m,x) &= m \triangleright e && \textit{where} && m \in \mathbb{M},\ x \in X,\ f(x) = e \in \mathbb{E} \\ f^\star(e) &= e && \textit{where} && e \in \mathbb{E} \end{aligned}$$

This generalizes the writer monad $(\mathbb{E} = \emptyset)$ *and the exception monad* $(\mathbb{M} = 1)$.

Example 2. A simple motivating example of a monoid-module pair $(\mathbb{M}, \mathbb{E})$ is the pair $(\mathbb{R}_+, \bar{\mathbb{R}}_+)$ where the monoid operation is addition with 0 as the unit and the monoid action is also addition.

More specifically, we are interested in *ordered monoids* and *(conservatively) complete monoid modules*. These are defined as follows.

Definition 5 (Ordered Monoids, (Conservatively) Complete Monoid Modules [7]). *We call a monoid* $(\mathbb{M}, 0, +)$ *an* ordered monoid *if it is equipped with a partial order* $\leqslant$, *such that* 0 *is the least element of this order and* $+$ *is right-monotone (but not necessarily left-monotone).*

An ordered $\mathbb{M}$-module *w.r.t. an ordered monoid* $(\mathbb{M}, +, 0, \leqslant)$, *is an* $\mathbb{M}$*-module* $(\mathbb{E}, \triangleright)$ *together with a partial order* $\sqsubseteq$ *and a least element* $\bot$, *such that* $\triangleright$ *is* *monotone on the right and* $(- \rhd \bot)$ *is monotone, i.e.*

$$\frac{}{\bot \sqsubseteq x} \qquad \frac{x \sqsubseteq y}{a \rhd x \sqsubseteq a \rhd y} \qquad \frac{a \leqslant b}{a \rhd \bot \sqsubseteq b \rhd \bot}$$

We call the last property restricted left monotonicity*.*

An ordered $\mathbb{M}$*-module is* (ω-)complete *if for every* ω*-chain* $s_1 \sqsubseteq s_2 \sqsubseteq \ldots$ *on* $\mathbb{E}$ *there is a least upper bound* $\bigsqcup_i s_i$ *and* $\rhd$ *is continuous on the right, i.e.*

$$\frac{}{\forall i.\, s_i \sqsubseteq \bigsqcup_i s_i} \qquad \frac{\forall i.\, s_i \sqsubseteq x}{\bigsqcup_i s_i \sqsubseteq x} \qquad \frac{}{a \rhd \bigsqcup_i s_i \sqsubseteq \bigsqcup_i a \rhd s_i}$$

(the law $\bigsqcup_i a \rhd s_i \sqsubseteq a \rhd \bigsqcup_i s_i$ *is derivable). Such an* $\mathbb{M}$*-module is* conservatively complete *if additionally for every* ω*-chain* $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ *in* $\mathbb{M}$*, such that the least upper bound* $\bigvee_i a_i$ *exists,* $\left(\bigvee_i a_i\right) \rhd \bot = \bigsqcup_i \; a_i \rhd \bot$*.*

A homomorphism $h\colon \mathbb{E} \to \mathbb{F}$ *of (conservatively) complete monoid* $\mathbb{M}$*-modules is required to be monotone and structure-preserving in the following sense:* $h(\bot) = \bot$*,* $h(a \rhd x) = a \rhd h(x)$*,* $h(\bigsqcup_i x_i) = \bigsqcup_i h(x_i)$*.*

The completeness requirement for $\mathbb{M}$-modules has a standard motivation coming from domain theory, where $\sqsubseteq$ is regarded as an *information order* and completeness is needed to ensure that the relevant semantic domain can accommodate infinite behaviours. The conservativity requirement additionally ensures that the least upper bounds, which exist in $\mathbb{M}$ agree with those in $\mathbb{E}$. Our main example is as follows (we will use it for building $\mathbf{H}_S$ and its iteration operator).

Definition 6 (Monoid Module of Trajectories). *The ordered monoid of* finite open trajectories $\left(\mathsf{Trj}_S, {}^\frown, \langle\emptyset, !\rangle, \leqslant\right)$ *over a given set* S*, is defined as follows:* $\mathsf{Trj}_S = \sum_{I \in [0,\mathbb{R}_+)} S^I$*, the unit is the empty trajectory* $\varepsilon = \langle\emptyset, !\rangle$*; summation is concatenation of trajectories* ${}^\frown$*, defined as follows:*

$$\langle [0,d_1), e_1\rangle {}^\frown \langle [0,d_2), e_2\rangle = \langle [0, d_1 + d_2), \lambda t.\; e_1^t \lhd t < d_1 \rhd e_2^{t-d_1}\rangle.$$

