Algebraic Temporal Effects: Temporal Verification of Recursively Typed Higher-Order Programs

TARO SEKIYAMA, National Institute of Informatics, Japan and SOKENDAI, Japan
HIROSHI UNNO, Tohoku University, Japan

We present a general form of temporal effects for recursive types. Temporal effects have been adopted by effect systems to verify both linear-time temporal safety and liveness properties of higher-order programs with recursive functions. A challenge in a generalization to recursive types is that recursive types can easily cause unstructured loops, which obscure the regularity of the infinite behavior of computation and make it harder to statically verify liveness properties. To solve this problem, we introduce temporal effects with a later modality, which enable us to capture the behavior of non-terminating programs by stratifying obscure loops caused by recursive types. While temporal effects in the prior work are based on certain concrete formal forms, such as logical formulas and automata-based lattices, our temporal effects, which we call *algebraic temporal effects*, are more abstract, axiomatizing temporal effects in an algebraic manner and clarifying the requirements for temporal effects that can reason about programs soundly. We formulate algebraic temporal effects, formalize an effect system built on top of them, and prove two kinds of soundness of the effect system: safety and liveness soundness. We also introduce two instances of algebraic temporal effects: one is *temporal regular effects*, which are based on ω-regular expressions, and the other is *temporal fixpoint effects*, which are based on a first-order fixpoint logic. Their usefulness is demonstrated via examples including concurrent and object-oriented programs.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Type theory**; • **Software and its engineering** → **Functional languages**.

Additional Key Words and Phrases: temporal verification, recursive types, effect systems, temporal effects

ACM Reference Format:
Taro Sekiyama and Hiroshi Unno. 2025. Algebraic Temporal Effects: Temporal Verification of Recursively Typed Higher-Order Programs. *Proc. ACM Program. Lang.* 9, POPL, Article 78 (January 2025), 31 pages. https://doi.org/10.1145/3704914

1 Introduction

1.1 Background: Effect Systems for Temporal Verification

Temporal program verification aims to reason about whether programs perform effectful operations in a well-organized manner. For example, it can ensure the safety of resource usage [Aldrich et al. 2009; Igarashi and Kobayashi 2002], which is a safety property that resources, such as memory cells and file objects, are manipulated in a correct order. Furthermore, temporal verification involves the aspect of liveness verification. For instance, it can check that an infinite computation releases every allocated resource eventually and every concurrent process is scheduled to be executed at some time [Owicki and Lamport 1982]. We focus on linear-time temporal verification, which reasons

Authors' Contact Information: Taro Sekiyama, National Institute of Informatics, Tokyo, Japan and SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org.

ACM 2475-1421/2025/1-ART78
https://doi.org/10.1145/3704914

about what sequences of events, called traces, are generated by programs. The examples mentioned above can be formulated as linear-time temporal properties.

One approach to (linear-time) temporal verification of higher-order programs is to design a type-and-effect system with a special form of effects, called *temporal effects* [Hofmann and Chen 2014; Koskinen and Terauchi 2014; Nanjo et al. 2018], which represent traces that a program may yield. A benefit of such systems, called *temporal effect systems*, is to inherit a common nature of effect systems: compositionality. That is, the temporal effect systems can verify expressions only using the static information of their subexpressions.

The existing temporal effect systems have focused on higher-order languages that are equipped with recursive functions as the only primitive that causes infinite computation. Such a language makes it easier to prove soundness of the liveness verification with the temporal effects. When infinite computation is caused only by a recursive function, the trace of the computation should be generated by executing the body of the recursive function infinitely many times. The temporal effect systems in the literature implement inference rules exploiting this nature of their target languages.

However, their formalism specialized in recursive functions is not satisfactory because there are programming constructs that are useful and powerful enough to perform infinite computation without structuring loops explicitly as recursive functions. A crucial example of such constructs is *recursive types*. The use of recursive types enables us to implement the fixed-point combinator [Morris 1969], but their significance is not only that. More critically, a language with recursive types can implement a variety of common programming features and paradigms such as recursive data structures, object-oriented programming [Mitchell 1990], and concurrency [Kobayashi and Igarashi 2013]. Extending temporal effect systems to recursive types enables verification of temporal properties of programs with such features and paradigms implementable through recursive types.

1.2 This Work

The purpose of this work is to generalize temporal effect systems to recursive types. A challenge involved in this generalization is that infinite computation caused by recursive types is unstructured, that is, it cannot be determined lexically in general. For example, consider the following program:

$$\mathsf{let}\, f : (\mathsf{rec}\, \alpha.\alpha \longrightarrow \mathsf{unit}) \longrightarrow \mathsf{unit} = \lambda x.(\mathsf{unfold}\, x)\, x\, \mathsf{in}\, f\ (\mathsf{fold}\, f)$$

which implements the Ω combinator with recursive type $\mathsf{rec}\, \alpha.\alpha \longrightarrow \mathsf{unit}$. The infinity of this program is caused by two separate operations for recursive types: fold and unfold. Thus, we need a means to identify when and what infinite computation arises by their unstructured use. Note that the use of fold and unfold does *not* necessarily give rise to infinite computation. For example, consider finite lists over integers given as a recursive type $\mathsf{rec}\, \alpha.\mathsf{unit} + (\mathsf{int} \times \alpha)$. We can implement a total function that returns whether a given list is empty using unfold. Therefore, we need a sophisticated method to reason about various computations caused by the flexible use of the fold and unfold operations.

A key fact we utilize to address this challenge is that in a language with recursive types, infinite computation is caused by an infinite series of unfolded computations. For example, the above divergent program applies function f to $\mathsf{fold}\, f$ that inhabits recursive type $\mathsf{rec}\, \alpha.\alpha \longrightarrow \mathsf{unit}$. The application leads to a computation that applies f again after unfolding $\mathsf{fold}\, f$. As a result, the unfolding and running of the computation involved in $\mathsf{fold}\, f$ is repeated infinitely and leads to the divergence. This fact indicates that identifying the trace of infinite computation can be reduced to identifying the events performed by unfolded computations.

An established approach to organizing possibly infinitely many unfoldings is to use a *later modality* $\blacktriangleright$ [Appel et al. 2007; Nakano 2000]. A type $\blacktriangleright T$ formed with the later modality represents values of type T generated and accessed "later." As shown by Nakano [2000], "later" can mean "after unfolding" by restricting self-referential type variables of recursive types to occur only beneath $\blacktriangleright$. Under such a restriction, we can enforce that each of the unfolded computations constituting an infinite computation has a type of the form $\blacktriangleright T$.

Because the present work is interested in the behavior of computations rather than their results, we introduce a later modality for temporal effects. The modality allows us to distinguish between the events performed by the current computation and those performed by "later" (i.e., unfolded) computations. Exploiting this ability, we provide a new temporal effect system that represents the infinite trace of a divergent program by effect $\phi_1 \circ \blacktriangleright(\phi_2 \circ \blacktriangleright(\phi_3 \circ \cdots))$, which means: the program first performs events represented by effect ϕ_1 and then executes an unfolded computation with effect $\phi_2 \circ \blacktriangleright(\phi_3 \circ \cdots)$, that is, it performs events in effect ϕ_2 and then executes an unfolded computation with effect $\phi_3 \circ \cdots$, etc.

Besides the support for recursive types, our effect system is abstracted over the representations of temporal effects. While the effect systems in the prior work [Hofmann and Chen 2014; Koskinen and Terauchi 2014; Nanjo et al. 2018] rely on temporal effects in some concrete forms, such as logical formulas and automata-based lattices, we parameterize our effect system over such concrete representations of temporal effects. Instead of relying on concrete forms, we define an algebraic structure required for temporal effects to instantiate the effect system. We call temporal effects with the proposed algebraic structure *algebraic temporal effects*, and build a temporal effect system and its metatheory on top of them. This approach with the algebraic formulation clarifies the assumptions and structures of temporal effects that can soundly reason about temporal properties.

As instances of algebraic temporal effects, we introduce *temporal regular effects*, which can specify finite traces in a regular language and infinite traces in an ω-regular language, and *temporal fixpoint effects*, which can specify finite and infinite traces using predicates in a fixpoint logic. Thanks to the expressive power of the fixpoint logic, temporal fixpoint effects can give non-regular specifications for traces, but, for the return of the expressivity, automated verification with temporal fixpoint effects would be more challenging. Our abstraction to algebraic temporal effects enables guaranteeing soundness of both of the instances, allowing the user to choose an appropriate form depending on their situation. That said, implementation matters like automated verification are beyond the scope of the present work, and we do not discuss further details on the comparison between the instances from the implementation perspective in this paper.

The contributions of this work are summarized as follows:

- We define algebraic temporal effects and build an effect system on top of them. The effect system is equipped with recursive types, later types (of the form $\blacktriangleright T$), and effect polymorphism. We demonstrate that our effect system can reason about temporal properties of higher-order programs with recursive functions, objects, and concurrency.
- We prove two kinds of soundness of the effect system. The first property is *safety soundness*, which states that the effect system correctly predicts the events performed by terminating computation. The second property, *liveness soundness*, ensures that an effect assigned to a divergent program accommodates the infinite trace generated by the program.
- We define temporal regular effects and temporal fixpoint effects as instances of algebraic temporal effects. The correctness of the verification with these instances is proven as corollaries of the soundness of the effect system.

The rest of the paper is organized as follows. Section 2 provides an overview of the present work. In Section 3, we formalize algebraic temporal effects, define the effect system on top of them, illustrate typing examples, and show the soundness properties of the effect system. Section 4 introduces the instances of algebraic temporal effects and presents the typechecking examples with them. In Section 5, we discuss the limitations and open problems in the current form of our work and certain directions of future research. Finally, after discussing related work in Section 6, we conclude the paper in Section 7. This paper only states key properties of the metatheory; the auxiliary lemmas and detailed proofs, as well as the complete definition of the effect system, can be found in the supplementary material.

Throughout the paper, we use the following notation. Let Σ be a set. We write Σ^* for the set of finite sequences, denoted by w, over Σ, and Σ^ω for the set of infinite sequences, denoted by ϖ, over Σ. We also write $\mathbb{N}$ for the set of natural numbers, and ε for the empty sequence. Given a (finite or infinite) sequence s and $i \in \mathbb{N}$, $s(i)$ stands for the $(i+1)$-th element of s (indices start at zero). For a set S, $\mathcal{P}_{\mathrm{fin}\setminus\emptyset}(S)$ is the set of all the nonempty finite subsets of S.

2 Overview

This section provides an overview of temporal effects, the challenge in generalizing them to recursive types, and our approach to addressing it.

2.1 Temporal Effects

Temporal effects specify both finite and infinite traces generated by expressions, taking the form of pairs (ϕ, ι) of a predicate ϕ on finite traces and a predicate ι on infinite traces. We call ϕ and ι *finite* and *infinite effects*, respectively. Both finite and infinite effects are equipped with subeffecting $\phi_1 \sqsubseteq \phi_2$ and $\iota_1 \sqsubseteq \iota_2$, which mean that all the traces in ϕ_1 and ι_1 are contained in ϕ_2 and ι_2, respectively. Temporal effect systems assign a computation type $T \;\&\; (\phi, \iota)$ composed of value type T and temporal effect (ϕ, ι), to an expression that, if terminating, produces a value of type T and a finite trace in predicate ϕ and, otherwise, generates an infinite trace in predicate ι (thus, (ϕ, ι) overapproximates the actual trace of the expression). The compositionality of temporal effect systems rests on composition operation $\triangleright$ on temporal effects, which combines the effects of expressions executed sequentially.

For example, consider the following program in ML-like syntax:

```
let rec repeat () =
  let file = rand_str () in
  if exists file then (
    open file; print_str (read ()); close ();
    repeat ()
  ) else ()
let _ = repeat ()
```

This program uses the following operations: `rand_str` to make a string at random; `exists` to check if the specified file exists; `open`, `read`, and `close` to manipulate files globally (i.e., they share the same file resource); and `print_str` to print a given string. The program calls function `repeat`, which repeats printing the contents of a file chosen at random (if the file does not exist, the function call terminates immediately).

Assume that we are interested in whether the program uses the file manipulation operations in a valid manner. This can be confirmed formally by checking that the series of the calls to `open`, `read`, and `close` conforms to regular expression $(\mathsf{open} \cdot (\mathsf{read}^*) \cdot \mathsf{close})^*$ when the program terminates and, otherwise, ω-regular expression $(\mathsf{open} \cdot (\mathsf{read}^*) \cdot \mathsf{close})^\omega$.

The use of temporal effects enables this reasoning in a formal manner. Now, we focus only on the file manipulation operations. We can find that a call to `repeat` terminates immediately (if the chosen file does not exist) or calls itself recursively after operating `open`, `read`, and `close`. Thus, the temporal effect (ϕ, ι) of `repeat` should accommodate both $(\varepsilon, \emptyset)$ and $(\mathsf{open} \cdot \mathsf{read} \cdot \mathsf{close}, \emptyset) \triangleright (\phi, \iota)$, where the empty trace set $\emptyset$ as an infinite effect means that the computation terminates. Temporal effect systems in the literature infer that a finite effect ϕ and an infinite effect ι meeting this requirement are $(\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{close})^*$ and $(\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{close})^\omega$, respectively. Then, the temporal effect systems assign function type $\mathsf{unit} \rightarrow \mathsf{unit}\ \&\ (\phi, \iota)$ to `repeat`.

It is noteworthy that this reasoning significantly rests on the structure of recursive functions, which expose where recursion occurs through variables representing the recursive functions themselves. For example, in the above program, the reference to variable `repeat` in the body of the function elucidates that the generated trace is open, `read`, and `close` followed by the one generated by the recursive call. The existing temporal effect systems utilize this structure to generate constraints on temporal effects (such as the above one on ϕ and ι).

2.2 Challenge with Recursive Types

Recursive types, which take the form $\mathrm{rec}\,\alpha.T$ in this paper, are powerful type representations with the ability to encode recursive functions, recursive data structures, concurrency, and objects. They—more precisely, *iso*recursive types—are equipped with two operations on expressions: `unfold` and `fold`. Operation `unfold` unfolds the recursive type of the argument, that is, `unfold M` is given type $T[\mathrm{rec}\,\alpha.T/\alpha]$ if expression `M` is of type $\mathrm{rec}\,\alpha.T$. Operation `fold` is the counterpart of `unfold`: `fold M` is given type $\mathrm{rec}\,\alpha.T$ if expression `M` is of type $T[\mathrm{rec}\,\alpha.T/\alpha]$.

These operations allow writing a variety of programs including divergent ones. For example, consider the following function:

```
let f x = print_str (read ()); (unfold x) x
```
(1)

It is easy to check that function `f` is of type $(\mathrm{rec}\,\alpha.\alpha \rightarrow \mathsf{unit}) \rightarrow \mathsf{unit}$ and, therefore, application `f (fold f)` is well typed. The application calls `read` and `print_str` in the order, and then runs `(unfold (fold f)) (fold f)`, which evaluates to `f (fold f)`. Hence, `f (fold f)` only performs `read` infinitely and will never close the file resource.

However, a naive adaption of temporal effects to recursively typed programs fails to deduce that `f` does not call `close`. To see the detail, assume that the function `f` is given the following type:

$$(\mathrm{rec}\,\alpha.\alpha \rightarrow \mathsf{unit}\ \&\ (\phi, \iota)) \rightarrow \mathsf{unit}\ \&\ (\phi, \iota)\ .$$

This type uses the temporal effect (ϕ, ι) to represent both traces yielded by applying `unfold x` and `f`, respectively, so that `fold f` is well typed (in fact, it is given the type $\mathrm{rec}\,\alpha.\alpha \rightarrow \mathsf{unit}\ \&\ (\phi, \iota)$ with the typing discipline for `fold` explained above). Because `f` only calls `read` and then `unfold x`, the only constraint on the temporal effect (ϕ, ι) is that the effect $(\mathtt{read}, \emptyset) \triangleright (\phi, \iota)$ of `f`'s body is a subeffect of the effect (ϕ, ι) specified in the return type of `f`—in other words, any trace denoted by $(\mathtt{read}, \emptyset) \triangleright (\phi, \iota)$ must be contained in (ϕ, ι). For a solution of the constraint, consider the effect $(\emptyset, (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega)$, which means that an expression given this effect does not terminate, performs `close` after one or more calls to `read`, and performs open before the next `read`. This temporal effect satisfies the constraint because all the infinite traces specified by $\mathtt{read} \cdot (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$, which comes from $(\mathtt{read}, \emptyset) \triangleright (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$, conform to $(\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$. As a result, the temporal effect system could erroneously conclude that the file, opened before calling `f`, will eventually be closed, although this is not the case.

A critical reason for this problem is that the naive support for recursive types considered above lacks the *productivity guarantee* [Sijtsma 1989] for infinite traces. This means that it does not ensure that a temporal effect assigned to a program contains an infinite trace where every finite prefix is eventually generated by the program. For the above example, any infinite trace in the predicate $(\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$ assigned to function `f` takes the form $\mathtt{read} \cdot \cdots \cdot \mathtt{read} \cdot \mathtt{close} \cdot \mathtt{open} \cdot \cdots$, but its prefix $\mathtt{read} \cdot \cdots \cdot \mathtt{read} \cdot \mathtt{close} \cdot \mathtt{open}$ is not generated by `f`.

The prior work [Hofmann and Chen 2014; Nanjo et al. 2018] on temporal effects guarantees the productivity by relying on the structure of recursive functions. For example, Hofmann and Chen's effect system involves a typing rule tailored to recursive functions, which assigns to a recursive function let rec $f\ x = M$ a function type with the latent temporal effect $(\phi, (r^* \cdot o) \cup r^\omega)$ if the body M of f has an effect $(\phi, (r \cdot X) \cup o)$ under the assumption that the latent effect of f is (ϕ, X), where r and o are (abstractions of) regular and ω-regular expressions, respectively, and X is a variable over infinite temporal effects. The assigned infinite effect $(r^* \cdot o) \cup r^\omega$ means that, if an application of f diverges, it is caused by a finite number of recursive calls to f followed by some other infinite computation (e.g., inner loops), or an infinite number of recursive calls to f. The typechecking condition on the body M ensures that some finite trace in r happens before each recursive call to f (if r is nonempty). Therefore, it is ensured that any finite prefix of an infinite trace in $(r^* \cdot o) \cup r^\omega$ happens actually.

