Two Guarded Recursive Powerdomains for Applicative Simulation

Rasmus Ejlers Møgelberg*
Department of Computer Science
IT University of Copenhagen
Denmark
mogel@itu.dk

Andrea Vezzosi*
Department of Computer Science
IT University of Copenhagen
Denmark
avez@itu.dk

Clocked Cubical Type Theory is a new type theory combining the power of guarded recursion with univalence and higher inductive types (HITs). This type theory can be used as a metalanguage for *synthetic guarded domain theory* in which one can solve guarded recursive type equations, also with negative variable occurrences, and use these to construct models for reasoning about programming languages. Combining this with HITs allows for the use of type constructors familiar from set-theory based approaches to semantics, such as quotients and finite powersets in these models.

In this paper we show how to reason about the combination of finite non-determinism and recursion in this type theory. Unlike traditional domain theory which takes an ordering of programs as primitive, synthetic guarded domain theory takes the notion of computation step as primitive in the form of a modal operator. We use this extra intensional information to define two guarded recursive (finite) powerdomain constructions differing in the way non-determinism interacts with the computation steps. As an example application of these we show how to prove applicative similarity a congruence in the cases of may- and must-convergence for the untyped lambda calculus with finite non-determinism. Such results are usually proved using operational reasoning and Howe's method. Here we use an adaptation of a denotational method developed by Pitts in the context of domain theory.

1 Introduction

Over the past 20 years, step-indexing techniques [4] have become one of the most used tools for constructing operational models of programming languages with combinations of advanced features such as recursive types, polymorphism, concurrency and non-determinism. Often such models are beyond the scope of traditional domain theoretic techniques, and also have the additional benefit of being more elementary. Guarded recursion is an abstract form of step-indexing, in which the explicit steps are replaced by abstract computation steps in the form of a delay modality $\triangleright$. This relieves the user of the bookkeeping involved in explicit step-indexing and reveals the underlying structure that makes these models work in the form of an introduction $X \to \triangleright X$, a guarded fixed point combinator of type $(\triangleright X \to X) \to X$ and solutions to guarded recursive domain equations. In its multiclocked version, where the delay modality $\triangleright^\kappa$ is indexed by a clock κ and clocks can be universally quantified, guarded recursion can moreover be used to encode coinductive types in type theory, allowing productivity requirements on these to be encoded in types [6].

Clocked Cubical Type Theory (CCTT) [24] is a type theory combining multiclocked guarded recursion with features from cubical type theory, in particular univalence and higher inductive types (HITs). The latter are a form of inductive types defined not only by constructors, but also by equations. HITs

*This work was supported by a research grant (13156) from VILLUM FONDEN.

Ana Sokolova (Eds.): MFPS 2021
EPTCS 351, 2021, pp. 200–217, doi:10.4204/EPTCS.351.13

have been used to construct topological spaces such as the circle and the torus in type theory, but can also be used for free structures, such as the free group on a set. In computer science, free structures can be used to form the monads generated by algebraic theories. For example, the finite powerset monad, often used to model finite non-determinism, can be generated by a binary union operation plus axioms of associativity, commutativity, and idempotency, and can therefore be naturally represented as a HIT [8, 19].

Combining HITs with guarded recursion provides a powerful metatheory in which one can reason about programming languages and programs. This paper presents a worked example of this. We study the untyped lambda calculus with finite non-determinism, and show how to construct a model of this in CCTT using a guarded recursive type. The model construction takes as parameter a monad T with a union operation $\cup : TX \times TX \to TX$ for modelling non-determinism, as well as a step operation $\mathsf{step}_T : \triangleright^\kappa(TX) \to TX$, which in combination with the fixed point operator allows us to model recursion.

We present two instantiations for T, corresponding to two different notions of observation on non-deterministic programs. The first describes a notion of non-deterministic computation where all possible branches of a computation are executed in parallel and we can observe all possible values that occur along the way, even if there are diverging branches of the computation. This monad corresponds to may-convergence and can be characterised as being generated by the operations $\cup$ and step_T with no equations between them.

The second instantiation corresponds to a notion of computation where all branches are evaluated in parallel, but partial results are only available when all branches have terminated. This is simply the composition $\mathsf{L}^\kappa\mathsf{P}_\mathsf{f}$ of the free monad L^κ generated by the step operation and the finite powerset monad P_f. It turns out that this composition is not itself a monad, but does have a sequencing operation sufficient for the purposes of this paper. We claim that this composition corresponds to must-convergence.

As an example application of this model, and to substantiate the claim that these constructors correspond to may- and must-convergence, respectively, we apply the model to a classical problem in lambda calculus, namely that of proving that applicative similarity is a congruence. This was first proved by Abramsky [1] for the lazy lambda calculus (without non-determinism) using domain theory and Stone duality, and this method has since been extended to calculi with non-determinism [31]. Here we use a different method due to Pitts [34] who used a domain theoretic model and a relation between syntax and semantics. We extend this proof to non-determinism for both may- and must-equivalence, and adapt it to guarded recursion.

The two instantiations of T mentioned above are what we consider as examples of guarded recursive powerdomains. In classical domain theory [2] a powerdomain is a domain theoretic correspondent to the powerset construction in set theory. A number of different powerdomains exist, each characterising a different notion of observation. One way of characterising the difference between these is in terms of enriched algebraic theories which allow them to be classified in terms of inequations such as $x \leq x \cup y$. In the case of guarded recursion the difference between the two guarded powerdomains studied here can be expressed in terms of equalities describing the interaction between $\cup$ and step_T. This paper can therefore also be read as a first study of the interaction of algebraic effects and guarded recursion.

1.1 Synthetic guarded domain theory

Until now most applications of guarded recursion and step-indexing have used these for operational reasoning for programming languages. While these techniques are very useful for proving properties of programs, we believe that there is a need for also developing guarded recursion as a tool for constructing denotational semantics. Denotational methods have the benefit of often being more modular than the operational methods, and often reveal the foundational mathematical building blocks of programming

languages. Denotational semantics often inspire new programming constructions or languages, as exemplified in monads [30], runners for computational effects [37, 3], homotopy type theory [36], or even guarded recursion itself [10].

Using guarded recursion for denotational semantics has several possible benefits over domain theory. The first is that it appears to be more expressive than domain theory as illustrated by the many uses of step-indexing for advanced programming languages. Another is that step-indexing and guarded recursion by many are considered more elementary tools. A third is that guarded recursion appears to be more amenable to effective formalisation in type theory and proof assistants, although some formalisations of classical domain theory do exist [9, 18], and in particular recent progress on such a formalisation in HoTT seems promising [22].