The relation $\leqslant$ *is defined as follows:* $\langle [0,d_1), e_1\rangle \leqslant \langle [0,d_2), e_2\rangle$ *if* $d_1 \leqslant d_2$ *and* $e_1^t = e_2^t$ *for every* $t \in [0, d_1)$*. We can additionally consider both sets* $\sum_{I \in [0,\bar{\mathbb{R}}_+)} S^I$ *and* $\sum_{I \in [0,\bar{\mathbb{R}}_+]} S^I$ *as* Trj_S*-modules, by defining the monoid action* $\rhd$ *also as concatenation of trajectories and by equipping these sets with the order* $\sqsubseteq$*:* $\langle I_1, e_1\rangle \sqsubseteq \langle I_2, e_2\rangle$ *if* $I_1 \subseteq I_2$ *and* $e_1^t = e_2^t$ *for all* $t \in I_1$*.*

Consider the following functors:

$$H'_S X = \sum\nolimits_{I \in [0,\mathbb{R}_+)} S^I \times X \cup \sum\nolimits_{I \in [0,\bar{\mathbb{R}}_+)} S^I \qquad (2)$$

$$H_S X = \sum\nolimits_{I \in [0,\mathbb{R}_+)} S^I \times X \cup \sum\nolimits_{I \in [0,\bar{\mathbb{R}}_+]} S^I \qquad (3)$$

Both of them extend to monads $\mathbf{H}'_S$ and $\mathbf{H}_S$ as they are instances of Definition 4. Moreover, it is laborious but straightforward to prove that both $H'_S X$ and $H_S X$ are conservatively complete Trj_S-modules on X [7], i.e. conservatively complete

Trj_S-modules, equipped with distinguished maps $\eta\colon X \to H'_S X$, $\eta\colon X \to H_S X$. In each case η sends $x \in X$ to $\langle \varepsilon, x\rangle$. The partial order on $H'_S X$ (which we will use for obtaining the least upper bound of a certain sequence of approximations) is given by the clauses below and relies on the previous order $\leqslant$ on trajectories:

$$\frac{}{\langle\langle I, e\rangle, x\rangle \sqsubseteq \langle\langle I, e\rangle, x\rangle} \qquad \frac{\langle I, e\rangle \leqslant \langle I', e'\rangle}{\langle I, e\rangle \sqsubseteq \langle\langle I', e'\rangle, x\rangle} \qquad \frac{\langle I, e\rangle \leqslant \langle I', e'\rangle}{\langle I, e\rangle \sqsubseteq \langle I', e'\rangle}$$

The monad given by (2) admits a sharp characterization, which is an instance of a general result [7]. In more detail,

Proposition 2. *The pair $(H'_S X, \eta)$ is a* free conservatively complete Trj_S-module *on X, i.e. for every conservatively complete Trj_S-module $\mathbb{E}$ and a map $f\colon X \to \mathbb{E}$, there is unique homomorphism $\hat{f}\colon H'_S X \to \mathbb{E}$ such that $\hat{f} \cdot \eta = f$.*

Intuitively, Proposition 2 ensures that $H'_S X$ is a *least* conservatively complete Trj_S-module generated by X. This characterization entails a construction of an iteration operator on $\mathbf{H}'_S$ as a least fixpoint. This, in fact, also transfers to $\mathbf{H}_S$ (as detailed in the proof of the following theorem).

Theorem 2. *Both $\mathbf{H}'_S$ and $\mathbf{H}_S$ are Elgot monads, for which $f^\dagger$ is computed as a least fixpoint of ω-continuous endomaps $g \mapsto [\eta, g]^\star \cdot f$ over the function spaces $X \to H'_S Y$ and $X \to H_S Y$ correspondingly.*

In this section's remainder, we formally connect the monad $\mathbf{H}_S$ with the monad $\mathbf{H}$, the latter introduced in our previous work and used for providing a semantics to the functional language HybCore. In the following section we provide a semantics for the current imperative language via the monad $\mathbf{H}_S$. Specifically, in this section we will show how to build $\mathbf{H}$ from $\mathbf{H}_S$ by considering additional semantic ingredients on top of the latter.

Let us subsequently write η^S, $(-)^\star_S$ and $(-)^\dagger_S$ for the unit, the Kleisli lifting and the Elgot iteration of $\mathbf{H}_S$. Note that $S, X \mapsto \mathbf{H}_S X$ is a *parametrized monad* in the sense of Uustalu [35], in particular H_S is functorial in S and for every $f\colon S \to S'$, $H_f\colon H_S \to H_{S'}$ is a monad morphism.