This approach in the prior work rests on the fact that the divergence caused by a recursive function let rec $f\ x = M$ is ascribed to the use of the self-referential variable f in the body M. It is difficult to take a similar approach for recursively typed, higher-order programs because the root cause of the divergence of such a program cannot be determined syntactically. For instance, consider example (1). In the example, whether a call to `f` diverges depends on actual arguments for `x`. While expression `f (fold f)` diverges due to the recursive call to `f`, expression `f (fold (fun x -> ()))` terminates. It indicates that, in the presence of recursive types and higher-order functions, we cannot identify (potential) loops in a lexical manner, unlike programs only with recursive functions. Thus, the flexibility of recursive types hinders ensuring the productivity of temporal effect systems.

2.3 Solution: Temporal Effects with Later Modality

To achieve a temporal effect system with the productivity in the presence of recursive types, we employ the *later modality* [Nakano 2000]. One actively studied application of the later modality is to ensure the productivity of coinductively defined infinite data structures [Atkey and McBride 2013; Birkedal and Møgelberg 2013; Clouston et al. 2015; Jaber and Riba 2021].[1] Inspired by the technique for the productivity of infinite data structures, we adapt it to reason about infinite executions complicated by recursive types.

The later modality $\blacktriangleright$ is a type constructor to guard the self references of recursive types. For example, recursive type rec $\alpha.\alpha \rightarrow$ unit is represented as rec $\alpha.\blacktriangleright\alpha \rightarrow$ unit in the form of guarded recursive types. Intuitively, a type $\blacktriangleright T$, which we call a *later type*, represents the values of unfolded expressions of type T. For example, if variable `x` is of type rec $\alpha.$unit $\rightarrow \blacktriangleright\alpha$, then expression `(unfold x) ()` has type $\blacktriangleright$(rec $\alpha.$unit $\rightarrow \blacktriangleright\alpha$), which means that the result originates from the computation arising by unfolding.[2] This indicates that the later modality can be seen as a type-based mechanism to reason about the usage of the unfolding operation.

To exploit the later modality for temporal liveness verification, we equip infinite effects with a later modality, and extend the modality to computation types: $\blacktriangleright(T \ \&\ (\phi, \iota)) \overset{\text{def}}{=} \blacktriangleright T \ \&\ (\phi, \blacktriangleright\iota)$. The role of the later modality over infinite effects is twofold.

First, it is used to fold the infinite behavior of a divergent computation, which enables us to stratify the infinite trace of the computation. To see it in more detail, consider a divergent expression M. Since the divergence is caused by infinitely many unfolded computations as discussed in Section 1.2, the evaluation of M reaches, after yielding some finite trace w_1, the state that places a folded

[1] The productivity of infinite data structures means that, informally speaking, any element of the infinite data structures can be computed in finite time.
[2] A similar discussion can be applied when the type variables of recursive types occur at negative positions, but the situation becomes more complex.

computation M_1 at a redex position and then unfolds M_1, which also diverges. Using temporal effects with the later modality, we can express this behavior as follows: provided that an infinite effect ι_1 is assigned to the unfolded M_1, the infinite effect ι of M is a supereffect of the composition $w_1 \circ \blacktriangleright\iota_1$, that is, $\iota \sqsupseteq w_1 \circ \blacktriangleright\iota_1$. For instance, in example (1), when `fold f` is passed as an argument to `f`, it is unfolded to `f` after `read`, and then `f` is called again. Thus, $\iota \sqsupseteq \texttt{read} \circ \blacktriangleright\iota$ should hold for some ι being the infinite effect of `f`. Returning to the execution of M, since the unfolded M_1 diverges, we can find that its evaluation also yields some finite trace w_2 and then produces a folded computation M_2 that will cause divergence after unfolding. This behavior is represented by $\iota_1 \sqsupseteq w_2 \circ \blacktriangleright\iota_2$, where ι_2 is the infinite effect of M_2, and allows us to derive $w_1 \circ \blacktriangleright\iota_1 \sqsupseteq w_1 \circ \blacktriangleright(w_2 \circ \blacktriangleright\iota_2)$.[3] By repeating this process, we can find that

$$\iota \sqsupseteq w_1 \circ \blacktriangleright\iota_1 \sqsupseteq w_1 \circ \blacktriangleright(w_2 \circ \blacktriangleright\iota_2) \sqsupseteq w_1 \circ \blacktriangleright(w_2 \circ \blacktriangleright(w_3 \circ \blacktriangleright\iota_3)) \sqsupseteq \cdots \sqsupseteq w_1 \circ \blacktriangleright(w_2 \circ \blacktriangleright(w_3 \circ \blacktriangleright(w_4 \circ \cdots)))$$

with some finite traces $w_2, w_3, w_4, \cdots$, each of which is generated between successive unfoldings leading to divergence. This observation indicates that the later operation $\blacktriangleright$ produces an infinite effect ($\blacktriangleright\iota_1$) by folding the finite traces ($w_2, w_3, w_4, \cdots$) between the successive unfoldings.

The second role of $\blacktriangleright$ over infinite effects is to ensure the productivity. This is achieved by requiring that $\blacktriangleright\iota_1$ contain the infinite trace $w_2 \cdot w_3 \cdot \cdots$ when $\blacktriangleright\iota_1 \sqsupseteq \blacktriangleright(w_2 \circ \blacktriangleright(w_3 \circ \cdots))$, because then it is easy to ensure that the infinite effect ι meeting $\iota \sqsupseteq w_1 \circ \blacktriangleright\iota_1$ accommodates the generated infinite trace $w_1 \cdot w_2 \cdot w_3 \cdot \cdots$. To accomplish this requirement, we assume that (1) the infinite effect $\blacktriangleright\iota_1$ supplies a set of finite effects $\{\phi_1, \cdots, \phi_n\}$ and that (2) when $\blacktriangleright\iota_1 \sqsupseteq \blacktriangleright(w_2 \circ \blacktriangleright(w_3 \circ \cdots))$ holds, there exists a finite effect ϕ_i in the supplied set that is an upper bound of all the finite traces $w_2, w_3, \cdots$ yielded between the successive unfoldings. Under these assumptions, the productivity is guaranteed (that is, $\blacktriangleright\iota_1$ accommodates $w_2 \cdot w_3 \cdot \cdots$) by interpreting the set of infinite traces captured by $\blacktriangleright\iota_1$ to be $\bigcup_{i \in [1,n]} \phi_i^\omega$, where ϕ_i^ω is the set of infinite traces constituting some infinitely many finite traces in the finite effect ϕ_i. We formulate assumption (1) by equipping infinite effects with a *finitization* operation *fin*, which maps infinite effects to finite effect sets $\{\phi_1, \cdots, \phi_n\}$. For assumption (2), we introduce inequational axioms with the operations $\blacktriangleright$ and *fin* (see inequations (9) and (10) in Definition 1).

The above interpretation of infinite effects means that our framework reduces the infinite behavior of a program (described by $\blacktriangleright\iota_1$) to the infinite repetition of the upper-bounded finite behavior between successive unfoldings (described by some $\phi_i \in \mathit{fin}(\blacktriangleright\iota_1)$). The later operation $\blacktriangleright$ plays a role in finding the behavior between successive unfoldings that is overapproximated by ϕ_i.

This view guides how we can implement $\blacktriangleright$ and *fin* for concrete instances of temporal effects. For instance, consider implementing infinite effects by ω-regular expressions. An ω-regular expression r^ω represents the infinite repetition of the finite behavior r. Thus, we can interpret that r in r^ω describes the finite behavior between successive unfoldings. Given an ω-regular expression $r_1 \cdot r_2^\omega$, r_1 expresses the behavior before the first unfolding. If a computation with the effect $r_1 \cdot r_2^\omega$ is folded (thus, it will be unfolded to be executed), the effect $r_1 \cdot r_2^\omega$ is also "folded" to $(r_1 \cup r_2)^\omega$ because the finite behavior between successive unfoldings in the folded computation is captured by r_1 or r_2. Based on these observations, we can implement the operations $\blacktriangleright$ and *fin* for ω-regular expressions as $\blacktriangleright(r_1 \cdot r_2^\omega) = (r_1 \cup r_2)^\omega$ and $\mathit{fin}(r^\omega) = \{r\}$. Because *fin* only applies to infinite effects of the form $\blacktriangleright\iota$, it is enough to consider the case that *fin* takes ω-regular expressions of the form r^ω. Temporal regular effects given in Section 4.1 generalize this idea to more general ω-regular expressions, such as the union of ω-regular expressions, where *fin* may return a set with multiple regular expressions, but the core idea behind it is the same as described here.

[3]We need the monotonicity of $\blacktriangleright$ with respect to $\sqsubseteq$ to derive it formally.

$$
\begin{array}{l}
\textbf{Variables}\ \ x, y, z, f \qquad \textbf{Type variables}\ \ \alpha \qquad \textbf{Effect variables}\ \ X \\
\begin{array}{rcll}
\textbf{Constants} & c & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid \cdots \\
\textbf{Values} & V & ::= & x \mid c \mid \lambda x.M \mid \Lambda X.M \mid \mathsf{fold}\ V \mid \mathsf{next}\ V \\
\textbf{Expressions} & M & ::= & V \mid o(\overline{V}) \mid V_1\ V_2 \mid V\ e \mid \mathsf{unfold}\ V \mid \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \mid \\
 & & & \mathsf{if}\ V\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \mid \mathsf{next}\ M \mid V_1 \circledast V_2 \mid \mathsf{prev}\ V \\
\textbf{Base types} & B & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \cdots \\
\textbf{Value types} & T & ::= & B \mid \alpha \mid T \to C \mid \forall X.C \mid \mathsf{rec}\,\alpha.T \mid \blacktriangleright T \\
\textbf{Computation types} & C & ::= & T\ \&\ e \\
\textbf{Syntactic effects} & e & ::= & \epsilon \mid X \mid e_1 \mathbin{\underline{\vee}} e_2 \mid e_1 \mathbin{\underline{\triangleright}} e_2 \mid \underline{\blacktriangleright} e \mid \mathbf{e} \\
\textbf{First-order types} & \tau & ::= & B \mid \blacktriangleright \tau \\
\textbf{Typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X \mid \Gamma, \alpha
\end{array}
\end{array}
$$

Fig. 1. Syntax.

Revisiting The Example. Finally, we show that how the extension introduced in this section prevents us to conclude that example (1) generates an infinite trace conforming to the specification $(\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$. As mentioned before, to typecheck example (1), the subeffecting $\mathtt{read} \circ \blacktriangleright\iota \sqsubseteq \iota$ needs to hold where ι is the latent infinite effect of the function `f`. Assume that finite effects and infinite effects are respectively represented as regular expressions and ω-regular expressions of the form $r_1 \cdot r_2^\omega$. Let the infinite effect ι in the example be $\mathtt{read}^* \cdot (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega$. Here, $\mathtt{read}^*$ represents finite traces generated between the call and recursive call to `f`. Because

$$
\begin{array}{rcl}
\mathtt{read} \circ \blacktriangleright\iota & = & \mathtt{read} \circ \blacktriangleright(\mathtt{read}^* \cdot (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open})^\omega) \\
 & = & \mathtt{read} \cdot (\mathtt{read}^* \cup (\mathtt{read}^+ \cdot \mathtt{close} \cdot \mathtt{open}))^\omega ,
\end{array}
$$

$\mathtt{read}^\omega$, which is the actually generated trace, is in $\mathtt{read} \circ \blacktriangleright\iota$ but not in ι. Thus, $\mathtt{read} \circ \blacktriangleright\iota \not\sqsubseteq \iota$, so we find the example program ill-typed as desired.[4] If $\iota = \mathtt{read}^* \cdot \mathtt{read}^\omega$, the required subeffecting

$$
\mathtt{read} \circ \blacktriangleright\iota = \mathtt{read} \cdot (\mathtt{read}^* \cup \mathtt{read})^\omega \sqsubseteq \mathtt{read}^* \cdot \mathtt{read}^\omega = \iota
$$

holds. Therefore, we can conclude that the infinite trace yielded by the example conforms to $\mathtt{read}^* \cdot \mathtt{read}^\omega$.

The crux to prevent the unsound reasoning about the example is the use of the later modality $\blacktriangleright$. The later modality folds the finite effect $\mathtt{read}^*$ specifying finite traces generated between successive (recursive) calls to `f` into the finite effect to be repeated infinitely. This folding effect of the later operation is guided by the requirement on *fin*, enforcing the infinite effect ι to involve the possibility that the call to `f` may diverge at the infinite trace $\mathtt{read}^\omega$. Thus, if an infinite effect that does not accommodate $\mathtt{read}^\omega$ is given as ι, the typechecking of example (1) results in failure. Algebraic temporal effects, introduced in Section 3.2, are an abstract formulation capturing this idea.

3 Algebraic Temporal Effect System

This section starts by defining a higher-order language with effectful operations, recursive types, later types, and effect polymorphism. We then introduce algebraic temporal effects and an effect system, illustrate typing examples, and show the properties of our system.

3.1 Language

Our language is based on those in the prior work on guarded recursion, an application of the later modality [Atkey and McBride 2013; Birkedal and Møgelberg 2013; Clouston et al. 2015], while our use of the later modality aims to directly reason about applications of the unfolding as in the work of Dreyer et al. [2011]. Inspired by the works on guarded recursion, we introduce the next constructor for later types. As an eliminator of later types, we introduce a term construct prev, which allows eliminating the later type constructor $\blacktriangleright$ from first-order types of the form $\blacktriangleright \cdots \blacktriangleright B$ with some base type B. We restrict the application of prev to first-order values because the unrestricted elimination of $\blacktriangleright$ from higher-order types can cause bypassing the enforcement of the effect folding via $\blacktriangleright$, which leads to unsoundness. The notion of prev is not original in this work. For example, Clouston et al. [2015] studied a calculus supporting prev that can eliminate $\blacktriangleright$ even from higher-order types under a certain condition concerning typing. We could relax the restriction on our prev as in Clouston et al.'s calculus, but it makes our language more complex. We strike a balance between simplicity and usefulness of prev by allowing prev to be applied only to first-order values.

Note that the present work aims to study a theoretical aspect of temporal verification for recursively typed programs. Thus, we allow our language to be equipped with some constructs—next, prev, $\circledast$, fold, and unfold—that work as annotations. We could design a surface language that hides the use of these annotations from programmers by providing only structured programming features, like recursive functions, algebraic data types/pattern matching, and objects, that can be implemented in our core language, but it is a matter of practice and beyond the scope of this work.

3.1.1 Syntax. The syntax of our language is shown in Figure 1. We assume that constants include the Boolean value true and false. Values, ranged over by V, consist of: variables; constants; lambda abstractions $\lambda x.M$; effect abstractions $\Lambda X.M$; foldings fold V; and next-values next V, which indicate that argument values V are the results of "later" computations. Expressions, ranged over by M, are: values; applications of possibly effectful operations $o(\overline{V})$, where o is a primitive operation and $\overline{V}$ is a finite sequence of values; function applications $V_1\ V_2$; effect applications $V\ e$; unfoldings unfold V; let-expressions let $x = M_1$ in M_2; if-expressions if V then M_1 else M_2; next-expressions next M, which mean that expressions M are "later" computations; later applications $V_1 \circledast V_2$, which allow function applications at the "later" time point (i.e., allow applying the function V_1 of a type $\blacktriangleright(T \rightarrow C)$ to the argument V_1 of a type $\blacktriangleright T$); or destructors for later types prev V. The term constructors next and $\circledast$ come from the previous work on the later modality, forming an applicative functor denotationally [Atkey and McBride 2013].

Value types T specify values, and computation types C, which are pairs of a value type and an effect, specify the behavior of computations. In addition to the type constructors explained in Section 2, the language supports effect-polymorphic types $\forall X.C$. As in the prior work [Birkedal and Møgelberg 2013; Clouston et al. 2015; Nakano 2000], we restrict types to meet a *guard condition.* A type variable α is *guarded* in a type if every occurrence of α in the type is beneath an occurrence of $\blacktriangleright$. We assume that, *for every recursive type* rec $\alpha.T'$ *occurring in any type, type variable* α *is guarded in type* T'. First-order types are base types with zero or more applications of the later modality, as explained above.

Syntactic effects (or effects for short) e are a syntactic representation of temporal effects. We use algebraic temporal effects as their denotations in subeffecting. The constructor ϵ represents that computation is pure, that is, no effectful operation is invoked. Join $\underline{\vee}$, sequential composition

[4]If we take $\iota = (\texttt{read}^+ \cdot \texttt{close} \cdot \texttt{open})^\omega$, our temporal effect system enforces $\blacktriangleright\iota$ to involve the possibility that the internal, unobservable computation continues infinitely, but then $\texttt{read} \circ \blacktriangleright\iota \sqsubseteq \iota$ does not hold because ι does not include such infinite internal computation.