Initial steps towards such a *synthetic guarded domain theory* were taken by Birkedal, Møgelberg and Paviotti [28, 32] who showed how to construct models of the programming languages PCF and FPC modelling recursion in these as guarded recursion, and proving the adequacy of these models entirely in a type theory with guarded recursion. This paper can be viewed as an extension of these works to non-determinism. Perhaps the main disadvantage of guarded recursion compared to domain theory is the intensional nature, allowing the model to distinguish between computations that produce the same result in a different number of steps. The present paper shows that using universal quantification over clocks allows to localise the steps and to prove properties that do not refer to steps, using the models.

1.2 Related work

Most proofs of applicative similarity being a congruence use operational arguments [25, 26], in particular Howe's method [20]. More recently, an abstract version of Howe's method has been developed [15] to handle languages with algebraic effects in a uniform way. This method uses domain theory to handle recursion. It would be interesting to see if the method described here generalises to a similar uniform method for computational effects, but this requires first developing a theory of algebraic effects in guarded type theory.

Step-indexing and guarded recursion based operational techniques have previously been used for languages with non-determinism. For example, Schwinghammer et al. [35] construct an operational model for reasoning about a typed programming language with recursive types, polymorphism and non-determinism and use it to prove contextual equivalences of programs. Bizjak et al. [11] show how to construct a similar model using guarded recursion and topos logic. These works use complex operational techniques including $\top\top$-closure. Our goal is different, namely to develop a theory of denotational semantics in a type theory with guarded recursion.

The above mentioned works on non-determinism [25, 26, 35, 11] study countable non-determinism, rather than finite non-determinism. This is generally considered a harder problem. For example, this forces the step-indexing used by Schwinghammer et al. [35] to be transfinite, whereas the underlying model of the Clocked Cubical Type Theory is based on natural number step-indexing. Likewise, defining powerdomains in domain theory for countable non-determinism is much harder than the finite case [5, 17]. We discuss the possibility of extending our approach to countable non-determinism in Section 8.

As described in Section 3, our partiality monad L^κ is strongly related to the coinductive partiality monad, and our use of it is similar to previous uses in semantics of recursion [9, 12, 16]. Interaction trees [39] are a general data structure combining the coinductive partiality monad with computational effects. Our guarded powerdomain monads can perhaps be seen as a form of guarded interaction trees for non-determinism, except that the use of HITs allows us to consider these up to an equational theory.

1.3 Overview

The paper is organised as follows: We first recall the basics of Cubical Type Theory in Section 2, in particular path types and higher inductive types. Section 3 then recalls Clocked Cubical Type Theory, the extension of Cubical Type Theory with multiclocked guarded recursion. Section 4 defines the guarded powerdomain for may-convergence and Section 5 presents our proof that applicative may-similarity is a congruence using a denotational model. Section 6 defines a guarded powerdomain for must-convergence, and Section 7 presents our proof that applicative must-similarity is a congruence. We conclude in Section 8.

2 Cubical type theory

Cubical Type Theory (CTT) [14] is a variant of Homotopy Type Theory (HoTT) [36] based on the cubical model of the univalence axiom, and specifically designed to compute with univalence. It moreover has the benefit of combining more easily with guarded recursion than HoTT, which was the reason for using it as a base for Clocked Cubical Type Theory as we shall describe in Section 3. Reading this paper does not require deep knowledge of CTT, and this section recalls the basic notions from CTT and HoTT that we shall need.

Perhaps the most fundamental difference between CTT and Martin-Löf type theory is that the identity type in the latter is replaced in CTT by a type of paths $\mathsf{Path}_A(x,y)$ between two elements $x,y:A$. We will often write the path type infix as $x=y$, and say that x and y are path equal if there is an element of $x=y$. Cubical Type Theory represent paths as maps from an abstract interval type $\mathbb{I}$, with endpoints 0 and 1. In particular a lambda abstraction like $\lambda i.t$ will build a path of type $t[0/i]=t[1/i]$. This allows more canonical proofs of equality than just reflexivity, and so to give computational content to principles like function extensionality and univalence. Path equality is still substitutive, in the sense that any element of type $P(x)$ can be transported along a path $x=y$ to construct an element of $P(y)$. In the following we will not rely on details about the specific primitives of CTT, which can be found in [14], and [38] for their incarnation in the Agda proof assistant.

Types can be classified according to the complexity of their path equality: We say a type is a *mere proposition* if any two elements are path equal, and that a type is a *homotopy set*, or simply a *set*, if its path equality type is a mere proposition. These predicates can be expressed in the type theory as types $\mathsf{isProp}(A)$ and $\mathsf{isSet}(A)$ for a type A. Given any universe of types U, we can form a universe of mere propositions Prop whose elements are pairs of an element of U and a proof that it is propositional. For $A:\mathsf{Prop}$ we will often write A itself rather than its first projection.

Cubical Type Theory also supports Higher Inductive Types (HITs), which allow to define an inductive type by declaring constructors also for its path equality, rather than only for its elements. For example, the propositional truncation $||A||$ of a type A is defined as a higher inductive type with the following constructors

$$|-|:A\to||A|| \qquad\qquad \mathsf{squash}:\Pi(xy:||A||).x=y$$

This defines the least proposition extending A in the sense that any map $f:A\to B$ into a proposition B, defines a unique map $\overline{f}:||A||\to B$, such that $\overline{f}(|a|)=f(a)$ for all a. If B is a set, then such an extension still exists if $\Pi(xy:A).fx=fy$ [23]. For any two propositions A and B their conjunction $A\wedge B$ is given by the cartesian product $A\times B$, their disjunction $A\vee B$ by truncating their disjoint union $||A+B||$, while the true and false propositions are given by the unit and the empty type. Univalence implies that any two

logically equivalent propositions are equal, so associativity and commutativity of disjunction and other such laws hold as path equalities. Given a predicate $P : A \to \mathsf{Prop}$, universal quantification $\forall a : A.Pa$ is given by the dependent function type $\Pi(a : A).Pa$, while existential quantification $\exists a : A.Pa$ is given by truncating the dependent pair type $||\Sigma a : A.Pa||$ as generally it will not be propositional otherwise. As $||Q||$ and Q coincide when Q is a proposition, so do $\exists a : A.Pa$ and $\Sigma a : A.Pa$ when Pa uniquely determines a.