Then we introduce the following technical natural transformations $\iota\colon H_S X \to X \uplus (S \uplus \{\bot\})$ and $\tau\colon H_{S \uplus Y} X \to H_S X$. First, let us define ι:

$$\iota(I, e, x) = \begin{cases} \mathsf{inr}\,\mathsf{inl}\, e^0, & \text{if } I \neq \emptyset \\ \mathsf{inl}\, x, & \text{otherwise} \end{cases} \qquad \iota(I, e) = \begin{cases} \mathsf{inr}\,\mathsf{inl}\, e^0, & \text{if } I \neq \emptyset \\ \mathsf{inr}\,\mathsf{inr}\,\bot, & \text{otherwise} \end{cases}$$

In words: ι returns the initial point for non-zero length trajectories, and otherwise returns either an accompanying value from X or $\bot$ depending on that if the given trajectory is convergent or divergent. The functor $(-) \uplus E$ for every E extends to a monad, called the *exception monad*. The following is easy to show for ι.

Lemma 1. *For every S, $\iota\colon H_S \to (-) \uplus (S \uplus \{\bot\})$ is a monad morphism.*

Next we define $\tau\colon H_{S \uplus Y} X \to H_S X$:

$$\tau(I, e, x) = \begin{cases} \langle I, e, x\rangle, & \text{if } I = I' \\ \langle I', e'\rangle, & \text{otherwise} \end{cases} \qquad \tau(I, e) = \langle I', e'\rangle$$

where $\langle I', e'\rangle$ is the largest such trajectory that for all $t \in I'$, $e^t = \mathsf{inl}\, e'^t$.

$$
\begin{aligned}
[\![\mathtt{x} := \mathtt{t}]\!](\sigma) &= \eta(\sigma \triangledown [\mathtt{t}\sigma/\mathtt{x}]) \\
[\![\bar{\mathtt{x}}' = \bar{\mathtt{u}}\ \mathtt{for}\ \mathtt{t}]\!](\sigma) &= \langle [0, \mathtt{t}\sigma), \lambda t.\, \sigma \triangledown [\phi_\sigma(t)/\bar{\mathtt{x}}], \sigma \triangledown [\phi_\sigma(\mathtt{t}\sigma)/\bar{\mathtt{x}}] \rangle \\
[\![\mathtt{p}\ ;\ \mathtt{q}]\!](\sigma) &= [\![\mathtt{q}]\!]^\star([\![\mathtt{p}]\!](\sigma)) \\
[\![\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}]\!](\sigma) &= [\![\mathtt{p}]\!](\sigma) \lhd \mathtt{b}\sigma \rhd [\![\mathtt{q}]\!](\sigma) \\
[\![\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}]\!](\sigma) &= (\lambda\sigma.\, (\hat{H}\,\mathsf{inr})([\![\mathtt{p}]\!](\sigma)) \lhd \mathtt{b}\sigma \rhd \eta(\mathsf{inl}\,\sigma))^\dagger(\sigma)
\end{aligned}
$$

Fig. 3: Denotational semantics.

Lemma 2. *For all S and Y, $\tau\colon H_{S \uplus Y} \to H_S$ is a monad morphism.*

We now arrive at the main result of this section.

Theorem 3. *The correspondence $S \mapsto H_S S$ extends to an Elgot monad as follows:*

$$
\begin{aligned}
\eta(x \in S) &= \eta^S(x), \\
(f\colon X \to H_S S)^\star &= \big(H_X X \xrightarrow{H_{\iota' \cdot f}\, \mathsf{id}} H_{S \uplus \{\bot\}} X \xrightarrow{\tau} H_S X \xrightarrow{f^\star_S} H_S S\big), \\
(f\colon X \to H_{S \uplus X}(S \uplus X))^\dagger &= \big(X \xrightarrow{f^\dagger_{S \uplus X}} H_{S \uplus X} S \xrightarrow{H_{[\mathsf{inl}, (\iota' \cdot f)^\natural]}\, \mathsf{id}} H_{S \uplus \{\bot\}} S \xrightarrow{\tau} H_S S\big).
\end{aligned}
$$

where $\iota' = [\mathsf{inl}, \mathsf{id}] \cdot \iota\colon H_S S \to S \uplus \{\bot\}$ and $(-)^\natural\colon (X \to (S \uplus X) \uplus \{\bot\}) \to (X \to S \uplus \{\bot\})$ is the iteration operator of the maybe-monad $(-) \uplus \{\bot\}$ (as in Example 1). Moreover, thus defined monad is isomorphic to $\mathbf{H}$.