Reduction rules $\boxed{M_1 \rightsquigarrow M_2}$

$$\begin{array}{rcll}
(\lambda x.M)\ V & \rightsquigarrow & M[V/x] & \text{R\_Beta} \\
(\mathsf{next}\ V_1) \circledast (\mathsf{next}\ V_2) & \rightsquigarrow & \mathsf{next}\ (V_1\ V_2) & \text{R\_NBeta} \\
(\Lambda X.M)\ e & \rightsquigarrow & M[e/X] & \text{R\_EBeta} \\
\mathsf{let}\ x = V_1\ \mathsf{in}\ M_2 & \rightsquigarrow & M_2[V_1/x] & \text{R\_Let} \\
\mathsf{unfold}\ (\mathsf{fold}\ V) & \rightsquigarrow & V & \text{R\_Unfold} \\
\mathsf{prev}\ (\mathsf{next}\ V) & \rightsquigarrow & V & \text{R\_Back} \\
\mathsf{if\ true\ then}\ M_1\ \mathsf{else}\ M_2 & \rightsquigarrow & M_1 & \text{R\_IfT} \\
\mathsf{if\ false\ then}\ M_1\ \mathsf{else}\ M_2 & \rightsquigarrow & M_2 & \text{R\_IfF}
\end{array}$$

Evaluation rules $\boxed{M_1 \longrightarrow M_2\ \&\ \phi}$

$$\frac{M_1 \rightsquigarrow M_2}{M_1 \longrightarrow M_2\ \&\ \mathbf{1}}\ \text{E\_Red} \qquad \frac{}{o(\overline{c}) \longrightarrow \delta_{\mathrm{v}}(o,\overline{c})\ \&\ \delta_{\mathrm{e}}(o)}\ \text{E\_Op}$$

$$\frac{M_1 \longrightarrow M_1'\ \&\ \phi}{\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ M_2 \longrightarrow \mathsf{let}\ x_1 = M_1'\ \mathsf{in}\ M_2\ \&\ \phi}\ \text{E\_Let} \qquad \frac{M \longrightarrow M'\ \&\ \phi}{\mathsf{next}\ M \longrightarrow \mathsf{next}\ M'\ \&\ \phi}\ \text{E\_Next}$$

Multi-step evaluation

$$M_1 \longrightarrow^n M_2\ \&\ \phi \overset{\text{def}}{=} \exists M_0', \cdots, M_n', \phi_1', \cdots, \phi_n'.$$
$$M_1 = M_0' \wedge M_2 = M_n' \wedge \phi = \mathbf{1} \cdot \phi_1' \cdot \cdots \cdot \phi_n' \wedge \forall\, i < n.\ M_i' \longrightarrow M_{i+1}'\ \&\ \phi_{i+1}'.$$

Divergence

$$M \Uparrow \varpi \overset{\text{def}}{=} \forall\, i, M', \phi.\ M \longrightarrow^i M'\ \&\ \phi \Longrightarrow \exists\, M''.\ M' \longrightarrow M''\ \&\ \varpi(i)$$

Fig. 2. Semantics.

$\underline{\triangleright}$, and effect-level later modality $\underline{\blacktriangleright}$ are supported syntactically. The metavariable $\mathbf{e}$ represents effect constructors specific to concrete representations of temporal effects. Because we aim at a general framework for temporal effects, we do not assume constructors specialized in specific effect representations, such as operations on regular expressions. Instead, we parameterize our language over effect constructors $\mathbf{e}$ and their denotations (i.e., how to interpret them as algebraic temporal effects). We give operations for (ω-)regular expressions as an instance in Section 4.1. When parsing syntactic effects, we assume $\underline{\blacktriangleright}$ has higher precedence than $\underline{\vee}$ and $\underline{\triangleright}$. For example, we may write $\underline{\blacktriangleright} e_1\ \underline{\triangleright}\ e_2$ to express $(\underline{\blacktriangleright} e_1)\ \underline{\triangleright}\ e_2$.

We use the following notation. For an expression M, $fv(M)$ is the set of all the variables occurring free in M. We also write $M[V/x]$ to denote the expression obtained by substituting value V for variable x in M in a capture-avoiding manner. We use similar notation for other kinds of substitution. The later modality is extended to computation types as: $\blacktriangleright(T\ \&\ e) \overset{\text{def}}{=} (\blacktriangleright T)\ \&\ (\underline{\blacktriangleright} e)$.

3.1.2 Semantics. The behavior of programs rests on effects performed by primitive operations. As the present work aims at a general theory of temporal effects, we define the effects of operations abstractly. There are two requirements for those effects. First, they should be sequentially composable because temporal verification is interested in the order in which effects occur. Second, they should include an effect that represents pure computation such as β-reduction. By these requirements, we assume that the effects of operations form a monoid $(\mathbb{F}, \cdot, \mathbf{1})$, where operation $(\cdot)$ enables the composition of multiple effects and the unit element $\mathbf{1}$ denotes the effect of pure computation. We use the metavariable ϕ to denote elements in $\mathbb{F}$.

The semantics consists of reduction relation $M_1 \rightsquigarrow M_2$, which represents the pure computation from M_1 to M_2, and evaluation relation $M_1 \longrightarrow M_2 \ \& \ \phi$, which represents the computation from M_1 to M_2 with effect ϕ. They are the smallest relations satisfying the rules in Figure 2.

The reduction rules are standard except for (R_Back), which represents that prev is an eliminator for next. The rule (R_NBeta) implements the beta-reduction under next.

The evaluation rules allow reduction (R_Red), applying operations (R_Op), and evaluating subexpressions ((E_Let) and (E_Next)). The rule (R_Op) assumes two metafunctions: δ_{v}, which maps tuples of an operation and zero or more constants to constants, and δ_{e}, which assigns an element of $\mathbb{F}$ to every operation.

Figure 2 also defines multi-step evaluation and divergence of programs. Multi-step evaluation $M_1 \longrightarrow^n M_2 \ \& \ \phi$ means that expression M_1 evaluates to M_2 with effect ϕ by n steps. We write $M_1 \longrightarrow^* M_2 \ \& \ \phi$ if $M_1 \longrightarrow^n M_2 \ \& \ \phi$ for some n. Assume that the metavariable ϖ denotes elements in $\mathbb{F}^{\omega}$. Then, program divergence $M \Uparrow \varpi$ defined in Figure 2 means that expression M can evaluate forever and, for any $i \in \mathbb{N}$, $\varpi(i)$ is the effect of the evaluation at the $(i+1)$-th step.

3.2 Algebraic Temporal Effects

This section defines an algebraic formulation of temporal effects. As in the prior work on temporal effects [Hofmann and Chen 2014; Koskinen and Terauchi 2014; Nanjo et al. 2018; Sekiyama and Unno 2023], algebraic temporal effects are defined to be pairs of finite effects, which represent the traces of terminating programs, and infinite effects, which represent those of divergent programs.

Definition 1 (Algebraic Temporal Effects). Algebraic temporal effects, *ranged over by* ζ, *are elements in set* $\mathbb{E}$, *which is the Cartesian product* $\mathbb{F} \times \mathbb{I}$ *of sets* $\mathbb{F}$ *and* $\mathbb{I}$ *such that:*

- *finite effect set* $\mathbb{F}$ *forms a monoid* $(\mathbb{F}, \cdot, \mathbf{1})$ *and a join semilattice* $(\mathbb{F}, +)$*;*
- *infinite effect set* $\mathbb{I}$ *forms a join semilattice* $(\mathbb{I}, \sqcup)$ *with the least element* $\mathbf{0}$*;*
- *there exist a later operation* $\blacktriangleright : \mathbb{I} \longrightarrow \mathbb{I}$*, a mixed-product operation* $\circ : \mathbb{F} \times \mathbb{I} \longrightarrow \mathbb{I}$*, and a finitization operation* $fin : \mathbb{I} \longrightarrow \mathcal{P}_{\mathrm{fin} \setminus \emptyset}(\mathbb{F})$*.*

Metavariable ι *ranges over the elements of* $\mathbb{I}$*. Partial orders* $\sqsubseteq$ *on* $\mathbb{F}$ *and on* $\mathbb{I}$ *are induced by the join operations:* $\phi_1 \sqsubseteq \phi_2 \stackrel{\mathrm{def}}{=} \phi_1 + \phi_2 = \phi_2$ *and* $\iota_1 \sqsubseteq \iota_2 \stackrel{\mathrm{def}}{=} \iota_1 \sqcup \iota_2 = \iota_2$*.*

We also define the three notions to state the assumptions on fin. The first is the join of a finite effect ϕ *and a set of finite effects* $S \subseteq \mathbb{F}$*:*

$$\phi + S \stackrel{\mathrm{def}}{=} \{\phi + \phi' \mid \phi' \in S\} .$$

The second is a binary relation $\precsim$ *on* $\mathcal{P}_{\mathrm{fin} \setminus \emptyset}(\mathbb{F})$*:*

$$S_1 \precsim S_2 \stackrel{\mathrm{def}}{=} \forall \phi_1 \in S_1 . \, \exists \phi_2 \in S_2 . \, \phi_1 \sqsubseteq \phi_2 .$$

Finally, an infinite effect ι *is called* infinitely expandable *when, for any* $n > 0$*, there exist some* $\phi_1, \cdots, \phi_n$*, and* ι' *such that* $\phi_1 \circ \blacktriangleright(\phi_2 \circ \blacktriangleright(\cdots (\phi_n \circ \blacktriangleright \iota') \cdots)) \sqsubseteq \iota$*. It defines a class of infinite effects that can be given to divergent expressions because the effect system presented shortly ensures that any infinite effect assigned to a well-typed divergent expression is infinitely expandable.*

Then, algebraic temporal effects are required to meet the following:

$$\begin{array}{lrcl@{\qquad}lrcl}
(1) & \phi \circ \mathbf{0} & = & \mathbf{0} & (2) & (\phi \cdot \phi_1) + (\phi \cdot \phi_2) & \sqsubseteq & \phi \cdot (\phi_1 + \phi_2) \\
(3) & \mathbf{1} \circ \iota & = & \iota & (4) & (\phi_1 \cdot \phi) + (\phi_2 \cdot \phi) & \sqsubseteq & (\phi_1 + \phi_2) \cdot \phi \\
(5) & \phi_1 \circ (\phi_2 \circ \iota) & = & (\phi_1 \cdot \phi_2) \circ \iota & (6) & (\phi \circ \iota_1) \sqcup (\phi \circ \iota_2) & = & \phi \circ (\iota_1 \sqcup \iota_2) \\
(7) & \blacktriangleright(\iota_1 \sqcup \iota_2) & = & (\blacktriangleright\iota_1) \sqcup (\blacktriangleright\iota_2) & (8) & (\phi_1 \circ \iota) \sqcup (\phi_2 \circ \iota) & \sqsubseteq & (\phi_1 + \phi_2) \circ \iota \\
(9) & \phi + \mathit{fin}(\blacktriangleright\iota_0) & \precsim & \mathit{fin}(\blacktriangleright(\phi \circ \blacktriangleright\iota_0)) & (10) & \iota_1 \sqsubseteq \iota_2 & \Longrightarrow & \mathit{fin}(\iota_1) \precsim \mathit{fin}(\iota_2)
\end{array}$$

where every metavariable appearing in each (in)equation is supposed to be universally quantified, and ι_0 in inequation (9) is supposed to be infinitely expandable.

Join operations $+$ and $\sqcup$ allow forming one effect from the effects of different control flows. Infinite effect $\mathbf{0}$ represents termination of programs. Mixed-product operation $\circ$ enables us to represent the effect of sequential composition of a terminating, preceding expression with a divergent, succeeding expression. As explained in Section 2.3, later operation $\blacktriangleright$ folds finite effects into infinite effects and finitization operation *fin* supplies a set of finite effects that are upper bounds of finite traces between successive unfoldings that cause divergence.

We define operations on algebraic temporal effects using the operations on $\mathbb{F}$ and those on $\mathbb{I}$.

Definition 2 (Operations on Algebraic Temporal Effects). *Later operation $\blacktriangleright$, join $\vee$, pure effect ϵ, and sequential composition $\triangleright$ on algebraic temporal effects are defined as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
\blacktriangleright(\phi, \iota) & \overset{\text{def}}{=} & (\phi, \blacktriangleright\iota) & (\phi_1, \iota_1) \vee (\phi_2, \iota_2) & \overset{\text{def}}{=} & (\phi_1 + \phi_2, \iota_1 \sqcup \iota_2) \\
\epsilon_{\mathbb{E}} & \overset{\text{def}}{=} & (\mathbf{1}, \mathbf{0}) & (\phi_1, \iota_1) \triangleright (\phi_2, \iota_2) & \overset{\text{def}}{=} & (\phi_1 \cdot \phi_2, \iota_1 \sqcup (\phi_1 \circ \iota_2)) .
\end{array}$$

The later operation on $\mathbb{E}$ is reduced to that on $\mathbb{I}$. The temporal effect $\epsilon_{\mathbb{E}}$ is the pure effect, given to terminating, pure expressions. The definition of sequential composition $\triangleright$ originates from the prior work on temporal effects. It indicates that, for a program that runs expressions M_1 with effect (ϕ_1, ι_1) and M_2 with effect (ϕ_2, ι_2) sequentially, it terminates when both M_1 and M_2 terminate (and then $\phi_1 \cdot \phi_2$ can be assigned as the finite effect of the entire program), and it diverges when M_1 diverges or when M_1 terminates and M_2 diverges (and then the infinite effect is ι_1 or $\phi_1 \circ \iota_2$).

Subeffecting on $\mathbb{E}$ is induced by join operation $\vee$. Subeffecting $\sqsubseteq$ on $\mathbb{E}$ is defined as

$$\zeta_1 \sqsubseteq \zeta_2 \overset{\text{def}}{=} \zeta_1 \vee \zeta_2 = \zeta_2 .$$

The equational theory of algebraic temporal effects allows us to prove certain properties of the effect equality and subeffecting, as shown in Section 3.5. Especially, inequation (9) requires $\mathit{fin}(\blacktriangleright(\phi \circ \blacktriangleright\iota_0))$ to contain an upper bound of the finite effect ϕ happening between successive unfoldings.

Inequations (9) and (10) for *fin* are introduced to ensure assumption (2) mentioned in Section 2.3. Recall that the assumption is that, when $\blacktriangleright\iota_0 \sqsupseteq \blacktriangleright(w_0 \circ \blacktriangleright(w_1 \circ \blacktriangleright(w_2 \circ \cdots)))$, there exists a finite effect $\phi \in \mathit{fin}(\blacktriangleright\iota_0)$ that is an upper bound of the finite traces $w_0, w_1, w_2, \cdots$. To see how the inequations ensure this assumption, consider $\blacktriangleright\iota_0 \sqsupseteq \blacktriangleright(w_0 \circ \blacktriangleright(w_1 \circ \blacktriangleright(w_2 \circ \cdots)))$. For every $i \geq 1$, let $\iota_i = w_i \circ \blacktriangleright(w_{i+1} \circ \cdots \blacktriangleright(w_n \circ \blacktriangleright\iota_0') \cdots)$ and assume that it is infinitely expandable. First, because $\blacktriangleright\iota_0 \sqsupseteq \blacktriangleright(w_0 \circ \blacktriangleright\iota_1)$, we have $\mathit{fin}(\blacktriangleright\iota_0) \succsim \mathit{fin}(\blacktriangleright(w_0 \circ \blacktriangleright\iota_1))$ by inequation (10). That is,

$$\forall \phi_0 \in \mathit{fin}(\blacktriangleright(w_0 \circ \blacktriangleright\iota_1)).\ \exists \phi_0' \in \mathit{fin}(\blacktriangleright\iota_0).\ \phi_0 \sqsubseteq \phi_0' . \tag{2}$$

By instantiating inequation (9) with w_0 and ι_1, we have $w_0 + \mathit{fin}(\blacktriangleright\iota_1) \precsim \mathit{fin}(\blacktriangleright(w_0 \circ \blacktriangleright\iota_1))$, that is,

$$\forall \phi_1 \in \mathit{fin}(\blacktriangleright\iota_1).\ \exists \phi_1' \in \mathit{fin}(\blacktriangleright(w_0 \circ \blacktriangleright\iota_1)).\ w_0 + \phi_1 \sqsubseteq \phi_1' . \tag{3}$$

By inequations (2) and (3),

$$\forall \phi_1 \in \mathit{fin}(\blacktriangleright\iota_1).\ \exists \phi_1' \in \mathit{fin}(\blacktriangleright(w_0 \circ \blacktriangleright\iota_1)).\ \exists \phi_0' \in \mathit{fin}(\blacktriangleright\iota_0).\ w_0 + \phi_1 \sqsubseteq \phi_1' \ \wedge\ \phi_1' \sqsubseteq \phi_0' .$$

Because $w_0 + \phi_1 \sqsubseteq \phi'_1 \ \wedge\ \phi'_1 \sqsubseteq \phi'_0 \Longrightarrow w_0 + \phi_1 \sqsubseteq \phi'_0$, we have

$$\forall\, \phi_1 \in \mathit{fin}\,(\blacktriangleright\iota_1).\ \exists\, \phi'_0 \in \mathit{fin}\,(\blacktriangleright\iota_0).\ w_0 + \phi_1 \sqsubseteq \phi'_0\ . \tag{4}$$

Next, instantiating inequation (9) with w_1 and ι_2 implies $w_1 + \mathit{fin}\,(\blacktriangleright\iota_2) \mathrel{\widetilde{\sqsubseteq}} \mathit{fin}\,(\blacktriangleright(w_1 \circ \blacktriangleright\iota_2))$, that is,

$$\forall\, \phi_2 \in \mathit{fin}\,(\blacktriangleright\iota_2).\ \exists\, \phi'_2 \in \mathit{fin}\,(\blacktriangleright(w_1 \circ \blacktriangleright\iota_2)).\ w_1 + \phi_2 \sqsubseteq \phi'_2\ . \tag{5}$$

Because $\blacktriangleright\iota_1 \ = \ \blacktriangleright(w_1 \circ \blacktriangleright\iota_2)$, inequations (4) and (5) imply

$$\forall\, \phi_2 \in \mathit{fin}\,(\blacktriangleright\iota_2).\ \exists\, \phi'_2 \in \mathit{fin}\,(\blacktriangleright(w_1 \circ \blacktriangleright\iota_2)).\ \exists\, \phi'_0 \in \mathit{fin}\,(\blacktriangleright\iota_0).\ w_1 + \phi_2 \sqsubseteq \phi'_2 \ \wedge\ w_0 + \phi'_2 \sqsubseteq \phi'_0\ .$$

Because $w_1 + \phi_2 \sqsubseteq \phi'_2 \ \wedge\ w_0 + \phi'_2 \sqsubseteq \phi'_0 \Longrightarrow w_0 + w_1 + \phi_2 \sqsubseteq \phi'_0$, we have

$$\forall\, \phi_2 \in \mathit{fin}\,(\blacktriangleright\iota_2).\ \exists\, \phi'_0 \in \mathit{fin}\,(\blacktriangleright\iota_0).\ w_0 + w_1 + \phi_2 \sqsubseteq \phi'_0\ .$$

By repeating this process for the remaining $\iota_3, \iota_4, \cdots$, we can find

$$\forall\, \phi_n \in \mathit{fin}\,(\blacktriangleright\iota_n).\ \exists\, \phi'_0 \in \mathit{fin}\,(\blacktriangleright\iota_0).\ w_0 + w_1 + \cdots + w_{n-1} + \phi_n \sqsubseteq \phi'_0\ .$$

for an arbitrary n. Because $\mathit{fin}\,(\blacktriangleright\iota_n)$ is nonempty, there exist some $\phi_n \in \mathit{fin}\,(\blacktriangleright\iota_n)$ and $\phi'_0 \in \mathit{fin}\,(\blacktriangleright\iota_0)$ such that $w_0 + w_1 + \cdots + w_{n-1} + \phi_n \sqsubseteq \phi'_0$. This means that $\mathit{fin}\,(\blacktriangleright\iota_0)$ includes an upper bound of the finite traces $w_0, w_1, w_2, \cdots$ that are, respectively, generated between successive unfoldings (as indicated by the use of $\blacktriangleright$ in $\blacktriangleright(w_0 \circ \blacktriangleright(w_1 \circ \blacktriangleright(w_2 \circ \cdots))))$.[5] Instances of algebraic temporal effects have to implement $\blacktriangleright$ and *fin* meeting inequations (9) and (10). One way to do so is to record finite effects between successive applications of $\blacktriangleright$ (i.e., ϕ in $\blacktriangleright(\phi \circ \blacktriangleright\iota)$) and have *fin* return an upper bound of all the finite effects recorded. This is exactly how the instances given in Section 4 implement $\blacktriangleright$ and *fin*.