2.1 Finite Powerset

The finite powerset $\mathsf{P_f}(A)$ [8, 19] of a type is another example of a higher inductive type defined by the following constructors

$$\begin{aligned}
\{-\} &: A \to \mathsf{P_f}(A) \\
\cup &: \mathsf{P_f}(A) \to \mathsf{P_f}(A) \to \mathsf{P_f}(A) \\
\mathsf{assoc} &: \Pi(X\,Y\,Z : \mathsf{P_f}(A)).X \cup (Y \cup Z) = (X \cup Y) \cup Z \\
\mathsf{comm} &: \Pi(X\,Y : \mathsf{P_f}(A)).X \cup Y = Y \cup X \\
\mathsf{idem} &: \Pi(X : \mathsf{P_f}(A)).X \cup X = X
\end{aligned}$$

plus two equalities ensuring that $\mathsf{P_f}(A)$ is a set [24]. Note that we restrict ourselves to *non-empty* finite powersets. We say that a pair of a type B and a binary operation $f : B \to B \to B$ is a *join-semilattice*[1] if B is a set and f is associative, commutative and idempotent. It can then be shown that $\mathsf{P_f}(A)$ is the free join-semilattice generated by A, and as such maps $\mathsf{P_f}(A) \to B$ which preserve $\cup$ correspond to maps $A \to B$ for any join-semilattice (B, f). We use this to define the membership predicate $x \in X$, as Prop forms a join-semilattice with disjunction. Membership satisfies the following equations

$$x \in \{y\} = ||x = y|| \qquad\qquad x \in (X \cup Y) = x \in X \vee x \in Y$$

If $X : \mathsf{P_f}(A)$ we write $\forall a \in X.\,Q(a)$ to mean $\Pi(a : A).\,a \in X \to Q(a)$ and similarly for $\exists a \in X.Q(a)$.

The finite powerset also supports the structure of a monad, and in particular, given $f : A \to \mathsf{P_f}(B)$ we write $\cup_{a \in X} f\,a$ for the bind operation, defined using the free join-semilattice structure. Given $f : A \to B$ we write $\mathsf{P_f}(f) : \mathsf{P_f}(A) \to \mathsf{P_f}(B)$ for the functor action of the finite powerset, defined as $\mathsf{P_f}(f)X \stackrel{\mathsf{def}}{=} \cup_{a \in X}\{f\,a\}$.

Finally, recall [29, Lemma 4.1] that if $f : A \to B$, $X : \mathsf{P_f}(A)$, and $b : B$, then

$$b \in \mathsf{P_f}(f)X \simeq \exists a \in X.f\,a = b \tag{1}$$

3 Clocked cubical type theory

Clocked Cubical Type Theory [24] extends Cubical Type Theory with the constructions of Clocked Type Theory [7, 27], a type theory with Nakano style guarded recursion, multiple clocks and ticks. This section recalls each of these concepts, but in a simplified form, omitting constructions related to tick irrelevance.

The fundamental notion in guarded recursion is that of a time step on a clock. Clocks are introduced as assumptions of the form $\kappa : \mathsf{clock}$ in the context, and time steps are represented as tick assumptions

[1] This is a slight misuse of terminology, since join-semilattices are usually assumed to also have a unit

$$\frac{\kappa : \mathsf{clock} \in \Gamma}{\Gamma, \alpha : \kappa \vdash} \qquad \frac{\Gamma, \mathsf{TimeLess}(\Gamma') \vdash t : \triangleright(\alpha : \kappa).A \qquad \Gamma, \beta : \kappa, \Gamma' \vdash}{\Gamma, \beta : \kappa, \Gamma' \vdash t\,[\beta] : A\,[\beta/\alpha]} \qquad \frac{\Gamma, \alpha : \kappa \vdash t : A}{\Gamma \vdash \lambda(\alpha : \kappa).t : \triangleright(\alpha : \kappa).A}$$

$$\frac{\Gamma, \kappa : \mathsf{clock} \vdash t : A}{\Gamma \vdash \Lambda\kappa.t : \forall\kappa.A} \qquad \frac{\Gamma \vdash t : \forall\kappa.A \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\Gamma \vdash t[\kappa'] : A[\kappa'/\kappa]}$$

Figure 1: Selected typing rules for Clocked Cubical Type Theory [24]. The telescope $\mathsf{TimeLess}(\Gamma')$ is composed of the timeless assumptions in Γ', i.e. interval variables and faces (as in Cubical Type Theory) as well as clock variables.

of the form $\alpha : \kappa$. The type $\triangleright(\alpha : \kappa).A$ classifies computations that in the next time step (as represented by the tick α) return elements of A. When α does not appear in A we simply write $\triangleright^\kappa A$ for this type. Elements of $\triangleright(\alpha : \kappa).A$ are introduced by *tick* abstraction $\lambda(\alpha : \kappa).t$ and eliminated by tick application $t\,[\alpha]$. The rules are similar to those for function types, except that tick application requires the eliminated term t not to depend on α, nor any of the variables bound before α. This rules out terms like $\lambda x.\lambda(\alpha : \kappa).x\,[\alpha]\,[\alpha] : \triangleright^\kappa\triangleright^\kappa A \to \triangleright^\kappa A$, which collapse two steps into one. Interval variables are considered timeless and therefore exempt from this restriction, which is necessary to prove that tick application preserves equalities:

$$(\lambda p.\lambda(\alpha : \kappa).\lambda i.\,(p\,i\,[\alpha])) : x =_{\triangleright(\alpha:\kappa).A} y \to (\triangleright(\alpha : \kappa).x\,[\alpha] =_A y\,[\alpha]) \tag{2}$$

In fact, the above map is an equivalence of types, and this extensionality principle is one of the main reasons CTT is used rather than HoTT. One consequence is that $\triangleright$ preserves truncation levels. In particular, if $\triangleright(\alpha : \kappa).\mathsf{isProp}(A\,[\alpha])$ then also $\mathsf{isProp}(\triangleright(\alpha : \kappa).(A\,[\alpha]))$ [29, Lemma 3.1] and similarly for sets.

The delay type allows to safely introduce a fixpoint combinator fix^κ of type $(\triangleright^\kappa A \to A) \to A$ and satisfying the path equality $\mathsf{fix}^\kappa t = t\,(\lambda(\alpha : \kappa).\mathsf{fix}^\kappa t)$ for any t. We can use fix^κ to define *guarded recursive types*, i.e. ones where the recursive occurrences are guarded by $\triangleright^\kappa$. An example is the partiality monad mapping $A : \mathsf{U}$ to $\mathsf{L}^\kappa A \stackrel{\mathsf{def}}{=} \mathsf{fix}^\kappa(\lambda(X : \triangleright^\kappa\mathsf{U}).A + \triangleright(\alpha : \kappa).X\,[\alpha])$. The path equality between this type and its unfolding gives rise to a type equivalence

$$\mathsf{L}^\kappa A \simeq A + \triangleright^\kappa \mathsf{L}^\kappa A$$

We use now_L and step_L to denote the two inclusions into $\mathsf{L}^\kappa A$ up to the above equivalence.

$$\mathsf{now}_\mathsf{L} : A \to \mathsf{L}^\kappa A \qquad\qquad \mathsf{step}_\mathsf{L} : \triangleright^\kappa \mathsf{L}^\kappa A \to \mathsf{L}^\kappa A$$