Proof (Proof Sketch). It is first verified that the monad axioms are satisfied using abstract properties of ι and τ, mainly provided by Lemmas 1 and 2. Then the isomorphism $\theta\colon H_S S \cong HS$ is defined as expected: $\theta([0, d), e, x) = \mathsf{inl}\langle [0, d], \hat{e}\rangle$ where $e^t = \hat{e}^t$ for $t \in [0, d)$, $\hat{e}^d = x$; and $\theta(I, e) = \mathsf{inr}\langle I, e\rangle$. It is easy to see that θ respects the unit. The fact that θ respects Kleisli lifting amounts to a (tedious) verification by case distinction. Checking the formula for $(-)^\dagger$ amounts to transferring the definition of $(-)^\dagger$, as defined in previous work [13], along θ. See the full proof in [15]. $\square$

6 Soundness and Adequacy

Let us start this section by providing a denotational semantics to our language using the results of the previous section. We will then provide a soundness and adequacy result that formally connects the thus established denotational semantics with the operational semantics presented in Section 3.

First, consider the monad in (3) and fix $S = \mathbb{R}^{\mathcal{X}}$. We denote the obtained instance of H_S as $\hat{H}$. Intuitively, we interpret a program $\mathtt{p}$ as a map $[\![\mathtt{p}]\!] : S \to \hat{H}S$ which given an environment (a map from variables to values) returns a trajectory over S. The definition of $[\![\mathtt{p}]\!]$ is inductive over the structure of $\mathtt{p}$ and is given in Figure 3.

In order to establish soundness and adequacy between the small-step operational semantics and the denotational semantics, we will use an auxiliary device. Namely, we will introduce a *big-step* operational semantics that will serve as midpoint between the two previously introduced semantics. We will show that the small-step semantics is equivalent to the big-step one and then establish soundness and adequacy between the big-step semantics and the denotational one. The desired result then follows by transitivity. The big-step rules are presented in Figure 4 and follow the same reasoning than the small-step ones. The expression $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'$ means that $\mathtt{p}$ paired with σ evaluates to $\mathtt{r}, \sigma'$ at time instant $\mathtt{t}$.

$$\textbf{(diff-stop}\Downarrow\textbf{)}\quad \frac{\mathtt{t} < \mathtt{s}\sigma}{\bar{\mathtt{x}}' = \bar{\mathtt{t}}\ \mathtt{for}\ \mathtt{s}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma\nabla[\phi_\sigma(\mathtt{t})/\bar{\mathtt{x}}]}$$

$$\textbf{(diff-skip}\Downarrow\textbf{)}\quad \frac{}{\bar{\mathtt{x}}' = \bar{\mathtt{t}}\ \mathtt{for}\ \mathtt{t}, \sigma, \mathtt{t}\sigma \Downarrow \mathit{skip}, \sigma\nabla[\phi_\sigma(\mathtt{t}\sigma)/\bar{\mathtt{x}}]}$$

$$\textbf{(asg}\Downarrow\textbf{)}\quad \frac{}{\mathtt{x} := \mathtt{t}, \sigma, 0 \Downarrow \mathit{skip}, \sigma\nabla[\mathtt{t}\sigma/\mathtt{x}]} \qquad \textbf{(seq-stop}\Downarrow\textbf{)}\quad \frac{\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma'}{\mathtt{p}\,;\mathtt{q}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma'}$$

$$\textbf{(seq-skip}\Downarrow\textbf{)}\quad \frac{\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{skip}, \sigma' \qquad \mathtt{q}, \sigma', \mathtt{t}' \Downarrow \mathtt{r}, \sigma''}{\mathtt{p}\,;\mathtt{q}, \sigma, \mathtt{t}+\mathtt{t}' \Downarrow \mathtt{r}, \sigma''} \qquad (\mathtt{r} \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(if-true}\Downarrow\textbf{)}\quad \frac{\mathtt{b}\sigma = \top \qquad \mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'}{\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'} \qquad (\mathtt{r} \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(if-false}\Downarrow\textbf{)}\quad \frac{\mathtt{b}\sigma = \bot \qquad \mathtt{q}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'}{\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'} \qquad (\mathtt{r} \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(wh-true}\Downarrow\textbf{)}\quad \frac{b\sigma = \top \qquad \mathtt{p}\,;\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'}{\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, \mathtt{t} \Downarrow \mathtt{r}, \sigma'} \qquad (\mathtt{r} \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(wh-false}\Downarrow\textbf{)}\quad \frac{b\sigma = \bot}{\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, 0 \Downarrow \mathit{skip}, \sigma}$$

Fig. 4: Big-step Operational Semantics

Next, we need the following result to formally connect both styles of operational semantics.