Note that inequation (9) assumes that ι_0 is infinitely expandable, which means that a computation may diverge; conversely, a computation given a *non*-infinitely-expandable effect must terminate because, if diverging, it involves infinitely many unfolded computations, which requires the assigned effect be infinitely expandable. Because the inequation is used only when divergent computation is focused on, we can assume that ι_0 is infinitely expandable. If the restriction to infinitely expandable effects was lifted, the instance given in Section 4.1 would not satisfy inequation (9).

3.3 Type-and-Effect System

This section defines our effect system, which is parameterized over algebraic temporal effects. We here assume that their set $\mathbb{E}$ is given. Well-formedness of typing contexts $\vdash \Gamma$, value types $\Gamma \vdash T$, computation type $\Gamma \vdash C$, and effects $\Gamma \vdash e$ are defined in a straightforward way; see the supplementary material for their definitions.

Figure 3 shows all the typing and subtyping rules. Typing judgments take the form $\Gamma \vdash M : C$. We abbreviate it to $\Gamma \vdash M : T$ if $C \ = \ T\ \&\ \epsilon$, which means that expression M terminates with no effect. Most of the typing rules are standard or straightforward variants of those in the prior work [Birkedal and Møgelberg 2013; Koskinen and Terauchi 2014; Nakano 2000] except for (T_Prev). The rule (T_Var) for variables uses the notation $\Gamma(x)$, which stands for the type assigned to variable x by typing context Γ; it is undefined if Γ assigns no type to x. The rules (T_Const) and (T_Op) rest on metafunction *ty*, which assigns a base type to every constant and a first-order function type $B_1 \to \cdots \to B_n \to B_0$, abbreviated to $\overline{B} \to B_0$, to every primitive operation. The effect assigned to application of an operation o is given by metafunction *eff* and we assume that it is closed. The rule (T_Let) uses sequential composition $\trianglerighteq$ to concatenate the effect of a preceding expression with that of a succeeding one. For next constructs, there are two typing rules: (T_Next) for computations and

[5]The discussion here considers only finitely many traces $w_0, w_1, \cdots, w_n$, but we can adapt it to infinitely many traces $w_0, w_1, \cdots$ using the assumption that *fin* returns finite sets. Readers interested in the detail are referred to the proof of Theorem 2 (Liveness Soundness) in the supplementary material.

Typing rules $\boxed{\Gamma \vdash M : C}$ $\boxed{\Gamma \vdash M : T} \overset{\text{def}}{=} \Gamma \vdash M : T \mathbin{\&} \epsilon$

$$\frac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)}\ \text{T\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash c : ty(c)}\ \text{T\_Const} \qquad \frac{\vdash \Gamma \quad ty(o) = \overline{B} \to B_0 \quad \overline{\Gamma \vdash V : B}}{\Gamma \vdash o(\overline{V}) : B_0 \mathbin{\&} \mathit{eff}(o)}\ \text{T\_Op}$$

$$\frac{\Gamma, x\!:\!T \vdash M : C}{\Gamma \vdash \lambda x.M : T \to C}\ \text{T\_Abs} \qquad \frac{\Gamma \vdash V_1 : T \to C \quad \Gamma \vdash V_2 : T}{\Gamma \vdash V_1\, V_2 : C}\ \text{T\_App}$$

$$\frac{\Gamma \vdash V : \mathsf{bool} \quad \Gamma \vdash M_1 : C \quad \Gamma \vdash M_2 : C}{\Gamma \vdash \mathsf{if}\ V\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : C}\ \text{T\_If} \qquad \frac{\Gamma \vdash M_1 : T_1 \mathbin{\&} e_1 \quad \Gamma, x\!:\!T_1 \vdash M_2 : T_2 \mathbin{\&} e_2}{\Gamma \vdash \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ M_2 : T_2 \mathbin{\&} e_1 \mathbin{\underline{\triangleright}} e_2}\ \text{T\_Let}$$

$$\frac{\Gamma \vdash V : T[\mathsf{rec}\,\alpha.T/\alpha]}{\Gamma \vdash \mathsf{fold}\ V : \mathsf{rec}\,\alpha.T}\ \text{T\_Fold} \qquad \frac{\Gamma \vdash V : \mathsf{rec}\,\alpha.T}{\Gamma \vdash \mathsf{unfold}\ V : T[\mathsf{rec}\,\alpha.T/\alpha]}\ \text{T\_Unfold}$$

$$\frac{\Gamma \vdash M : C}{\Gamma \vdash \mathsf{next}\, M : \blacktriangleright C}\ \text{T\_Next} \qquad \frac{\Gamma \vdash V : T}{\Gamma \vdash \mathsf{next}\ V : \blacktriangleright T}\ \text{T\_NextV} \qquad \frac{\Gamma \vdash V : \blacktriangleright \tau}{\Gamma \vdash \mathsf{prev}\ V : \tau}\ \text{T\_Prev}$$

$$\frac{\Gamma \vdash V_1 : \blacktriangleright(T \to C) \quad \Gamma \vdash V_2 : \blacktriangleright T}{\Gamma \vdash V_1 \circledast V_2 : \blacktriangleright C}\ \text{T\_LApp} \qquad \frac{\Gamma \vdash M : C \quad \Gamma; \emptyset \vdash C <: C'}{\Gamma \vdash M : C'}\ \text{T\_Sub}$$

$$\frac{\Gamma, X \vdash M : C}{\Gamma \vdash \Lambda X.M : \forall X.C}\ \text{T\_EAbs} \qquad \frac{\Gamma \vdash V : \forall X.C \quad \Gamma \vdash e}{\Gamma \vdash V\, e : C[e/X]}\ \text{T\_EApp}$$

Subtyping rule for computation types $\boxed{\Gamma; \Delta \vdash C_1 <: C_2}$

$$\frac{\Gamma; \Delta \vdash T_1 <: T_2 \quad \Gamma \vdash e_1 \sqsubseteq e_2}{\Gamma; \Delta \vdash T_1 \mathbin{\&} e_1 <: T_2 \mathbin{\&} e_2}$$

Subtyping rules for value types $\boxed{\Gamma; \Delta \vdash T_1 <: T_2}$

$$\frac{\Gamma \vdash T \quad \vdash \Delta \quad dom(\Delta) \cap ftv(T) = \emptyset}{\Gamma; \Delta \vdash T <: T} \qquad \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta \quad \vdash \Delta \quad \alpha <: \beta \in \Delta}{\Gamma; \Delta \vdash \alpha <: \beta}$$

$$\frac{\Gamma; \Delta \vdash T_2 <: T_1 \quad \Gamma; \Delta \vdash C_1 <: C_2}{\Gamma; \Delta \vdash T_1 \to C_1 <: T_2 \to C_2} \qquad \frac{\Gamma, X; \Delta \vdash C_1 <: C_2}{\Gamma; \Delta \vdash \forall X.C_1 <: \forall X.C_2}$$

$$\frac{\alpha \notin ftv(T_2) \quad \beta \notin ftv(T_1) \quad \Gamma, \alpha, \beta; \Delta, \alpha <: \beta \vdash T_1 <: T_2}{\Gamma; \Delta \vdash \mathsf{rec}\,\alpha.T_1 <: \mathsf{rec}\,\beta.T_2} \qquad \frac{\Gamma; \Delta \vdash T_1 <: T_2}{\Gamma; \Delta \vdash \blacktriangleright T_1 <: \blacktriangleright T_2}$$

Subeffecting rule $\boxed{\Gamma \vdash e_1 \sqsubseteq e_2}$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2 \quad \forall d^{\mathcal{E}} \in \mathcal{D}[dom^{\mathsf{eff}}(\Gamma)].\ \mathbb{E}[e_1](d^{\mathcal{E}}) \sqsubseteq \mathbb{E}[e_2](d^{\mathcal{E}})}{\Gamma \vdash e_1 \sqsubseteq e_2}$$

Fig. 3. Typing and subtyping rules.

(T_NextV) for values. The difference between them is effect assignment: while (T_Next) assigns effect $\underline{\blacktriangleright} e$ to expression next M if M has effect e, (T_NextV) assigns the pure effect ϵ. Note that the interpretation of $\underline{\blacktriangleright}\epsilon$ in algebraic temporal effects may be different from that of ϵ in general; indeed, we assign different denotations to them in temporal regular effects. The rule (T_Prev) represents that the eliminator prev makes first-order values produced by unfolded computation currently available. The rule (T_LApp) expresses that $\circledast$ allows function applications for functions and arguments of later types.

The subtyping rules are standard in the presence of recursive types and later types [Bengtson et al. 2011; Nakano 2000]. The metavariable Δ represents finite sequences of type variable bindings of the form $\alpha <: \beta$, which means that type variable α is a subtype of type variable β. The well-formedness judgment $\vdash \Delta$ means that the type variables bound by Δ are distinct from each other. Subtyping between computation types compare effects using subeffecting.

Subeffecting relies on the interpretation of syntactic effects to algebraic temporal effects. We assume that an algebraic temporal effect $\delta_{\mathbb{E}}(\mathbf{e})$ is assigned for every effect constructor $\mathbf{e}$.

Definition 3 (Effect Interpretation). *The interpretation $\mathbb{E}[e](d^{\mathcal{E}})$ of an effect e with a mapping $d^{\mathcal{E}}$ from effect variables to algebraic temporal effects is an algebraic temporal effect, defined as follows:*

$$\begin{array}{llllll}
\mathbb{E}[\epsilon](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & \epsilon_{\mathbb{E}} & \mathbb{E}[X](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & d^{\mathcal{E}}(X) \\
\mathbb{E}[\mathbf{e}](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & \delta_{\mathbb{E}}(\mathbf{e}) & \mathbb{E}[\underline{\blacktriangleright} e](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & \blacktriangleright \mathbb{E}[e](d^{\mathcal{E}}) \\
\mathbb{E}[e_1 \underline{\vee} e_2](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & \mathbb{E}[e_1](d^{\mathcal{E}}) \vee \mathbb{E}[e_2](d^{\mathcal{E}}) & \mathbb{E}[e_1 \underline{\triangleright} e_2](d^{\mathcal{E}}) & \stackrel{\text{def}}{=} & \mathbb{E}[e_1](d^{\mathcal{E}}) \triangleright \mathbb{E}[e_2](d^{\mathcal{E}})
\end{array}$$

The subeffecting rule uses the notation $dom^{\text{eff}}(\Gamma)$, which denotes the set of effect variables bound by typing context Γ, and $\mathcal{D}[S]$, which, given a set S of effect variables, denotes the set of functions that map effect variables in S to algebraic temporal effects. Hence, it means that effect e_1 is a subeffect of e_2 if the interpretation of e_1 is a subeffect of that of e_2 under any interpretation $d^{\mathcal{E}}$ of free effect variables in e_1 and e_2.

3.4 Examples

This section illustrates how programs in our language behave and are typechecked through a few examples. For that, in this section we assume that: the finite effect set $\mathbb{F}$ accommodates Σ^* (i.e., finite sequences) over a finite symbol set Σ; for $w_1, w_2 \in \Sigma^*$, the finite effect composition $w_1 \cdot w_2$ is defined to be the concatenation of w_1 and w_2; and the finite effect $\mathbf{1}$ coincides with the empty sequence ε. We also assume an operation $\mathsf{ev}[\mathbf{a}]$, which raises event $\mathbf{a} \in \Sigma$. This behavior is formulated by letting $\delta_{\mathsf{e}}(\mathsf{ev}[\mathbf{a}]) \stackrel{\text{def}}{=} \mathbf{a}$. Furthermore, we use the following shorthand for simplicity:

$$\begin{array}{lll}
M_1; M_2 \stackrel{\text{def}}{=} \mathsf{let}\, x_1 = M_1\, \mathsf{in}\, M_2 & \mathsf{unfold}^{\blacktriangleright} \stackrel{\text{def}}{=} \mathsf{next}\,(\lambda x.\mathsf{unfold}\, x) & \mathsf{fold}^{\blacktriangleright} \stackrel{\text{def}}{=} \mathsf{next}\,(\lambda x.\mathsf{fold}\, x) \\
T_1 \to T_2 \stackrel{\text{def}}{=} T_1 \to (T_2 \ \&\ \epsilon) & \forall X.T \stackrel{\text{def}}{=} \forall X.(T \ \&\ \epsilon) &
\end{array}$$

We often place a non-value expression at the position where a value is expected. In such a case, we assume that let constructs are inserted appropriately. For example, a function application $M_1\, M_2$ is regarded as $\mathsf{let}\, x = M_1\, \mathsf{in}\, \mathsf{let}\, y = M_2\, \mathsf{in}\, x\, y$ implicitly if neither M_1 nor M_2 is a value. Note that $\mathsf{unfold}^{\blacktriangleright}$ unfolds a recursive type beneath $\blacktriangleright$. Namely, given a value V of type $\blacktriangleright \mathsf{rec}\, \alpha.T$, the application $\mathsf{unfold}^{\blacktriangleright} \circledast V$ is of the type $\blacktriangleright T[\mathsf{rec}\, \alpha.T/\alpha] \ \&\ \underline{\blacktriangleright}\epsilon$. Similarly, given a value V of type $\blacktriangleright T[\mathsf{rec}\, \alpha.T/\alpha]$, the application $\mathsf{fold}^{\blacktriangleright} \circledast V$ has the type $\blacktriangleright \mathsf{rec}\, \alpha.T \ \&\ \underline{\blacktriangleright}\epsilon$.

Example 1. The first example is a variant of the Ω combinator presented in Section 1. It is changed so that function f raises event $\mathbf{a}$ when applied; therefore, the program diverges with infinite trace $\mathbf{a}^{\omega} = \mathbf{a} \cdot \mathbf{a} \cdot \cdots$. In our language, the example is described as follows:

$$\mathsf{let}\, f = V\, \mathsf{in}\, f\, (\mathsf{next}\, (\mathsf{fold}\, f))$$

where

$$V \stackrel{\text{def}}{=} \lambda x.(\mathsf{ev}[\mathbf{a}](); M) \qquad M \stackrel{\text{def}}{=} \mathsf{let}\, y = (\mathsf{unfold}^{\blacktriangleright} \circledast x) \circledast (\mathsf{next}\, x)\, \mathsf{in}\, \mathsf{prev}\, y\, .$$

First, let us confirm the behavior of the program. (here, $M_1 \mathbin{\&} \phi_1 \longrightarrow^* M_2 \mathbin{\&} \phi_2$ means $M_1 \longrightarrow^* M_2 \mathbin{\&} \phi_2$, and ϕ_1 is the effect caused by the previous computation):

$$
\begin{array}{cl}
 & \mathsf{let}\, f = V \,\mathsf{in}\, f\,(\mathsf{next}\,(\mathsf{fold}\, f)) \\
\longrightarrow & V\,(\mathsf{next}\,(\mathsf{fold}\, V)) \mathbin{\&} \mathbf{1} \\
\longrightarrow & \mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x] \mathbin{\&} \mathbf{1} \\
\longrightarrow & M[\mathsf{next}\,(\mathsf{fold}\, V)/x] \mathbin{\&} \mathbf{a} \\
= & \mathsf{let}\, y = (\mathsf{unfold}^{\blacktriangleright} \circledast (\mathsf{next}\,(\mathsf{fold}\, V))) \circledast (\mathsf{next}\,(\mathsf{next}\,(\mathsf{fold}\, V)))\,\mathsf{in}\,\mathsf{prev}\, y \mathbin{\&} \mathbf{a} \\
\longrightarrow^* & \mathsf{let}\, y = (\mathsf{next}\, V) \circledast (\mathsf{next}\,(\mathsf{next}\,(\mathsf{fold}\, V)))\,\mathsf{in}\,\mathsf{prev}\, y \mathbin{\&} \mathbf{1} \\
\longrightarrow & \mathsf{let}\, y = \mathsf{next}\,(V\,(\mathsf{next}\,(\mathsf{fold}\, V)))\,\mathsf{in}\,\mathsf{prev}\, y \mathbin{\&} \mathbf{1} \\
\longrightarrow & \mathsf{let}\, y = \mathsf{next}\,(\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x])\,\mathsf{in}\, y \mathbin{\&} \mathbf{1}
\end{array}
$$