An element of $\mathsf{L}^\kappa A$ represents a possibly non-terminating computation of an element of A. For example, the element $\bot = \mathsf{fix}^\kappa(\mathsf{step}_\mathsf{L})$ represents divergence. We say that (A, δ) is a *delay algebra* if δ has type $\triangleright^\kappa A \to A$. The pair $(\mathsf{L}^\kappa A, \mathsf{step}_\mathsf{L})$ is the free delay algebra generated by A in the sense that any map $f : A \to B$ where (B, δ) is a delay algebra defines a unique $\overline{f} : \mathsf{L}^\kappa A \to B$ such that

$$\overline{f}(\mathsf{now}_\mathsf{L}\, a) = f\, a \qquad\qquad \overline{f}(\mathsf{step}_\mathsf{L}\, x) = \delta(\lambda(\alpha : \kappa).\overline{f}(x\,[\alpha]))$$

The monad structure is then defined with now_L as the unit, and multiplication $\mu_\mathsf{L} : \mathsf{L}^\kappa\mathsf{L}^\kappa A \to \mathsf{L}^\kappa A$ defined as the unique (delay algebra)-homomorphism extending the identity.

The *clock quantification* type former $\forall\kappa.A$ is introduced by clock abstraction, and eliminated by application to a clock. It behaves much like a Π-type, except that clock is not a type, but has a similar status to the interval $\mathbb{I}$. Clock quantification localises guarded recursion on a clock, and in particular supports a map $\mathsf{force} : \forall\kappa.\triangleright^\kappa A \to \forall\kappa.A$, inverse to $\lambda x.\lambda\kappa.\lambda(\alpha:\kappa).x[\kappa]$, allowing to safely eliminate $\triangleright^\kappa$.

The main use case for clock quantification is to encode coinductive types. For example $\forall\kappa.\mathsf{L}^\kappa A$ is the final coalgebra for the functor $F(X) = A + X$, if A is *clock irrelevant*, i.e., if the canonical map $A \to \forall\kappa.A$ is an equivalence. The notion of clock irrelevance is closed under all basic type formers as well as inductive types and (under certain restrictions [24]) higher inductive types. In particular, the inductive types used in this paper to represent syntax are all clock irrelevant. This encoding of coinductive types is originally due to Atkey and McBride [6] and presumes the existence of a clock constant κ_0, which we achieve by just initially assuming $\kappa_0 : \mathsf{clock}$.

More generally for every (indexed) functor F which commutes with clock quantification we have that $\forall\kappa.\mathsf{fix}^\kappa(\lambda X.F(\triangleright(\alpha:\kappa).X[\alpha]))$ is the final coalgebra of F, i.e. its coinductive fixpoint [24]. The collection of functors commuting with clock quantification is closed under a long list of constructors including truncations, finite powersets and sum types as expressed in the following type equivalences.

$$\forall\kappa.||A|| \simeq ||\forall\kappa.A|| \qquad \forall\kappa.\mathsf{P_f}(A) \simeq \mathsf{P_f}(\forall\kappa.A) \qquad \forall\kappa.(A+B) \simeq \forall\kappa.A + \forall\kappa.B$$

4 A powerdomain for may-convergence

Define the may powerdomain as the unique solution to the guarded recursive equation

$$\mathsf{P}^\kappa_\Diamond(A) \simeq \mathsf{P_f}(A + \triangleright^\kappa \mathsf{P}^\kappa_\Diamond(A))$$

Formally, $\mathsf{P}^\kappa_\Diamond$ can be defined as a fixed point $\mathsf{fix}^\kappa(\lambda(X : \triangleright^\kappa \mathsf{U}).\mathsf{P_f}(A + \triangleright(\alpha:\kappa).X[\alpha]))$ similarly to the definition of $\mathsf{L}^\kappa A$, but in the rest of this paper we will not give such definitions explicitly. The type constructor $\mathsf{P}^\kappa_\Diamond$ comes equipped with the following operations

$$\cup : \mathsf{P}^\kappa_\Diamond(A) \to \mathsf{P}^\kappa_\Diamond(A) \to \mathsf{P}^\kappa_\Diamond(A) \qquad \mathsf{now}_\Diamond : A \to \mathsf{P}^\kappa_\Diamond(A) \qquad \mathsf{step}_\Diamond : \triangleright^\kappa \mathsf{P}^\kappa_\Diamond(A) \to \mathsf{P}^\kappa_\Diamond(A)$$

where $\cup$ is inherited from $\mathsf{P_f}$ and therefore defines a join-semilattice and

$$\mathsf{now}_\Diamond(a) = \{\mathsf{inl}(a)\} \qquad \mathsf{step}_\Diamond(a) = \{\mathsf{inr}(a)\}$$

defines a unit and a delay algebra structure. In particular, this means that $\mathsf{P}^\kappa_\Diamond(A)$ can represent diverging computations ($\bot = \mathsf{fix}^\kappa(\mathsf{step}_\Diamond)$), as well as values. An element of $\mathsf{P}^\kappa_\Diamond(A)$ may also have both converging and diverging branches, as for example $\{a, \bot\}$.

The next lemma states that for a set A, $\mathsf{P}^\kappa_\Diamond(A)$ is the free algebra for the theory combining delay and union with no interaction between the two.

Lemma 1. *Let A be a set, let B be a set with both a join-semilattice structure and a delay algebra structure, and let $f : A \to B$. Then there is a unique map $\mathsf{P}^\kappa_\Diamond(A) \to B$ extending f and commuting with the join-semilattice and delay-algebra structures.*

In terms of algebraic theories, $\mathsf{P}^\kappa_\Diamond$ can therefore be seen as generated by the theories of join-semilattices and delay-algebras with no operations between them. As a special case of the lemma one can define a bind operation mapping $a : \mathsf{P}^\kappa_\Diamond(A)$ and $f : A \to \mathsf{P}^\kappa_\Diamond B$ to $a \mathbin{>\!\!>=} f : \mathsf{P}^\kappa_\Diamond B$ which then equips $\mathsf{P}^\kappa_\Diamond$ with a monad structure with unit given by $\mathsf{now}_\Diamond$. Bind moreover commutes with these operations as in the following equations

$$(\mathsf{step}_\Diamond\, a \mathbin{>\!\!>=} f) = \mathsf{step}_\Diamond\, \lambda(\alpha:\kappa).(a[\alpha] \mathbin{>\!\!>=} f) \tag{3}$$

$$((a \cup b) \mathbin{>\!\!>=} f) = (a \mathbin{>\!\!>=} f) \cup (b \mathbin{>\!\!>=} f) \tag{4}$$

5 Applicative may-simulation

The type constructor $\mathsf{P}^\kappa_\Diamond$ should be seen as a guarded recursive powerdomain for may-convergence. Intuitively, this is true because an element of $\mathsf{P}^\kappa_\Diamond(A)$ describes a set of values (of type A) that the computation has returned now, and a set of computations that we can choose to further evaluate, but may also choose not to, if we are only interested in testing if a program may evaluate to a particular value. This should be seen in contrast to the must-powerdomain of Section 6 which will force all branches of a computation tree to be evaluated fully before the result values can be inspected. In this section we substantiate that claim by using $\mathsf{P}^\kappa_\Diamond$ to prove applicative may-similarity a congruence.