Lemma 3. *Given a program* $\mathtt{p}$*, an environment* σ *and a time instant* $\mathtt{t}$

1. *if* $\mathtt{p}, \sigma, \mathtt{t} \rightarrow \mathtt{p}', \sigma', \mathtt{t}'$ *and* $\mathtt{p}', \sigma', \mathtt{t}' \Downarrow \mathit{skip}, \sigma''$ *then* $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{skip}, \sigma''$*;*
2. *if* $\mathtt{p}, \sigma, \mathtt{t} \rightarrow \mathtt{p}', \sigma', \mathtt{t}'$ *and* $\mathtt{p}', \sigma', \mathtt{t}' \Downarrow \mathit{stop}, \sigma''$ *then* $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma''$*.*

Proof. The proof follows by induction over the derivation of the small step relation. □

Theorem 4. *The small-step semantics and the big-step semantics are related as follows. Given a program* $\mathtt{p}$*, an environment* σ *and a time instant* $\mathtt{t}$

1. $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{skip}, \sigma'$ *iff* $\mathtt{p}, \sigma, \mathtt{t} \to^\star \mathit{skip}, \sigma', 0$*;*
2. $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma'$ *iff* $\mathtt{p}, \sigma, \mathtt{t} \to^\star \mathit{stop}, \sigma', 0$*.*

Proof. The right-to-left direction is obtained by induction over the length of the small-step reduction sequence using Lemma 3. The left-to-right direction follows by induction over the proof of the big-step judgement using Proposition 1. □

Finally, we can connect the operational and the denotational semantics in the expected way.

Theorem 5 (Soundness and Adequacy). *Given a program* $\mathtt{p}$*, an environment* σ *and a time instant* $\mathtt{t}$

1. $\mathtt{p}, \sigma, \mathtt{t} \to^\star \mathit{skip}, \sigma', 0$ *iff* $[\![\mathtt{p}]\!](\sigma) = (\mathtt{h}\colon [0,\mathtt{t}) \to \mathbb{R}^{\mathcal{X}}, \sigma')$*;*
2. $\mathtt{p}, \sigma, \mathtt{t} \to^\star \mathit{stop}, \sigma', 0$ *iff either* $[\![\mathtt{p}]\!](\sigma) = (\mathtt{h}\colon [0,\mathtt{t}') \to \mathbb{R}^{\mathcal{X}}, \sigma'')$ *or* $[\![\mathtt{p}]\!](\sigma) = \mathtt{h}\colon [0,\mathtt{t}') \to \mathbb{R}^{\mathcal{X}}$*, and in either case with* $\mathtt{t}' > \mathtt{t}$ *and* $h(\mathtt{t}) = \sigma'$*.*

Here, "soundness" corresponds to the left-to-right directions of the equivalences and "adequacy" to the right-to-left ones.

Proof. By Theorem 4, we equivalently replace the goal as follows:

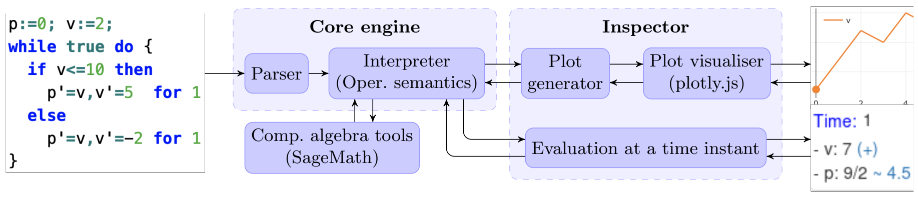
1. $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{skip}, \sigma'$ iff $[\![\mathtt{p}]\!](\sigma) = (\mathtt{h}\colon [0,\mathtt{t}) \to \mathbb{R}^{\mathcal{X}}, \sigma')$;
2. $\mathtt{p}, \sigma, \mathtt{t} \Downarrow \mathit{stop}, \sigma'$ iff either $[\![\mathtt{p}]\!](\sigma) = (\mathtt{h}\colon [0,\mathtt{t}') \to \mathbb{R}^{\mathcal{X}}, \sigma'')$ or $[\![\mathtt{p}]\!](\sigma) = \mathtt{h}\colon [0,\mathtt{t}') \to \mathbb{R}^{\mathcal{X}}$, and in either case with $\mathtt{t}' > \mathtt{t}$ and $h(\mathtt{t}) = \sigma'$.

Then the "soundness" direction is obtained by induction over the derivation of the rules in Fig. 4. The "adequacy" direction follows by structural induction over $\mathtt{p}$; for while-loops, we call the fixpoint law $[\eta, f^\dagger]^\star \cdot f = f^\dagger$ of Elgot monads. □

7 Implementation

This section presents our prototype implementation – Lince – which is available online both to run in our servers and to be compiled and executed locally (http://arcatools.org/lince). Its architecture is depicted in Figure 5. The dashed rectangles correspond to its main components. The one on the left (**Core engine**) provides the parser respective to the while-language and the engine to evaluate hybrid programs using the small-step operational semantics of Section 3. The one on the right (**Inspector**) depicts trajectories produced by hybrid programs according to parameters specified by the user and provides an interface to evaluate hybrid programs at specific time instants (the initial environment $\sigma : \mathcal{X} \to \mathbb{R}$ is assumed to be the function constant on zero). As already mentioned, plots are generated by automatically evaluating at different time instants the program given as input. Incoming arrows in the figure denote an input relation and outgoing arrows denote an output relation. The two main components are further explained below.