The above evaluation shows that the program's behavior conforms to the following pattern:

$$E_i[\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x]] \quad \longrightarrow^* \quad E_{i+1}[\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x]] \mathbin{\&} \mathbf{a}$$

for any natural number i, where $E_0, E_1, \cdots$ are evaluation contexts defined as follows: $E_0 \overset{\mathrm{def}}{=} [\,]$; and $E_{i+1} \overset{\mathrm{def}}{=} \mathsf{let}\, y = \mathsf{next}\, E_i \,\mathsf{in}\,\mathsf{prev}\, y$ (note that $[\,]$ is the hole and $E[M]$ denotes the expression obtained by filling the hole in E with the expression M). Therefore, the program diverges with $\mathbf{a}^{\omega}$:

$$
\begin{array}{rcl}
V\,(\mathsf{next}\,(\mathsf{fold}\, V)) & \longrightarrow & E_0[\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x]] \mathbin{\&} \mathbf{1} \\
 & \longrightarrow^* & E_1[\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x]] \mathbin{\&} \mathbf{a} \\
 & \longrightarrow^* & E_2[\mathsf{ev}[\mathbf{a}]();M[\mathsf{next}\,(\mathsf{fold}\, V)/x]] \mathbin{\&} \mathbf{a} \\
 & \longrightarrow^* & \cdots
\end{array}
$$

Next, let us see the effect constraint imposed on the program. Let $T \overset{\mathrm{def}}{=} \mathsf{rec}\,\alpha.{\blacktriangleright}\alpha \rightarrow \mathsf{unit} \mathbin{\&} e$ for some effect e. Then, function V has type ${\blacktriangleright}T \rightarrow \mathsf{unit} \mathbin{\&} e$ if the subeffecting $\mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}e \mathbin{\underline{\triangleright}} \epsilon \sqsubseteq e$ holds (where $\underline{\blacktriangleright}\epsilon$, $\underline{\blacktriangleright}e$, and ϵ comes from $\mathsf{unfold}^{\blacktriangleright} \circledast x$, its application to $\mathsf{next}\, x$, and $\mathsf{prev}\, z$, respectively; readers interested in the detail are referred to the supplementary material), and the remaining is typechecked by:

$$
\dfrac{\begin{array}{c} f : {\blacktriangleright}T \rightarrow \mathsf{unit} \mathbin{\&} e \vdash f : ({\blacktriangleright}T \rightarrow \mathsf{unit} \mathbin{\&} e) \\ f : {\blacktriangleright}T \rightarrow \mathsf{unit} \mathbin{\&} e \vdash \mathsf{next}\,(\mathsf{fold}\, f) : {\blacktriangleright}T \end{array}}{f : {\blacktriangleright}T \rightarrow \mathsf{unit} \mathbin{\&} e \vdash f(\mathsf{next}\,(\mathsf{fold}\, f)) : \mathsf{unit} \mathbin{\&} e}\ (\mathrm{T\_A{\scriptsize PP}})
$$

By the equational theory of algebraic temporal effects, we can find that

$$
\begin{array}{rl}
e \sqsupseteq & \mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}e \mathbin{\underline{\triangleright}} \epsilon \\
\sqsupseteq & \mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}(\mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}e \mathbin{\underline{\triangleright}} \epsilon) \mathbin{\underline{\triangleright}} \epsilon \\
\sqsupseteq & \mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}(\mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}(\mathit{eff}(\mathsf{ev}[\mathbf{a}]) \mathbin{\underline{\triangleright}} \cdots) \mathbin{\underline{\triangleright}} \epsilon) \mathbin{\underline{\triangleright}} \epsilon \ \sqsupseteq\ \cdots .
\end{array}
$$

It indicates that (the interpretation of) the effect e only involves the infinite trace $\mathbf{a}^{\omega}$. In turn, by the productivity of our temporal effect system, it is ensured that the program generates the trace $\mathbf{a}^{\omega}$. Section 4.1 provides a concrete effect representation for e that enables to prove this claim formally.

Example 2. The second example is a fixed-point combinator with recursive types. In our language, it can be expressed by function fix defined as follows:

$$V \overset{\mathrm{def}}{=} \lambda g.\lambda y.f\,((\mathsf{unfold}^{\blacktriangleright} \circledast g) \circledast (\mathsf{next}\, g))\, y \qquad \mathsf{fix} \overset{\mathrm{def}}{=} \lambda f.V\,(\mathsf{next}\,(\mathsf{fold}\, V))$$

Given type T_1, T_2, and effect e, let $T_0 \overset{\mathrm{def}}{=} T_1 \rightarrow (T_2 \mathbin{\&} e)$. The effect system assigns a type $({\blacktriangleright}T_0 \rightarrow T_0) \rightarrow T_0$ to the fixed-point combinator fix if $\underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathbin{\underline{\triangleright}} e \sqsubseteq e$ holds. The assigned type is similar to the type assigned to a fixed-point combinator by Nakano [2000]. The supplementary material gives a typing derivation for this type assignment.

Example 3. Next, we consider a simple scheduler sc for two nonterminating concurrent processes. It is defined using fixed-point combinator fix, as follows:

$$\begin{aligned} V &\overset{\text{def}}{=} \lambda x'.(\mathsf{unfold}\ y)\ (\lambda y'.\mathsf{let}\ z = f \circledast x'\ \mathsf{in}\ z \circledast y') \\ \mathsf{sc} &\overset{\text{def}}{=} \mathsf{fix}\,\lambda f.\lambda x.\lambda y.(\mathsf{unfold}\ x)\ V\,. \end{aligned}$$

Function sc takes two functions x and y, which represent concurrent processes executed alternately. When called, sc begins by executing (the result of unfolding) the first process x. The argument V is a function that may be invoked by the first process x to switch the control to the second process y. To call V, x passes its continuation x'. Then, V executes (the result of unfolding) the process y. Similarly to x, y also takes as an argument a function applied when switching the control back to x. The function argument to y takes y's continuation y', and then resumes x's continuation x' by calling scheduler sc recursively via function f passed by fixed-point combinator fix.

Scheduler sc has a type T_0 defined as follows:

$$\begin{aligned} T_x &\overset{\text{def}}{=} \mathsf{rec}\,\alpha.((\blacktriangleright\alpha \to (B\ \&\ e_y)) \to (B\ \&\ e_x)) \\ T_y &\overset{\text{def}}{=} \mathsf{rec}\,\alpha.((\blacktriangleright\alpha \to (\blacktriangleright B\ \&\ \underline{\blacktriangleright}\epsilon \trianglerighteq \underline{\blacktriangleright}e_x)) \to (B\ \&\ e_y)) \\ T_0 &\overset{\text{def}}{=} T_x \to (T_y \to (B\ \&\ e_x)) \end{aligned}$$

where e_x and e_y are the effects performed by processes x and y, respectively, and B is the type of the computation results of x and y (if any). The type T_x of x is defined recursively because x passes its continuation to the argument, and the continuation is expected to behave as x. For a similar reason, the type of y is also expressed using a recursive type. The effects $\underline{\blacktriangleright}\epsilon$ and $\underline{\blacktriangleright}e_x$ in the type of y are caused by $f \circledast x'$ and $z \circledast y'$ in V, respectively. A typing derivation for sc is found in the supplementary material.

As example processes to be scheduled by sc, we consider the following two functions px and py:

$$\begin{aligned} \mathsf{px} &\overset{\text{def}}{=} \mathsf{fix}\,\lambda f.\lambda g.\mathsf{ev}[\mathbf{x}]();\mathsf{let}\ z = \mathsf{fold}^{\blacktriangleright} \circledast f\ \mathsf{in}\ g\ z \\ \mathsf{py} &\overset{\text{def}}{=} \mathsf{fix}\,\lambda f.\lambda g.\mathsf{ev}[\mathbf{y}]();\mathsf{let}\ z = \mathsf{fold}^{\blacktriangleright} \circledast f\ \mathsf{in}\ \mathsf{prev}\,(g\ z). \end{aligned}$$

They repeatedly raise events $\mathbf{x}$ and $\mathbf{y}$, respectively, and then pass the control to the other process. Their continuations are (the results of folding) themselves represented by variable f. To typecheck sc with px and py, assume $\mathit{eff}(\mathsf{ev}[\mathbf{x}]) \trianglerighteq \underline{\blacktriangleright}\epsilon \trianglerighteq e_y \sqsubseteq e_x$ and $\mathit{eff}(\mathsf{ev}[\mathbf{y}]) \trianglerighteq \underline{\blacktriangleright}\epsilon \trianglerighteq (\underline{\blacktriangleright}\epsilon \trianglerighteq \underline{\blacktriangleright}e_x) \sqsubseteq e_y$. Under these assumptions, functions px and py have type $(\blacktriangleright T_x \to (B\ \&\ e_y)) \to (B\ \&\ e_x)$ and $(\blacktriangleright T_y \to (\blacktriangleright B\ \&\ \underline{\blacktriangleright}\epsilon \trianglerighteq \underline{\blacktriangleright}e_x)) \to (B\ \&\ e_y)$, respectively (again, readers interested in the typing derivations are referred to the supplementary material). Because the types of px and py are matched with the results of unfolding the types T_x and T_y, fold px and fold py have the types T_x and T_y, respectively. Then, expression sc (fold px) (fold py) has type $B\ \&\ e_x$. We can find that $e_x \sqsupseteq \mathit{eff}(\mathsf{ev}[\mathbf{x}]) \trianglerighteq \underline{\blacktriangleright}\epsilon \trianglerighteq (\mathit{eff}(\mathsf{ev}[\mathbf{y}]) \trianglerighteq \underline{\blacktriangleright}\epsilon \trianglerighteq (\underline{\blacktriangleright}\epsilon \trianglerighteq \underline{\blacktriangleright}e_x))$ holds. It indicates that the expression repeats the sequential raise of events $\mathbf{x}$ and $\mathbf{y}$ infinitely.

The use of prev in py enables returning the result from the continuation of process x, if any, as the result of process y. If prev were unsupported, there would be no means to bridge the gap between the type $\blacktriangleright B$ of the return value of x's continuation and the type B of the return value of y.

Example 4. Finally, we verify the following program with functional objects in a syntax like Java.

```
interface P { void print(); }
class Int extends P {
  int x; void print() { ev[I](); print_int(x); }
}
class Seq extends P {
  P x; P y; void print() { x.print(); y.print(); }
}
class Omega extends P {
  void print() { ev[O](); this.print(); }
}
// main expression
new Seq(new Seq(new Int(0), new Int(1)), new Omega()).print();
```

We adopt the convention that, given a class `C` with fields $\mathtt{f}_1, \cdots, \mathtt{f}_n$, expression `new C(`$V_1$`, `$\cdots$`, `$V_n$`)` creates a new object of class `C` by assigning values $V_1, \cdots, V_n$ to fields $\mathtt{f}_1, \cdots, \mathtt{f}_n$. All the classes give method `print`, but its behavior varies depending on the class. The objects of class `Int` print the value of field `x` after raising event $\mathbf{I}$. The objects of class `Seq` call `print` of fields `x` and `y` sequentially. The objects of class `Omega` raise event $\mathbf{O}$ and then call method `print` of themselves recursively.

To encode this program, we use type T, defined as follows, that expresses class `P`.

$$T \overset{\text{def}}{=} \mathrm{rec}\,\alpha.(\blacktriangleright\alpha \to \mathrm{unit} \to (\mathrm{unit}\ \&\ X))$$

Hereinafter, we write T_e for type $\blacktriangleright T[e/X] \to \mathrm{unit} \to (\mathrm{unit}\ \&\ e)$. It expresses class `P` with method `print` to which effect e is assigned. Type T_e indicates that functions representing objects of class `P` take themselves as first arguments and then return functions representing `print` of type $\mathrm{unit} \to (\mathrm{unit}\ \&\ e)$. A call to `print` of an object V with type T_e is denoted by expression

$$V.\mathrm{print}\ () \overset{\text{def}}{=} \mathrm{let}\ x = V\ (\mathrm{next}\ (\mathrm{fold}\ V))\ \mathrm{in}\ x\ ()$$

of type $\mathrm{unit}\ \&\ e$. Similarly, a call to `print` of a self-referential variable this, which has type $\blacktriangleright T[e/X]$, is encoded by expression

$$\mathrm{self.print}\ () \overset{\text{def}}{=} \mathrm{let}\ x = (\mathrm{unfold}^{\blacktriangleright} \circledast \mathrm{this}) \circledast (\mathrm{next\ this}) \circledast (\mathrm{next}\ ())\ \mathrm{in\ prev}\ x$$

of type $\mathrm{unit}\ \&\ \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}e$. Then, for each class `C`, object constructor `new C(`$\cdots$`)` is encoded into a function as follows:

$$\begin{array}{rcll}
\mathrm{cInt} & \overset{\text{def}}{=} & \lambda x.\lambda \mathrm{this}.\lambda z.\mathrm{ev}[\mathbf{I}]();\mathrm{print\_int}(x) & : \mathrm{int} \to T_{\mathit{eff}(\mathrm{ev}[\mathbf{I}])} \\
\mathrm{cSeq} & \overset{\text{def}}{=} & \Lambda X_1.\Lambda X_2.\lambda x.\lambda y.\lambda \mathrm{this}.\lambda z.x.\mathrm{print}\ ();y.\mathrm{print}\ () & : \forall X_1.\forall X_2.(T_{X_1} \to T_{X_2} \to T_{X_1 \mathrel{\underline{\triangleright}} X_2}) \\
\mathrm{cOmega} & \overset{\text{def}}{=} & \lambda \mathrm{this}.\lambda z.\mathrm{ev}[\mathbf{O}]();\mathrm{self.print}\ () & : T_{e_{\mathbf{O}}}
\end{array}$$

where we assume $\mathit{eff}(\mathrm{ev}[\mathbf{O}]) \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}e_{\mathbf{O}} \sqsubseteq e_{\mathbf{O}}$ for some effect $e_{\mathbf{O}}$. The main expression is denoted by

$$\begin{array}{l}
\mathrm{let}\ x = \mathrm{cSeq}\ \mathit{eff}(\mathrm{ev}[\mathbf{I}])\ \mathit{eff}(\mathrm{ev}[\mathbf{I}])\ (\mathrm{cInt}\ 1)\ (\mathrm{cInt}\ 2)\ \mathrm{in} \\
\mathrm{let}\ y = \mathrm{cSeq}\ (\mathit{eff}(\mathrm{ev}[\mathbf{I}]) \mathrel{\underline{\triangleright}} \mathit{eff}(\mathrm{ev}[\mathbf{I}]))\ e_{\mathbf{O}}\ x\ \mathrm{cOmega}\ \mathrm{in} \\
y.\mathrm{print}\ () : \mathrm{unit}\ \&\ \mathit{eff}(\mathrm{ev}[\mathbf{I}]) \mathrel{\underline{\triangleright}} \mathit{eff}(\mathrm{ev}[\mathbf{I}]) \mathrel{\underline{\triangleright}} e_{\mathbf{O}}\ .
\end{array}$$

By the above subeffecting assumption on $e_{\mathbf{O}}$, the effect $\mathit{eff}(\mathrm{ev}[\mathbf{I}]) \mathrel{\underline{\triangleright}} \mathit{eff}(\mathrm{ev}[\mathbf{I}]) \mathrel{\underline{\triangleright}} e_{\mathbf{O}}$ of the program accommodates the trace $\mathbf{I} \cdot \mathbf{I} \cdot \mathbf{O}^{\omega}$ it generates.

3.5 Properties

This section states the properties of algebraic temporal effects and then shows two soundness properties: safety and liveness soundness, which guarantee the correctness of the effects predicted for terminating and diverging programs, respectively. Hereinafter, for algebraic temporal effect $\zeta = (\phi, \iota)$, we write $\zeta.F$ and $\zeta.I$ to denote ϕ and ι, respectively. We assume that, for every operation o, $\mathbb{E}[\mathit{eff}(o)](\emptyset) = (\delta_{\mathrm{e}}(o), \mathbf{0})$, which means that the syntactic effect $\mathit{eff}(o)$ statically assigned to the operation o is matched with the finite effect $\delta_{\mathrm{e}}(o)$ that happens at run time in calling o.

Lemma 1. *The following (in)equations hold.*

(Reflexivity) $\forall \zeta.\ \zeta \sqsubseteq \zeta$.
(Transitivity) $\forall \zeta_1, \zeta_2, \zeta_3.\ \zeta_1 \sqsubseteq \zeta_2 \ \land\ \zeta_2 \sqsubseteq \zeta_3 \Longrightarrow \zeta_1 \sqsubseteq \zeta_3$.
(Monotonicity of Later) $\forall \zeta_1, \zeta_2.\ \zeta_1 \sqsubseteq \zeta_2 \Longrightarrow \blacktriangleright\zeta_1 \sqsubseteq \blacktriangleright\zeta_2$.
(Monotonicity of Composition) $\forall \zeta_1, \zeta_2.\ \zeta_1 \sqsubseteq \zeta_2 \Longrightarrow \forall \zeta.\ \zeta_1 \triangleright \zeta \sqsubseteq \zeta_2 \triangleright \zeta \ \land\ \zeta \triangleright \zeta_1 \sqsubseteq \zeta \triangleright \zeta_2$.
(Associativity of Composition) $\forall \zeta_1, \zeta_2, \zeta_3.\ \zeta_1 \triangleright (\zeta_2 \triangleright \zeta_3) \ =\ (\zeta_1 \triangleright \zeta_2) \triangleright \zeta_3$.
(Pre-Effect Identity) $\forall \zeta.\ \epsilon_{\mathbb{E}} \triangleright \zeta \ =\ \zeta$.

Proof. By the algebraic properties of temporal effects. □

3.5.1 *Safety Soundness.* Safety soundness is formulated as type safety, which is shown via progress and preservation [Wright and Felleisen 1994].

Theorem 3.1 (Safety Soundness). *If $\emptyset \vdash M : T\ \&\ e$ and $M \longrightarrow^* M'\ \&\ \phi$ and M' cannot evaluate further, then $M' = V$ and $\emptyset \vdash V : T$ for some V, and $\phi \sqsubseteq \mathbb{E}[e](\emptyset).F$.*

Subeffecting $\phi \sqsubseteq \mathbb{E}[e](\emptyset).F$ means that the effect ϕ performed by M is contained in the finite part of the assigned effect e.