We start by recalling the untyped lambda calculus with binary non-determinism and the notion of applicative may-similarity. We use informal binding notation here for readability, using the grammar

$$M,N ::= MN \mid \hat{\lambda}\, x.M \mid M \,\mathsf{or}\, N$$

for terms. This could for example be implemented more formally as an inductive family using de Bruijn indices. Note the use of $\hat{\lambda}$ to distinguish it from the meta-level lambda. A value is a closed term of the form $\hat{\lambda}\, x.M$, and we shall use Λ and Val for the types of closed lambda terms and values respectively, which we will assume are sets (and indeed are if formalised using de Bruijn indices). These are moreover clock irrelevant, which can be shown by embedding them into the inductive types of all (also open) terms, and the fact that all inductive types are clock irrelevant [24].

We define two operational semantics. The first one is a big-step operational semantics formulated as a relation $\Downarrow_\Diamond : \Lambda \times \mathsf{Val} \to \mathsf{Prop}$ defined inductively in the standard way

$$\frac{V = W}{V \Downarrow_\Diamond W} \qquad \frac{M \Downarrow_\Diamond \hat{\lambda}\, x.M' \qquad N \Downarrow_\Diamond V' \qquad M'[V'/x] \Downarrow_\Diamond V}{MN \Downarrow_\Diamond V} \qquad \frac{M \Downarrow_\Diamond V \vee N \Downarrow_\Diamond V}{M \,\mathsf{or}\, N \Downarrow_\Diamond V}$$

One can also similarly define a small-step operational semantics and prove this equivalent to the big-step semantics using standard methods.

The second operational semantics is less familiar but has the benefit of being suitable for guarded recursive reasoning. It uses the monad $\mathsf{P}^\kappa_\Diamond$ to model recursion and non-determinism, but since parts of the development in this section will be reused later, we will define the operational semantics relative to a monad T, which can be instantiated to be $\mathsf{P}^\kappa_\Diamond$. We will assume that T has operations $\cup$ and step_T providing T with a join-semilattice structure as well as a delay-algebra structure satisfying the equations (3) and (4). In fact we shall see later that not all axioms for monads are needed for our developments.

Using this, we define an evaluation function $\mathsf{eval} : \Lambda \to T(\mathsf{Val})$ as

$$\begin{aligned}
\mathsf{eval}\,(\hat{\lambda}\, x.M) &= \mathsf{pure}\,(\hat{\lambda}\, x.M) \\
\mathsf{eval}\,(MN) &= \mathsf{eval}\,M >\!\!>\!= \lambda(\hat{\lambda}\, x.M').\, \mathsf{eval}\,N >\!\!>\!= \lambda V.\, \mathsf{step}_T\,(\lambda(\alpha : \kappa).\mathsf{eval}\,(M'[V/x])) \\
\mathsf{eval}\,(M \,\mathsf{or}\, N) &= \mathsf{eval}\,M \cup \mathsf{eval}\,N
\end{aligned}$$

where pure refers to the unit of the monad T. Note that the match on $(\hat{\lambda}\, x.M')$ in the application case is exhaustive, since values can only be lambda abstractions. To lift this to a big step relation, we will assume given a lifting of T to predicates as follows.

Definition 1. *A lifting of a monad T to predicates is a function that maps a predicate $P : A \to \mathsf{Prop}$ to a predicate $\hat{T}(P) : T(A) \to \mathsf{Prop}$ satisfying the following properties*

1. $\hat{T}(P)(T(f)(a)) = \hat{T}(P \circ f)(a)$

2. $\hat{T}(P)(\mathsf{pure}(a)) = P(a)$
3. $\hat{T}(P)(\mathsf{step}_T(a)) = \triangleright(\alpha:\kappa).\hat{T}(P)(a[\alpha])$
4. $\hat{T}(P)(m >>= \lambda x.t) = \hat{T}(\lambda x.\hat{T}(P)(t))(m)$
5. $\hat{T}(P)(a \cup b) = \hat{T}(P)(a) \wedge \hat{T}(P)(b)$

Note that a unique such lifting can be defined for $\mathsf{P}^\kappa_\Diamond$ using Lemma 1 since Prop is a set, and $\wedge$ and $\triangleright$ define a join-semilattice and delay-algebra structure on Prop, so $\hat{T}(P)$ can be defined as the extension of P.

Specialising to $T = \mathsf{P}^\kappa_\Diamond$ we make the following definition.

Definition 2. *Let $\hat{T}$ be the lifting of $\mathsf{P}^\kappa_\Diamond$ to predicates, let $Q : \mathsf{Val} \to \mathsf{Prop}$ be a predicate on values and let $M : \Lambda$. Define*

$$M \Downarrow^\kappa_\Diamond Q \stackrel{def}{=} \hat{T}Q(\mathsf{eval}(M)) : \mathsf{Prop}$$

One can then define a more standard big step operational semantics as

$$M \Downarrow^\kappa_\Diamond V \stackrel{\mathsf{def}}{=} M \Downarrow^\kappa_\Diamond (\lambda W.(V = W))$$

Intuitively $M \Downarrow^\kappa_\Diamond Q$ means that if M terminates to a value, that value will satisfy Q. In particular, $M \Downarrow^\kappa_\Diamond V$ will hold for any V if M diverges, unlike the statement $M \Downarrow_\Diamond V$ which guarantees termination.

To express the precise relationship between the two semantics, we introduce a predicate of may-convergence on the powerdomain $\mathsf{P}^\kappa_\Diamond$. Since termination cannot be expressed as a predicate on guarded recursive types directly, it must be expressed as a predicate on $\forall\kappa.\mathsf{P}^\kappa_\Diamond(-)$, which captures global behaviour of $\mathsf{P}^\kappa_\Diamond$ [28]. If $m : \forall\kappa.\mathsf{P}^\kappa_\Diamond(A)$ and $a : A$ define $m \Downarrow^\forall_{\mathsf{P}_\Diamond} a$ as the proposition inductively generated by the following introductions

$$\frac{}{\lambda\kappa.\mathsf{now}_\Diamond(a) \Downarrow^\forall_{\mathsf{P}_\Diamond} a} \qquad \frac{m \Downarrow^\forall_{\mathsf{P}_\Diamond} a}{\lambda\kappa.\mathsf{step}_\Diamond(\lambda(\alpha:\kappa).m[\kappa]) \Downarrow^\forall_{\mathsf{P}_\Diamond} a} \qquad \frac{m \Downarrow^\forall_{\mathsf{P}_\Diamond} a \vee m' \Downarrow^\forall_{\mathsf{P}_\Diamond} a}{\lambda\kappa.m[\kappa] \cup m'[\kappa] \Downarrow^\forall_{\mathsf{P}_\Diamond} a}$$

This means that these rules should be read as constructors of a HIT which also has propositional truncation operators as a constructor. The relationship between the two semantics can then be expressed as follows.