Core engine. Our implementation extensively uses the computer algebra tool SageMath [31]. This serves two purposes: (1) to solve systems of differential



```
p:=0; v:=2;
while true do {
  if v<=10 then
    p'=v,v'=5  for 1
  else
    p'=v,v'=-2 for 1
}
```

Core engine
Parser
Interpreter (Oper. semantics)
Comp. algebra tools (SageMath)
Inspector
Plot generator
Plot visualiser (plotly.js)
Evaluation at a time instant
v
Time: 1
- v: 7 (+)
- p: 9/2 ~ 4.5

Fig. 5: Depiction of Lince's architecture

equations (present in hybrid programs); and (2) to correctly evaluate if-then-else statements. Regarding the latter, note that we do not merely use predicate functions in programming languages for evaluating Boolean conditions, essentially because such functions tend to give wrong results in the presence of real numbers (due to the finite precision problem). Instead of this, Lince uses SageMath and its ability to perform advanced symbolic manipulation to check whether a Boolean condition is true or not. However, note that this will not always give an output, fundamentally because solutions of linear differential equations involve transcendental numbers and real-number arithmetic with such numbers is undecidable [20]. We leave as future work the development of more sophisticated techniques for avoiding errors in the computational evaluation of hybrid programs.

Inspector. The user interacts with Lince at two different stages: (a) when inputting a hybrid program and (b) when inspecting trajectories using Lince's output interfaces. The latter case consists of adjusting different parameters for observing the generated plots in an optimal way.

Event-triggered programs. Observe that the differential statements $\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\ \mathtt{for}\ \mathtt{t}$ are *time-triggered*: they terminate precisely when the instant of time $\mathtt{t}$ is achieved. In the area of hybrid systems it is also usual to consider *event-triggered* programs: those that terminate *as soon as* a specified condition ψ becomes true [38,6,11]. So we next consider atomic programs of the type $\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\ \mathtt{until}\ \psi$ where ψ is an element of the free Boolean algebra generated by $\mathtt{t} \leqslant \mathtt{s}$ and $\mathtt{t} \geqslant \mathtt{s}$ where $\mathtt{t}, \mathtt{s} \in \mathtt{LTerm}(\mathcal{X})$, signalling the termination of the program. In general, it is impossible to determine with *exact* precision when such programs terminate (again due to the undecidability of real-number arithmetic with transcendental numbers). A natural option is to tackle this problem by checking the condition ψ periodically, which essentially reduces event-triggered programs into time-triggered ones. The cost is that the evaluation of a program might greatly diverge from the nominal behaviour, as discussed for instance in documents [4,6] where an analogous approach is discussed for the well-established simulation tools Simulink and Modelica. In our case, we allow programs of the form $\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\ \mathtt{until}_\epsilon\ \psi$ in the tool and define them as the abbreviation of $\mathtt{while}\ \neg\psi\ \mathtt{do}\ \{\ \mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\ \mathtt{for}\ \epsilon\ \}$. This sort of abbreviation has the advantage of avoiding spurious evaluations of hybrid programs w.r.t. the established semantics. We could indeed easily allow such event-triggered programs natively in our language (i.e. without recurring to

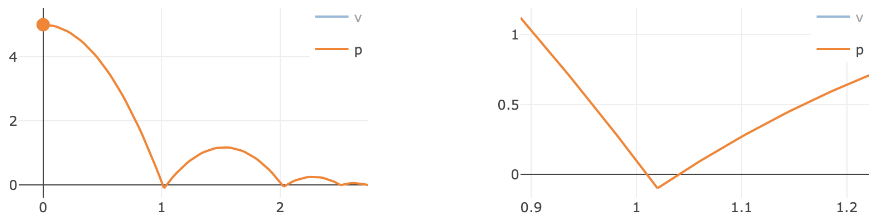


Fig. 6: Position of the bouncing ball over time (plot on the left); zoomed in position of the bouncing ball at the first bounce (plot on the right).

abbreviations) and extend the semantics accordingly. But we prefer not to do this at the moment, because we wish first to fully understand the ways of limiting spurious computational evaluations arising from event-triggered programs.

Remark 3. SIMULINK and MODELICA are powerful tools for simulating hybrid systems, but lack a well-established, formal semantics. This is discussed for example in [3,9], where the authors aim to provide semantics to subsets of SIMULINK and MODELICA. Getting inspiration from control theory, the language of SIMULINK is circuit-like, block-based; the language of MODELICA is *acausal* and thus particularly useful for modelling electric circuits and the like which are traditionally modelled by systems of equations.