3.5.2 *Liveness Soundness.* Liveness soundness states that the infinite effects assigned to divergent programs correctly predict infinite sequences of the effects performed by the programs. To formalize liveness soundness, we characterize how programs diverge. To this end, we refer to Example 1 in Section 3.4. In what follows, we call evaluation contexts *guarded* if their holes are placed immediately beneath the next constructor. Formally, they are described by the following grammar:

$$E \ ::=\ \mathsf{next}\ [\,]\ \mid\ \mathsf{let}\ x = E\ \mathsf{in}\ M_2\ .$$

We can find two observations from Example 1. First, a divergent program evaluates to an expression of the form $E_0[M_0]$ for some M_0 and E_0. Second, there exist infinitely many M_i and E_i ($i > 0$) such that M_{i-1} evaluates to $E_i[M_i]$. These two phenomenons are not specific to Example 1; they occur in any divergent program. Using this fact, we can prove the following property.

Lemma 2 (Infinite Guarded Reduction Chain). *If $\emptyset \vdash M : C$ and $M \Uparrow \varpi$, then there exist infinitely many expressions M_i, value types T_i, effects e_i, and finite effects ϕ_i ($i \geq 0$) such that: $M = M_0$; $C = T_0\ \&\ e_0$; and, for any i, $\emptyset \vdash M_i : T_i\ \&\ e_i$ and $\exists E_{i+1}.\ M_i \longrightarrow^* E_{i+1}[M_{i+1}]\ \&\ \phi_i$ and $\phi_i \circ \blacktriangleright\mathbb{E}[e_{i+1}](\emptyset).I \sqsubseteq \mathbb{E}[e_i](\emptyset).I$.*

In subeffecting $\phi_i \circ \blacktriangleright\mathbb{E}[e_{i+1}](\emptyset).I \sqsubseteq \mathbb{E}[e_i](\emptyset).I$, the infinite subeffect $\phi_i \circ \blacktriangleright\mathbb{E}[e_{i+1}](\emptyset).I$ represents the evaluation $M_i \longrightarrow^* E_{i+1}[M_{i+1}]\ \&\ \phi_i$ precisely: the evaluation performs ϕ_i, and the remaining computation $E_{i+1}[M_{i+1}]$ has infinite effect $\blacktriangleright\mathbb{E}[e_{i+1}](\emptyset).I$ because divergent expression M_{i+1} with effect e_{i+1} is at the redex position under constructor next. Our proof of this lemma rests on the fact that a divergent program evaluates to an expression of the form $E[M]$ for some divergent expression M and guarded evaluation context E. We prove it using a logical relation. Interested readers are referred to the supplementary material.

Now, we show liveness soundness.

Theorem 3.2 (Liveness Soundness). *If $\emptyset \vdash M : T\ \&\ e$ and $M \Uparrow \varpi$, then there exist some $\iota \in \mathbb{I}$, infinitely many $\phi_0, \phi_1, \cdots \in \mathbb{F}$, and $i_1, i_2, \cdots \geq 0$ such that: (1) $\phi_0 = \varpi(0) \cdot \cdots \cdot \varpi(i_1 - 1)$; (2) for any $j > 0$, $\phi_j = \varpi(i_j) \cdot \cdots \cdot \varpi(i_{j+1} - 1)$ and $i_j < i_{j+1}$; (3) $\phi_0 \circ \blacktriangleright\iota \sqsubseteq \mathbb{E}[e](\emptyset).I$; and (4) there exists some $\phi \in \mathit{fin}\,(\blacktriangleright\iota)$ such that $\forall j > 0.\ \phi_j \sqsubseteq \phi$.*

To see what this theorem means, assume that a program M uses only the event-raising operation $\mathsf{ev}[\mathbf{a}]$. The assumption $M \Uparrow \varpi$ means that M generates some infinite trace ϖ. The theorem means that the trace ϖ can be split into a finite prefix w_0 (corresponding to ϕ_0 in the theorem) and an infinite suffix ϖ_0, and, furthermore, ϖ_0 can be split into infinitely many finite traces $w_1, w_2, \cdots$ (corresponding to $\phi_1, \phi_2, \cdots$ in the theorem) such that some $\phi \in \mathit{fin}\,(\blacktriangleright\iota)$ contains all of them. Because the infinite effect $\mathbb{E}[e](\emptyset).I$ of the syntactic effect e assigned to M overapproximates $w_0 \circ \blacktriangleright\iota$, it recognizes the infinite trace $w_0 \cdot w_1 \cdot w_2 \cdot \cdots$.

4 Instances of Algebraic Temporal Effects

This section provides two instances of algebraic temporal effects: *temporal regular effects* and *temporal fixpoint effects.*

4.1 Temporal Regular Effects

This section introduces temporal regular effects, regular effects for short. Regular effects enable us to check that the finite traces of terminating programs conform to a regular expression and that the infinite traces of divergent programs conform to an ω-regular expression.

4.1.1 Definition and Properties. We employ regular expressions as finite effects in regular effects.

Definition 4 (Regular Expressions). *Regular expressions over a set Σ are defined as follows:*

$$r \quad ::= \quad \emptyset \mid \mathbf{1}_\mathrm{r} \mid \mathbf{a} \mid r_1 \cdot_\mathrm{r} r_2 \mid r_1 +_\mathrm{r} r_2 \mid r^*$$

where $\mathbf{a} \in \Sigma$. We write r^+ for $r \cdot_\mathrm{r} r^*$. The language $\mathcal{L}(r) \subseteq \Sigma^*$ of a regular expression r is the set of all the finite words conforming to r, defined in a standard manner. A partial order $\sqsubseteq_\mathrm{r}$ on regular expressions is defined as: $r_1 \sqsubseteq_\mathrm{r} r_2$ if and only if $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$. Regular expressions r_1 and r_2 are equivalent, written $r_1 \equiv_\mathrm{r} r_2$, if $\mathcal{L}(r_1) = \mathcal{L}(r_2)$. Re$(\Sigma)$ is the set of regular expressions modulo $\equiv_\mathrm{r}$.

To give an instance of infinite effects with ω-regular expressions, we use the fact that any ω-regular expression can be represented in the form $r_{11} \cdot_\mathrm{r} r_{12}^\omega +_\mathrm{r} \cdots +_\mathrm{r} r_{n1} \cdot_\mathrm{r} r_{n2}^\omega$ $(n > 0)$ [Perrin and Pin 2004]. Our formulation of infinite effects represents such an ω-regular expression by a finite set of pairs of regular expressions as $\{(r_{11}, r_{12}), \cdots, (r_{n1}, r_{n2})\}$. Once deciding to adopt this formulation, a join and mixed-product operation can be defined naturally as shown shortly. A later operation is implemented as explained in Section 2.3. That is, it maps a set $\{(r_{11}, r_{12}), \cdots, (r_{n1}, r_{n2})\}$ to $\{(\mathbf{1}_\mathrm{r}, r_{11} +_\mathrm{r} r_{12}), \cdots, (\mathbf{1}_\mathrm{r}, r_{n1} +_\mathrm{r} r_{n2})\}$ by folding each r_{i1} into r_{i2} because r_{i1} represents the finite effect between two successive unfoldings when the later operation is applied.

We need to handle the representation of zero infinite effect $\mathbf{0}$ with care. One seemingly reasonable approach might be to take the empty set as $\mathbf{0}$ and define the later operation on it to return $\emptyset$ as is. However, in general, the later modality with $\blacktriangleright\mathbf{0} = \mathbf{0}$ hinders concluding that programs with the infinite effect $\mathbf{0}$ terminate although we intend it to denote the termination of programs. One—perhaps the only—way to guarantee that a program with the infinite effect $\mathbf{0}$ terminates (more precisely, guarantee that it generate no infinite trace) is to define $\mathit{fin}\,(\mathbf{0})$ to be the set $\{\phi_0\}$ with the finite effect ϕ_0 that contains no finite trace (for regular expressions in Definition 4, ϕ_0 corresponds to $\emptyset$). With such a definition, it is easy to check that any program M with $\mathbf{0}$ generates no infinite trace. If $M \Uparrow \varpi$, the liveness soundness (Theorem 3.2) implies that some nonempty finite traces comprising ϖ are contained in ϕ_0, although ϕ_0 is assumed to contain no finite trace; thus, the assumption $M \Uparrow \varpi$ implies a contradiction, that is, the program M terminates. The infinite effect $\mathbf{0}$ meeting $\blacktriangleright\mathbf{0} = \mathbf{0}$ disables this reasoning. To see it, assume $\blacktriangleright\mathbf{0} = \mathbf{0}$. Then, $\mathbf{0}$ is infinitely expandable because $\forall\phi.\ \phi \circ \blacktriangleright\mathbf{0} = \phi \circ \mathbf{0} = \mathbf{0}$. Thus, by inequation (9) in Definition 1, $\forall\phi.\ \phi + \mathit{fin}\,(\blacktriangleright\mathbf{0}) \precsim \mathit{fin}\,(\blacktriangleright(\phi \circ \blacktriangleright\mathbf{0}))$. Since $\blacktriangleright\mathbf{0} = \blacktriangleright(\phi \circ \blacktriangleright\mathbf{0}) = \mathbf{0}$, we have

$$\forall\phi.\ \phi + \mathit{fin}\,(\mathbf{0}) \precsim \mathit{fin}\,(\mathbf{0})\ . \tag{6}$$

However, property (6) makes it impossible to define $\mathit{fin}\,(\mathbf{0}) = \{\phi_0\}$ for some ϕ_0 that contains no finite trace. If both property (6) and $\mathit{fin}\,(\mathbf{0}) = \{\phi_0\}$ hold, we have $\forall\,\phi.\ \phi + \phi_0 \sqsubseteq \phi_0$, that is, $\forall\,\phi.\ \phi \sqsubseteq \phi_0$, which means that ϕ_0 is the "top" finite effect and is contradictory with the requirement that ϕ_0 contain no finite trace.

Our solution to this problem is to represent infinite effects for termination by natural numbers[6] and to define the later operation to increment the numbers. Intuitively, number n as an infinite effect means that, on the course of the evaluation, the next constructor is nested at most n times. Because infinite computation needs next constructors nested infinitely many times, a program given n as an infinite effect is guaranteed to terminate.

We define subeffecting on infinite effects so that $r_{11} \cdot r_{12}^{\omega} \sqsubseteq r_{21} \cdot r_{22}^{\omega}$ if and only if $r_{11} \sqsubseteq r_{21} \cdot_{\mathrm{r}} r_{22}^{*}$ and $r_{12} \sqsubseteq r_{22}^{+}$. This means that a finite effect that (potentially) happens between successive unfoldings can be split into multiple finite effects, and each of them can be regarded as if it happens between some successive unfoldings. This definition guarantees the productivity while expressive enough to allow, e.g., $\mathbf{ab} \cdot (\mathbf{c}^{\omega}) \sqsubseteq \mathbf{a} \cdot (\mathbf{b} +_{\mathrm{r}} \mathbf{c})^{\omega}$ and $(\mathbf{aa})^{\omega} \sqsubseteq \mathbf{a}^{\omega}$. We also slightly modify the finitization function given in Section 2.3 to show the monotonicity with respect to the subeffecting.

Definition 5 (Temporal Regular Effects). *Given a set Σ, let $Re^{\omega}(\Sigma) \stackrel{\mathrm{def}}{=} \mathcal{P}_{\mathrm{fin}\setminus\emptyset}(Re(\Sigma) \times Re(\Sigma))$ and $\mathcal{I}(\Sigma) \stackrel{\mathrm{def}}{=} Re^{\omega}(\Sigma) \cup \mathbb{N}$. We use the metavariable ρ to denote the elements of $\mathcal{I}(\Sigma)$. We define binary relation $\preceq$ on $\mathcal{I}(\Sigma)$ as follows:*

$$\begin{array}{l}\{(m,n) \in \mathbb{N} \times \mathbb{N} \mid m \le n\} \ \cup\ (\mathbb{N} \times Re^{\omega}(\Sigma)) \cup \\ \{(\rho_1,\rho_2) \in Re^{\omega}(\Sigma) \times Re^{\omega}(\Sigma) \mid \\ \quad \forall\,(r_{11},r_{12}) \in \rho_1.\ \exists\,(r_{21},r_{22}) \in \rho_2.\ r_{11} \sqsubseteq_{\mathrm{r}} r_{21} \cdot_{\mathrm{r}} r_{22}^{*} \ \wedge\ r_{12} \sqsubseteq_{\mathrm{r}} r_{22}^{+}\}\end{array}$$

Binary relation $\equiv_{\mathcal{I}(\Sigma)}$ on $\mathcal{I}(\Sigma)$ is $\{(\rho_1,\rho_2) \mid \rho_1 \preceq \rho_2 \ \wedge\ \rho_2 \preceq \rho_1\}$.

A temporal regular effect over Σ is defined to be an element of $\mathbb{E}^{\mathrm{r}}(\Sigma) \stackrel{\mathrm{def}}{=} Re(\Sigma) \times (\mathcal{I}(\Sigma)/{\equiv_{\mathcal{I}(\Sigma)}})$ where:

- *set $Re(\Sigma)$ is equipped with operation $\cdot_{\mathrm{r}}$ and element $\mathbf{1}_{\mathrm{r}}$ to form a monoid, and with operation $+_{\mathrm{r}}$ to form a join semilattice;*
- *set $\mathcal{I}_n(\Sigma)/{\equiv_{\mathcal{I}_n(\Sigma)}}$ is equipped with operation $\overset{\mathrm{r}}{\sqcup}$, defined as*

$$(\rho_1,\rho_2) \mapsto \begin{cases} \max(\rho_1,\rho_2) & (\text{if } \rho_1,\rho_2 \in \mathbb{N}) \\ \rho_i & (\text{if } \rho_i \in Re^{\omega}(\Sigma) \text{ and } \rho_{3-i} \in \mathbb{N} \text{ for } i \in \{1,2\}) \\ \rho_1 \cup \rho_2 & (\text{if } \rho_1,\rho_2 \in Re^{\omega}(\Sigma)), \end{cases}$$

  to form a join semilattice with number 0 as the least element;
- *later operation $\overset{\mathrm{r}}{\blacktriangleright}$ is defined as*

$$\rho \mapsto \begin{cases} \rho + 1 & (\text{if } \rho \in \mathbb{N}) \\ \{(\mathbf{1}_{\mathrm{r}}, r_1 +_{\mathrm{r}} r_2) \mid (r_1,r_2) \in \rho\} & (\text{if } \rho \in Re^{\omega}(\Sigma)); \end{cases}$$

- *mixed product $\overset{\mathrm{r}}{\circ}$ is defined as*

$$(r,\rho) \mapsto \begin{cases} \rho & (\text{if } \rho \in \mathbb{N}) \\ \{(r \cdot_{\mathrm{r}} r_1, r_2) \mid (r_1,r_2) \in \rho\} & (\text{if } \rho \in Re^{\omega}(\Sigma)); \end{cases}$$

  and

[6] We can generalize to a well-founded set with a least element and a successor function as seen in Section 4.2.

- *finitization $\mathit{fin}_{\mathrm{r}}$ is defined as*

$$\rho \mapsto \begin{cases} \{\varnothing\} & (\mathit{if}\ \rho \in \mathbb{N}) \\ \{r_2^+ \mid (r_1, r_2) \in \rho\} & (\mathit{if}\ \rho \in \mathit{Re}^{\omega}(\Sigma))\,. \end{cases}$$

The relation $\preceq$ indicates that infinite effects may overapproximate the number of unfoldings in a terminating expression (by allowing number n to be a subeffect of number m if $n \leq m$) and they can regard terminating expressions as divergent ones (by allowing number n to be a subeffect of any set in $\mathit{Re}^{\omega}(\Sigma)$). It represents the subeffecting on infinite effects, coinciding with the partial order $\sqsubseteq$ induced by $\overset{\mathrm{r}}{\sqcup}$.

Lemma 3. $\forall \rho_1, \rho_2 \in \mathcal{I}_n(\Sigma).\ \rho_1 \preceq \rho_2 \iff \rho_1 \sqsubseteq \rho_2$.

We first show that regular effects are algebraic temporal effects.

Theorem 4.1. *For any Σ, temporal regular effects over Σ are algebraic temporal effects.*

Soundness of safety and liveness verification with regular effects are immediately derived from the soundness of the effect system. In what follows, suppose $\mathbb{E}^{\mathrm{r}}_n(\Sigma)$ as $\mathbb{E}$ and assume that, for every operation o, $\mathcal{L}(\delta_{\mathrm{e}}(o))$ is a singleton set (that is, every operation raises only a single event). We define the language $\mathcal{L}(\rho)$ of $\rho \in \mathit{Re}^{\omega}(\Sigma)$ as

$$\{w_0 \cdot w_1 \cdot w_2 \cdot \cdots \in \Sigma^{\omega} \mid \exists\, (r_1, r_2) \in \rho.\ w_0 \in \mathcal{L}(r_1)\ \wedge\ \forall\, i > 0.\ w_i \in \mathcal{L}(r_2) \setminus \{\varepsilon\}\}\,.$$

Given $\varpi \in \mathit{Re}(\Sigma)^{\omega}$ such that, for any $i \in \mathbb{N}$, $\mathcal{L}(\varpi(i)) \in \Sigma \cup \{\varepsilon\}$ and $\exists j > i.\ \mathcal{L}(\varpi(j)) \in \Sigma$, we can construct an infinite trace in Σ^{ω} by concatenating the interpretations $\mathcal{L}(\varpi(i))$ of all $\varpi(i)$ in the order. We denote such an infinite trace by ϖ^{c}.

Corollary 1 (Safety Soundness of Regular Effects). *If $\emptyset \vdash M : T \ \&\ e$ and $M \longrightarrow^{*} M' \ \&\ w$ and M' cannot evaluate further, there exist some V and $r = \mathbb{E}[e](\emptyset).F \in \mathit{Re}(\Sigma)$ such that $M' = V$ and $\emptyset \vdash V : T$ and $w \in \mathcal{L}(r)$.*

Corollary 2 (Liveness Soundness of Regular Effects). *If $\emptyset \vdash M : T \ \&\ e$ and $M \Uparrow \varpi$ and ϖ^{c} is well defined, then there exists some $\rho \in \mathit{Re}^{\omega}(\Sigma)$ such that $\mathbb{E}[e](\emptyset).I = \rho$ and $\varpi^{\mathrm{c}} \in \mathcal{L}(\rho)$.*

Note that we can ensure that, when $M \Uparrow \varpi$, the infinite trace ϖ^{c} is always well defined by, e.g., assuming that M raises a "dummy" event each time unfolding is performed.