Proposition 1. *The statements $M \Downarrow_\Diamond V$ and $(\lambda\kappa.\mathsf{eval}(M)) \Downarrow^\forall_{\mathsf{P}_\Diamond} V$ are logically equivalent.*

Finally, the next lemma makes the intuition for $M \Downarrow^\kappa_\Diamond Q$ stated above precise. As the notation used in the lemma suggests, κ can appear free in Q^κ.

Lemma 2. *Let A be clock irrelevant, Q^κ a family over A for each κ, and $m : \forall\kappa.\mathsf{P}^\kappa_\Diamond(A)$. The statements $\forall\kappa.\hat{\mathsf{P}}^\kappa_\Diamond Q^\kappa(m[\kappa])$ and $\forall a.m \Downarrow^\forall_{\mathsf{P}_\Diamond} a \to \forall\kappa.Q^\kappa(a)$ are logically equivalent. As a consequence $\forall\kappa.M \Downarrow^\kappa_\Diamond Q^\kappa$ is equivalent to $M \Downarrow_\Diamond V \to \forall\kappa.Q^\kappa(V)$.*

5.1 Applicative may-similarity

We now recall the notion of applicative similarity, as originally studied by Abramsky [1] for the pure lambda calculus and adapt it to finite non-determinism in the case of may-convergence. Say that a relation R on closed terms is an applicative may-simulation if MRN and $M \Downarrow_\Diamond \lambda\!\!\lambda\, x.M'$ implies

$$\exists N'.N \Downarrow_\Diamond \lambda\!\!\lambda\, y.N' \wedge (\forall(V : \mathsf{Val}).M'[V/x]RN'[V/x])$$

Applicative may-similarity is the greatest applicative may-simulation. We define this by universally quantifying a clock in a guarded recursive definition. First define

$$\leq^{\kappa}_{\mathsf{Val}} : \mathsf{Val} \to \mathsf{Val} \to \mathsf{Prop}$$
$$\leq^{\kappa}_{\Diamond} : \Lambda \to \Lambda \to \mathsf{Prop}$$
$$\lambda\!\!\lambda\, x.M \leq^{\kappa}_{\mathsf{Val}} \lambda\!\!\lambda\, y.N \stackrel{\mathsf{def}}{=} \triangleright(\alpha:\kappa).(\forall V:\mathsf{Val}.M[V/x] \leq^{\kappa}_{\Diamond} N[V/y])$$
$$M \leq^{\kappa}_{\Diamond} N \stackrel{\mathsf{def}}{=} M \Downarrow^{\kappa}_{\Diamond} \lambda V.(\exists W.N \Downarrow_{\Diamond} W \wedge V \leq^{\kappa}_{\mathsf{Val}} W)$$

The statement $M \leq^{\kappa}_{\Diamond} N$ should be read as stating that if M terminates, then also N terminates, and moreover, applying the resulting terms to the same value results in the related results later. Note the asymmetry in the use of operational semantics: The evaluation of M uses the guarded operational semantics which ensures that if M diverges, then $M \leq^{\kappa}_{\Diamond} N$ is true. On the other hand, once M converges to a value, N must also converge, and expressing this requires the inductive operational semantics. The delay in the definition of $\leq^{\kappa}_{\mathsf{Val}}$ ensures well-definedness: unfolding the definition of $\leq^{\kappa}_{\Diamond}$ in $\leq^{\kappa}_{\mathsf{Val}}$ gives a guarded recursive definition.

Applicative similarity is extended to open terms by defining $M \leq^{\kappa}_{\Diamond} N$ to mean $M\sigma \leq^{\kappa}_{\Diamond} N\sigma$ for all substitutions σ mapping all free variables in M and N to values. Finally, we localise the steps in the definition of $M \leq^{\kappa}_{\Diamond} N$ by universally quantifying κ and thereby pass to a coinductive type:

$$M \leq_{\Diamond} N \stackrel{\mathsf{def}}{=} \forall\kappa.M \leq^{\kappa}_{\Diamond} N$$

Lemma 3. *$\leq_{\Diamond}$ is the greatest applicative may-simulation.*

We now proceed to prove that applicative may-similarity is a congruence. Most proofs of this use syntactic arguments, but here we use a semantic method developed by Pitts [34] in the context of domain theory, which we adapt to guarded recursion. As a first step we construct a denotational semantics of the untyped lambda calculus.

5.2 Denotational semantics

Like the operational semantics, the denotational semantics is parametrised by a monad T equipped with a join-semilattice structure and a delay algebra structure, satisfying (3) and (4). Closed terms will be interpreted as elements of the type D^{κ} defined by the following equations.

$$\mathsf{SVal}^{\kappa} \stackrel{\mathsf{def}}{=} \triangleright^{\kappa}(\mathsf{SVal}^{\kappa} \to T(\mathsf{SVal}^{\kappa})) \qquad\qquad D^{\kappa} \stackrel{\mathsf{def}}{=} T(\mathsf{SVal}^{\kappa})$$

Here the definition of SVal^{κ} should be read as a guarded recursive definition, and states that semantic values can be considered effectful computations on semantic values, but that this unfolding takes a single computation step. Define a semantic application $\cdot : D^{\kappa} \times D^{\kappa} \to D^{\kappa}$ as

$$d \cdot d' = d \mathbin{>\!\!>\!=} \lambda f.\ d' \mathbin{>\!\!>\!=} \lambda v.\ \mathsf{step}_T\,(\lambda(\alpha:\kappa).f\,[\alpha]\,v)$$

and using this we define the operational semantics $[\![-]\!]^{\kappa} : \Lambda(n) \to (\mathsf{SVal}^{\kappa})^n \to D^{\kappa}$ where $\Lambda(n)$ is the set of terms with at most n free variables, as

$$[\![x_i]\!]^{\kappa}\rho = \mathsf{pure}\,(\rho\, i) \qquad [\![\lambda\!\!\lambda\, x_{n+1}.M]\!]^{\kappa}\rho = \mathsf{pure}\,(\lambda(\alpha:\kappa).(\lambda d.[\![M]\!]^{\kappa}(\rho,d)))$$
$$[\![MN]\!]^{\kappa}\rho = [\![M]\!]^{\kappa}\rho \cdot ([\![N]\!]^{\kappa}\rho) \qquad [\![M \,\mathsf{or}\, N]\!]^{\kappa}\rho = [\![M]\!]^{\kappa}\rho \cup [\![N]\!]^{\kappa}\rho$$

Note that the interpretation of values factor through $\mathsf{pure} : \mathsf{SVal}^{\kappa} \to D^{\kappa}$ via $[\![-]\!]^{\kappa}_{\mathsf{Val}} : \mathsf{Val} \to \mathsf{SVal}^{\kappa}$.