Example 3 (Bouncing Ball). As an illustration of the approach described above for event-triggered programs, take a bouncing ball dropped at a positive height $\mathtt{p}$ and with no initial velocity $\mathtt{v}$. Due to the gravitational acceleration $\mathtt{g}$, it falls to the ground and bounces back up, losing part of its kinetic energy in the process. This can be approximated by the following hybrid program

$$(\mathtt{p}' = \mathtt{v}, \mathtt{v}' = \mathtt{g}\ \mathtt{until}_{0.01}\ \mathtt{p} \leqslant 0 \wedge \mathtt{v} \leqslant 0)\,;(\mathtt{v} := \mathtt{v} \times -0.5)$$

where $\mathtt{0.5}$ is the dampening factor of the ball. We now want to drop the ball from a specific height (e.g. $\mathtt{5}$ meters) and let it bounce until it stops. Abbreviating the previous program into $\mathtt{b}$, this behaviour can be approximated by $\mathtt{p} := 5\,;\mathtt{v} := 0\,;\mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\ \mathtt{b}\ \}$. Figure 6 presents the trajectory generated by the ball (calculated by LINCE). Note that since $\epsilon = 0.01$ the ball reaches below ground, as shown in Figure 6 on the right. Other examples of event- and time-triggered programs can be seen in LINCE's website.

8 Conclusions and future work

We introduced small-step and big-step operational semantics for hybrid programs suitable for implementation purposes and provided a denotational counterpart via the notion of Elgot monad. These semantics were then linked by a soundness and adequacy theorem [37]. We regard these results as a stepping stone for developing computational tools and techniques for hybrid programming; which we attested with the development of LINCE. With this work as basis, we plan to explore the following research lines in the near future.

Program equivalence. Our denotational semantics entails a natural notion of program equivalence (denotational equality) which inherently includes classical laws of iteration and a powerful *uniformity* principle [33], thanks to the use of Elgot monads. We intend to further explore the equational theory of our language so that we can safely refactor/simplify hybrid programs. Note that the theory includes equational schema like $(\mathtt{x} := \mathtt{a}\,;\mathtt{x} := \mathtt{b}) = \mathtt{x} := \mathtt{b}$ and $(\mathtt{wait}\ \mathtt{a}\,;\mathtt{wait}\ \mathtt{b}) = \mathtt{wait}\ (\mathtt{a} + \mathtt{b})$ thus encompassing not only usual laws of programming but also axiomatic principles behind the notion of time.

New program constructs. Our while-language is intended to be as simple as possible whilst harbouring the core, uncontroversial features of hybrid programming. This was decided so that we could use the language as both a theoretical and practical basis for advancing hybrid programming. A particular case that we wish to explore next is the introduction of new program constructs, including e.g. non-deterministic or probabilistic choice and exception operations `raise`(`exc`). Denotationally, the fact that we used monadic constructions readily provides a palette of techniques for this process, e.g. tensoring and distributive laws [22,23].

Robustness. A core aspect of hybrid programming is that programs should be *robust*: small variations in their input should *not* result in big changes in their output [32,21]. We wish to extend LINCE with features for detecting non-robust programs. A main source of non-robustness are conditional statements `if b then p else q`: very small changes in their input may change the validity of `b` and consequently cause a switch between (possibly very different) execution branches. Currently, we are working on the systematic detection of non-robust conditional statements in hybrid programs, by taking advantage of the notion of δ-perturbation [20].

Acknowledgements The first author would like to acknowledge support of German Research Council (DFG) under the project *A High Level Language for Monad-based Processes (GO 2161/1–2)*. The second author was financed by the 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 project POCI-01-0145-FEDER-030947. The third author was partially supported by National Funds through FCT/MCTES, within the CISTER Research Unit (UIDB/04234/2020); by COMPETE 2020 under the PT2020 Partnership Agreement, through ERDF, and by national funds through the FCT, within project POCI-01-0145-FEDER-029946; by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through ERDF and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550; and by the FCT within project ECSEL/0016/2019 and the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