4.1.2 Examples. This section revisits Examples 1 and 3 in Section 3.4. We suppose that temporal regular effects can be used as syntactic effects directly by providing them as effect constructors $\mathbf{e}$.

Example 1 is a program raising event $\mathbf{a}$ infinitely. The effect e assigned to it has to meet $\mathit{eff}(\mathrm{ev}[\mathbf{a}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ \underline{\blacktriangleright}e\ \underline{\triangleright}\ \epsilon \sqsubseteq e$. For e, consider regular effect $(r_1, \{(r_{21}, r_{22})\})$ for some r_1, r_{21}, and r_{22}. Because

$$\begin{array}{rl} & \mathbb{E}[\mathit{eff}(\mathrm{ev}[\mathbf{a}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ \underline{\blacktriangleright}e\ \underline{\triangleright}\ \epsilon](\emptyset) \\ = & (\mathbf{a}, 0) \triangleright (\mathbf{1}_{\mathrm{r}}, 1) \triangleright (r_1, \{(\mathbf{1}_{\mathrm{r}}, r_{21} +_{\mathrm{r}} r_{22})\}) \triangleright (\mathbf{1}_{\mathrm{r}}, 0) \\ = & (\mathbf{a} \cdot_{\mathrm{r}} r_1, \{(\mathbf{a}, r_{21} +_{\mathrm{r}} r_{22})\})\,, \end{array}$$

a sufficient condition to solve the subeffect constraint on e is the conjunction of $\mathbf{a} \cdot_{\mathrm{r}} r_1 \sqsubseteq r_1$ and $\mathbf{a} \sqsubseteq r_{21}$ and $r_{21} +_{\mathrm{r}} r_{22} \sqsubseteq r_{22}$. We can solve this by letting $r_1 = \varnothing$ and $r_{21} = r_{22} = \mathbf{a}$. Thus, by Corollary 2, it turns out that the programs generate an infinite trace conforming to $\mathbf{a} \cdot \mathbf{a}^{\omega}$.

Example 3 provides a concurrent program that raises event $\mathbf{x}$ and $\mathbf{y}$ alternately. The effect e_x assigned to the program is required to satisfy $\mathit{eff}(\mathrm{ev}[\mathbf{x}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ (\mathit{eff}(\mathrm{ev}[\mathbf{y}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ (\underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ \underline{\blacktriangleright}e_x)) \sqsubseteq e_x$. For e_x, consider regular effect $(r_1, \{(r_{21}, r_{22})\})$ for some r_1, r_{21}, and r_{22}. Because

$$\begin{array}{rl} & \mathbb{E}[\mathit{eff}(\mathrm{ev}[\mathbf{x}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ (\mathit{eff}(\mathrm{ev}[\mathbf{y}])\ \underline{\triangleright}\ \underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ (\underline{\blacktriangleright}\epsilon\ \underline{\triangleright}\ \underline{\blacktriangleright}e_x))](\emptyset) \\ = & (\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y} \cdot_{\mathrm{r}} r_1, \{(\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y}, r_{21} +_{\mathrm{r}} r_{22})\})\,, \end{array}$$

the subeffect constraint on e is reduced to the conjunction of $\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y} \cdot_{\mathrm{r}} r_1 \sqsubseteq r_1$ and $\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y} \sqsubseteq r_{21}$ and $r_{21} +_{\mathrm{r}} r_{22} \sqsubseteq r_{22}$. By solving this, we can find that scheduler sc with processes px and py generates an infinite trace conforming to $\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y} \cdot_{\mathrm{r}} (\mathbf{x} \cdot_{\mathrm{r}} \mathbf{y})^{\omega}$.

To confirm that the constraint solving on regular expressions is possible actually, we have implemented a constraint solver based on grammar-based synthesis [Alur et al. 2015] of regular expressions. Our tool, which is available at https://github.com/hiroshi-unno/coar, solved the constraints of Examples 1–3 within 1 second. The investigation of the scalability to more complex constraints and further improvement of efficiency is left as future work.

4.2 Temporal Fixpoint Effects

This section presents temporal fixpoint effects, or fixpoint effects for short, which are another instance of algebraic temporal effects. In fixpoint effects, finite and infinite effects are described in a first-order fixpoint logic over traces. We assume an alphabet Σ in this section.

4.2.1 Definition and Properties. A part of the syntax of the fixpoint logic is given as follows:

$$\begin{array}{llll} \textbf{Sorts} & s ::= B \mid \Sigma^* \mid \Sigma^\omega & \textbf{Predicates} & P ::= \mathcal{X} \mid \mu\mathcal{X}(\overline{x:s}).p \mid \nu\mathcal{X}(\overline{x:s}).p \mid (=) \mid \cdots \\ \textbf{Terms} & t ::= x \mid c \mid \mathbf{a} \mid F(\bar{t}) & \textbf{Formulas} & p ::= \bot \mid P(\bar{t}) \mid \neg p \mid p_1 \vee p_2 \mid \exists\, x : s.\, p \end{array}$$

where $\mathcal{X}$ is a predicate variable and F is some term-level function (such as $(\cdot)$ to concatenate finite traces and $(\circ)$ to concatenate a finite and infinite trace). The sort Σ^* and Σ^ω represent the sets of finite and infinite traces over Σ, respectively. The μ and ν operators added to predicates are the least and greatest fixpoint operators, respectively. They are useful to state traces in the logic. For example, the sets of finite and infinite traces consisting only of event $\mathbf{a}$ are expressed using the μ and ν operators, respectively, as follows: $\mu\mathcal{X}(x:\Sigma^*).(x = \varepsilon) \vee (\exists\, y:\Sigma^*.\, x = \mathbf{a} \cdot y \wedge \mathcal{X}(y))$ and $\nu\mathcal{X}(x:\Sigma^\omega).\exists\, y:\Sigma^\omega.\, x = \mathbf{a} \circ y \wedge \mathcal{X}(y)$. Note that other standard logical connectives, such as conjunction and implication, can be expressed in the logic. We write $\theta \models p$ if the formula p is valid under a model θ, which is a mapping from variables occurring in p to their denotations. We omit θ if it is the empty mapping. Readers interested in a more detail are referred to the supplementary material or the prior work [Kobayashi et al. 2019; Sekiyama and Unno 2023; Unno et al. 2023]. We also write $\lambda x : s.\, p$ for predicate $\mu\mathcal{X}(x : s).p$ when $\mathcal{X}$ does not occur in p.

Fixpoint effects are defined in a way similar to regular effects, but there are two main differences. First, finite and infinite effects of fixpoint effects are defined using logical predicates of the form $\lambda x : \Sigma^*.\, p$, instead of regular expressions. Second, while regular effects use natural numbers to represent termination with infinite effects, fixpoint effects allow an arbitrary set S with necessary operations and relations to ensure termination.

Definition 6 (Temporal Fixpoint Effects). *Let*

- ***Pred*** *be the set of closed predicates of the form* $\lambda x : \Sigma^*.\, p$*;*
- $\sqsubseteq_{\Sigma^*}$ *be a partial order on* ***Pred****, which contains* $((\lambda x : \Sigma^*.\, p_1), (\lambda x : \Sigma^*.\, p_2))$ *if* $\models \forall\, x : \Sigma^*.\, (p_1 \Rightarrow p_2)$*;*
- S *be a set equipped with a partial, well-founded order* $\prec_{\mathrm{wf}}$*, the least element* $\mathbf{0}_{\mathrm{wf}}$*, a join operation* $\sqcup_{\mathrm{wf}}$*, and a monotonic function succ such that, for any* $\rho \in S$*,* $\rho \prec_{\mathrm{wf}} succ(\rho)$*; and*
- $\mathcal{I}_{\mathrm{fix}}$ *be the set* $\mathcal{P}_{\mathrm{fin}\setminus\emptyset}(\textbf{Pred} \times \textbf{Pred}) \cup S$*.*

We define a binary relation $\preceq$ *on* $\mathcal{I}_{\mathrm{fix}}$ *as follows:*

$$\begin{aligned} &\{(\rho_1, \rho_2) \in S^2 \mid \rho_1 \prec_{\mathrm{wf}} \rho_2\} \cup (S \times (\mathcal{I}_{\mathrm{fix}} \setminus S)) \\ \cup\ &\{(\rho_1, \rho_2) \in (\mathcal{I}_{\mathrm{fix}} \setminus S)^2 \mid \forall\, (P_{11}, P_{12}) \in \rho_1.\ \exists\, (P_{21}, P_{22}) \in \rho_2.\ P_{11} \sqsubseteq_{\Sigma^*} P_{21} \ \wedge\ P_{12} \sqsubseteq_{\Sigma^*} P_{22}\}\ . \end{aligned}$$

A binary relation $\equiv_S^{\mathrm{fix}}$ *on* $\mathcal{I}_{\mathrm{fix}}$ *is defined to be* $\{(\rho_1, \rho_2) \mid \rho_1 \preceq \rho_2 \ \wedge\ \rho_2 \preceq \rho_1\}$*.*

Temporal fixpoint effects are elements in set $\mathbb{E}_{\mathrm{fix}}$ *defined to be* $\textbf{Pred} \times (\mathcal{I}_{\mathrm{fix}}/{\equiv_S^{\mathrm{fix}}})$ *where:*

- *set* ***Pred*** *is equipped with operations* $\cdot_{\mathsf{fix}}$, $+_{\mathsf{fix}}$, *and element* $\mathbf{1}_{\mathsf{fix}}$ *defined as*

$$\begin{aligned}(\lambda x_1 : \Sigma^*.\, p_1) \cdot_{\mathsf{fix}} (\lambda x_2 : \Sigma^*.\, p_2) &\stackrel{\text{def}}{=} \lambda x : \Sigma^*.\, \exists\, x_1 : \Sigma^*.\, \exists\, x_2 : \Sigma^*.\, (x = x_1 \cdot x_2) \land p_1 \land p_2 \\ (\lambda x : \Sigma^*.\, p_1) +_{\mathsf{fix}} (\lambda x : \Sigma^*.\, p_2) &\stackrel{\text{def}}{=} \lambda x : \Sigma^*.\, p_1 \lor p_2 \\ \mathbf{1}_{\mathsf{fix}} &\stackrel{\text{def}}{=} \lambda x : \Sigma^*.\, x = \varepsilon\end{aligned}$$

(where x, x_1, *and* x_2 *are distinct from each other) to form a monoid* $(\boldsymbol{Pred}, \cdot_{\mathsf{fix}}, \mathbf{1}_{\mathsf{fix}})$ *and a join semilattice* $(\boldsymbol{Pred}, +_{\mathsf{fix}})$*;*

- *set* $\mathcal{I}_{\mathsf{fix}}/\equiv_S^{\mathsf{fix}}$ *is equipped with operation* $\sqcup_{\mathsf{fix}}$, *which is defined as*

$$(\rho_1, \rho_2) \mapsto \begin{cases} \rho_1 \sqcup_{\mathsf{wf}} \rho_2 & (\textit{if } \rho_1, \rho_2 \in S) \\ \rho_i & (\textit{if } \rho_i \in \mathcal{I}_{\mathsf{fix}} \setminus S \textit{ and } \rho_{3-i} \in S \textit{ for } i \in \{1,2\}) \\ \rho_1 \cup \rho_2 & (\textit{if } \rho_1, \rho_2 \in \mathcal{I}_{\mathsf{fix}} \setminus S), \end{cases}$$

to form a join semilattice with $\mathbf{0}_{\mathsf{wf}}$ *as the least element;*

- *later operation* $\blacktriangleright_{\mathsf{fix}}$ *is defined as*

$$\rho \mapsto \begin{cases} \mathit{succ}(\rho) & (\textit{if } \rho \in S) \\ \{(\mathbf{1}_{\mathsf{fix}}, P_1 +_{\mathsf{fix}} P_2) \mid (P_1, P_2) \in \rho\} & (\textit{if } \rho \in \mathcal{I}_{\mathsf{fix}} \setminus S); \end{cases}$$

- *mixed-product operation* $\circ_{\mathsf{fix}}$ *is defined as*

$$(P, \rho) \mapsto \begin{cases} \rho & (\textit{if } \rho \in S) \\ \{(P \cdot_{\mathsf{fix}} P_1, P_2) \mid (P_1, P_2) \in \rho\} & (\textit{if } \rho \in \mathcal{I}_{\mathsf{fix}} \setminus S); \end{cases}$$

and

- *finitization* $\mathit{fin}_{\mathsf{fix}}$ *is defined as*

$$\rho \mapsto \begin{cases} \{\, \lambda x : \Sigma^*.\, \bot \,\} & (\textit{if } \rho \in S) \\ \{P_2 \mid (P_1, P_2) \in \rho\} & (\textit{if } \rho \in \mathcal{I}_{\mathsf{fix}} \setminus S). \end{cases}$$

This definition can be read as follows. Predicate $P_1 \cdot_{\mathsf{fix}} P_2$ contains finite traces that can be split into two traces accepted by P_1 and P_2, respectively, and predicate $P_1 +_{\mathsf{fix}} P_2$ contains finite traces accepted by P_1 or P_2. The predicate $\mathbf{1}_{\mathsf{fix}}$ expresses the singleton set of the empty trace. The join operation on infinite effects and the later, mixed-product, and finitization operations are defined similarly to those in regular effects.

Now, we state the soundness of fixpoint effects after defining the languages of fixpoint effects.

Definition 7 (Languages of Fixpoint Effects). *Given a predicate* $\lambda x : \Sigma^*.\, p \in \boldsymbol{Pred}$, *the language* $\mathcal{L}(\lambda x : \Sigma^*.\, p) \subseteq \Sigma^*$ *is defined by* $\{w \mid \{x \mapsto w\} \models p\}$. *Given an infinite fixpoint effect* $\rho \in \mathcal{I}_{\mathsf{fix}} \setminus S$, *the language* $\mathcal{L}(\rho) \subseteq \Sigma^\omega$ *is defined as:*

$$\mathcal{L}(\rho) \stackrel{\text{def}}{=} \{w_0 \cdot w_1 \cdot w_2 \cdot \cdots \mid \exists\, (P_1, P_2) \in \rho.\ w_0 \in \mathcal{L}(P_1) \ \land\ \forall\, i > 0.\ w_i \in \mathcal{L}(P_2) \setminus \{\varepsilon\}\}\,.$$

Given an event $\mathbf{a}$, we write $P_{\mathbf{a}}$ for the predicate $\lambda x : \Sigma^*.\, x = \mathbf{a}$. As in Section 4.1.1, we assume that, for every operation o, $\delta_{\mathrm{e}}(o)$ is a predicate $P_{\mathbf{a}}$ for some event $\mathbf{a}$. Moreover, for simplicity, we identify a finite (resp. infinite) sequence of predicates $P_{\mathbf{a}_1}, P_{\mathbf{a}_2}, \cdots$ with a finite trace in Σ^* (resp. an infinite trace in Σ^ω). This allows us to write $M \longrightarrow^* M' \mathbin{\&} w$ and $M \Uparrow \varpi$ when M reaches M' with the finite trace $w \in \Sigma^*$ and M diverges with the infinite trace $\varpi \in \Sigma^\omega$, respectively.

Theorem 4.2. *Temporal fixpoint effects are algebraic temporal effects.*

Corollary 3 (Safety Soundness of Fixpoint Effects). *If* $\emptyset \vdash M : T \mathbin{\&} e$ *and* $M \longrightarrow^* M' \mathbin{\&} w$ *and* M' *cannot evaluate further, there exist some* V *and* $P = \mathbb{E}[e](\emptyset).F \in \boldsymbol{Pred}$ *such that* $M' = V$ *and* $\emptyset \vdash V : T$ *and* $w \in \mathcal{L}(P)$.

Corollary 4 (Liveness Soundness of Fixpoint Effects). *If* $\emptyset \vdash M : T \ \&\ e$ *and* $M \Uparrow \varpi$*, then there exists some* $\rho \in \mathcal{I}_{\mathsf{fix}} \setminus S$ *such that* $\mathbb{E}[e](\emptyset).I = \rho$ *and* $\varpi \in \mathcal{L}(\rho)$*.*

As in regular effects, this theorem means that, if an infinite fixpoint effect $\{(P_{11}, P_{12}), \cdots, (P_{n1}, P_{n2})\}$ is assigned to a divergent program, the infinite trace yielded by the program is recognized by the predicate $\lambda x : \Sigma^\omega.\ \exists\, y : \Sigma^*.\ \exists\, z : \Sigma^\omega.\ (x = y \circ z) \land P_{i1}(y) \land P(z)$ where

$$P \overset{\mathrm{def}}{=} \nu\mathcal{X}(z : \Sigma^\omega).\exists\, z_1 : \Sigma^*.\ \exists\, z_2 : \Sigma^\omega.\ (z = z_1 \circ z_2) \land P_{i2}(z_1) \land \mathcal{X}(z_2)$$

for some i.