Theorem 1 (Soundness)**.** $T([\![-]\!]^\kappa_{\mathsf{Val}})(\mathsf{eval}\, M) = [\![M]\!]^\kappa$

We now specialise T to $\mathsf{P}^\kappa_\Diamond$ for the following notation and corollary. If $Q : A \to \mathsf{Prop}$ and $m : \mathsf{P}^\kappa_\Diamond A$, we shall use the infix notation $m \downarrow^\kappa_\Diamond Q$ for $\hat{T}Qm$, where $\hat{T}$ is the lifting of $\mathsf{P}^\kappa_\Diamond$.

Corollary 1. *The statements* $[\![M]\!]^\kappa \downarrow^\kappa_\Diamond Q$ *and* $M \Downarrow^\kappa_\Diamond Q \circ [\![-]\!]^\kappa_{\mathsf{Val}}$ *are equivalent.*

5.3 Relating syntax and semantics

We now construct a relation between syntax and semantics that will allow us to use the model to reason about the operational semantics. The relation is similar to the one constructed by Pitts [34] in the setting of domain theory, but whereas Pitts must provide a technical argument for the existence of the relation, which is far from obvious in the domain theoretic setting, in our setting the relation exists simply by guarded recursion.

In this section we specialise the model from the general monad T to $\mathsf{P}^\kappa_\Diamond$, and define two relations, one on values and one on general terms as follows

$$\begin{aligned}
\preceq^\kappa &: D^\kappa \times \Lambda \to \mathsf{Prop} \\
\preceq^\kappa_{\mathsf{Val}} &: \mathsf{SVal}^\kappa \times \mathsf{Val} \to \mathsf{Prop} \\
d \preceq^\kappa M &\stackrel{\mathsf{def}}{=} d \downarrow^\kappa_\Diamond \lambda v.(\exists V.M \Downarrow_\Diamond V \wedge v \preceq^\kappa_{\mathsf{Val}} V) \\
v \preceq^\kappa_{\mathsf{Val}} \lambda x.M &\stackrel{\mathsf{def}}{=} \triangleright(\alpha:\kappa).(\forall v', V'.v' \preceq^\kappa_{\mathsf{Val}} V' \to (v[\alpha](v')) \preceq^\kappa M[V'/x])
\end{aligned}$$

This is well-defined, because unfolding the definition of $\preceq^\kappa$ in the definition of $\preceq^\kappa_{\mathsf{Val}}$ gives a guarded recursive definition of $\preceq^\kappa_{\mathsf{Val}}$. If $\rho : (\mathsf{SVal}^\kappa)^n$ and $\sigma : \mathsf{Val}^n$ write $\rho \preceq^\kappa_{\mathsf{Val}} \sigma$ to mean $\rho_1 \preceq^\kappa_{\mathsf{Val}} \sigma_1 \wedge \ldots \wedge \rho_n \preceq^\kappa_{\mathsf{Val}} \sigma_n$.

Lemma 4 (Fundamental lemma)**.** *If* $\rho \preceq^\kappa_{\mathsf{Val}} \sigma$ *then* $[\![M]\!]^\kappa \rho \preceq^\kappa M\sigma$.

The fundamental lemma is proved by induction on M. Using this, one can prove the following correspondence between $\preceq^\kappa$ and $\leq_\Diamond$

Lemma 5. *If* M *and* N *are closed terms then* $M \leq_\Diamond N$ *is equivalent to* $\forall\kappa.[\![M]\!]^\kappa \preceq^\kappa N$.

The left to right direction is proved by showing that $\preceq^\kappa$ is upward closed in its second argument. The other direction is proved using guarded recursion. Note that as a consequence of Lemma 4 and 5 it follows that $\leq_\Diamond$ is a reflexive relation.

Theorem 2. $\leq_\Diamond$ *is a congruence, i.e., if* $M \leq_\Diamond N$ *and* $C[-]$ *is a context then also* $C[M] \leq_\Diamond C[N]$.

Proof. Using reflexivity it suffices to show that if $M \leq_\Diamond N$ and $M' \leq_\Diamond N'$ then $MM' \leq_\Diamond NN'$, $M \,\mathsf{or}\, M' \leq_\Diamond N \,\mathsf{or}\, N'$ and $\lambda x.M \leq_\Diamond \lambda x.N$. The cases of application and choice can be reduced to the statements that if $d \preceq^\kappa M$ and $d' \preceq^\kappa N$ then $d \cdot d' \preceq^\kappa MN$ and $d \cup d' \preceq^\kappa M \,\mathsf{or}\, N$, which can be proved by guarded recursion. To prove $\lambda x.M \leq_\Diamond \lambda x.N$ it suffices to prove that $\triangleright^\kappa(\forall V.M[V/x] \leq_\Diamond N[V/x])$. By definition of applicative may-similarity for open terms, however, we know that $M[V/x] \leq_\Diamond N[V/x]$. □

6 A powerdomain for must-convergence

We now introduce our powerdomain construction $\mathsf{P}^\kappa_\Box$ for must-convergence. This should have an inclusion $\mathsf{now}_\Box : A \to \mathsf{P}^\kappa_\Box(A)$, a join-semilattice structure $\cup$ and a delay algebra structure $\mathsf{step}_\Box$. However,

when considering must-convergence, a term $M \operatorname{or} N$ diverges if M diverges even if N converges. To enforce that in our powerdomain we use equations to enforce parallel evaluation of subcomputations and stating that terminating values are postponed until all subcomputations have been evaluated fully:

$$\mathsf{step}_\square(x) \cup \mathsf{step}_\square(y) = \mathsf{step}_\square(\lambda(\alpha:\kappa).x[\alpha] \cup y[\alpha]) \tag{5}$$

$$\mathsf{step}_\square(x) \cup \mathsf{now}_\square(y) = \mathsf{step}_\square(\lambda(\alpha:\kappa).(x[\alpha] \cup \mathsf{now}_\square(y))) \tag{6}$$

These equations (together with the derivable symmetric version of (6)) allow steps to bubble up the syntax tree, to a normal form consisting of a (possibly infinite) sequence of computation steps followed by a finite set of values. Following this intuition we define

$$\mathsf{P}^\kappa_\square(A) \stackrel{\mathsf{def}}{=} \mathsf{L}^\kappa(\mathsf{P}_\mathsf{f}(A))$$

This has the benefit over, say a HIT given by the equations above, of giving direct access to the set of possible values returned by a computation that must converge.