References

1. J. Adámek, H. Herrlich, and G. Strecker. *Abstract and concrete categories.* John Wiley & Sons Inc., New York, 1990.
2. J. Adámek, S. Milius, and J. Velebil. Elgot theories: a new perspective on the equational properties of iteration. *Mathematical Structures in Computer Science*, 21(2):417–480, 2011.
3. O. Bouissou and A. Chapoutot. An operational semantics for Simulink's simulation engine. In *ACM SIGPLAN Notices*, vol. 47, pp. 129–138. ACM, 2012.
4. D. Broman. Hybrid simulation safety: Limbos and zero crossings. In *Principles of Modeling*, pp. 106–121. Springer, 2018.
5. Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
6. D. A. Copp and R. G. Sanfelice. A zero-crossing detection algorithm for robust simulation of hybrid systems jumping on surfaces. *Simulation Modelling Practice and Theory*, 68:1–17, 2016.
7. T. L. Diezel and S. Goncharov. Towards Constructive Hybrid Semantics. In Z. M. Ariola, ed., *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, vol. 167 of *LIPIcs*, pp. 24:1–24:19, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
8. C. Elgot. Monadic computation and iterative algebraic theories. In *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 175–230. Elsevier, 1975.
9. S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP semantics for Modelica. In *International Symposium on Unifying Theories of Programming*, pp. 44–64. Springer, 2016.
10. P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach.* John Wiley & Sons, 2014.
11. R. Goebel, R. G. Sanfelice, and A. R. Teel. Hybrid dynamical systems. *IEEE Control Systems*, 29(2):28–93, 2009.
12. S. Goncharov, J. Jakob, and R. Neves. A semantics for hybrid iteration. In *29th International Conference on Concurrency Theory, CONCUR 2018.* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
13. S. Goncharov, J. Jakob, and R. Neves. A semantics for hybrid iteration. *CoRR*, abs/1807.01053, 2018.
14. S. Goncharov and R. Neves. An adequate while-language for hybrid computation. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, PPDP '19, pp. 11:1–11:15, New York, NY, USA, 2019. ACM.
15. S. Goncharov, R. Neves, and J. Proença. Implementing hybrid semantics: From functional to imperative. *CoRR*, abs/2009.14322, 2020.
16. S. Goncharov, L. Schröder, C. Rauch, and M. Piróg. Unifying guarded and unguarded iteration. In *International Conference on Foundations of Software Science and Computation Structures*, pp. 517–533. Springer, 2017.
17. T. A. Henzinger. The theory of hybrid automata. In *LICS96': Logic in Computer Science, 11th Annual Symposium, New Jersey, USA, July 27-30, 1996*, pp. 278–292. IEEE, 1996.
18. P. Höfner and B. Möller. An algebra of hybrid systems. *The Journal of Logic and Algebraic Programming*, 78(2):74 – 97, 2009.
19. J. J. Huerta y Munive and G. Struth. Verifying hybrid systems with modal kleene algebra. In J. Desharnais, W. Guttmann, and S. Joosten, eds., *Relational*

and Algebraic Methods in Computer Science, pp. 225–243, Cham, 2018. Springer International Publishing.

20. S. Kong, S. Gao, W. Chen, and E. Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*, pp. 200–205. Springer, 2015.
21. D. Liberzon and A. S. Morse. Basic problems in stability and design of switched systems. *IEEE Control systems*, 19(5):59–70, 1999.
22. C. Lüth and N. Ghani. Composing monads using coproducts. In M. Wand and S. L. P. Jones, eds., *ICFP'02: Functional Programming, 7th ACM SIGPLAN International Conference, Pittsburgh, USA, October 04 - 06, 2002*, pp. 133–144. ACM, 2002.
23. E. Manes and P. Mulry. Monad compositions I: general constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.
24. E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pp. 14–23. IEEE Computer Society, 1989.
25. E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.
26. R. Neves. *Hybrid programs*. PhD thesis, Minho University, 2018.
27. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time maude. *Higher-order and symbolic computation*, 20(1-2):161–196, 2007.
28. A. Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.
29. A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.
30. R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *DAC'10: Design Automation Conference, 47th ACM/IEEE Conference, Anaheim, USA, June 13-18, 2010*, pp. 731–736. IEEE, 2010.
31. W. Stein et al. *Sage Mathematics Software (Version 6.4.1)*. The Sage Development Team, 2015. `http://www.sagemath.org`.
32. R. Shorten, F. Wirth, O. Mason, K. Wulff, and C. King. Stability criteria for switched and hybrid systems. *Society for Industrial and Applied Mathematics (review)*, 49(4):545–592, 2007.
33. A. Simpson and G. Plotkin. Complete axioms for categorical fixed-point operators. In *Logic in Computer Science, LICS 2000*, pp. 30–41, 2000.
34. K. Suenaga and I. Hasuo. Programming with infinitesimals: A while-language for hybrid system modeling. In *International Colloquium on Automata, Languages, and Programming*, pp. 392–403. Springer, 2011.
35. T. Uustalu. Generalizing substitution. *RAIRO-Theoretical Informatics and Applications*, 37(4):315–336, 2003.
36. R. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *Theories of Concurrency, CONCUR 1990*, vol. 458, pp. 278–297, 1990.
37. G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.
38. H. Witsenhausen. A class of hybrid-state continuous-time dynamic systems. *IEEE Transactions on Automatic Control*, 11(2):161–167, 1966.