4.2.2 Example. As an example, we consider a variant of Example 4. Recall that class cInt has method `print` and integer field x. We now suppose that `print` of cInt is changed to

$$\lambda x.\lambda \mathsf{this}.\lambda z.\mathsf{if}\ x = 0\ \mathsf{then}\ ()\ \mathsf{else}\ (\mathsf{ev}[\mathbf{I}]();\mathsf{this.print}\ (x-1);\mathsf{ev}[\mathbf{J}]())\ ,$$

which raises events **I** and **J** before and after, respectively, the recursive call. We assume that any finite and infinite fixpoint effect can be described as a syntactic effect. A call to this method terminates at a finite trace $\mathbf{I}^x \cdot \mathbf{J}^x$ if x is nonnegative and, otherwise, diverges with the infinite trace $\mathbf{I}^\omega$. Note that no regular expression can specify the set of such finite traces. Then, we can assign to cInt type $\mathsf{int} \longrightarrow T_e$ with effect $e \overset{\mathrm{def}}{=} (\lambda y : \Sigma^*.\ P(y), \{(P_{\mathbf{I}}, P_{\mathbf{I}})\})$ where

$$P \overset{\mathrm{def}}{=} \mu\mathcal{X}(z : \Sigma^*).(y = \varepsilon) \lor \exists\, z' : \Sigma^*.\ ((z = \mathbf{I} \cdot z' \cdot \mathbf{J}) \land \mathcal{X}(z'))$$

(recall that type T_e is the shorthand introduced for Example 4 in Section 3.4). This effect addresses both of the case that the call to `print` terminates (by $\lambda y : \Sigma^*.\ P(y)$) and the one that the call diverges (by $\{(P_{\mathbf{I}}, P_{\mathbf{I}})\}$). The program in Example 4 creates two objects of cInt, sequentializes the calls to `print` of them, and then invokes `print` of an object of cOmega. The effect $e_{\mathbf{O}}$ of cOmega's `print` has to meet $\mathit{eff}(\mathsf{ev}[\mathbf{O}]) \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright}\epsilon \mathrel{\underline{\triangleright}} \underline{\blacktriangleright} e_{\mathbf{O}} \sqsubseteq e_{\mathbf{O}}$. For $e_{\mathbf{O}}$, we can assign effect $(\lambda x : \Sigma^*.\ \bot, \{(P_{\mathbf{O}}, P_{\mathbf{O}})\})$. Then, the effect assigned to the entire program is

$$(\lambda x : \Sigma^*.\ \bot, \{(P_{\mathbf{I}}, P_{\mathbf{I}}), (P \cdot_{\mathsf{fix}} P_{\mathbf{I}}, P_{\mathbf{I}}), (P \cdot_{\mathsf{fix}} P \cdot_{\mathsf{fix}} P_{\mathbf{O}}, P_{\mathbf{O}})\})\ ,$$

which means that the program diverges with one of the the infinite traces $\mathbf{I}^\omega$, $\mathbf{I}^n \cdot \mathbf{J}^n \cdot \mathbf{I}^\omega$, or $\mathbf{I}^n \cdot \mathbf{J}^n \cdot \mathbf{I}^m \cdot \mathbf{J}^m \cdot \mathbf{O}^\omega$ for some $n, m \geq 0$.

5 Discussion

This section discusses the limitations and open problems of the present work as well as certain directions of future research.

5.1 Value Dependency

A limitation in the current form of our temporal effect system is the lack of support for *value dependency*, a crucial feature to enable precise type-based verification. For example, consider a recursive function `let rec f n = ev[a](); if n <= 0 then () else f(n-1)`. A call to this function terminates after $max(0, n)$ recursive calls. Our temporal effects are unaware of program values, so they cannot precisely express this termination condition. More specifically, using temporal regular effects, an effect given to the function `f` can only specify (a^*, a^ω), which cannot guarantee the termination of the call to `f`. This problem could be resolved if we extend algebraic temporal effects to be value-dependent as Nanjo et al. [2018] and Sekiyama and Unno [2023] did. Bizjak et al. [2016] extended guarded type theory involving later types to dependent typing using *delayed substitutions*. We could adapt this approach to algebraic temporal effects, but it is left for future.

5.2 Expressivity of Instances of Algebraic Temporal Effects

Our framework reduces the reasoning about the infinite behavior of a program to the reasoning about the finite behavior to be repeated infinitely. This reduction is carried out by the effect system with the help of later types, and instances of algebraic temporal effects can focus on finding upper bounds of the finite effects between successive unfoldings (determined by the later types). This is the reason why the infinite effects of the instances presented in Section 4 only involve representations for specifying finite traces: the infinite effects of regular effects only involve regular expressions, not ω-regular expressions themselves; and those of fixpoint effects only involve fixpoint logical formulas over Σ^*, not ones over Σ^ω. However, these representations of infinite effects are, at least superficially, different from those in the previous work on temporal effects [Hofmann and Chen 2014; Nanjo et al. 2018; Sekiyama and Unno 2023], where infinite effects are represented by ω-regular expressions or logical formulas over Σ^ω. Our effect system with the presented instances can reason about the infinite behavior, as stated in the liveness soundness of each instance (Corollaries 2 and 4), but it is worth exploring whether it is possible to directly employ ω-regular expressions or logical formulas over Σ^ω as representations of infinite effects in the presence of recursive types, possibly by extending algebraic temporal effects or the effect system, and, if it is possible, how expressive they are compared to regular effects or fixpoint effects.

5.3 Implementation

The present work sheds light on the nontrivial theoretical aspects of algebraic temporal effects in the presence of recursive types, but we also plan to implement a verifier based on the proposed effect system with a specific instance of algebraic temporal effects, such as regular effects and fixpoint effects. Here, we discuss potential challenges in the implementation. In our verifier, the verification process will consist of three phases: type inference, constraint generation, and constraint solving.

Phase 1: Type Inference. The first phase is type inference, inferring the value type of each expression, while the remaining phases are for inferring and checking effects. If we can admit the use of term constructors for annotations (fold, unfold, next, prev, $\circledast$, and effect abstractions and applications), we expect the type inference to be decidable. Otherwise, we would need to design a surface language that hides the use of such term constructors. This might be possible by focusing on structured programming features, such as algebraic data types, objects, and concurrency. We believe that the constructs fold and unfold for recursive types can be completely hidden, but whether the constructs for later types (next, prev, and $\circledast$) can be hidden requires an exploration. We may also need to restrict effect polymorphism to be in a prenex form to make the type inference decidable.

Phase 2: Constraint Generation. This phase generates constraints on effect variables in terms of subeffecting, which are sufficient conditions for programs to be well typed. We could implement this phase according to the typing rules of the proposed effect system. The user often writes effect annotations to specify the expected behavior of some expression or to give a hint to help constraint solving (phase 3). Such information would also be collected during this phase.

Phase 3: Constraint Solving. Finally, the verification process checks that the generated constraints are met by finding solutions, i.e., concrete effects that satisfy the constraints. Because algebraic temporal effects are pairs of finite and infinite effects, we would need solvers for finite and infinite effects, respectively, depending on chosen instances of algebraic temporal effects.

For regular effects, we have developed a solver for finite effect constraints on variables that represent unknown regular expressions, as mentioned in the last paragraph of Section 4.1. For infinite effects, which are finite sets of pairs of regular expressions, we could apply the solver for finite effects if the cardinality of the finite set that represents an infinite effect is fixed. The development of a solver that can lift the restriction of fixing the cardinality is under exploration.

For fixpoint effects, finite effects are fixpoint logical formulas. A solver for fixpoint logical formulas could be implemented based on the techniques proposed by the previous work [Nanjo et al. 2018; Unno et al. 2023]. We believe that a solver for infinite effects could be implemented in a similar strategy as the solver for infinite effects of regular effects.

6 Related Work

6.1 Temporal Verification of Higher-Order Programs

Many approaches to temporal safety verification have been proposed thus far. Typestate [Strom and Yemini 1986] and resource usage analysis [Igarashi and Kobayashi 2002] address temporal safety properties for each resource, such as individual files and memory cells, in the presence of recursion. Skalka and Smith [2004] introduced an effect system to reason about finite traces generated by programs. Gordon [2017] generalized it to an effect system with an abstract algebraic formulation of effects, called *effect quantales*. Effect quantales expose algebraic structures of effects for temporal safety verification, similar to our formulation of finite effects in that they form a monoid and join semilattice with the equational laws to distribute the (finite) effect composition over joins (i.e., $\phi \cdot (\phi_1 + \phi_2) = (\phi \cdot \phi_1) + (\phi \cdot \phi_2)$ and $(\phi_1 + \phi_2) \cdot \phi = (\phi_1 \cdot \phi) + (\phi_2 \cdot \phi)$ in our notation). Algebraic temporal effects only require inequations for distribution of the composition $(\cdot)$ (inequations (2) and (4) in Definition 1) instead of equations because they are sufficient to prove soundness, at least in our case. Gordon [2021] demonstrated the experssivity of effect quantales by encoding a variety of effect-based verification frameworks. Note that the system of Skalka and Smith and that of Gordon only target higher-order programs with recursive functions. Skalka [2008] extended the prior system [Skalka and Smith 2004] to verify temporal safety properties of object-oriented programs.

Kobayashi and Ong [2009] introduced an intersection type system that enables verification of branching-time temporal properties of higher-order programs with recursive functions and finite data domains. Tsukada and Ong [2014] extended Kobayashi and Ong's system to infinite intersection types, which can express recursive types by expanding them infinitely. Therefore, our effect system could be reduced to their system, and then the soundness of our effect system could be derived from that of their system under some restriction such as finite data domains. In contrast to Tsukada and Ong's system, our effect system does not need the restriction on data domains and enables program reasoning only with types in finite representations, which we believe are desirable for automated verification. Kobayashi and Igarashi [2013] proposed a recursive intersection type system for verifying *safety* properties of programs with recursive types.

Several works have proposed effect systems for verifying linear-time temporal properties of programs with recursive functions. The effect system of Hofmann and Chen [2014] can verify that the traces generated by programs conform to given ω-regular expressions. They target first-order programs and an extension to higher-order programs is sketched, but completing the extension is addressed later [Hofmann and Ledent 2017; Salvati and Walukiewicz 2015]. Koskinen and Terauchi [2014] introduced temporal effects, which are pairs of a set of finite traces and a set of infinite traces. Their effect system can address higher-order programs, but the reasoning about recursive functions—which is the only source of divergence in their target language—rests on an oracle. Nanjo et al. [2018] improve this situation by inferring traces of recursive functions using least and greatest fixpoint predicates instead of an oracle. Temporal effects in the prior work take the form of pairs of trace sets or pairs of logical formulas on traces. Instead of adopting such a concrete form, we express temporal effects in an abstract form with an algebraic characterization.

There also exist the works on temporal verification in the presence of control operators [Gordon 2020; Sekiyama and Unno 2023; Song et al. 2022]. The effect systems of Gordon [2020] and Sekiyama and Unno [2023] address recursive functions but not recursive types. Song et al. [2022] do not deal with recursive functions explicitly, but their system allows writing recursive programs using effect-level recursion. Song et al.'s system supports only temporal safety properties, and it is nontrivial to extend it to verification of liveness properties of programs with general recursive types.

As far as we know, our work is the first to enable temporal *liveness* verification of programs that can manipulate *general* recursive data structures. The existing type systems for liveness verification of recursive data structures [Unno et al. 2018; Vazou et al. 2014; Xi 2002] are unsound without restricting self-referential type variables of recursive data types to positive positions (otherwise, recursive types can cause infinite computation). There is no such a limitation in this work.

6.2 Modality for Recursion

Our effect system exploits the later modality introduced by Nakano [2000] to structure unfolding operations of recursive types. The later modality has several other applications such as reactive programming [Krishnaswami and Benton 2011; Krishnaswami et al. 2012] and guarded recursion [Atkey and McBride 2013; Birkedal and Møgelberg 2013; Clouston et al. 2015]. These prior works utilize the later modality to ensure the productivity of values or the totality of programs with recursive types—that is, they are interested in at which time values are produced. By contrast, this work is interested in at which time unfolding and effects occur. Thus, we allow simply eliminating the later modality attached to first-order types (the modality elimination from higher-order types is not allowed because higher-order values may involve unfolding or effects). While some of the prior works introduced alternative, more sophisticated approaches to eliminating the modality [Atkey and McBride 2013; Clouston et al. 2015], we did not employ them to make the effect system as simple as possible. Jaber and Riba [2021] used guarded recursive types for verifying infinite objects like streams and infinite trees. By contrast, our proposal focuses on the verification of infinite executions exhibited by control structures, like higher-order recursive functions, objects, and concurrency, in a unified, abstract framework. It is an interesting direction to explore how their refinement specifications can be incorporated into our framework.

Iris [Jung et al. 2018], a framework for concurrent separation logic, also relies on a later modality. The semantics of Iris is indexed by the steps of computation to avoid the circularity incurred in introducing higher-order ghost state, which is essential to encode advanced features of Iris. The later modality of Iris is a mechanism to allow users to access to propositions that hold at the next step in an abstract manner. A key difference from Iris is that the logic of Iris focuses on safety properties such as partial correctness, and liveness properties are not focused. Tassarotti et al. [2017] extend Iris to address concurrent termination-preserving refinement, which is a liveness property, but the semantics of their extension is indexed by computation steps, which imposes a restriction on programs to be verified such that their possible behavior is bounded. Spies et al. [2021] lift this restriction by extending the semantics of Iris to be indexed by ordinals, demonstrating that the extension can verify termination and termination-preserving refinement. A key difference from these works is that their liveness verification is based on simulation. Therefore, expressions to be verified need to be able to be evaluated, but this prevents the verification of open terms. By contrast, our approach estimates the behavior of expressions as effects, which work even for open terms if appropriate assumptions on variables are provided. Combining these two different approaches is an interesting direction for further research.

7 Conclusion

We proposed algebraic temporal effects, a novel abstract form of temporal effects with a modality, for temporal verification of programs with recursive types. The reasoning with our type system is guaranteed to be sound and we demonstrate that our type system can reason about higher-order programs with recursive types extensively. For future work, besides addressing the limitations and open problems described in Section 5, we are also interested in adapting our technique with the modality to other features involving circularity such as general mutable references and to the calculi studied in the previous work on temporal verification [Hofmann and Chen 2014; Koskinen and Terauchi 2014; Nanjo et al. 2018; Sekiyama and Unno 2023]. Another future direction is to support the verification of temporal properties per resource as in resource usage analysis [Igarashi and Kobayashi 2002]; the current system focuses only on global traces.

Acknowledgments

We thank the anonymous reviewers for their fruitful comments on the submission, which help us improve the presentation of the paper significantly. This research was partially supported by JSPS KAKENHI Grant Numbers JP22K17875, JP24H00699, JP20H00582, JP20H04162, JP19H04084, JP20H05703, JP22H03564, and JP23K24826 as well as JST CREST Grant Number JPMJCR21M3.

References

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1015–1022. https://doi.org/10.1145/1639950.1640073

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 109–122. https://doi.org/10.1145/1190216.1190235

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 197–208. https://doi.org/10.1145/2500365.2500597

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 8:1–8:45. https://doi.org/10.1145/1890028.1890031

Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*. IEEE Computer Society, 213–222. https://doi.org/10.1109/LICS.2013.27

Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9634)*, Bart Jacobs and Christof Löding (Eds.). Springer, 20–35. https://doi.org/10.1007/978-3-662-49630-5_2

Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015 (Lecture Notes in Computer Science, Vol. 9034)*, Andrew M. Pitts (Ed.). Springer, 407–421. https://doi.org/10.1007/978-3-662-46678-0_26

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming, ECOOP 2017 (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.13

Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.23

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79. https://doi.org/10.1145/3450272

Martin Hofmann and Wei Chen. 2014. Abstract interpretation from Büchi automata. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 51:1–51:10. https://doi.org/10.1145/2603088.2603127

Martin Hofmann and Jérémy Ledent. 2017. A cartesian-closed category for higher-order model checking. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005120

Atsushi Igarashi and Naoki Kobayashi. 2002. Resource usage analysis. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 331–342. https://doi.org/10.1145/503272.503303

Guilhem Jaber and Colin Riba. 2021. Temporal Refinements for Guarded Recursive Types. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 548–578. https://doi.org/10.1007/978-3-030-72019-3_20

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Naoki Kobayashi and Atsushi Igarashi. 2013. Model-Checking Higher-Order Programs with Recursive Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013 as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013 (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 431–450. https://doi.org/10.1007/978-3-642-37036-6_24

Naoki Kobayashi, Takeshi Nishikawa, Atsushi Igarashi, and Hiroshi Unno. 2019. Temporal Verification of Programs via First-Order Fixpoint Logic. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 413–436. https://doi.org/10.1007/978-3-030-32304-2_20

Naoki Kobayashi and C.-H. Luke Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009*. IEEE Computer Society, 179–188. https://doi.org/10.1109/LICS.2009.29

Eric Koskinen and Tachio Terauchi. 2014. Local Temporal Reasoning. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS '14)*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 59:1–59:10. https://doi.org/10.1145/2603088.2603138

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011*. IEEE Computer Society, 257–266. https://doi.org/10.1109/LICS.2011.38

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, John Field and Michael Hicks (Eds.). ACM, 45–58. https://doi.org/10.1145/2103656.2103665

John C. Mitchell. 1990. Toward a Typed Foundation for Method Specialization and Inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, Frances E. Allen (Ed.). ACM Press, 109–124. https://doi.org/10.1145/96709.96719

James H. Morris. 1969. *Lambda-calculus models of programming languages*. Ph. D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/64850

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. https://doi.org/10.1109/LICS.2000.855774

Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768. https://doi.org/10.1145/3209108.3209204

Susan S. Owicki and Leslie Lamport. 1982. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 455–495. https://doi.org/10.1145/357172.357178

Dominique Perrin and Jean-Eric Pin. 2004. *Infinite words - automata, semigroups, logic and games*. Pure and applied mathematics series, Vol. 141. Elsevier Morgan Kaufmann.

Sylvain Salvati and Igor Walukiewicz. 2015. A Model for Behavioural Properties of Higher-order Programs. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany (LIPIcs, Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 229–243. https://doi.org/10.4230/LIPIcs.CSL.2015.229

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (2023), 32 pages. https://doi.org/10.1145/3571264

Ben A. Sijtsma. 1989. On the Productivity of Recursive List Definitions. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 633–649. https://doi.org/10.1145/69558.69563

Christian Skalka. 2008. Types and trace effects for object orientation. *High. Order Symb. Comput.* 21, 3 (2008), 239–282. https://doi.org/10.1007/s10990-008-9032-6

Christian Skalka and Scott F. Smith. 2004. History Effects and Verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004 (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 107–128. https://doi.org/10.1007/978-3-540-30477-7_8

Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022 (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 88–109. https://doi.org/10.1007/978-3-031-21037-2_5

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 80–95. https://doi.org/10.1145/3453483.3454031

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Takeshi Tsukada and C.-H. Luke Ong. 2014. Compositional higher-order model checking via ω-regular games over Böhm trees. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 78:1–78:10. https://doi.org/10.1145/2603088.2603133

Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 12:1–12:29. https://doi.org/10.1145/3158100

Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 2111–2140. https://doi.org/10.1145/3571265

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *High. Order Symb. Comput.* 15, 1 (2002), 91–131. https://doi.org/10.1023/A:1019916231463

Received 2024-07-11; accepted 2024-11-07