By definition $\mathsf{P}^\kappa_\square$ carries a delay-algebra structure, and the inclusion of A into $\mathsf{P}^\kappa_\square(A)$ can be defined as

$$\mathsf{now}_\square(a) \stackrel{\mathsf{def}}{=} \mathsf{now}_\mathsf{L}(\{a\})$$

The join-semilattice can be defined by guarded recursion using the equations (5), (6), the symmetrisation of (6) and

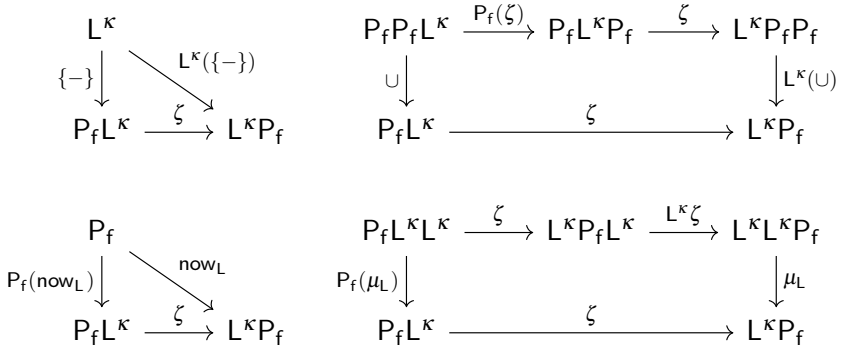
$$\mathsf{now}_\mathsf{L}\, x \cup \mathsf{now}_\mathsf{L}\, y \stackrel{\mathsf{def}}{=} \mathsf{now}_\mathsf{L}(x \cup y)$$

A natural question is whether $\mathsf{P}^\kappa_\square$ defines a monad. Since it is the composite of two monads, it is sufficient that there is a distributive law of monads, and indeed a natural candidate is easily defined as

$$\zeta : \mathsf{P}_\mathsf{f}\mathsf{L}^\kappa \to \mathsf{L}^\kappa\mathsf{P}_\mathsf{f} \qquad\qquad \zeta(X) \stackrel{\mathsf{def}}{=} \cup_{x\in X}\mathsf{L}^\kappa(\{-\})(x)$$

However, this only defines a distributive law of the monad P_f over L^κ considered as a functor, not a monad.

Proposition 2. *Of the four diagrams for distributive laws over monads:*



all but the last commute.

A counterexample to the last is $\{\mathsf{step}_\mathsf{L}(\lambda(\alpha:\kappa).\mathsf{now}_\mathsf{L}(\mathsf{now}_\mathsf{L}\, x)), \mathsf{now}_\mathsf{L}(\mathsf{step}_\mathsf{L}\,\lambda(\beta:\kappa).\mathsf{now}_\mathsf{L}\, x)\}$ which is mapped by the lower composite to $\mathsf{step}_\mathsf{L}(\lambda(\alpha:\kappa).\mathsf{now}_\mathsf{L}\{x\}))$ and by the upper to $\mathsf{step}_\mathsf{L}\,\lambda(\alpha:\kappa).\mathsf{step}_\mathsf{L}\,\lambda(\beta:\kappa).\mathsf{now}_\mathsf{L}\{x\}$. Note that these only differ by a finite number of computation steps, i.e., are equal up to weak bisimilarity. We conjecture that this is generally true and that $\mathsf{P}^\kappa_\square$ is a monad up to weak bisimilarity.

As a consequence, using ζ to define the multiplication of $\mathsf{P}^\kappa_\square$ does not define a monad. Nevertheless, it does define a bind operation.

Lemma 6. *The bind operation induced by ζ maps $f : A \to \mathsf{P}^\kappa_\square(B)$ and $a : \mathsf{P}^\kappa_\square(A)$ to*

$$a \mathbin{>\!\!>\!=_{\mathsf{L}}} \lambda X.\, \cup_{x \in X} f(x)$$

and satisfies the equations $(\mathsf{now}_\square(a) \mathbin{>\!\!>\!=} f) = f(a)$ and $(a \mathbin{>\!\!>\!=} \mathsf{now}_\square) = a$, and moreover defines a homomorphism of delay-algebras as well as join-semilattices in a. It does not satisfy the associativity axiom.

Since the associativity axiom is not used in our development, the proofs done in the previous section for a general monad T carry over to this case, as we shall see.

7 Applicative must-simulation

We now show how our techniques from the may-convergence case apply to show that applicative must-similarity is a congruence also in the case of must-convergence. First we set up the operational semantics. In the case of the standard big-step semantics, define the predicate $\Downarrow_\square \subseteq \Lambda \times \mathsf{P}_\mathsf{f}(\mathsf{Val})$ as

$$\frac{M \Downarrow_\square X \qquad N \Downarrow_\square Y}{M \,\mathsf{or}\, N \Downarrow_\square X \cup Y} \qquad\qquad \frac{}{\boldsymbol{\lambda}\, x.M \Downarrow_\square \{\boldsymbol{\lambda}\, x.M\}}$$

$$\frac{M \Downarrow_\square X \qquad N \Downarrow_\square Y \qquad \forall (\boldsymbol{\lambda}\, y.M') \in X, V \in Y.\ M'[V/y] \Downarrow_\square Z_{\boldsymbol{\lambda}\, y.M', V}}{M N \Downarrow_\square \cup_{V' \in X, V \in Y} Z_{V',V}}$$

The judgement $M \Downarrow_\square X$ states that M must converge and that the possible values that it can converge to is X.

The evaluation function $\mathsf{eval} : \Lambda \to \mathsf{P}^\kappa_\square(\mathsf{Val})$ is defined by specialising the general definition given in Section 5. We also define a relation $M \Downarrow^\kappa_\square Q$ stating that if M terminates, it will terminate to a set of values satisfying $Q : \mathsf{P}_\mathsf{f}(\mathsf{Val}) \to \mathsf{Prop}$. Note that Q is a predicate on sets of values, rather than values themselves (as was the case for $M \Downarrow^\kappa_\Diamond Q$). This allows us to express properties e.g. by existential quantification over outcome values, as needed e.g. in the definition of must-similarity below. To define $M \Downarrow^\kappa_\square Q$, consider first a lifting $\hat{\mathsf{L}^\kappa} Q$ of predicates $Q : A \to \mathsf{Prop}$ to $\mathsf{L}^\kappa A$ defined as

$$\hat{\mathsf{L}^\kappa} Q(\mathsf{now}_\mathsf{L}\, a) \stackrel{\mathsf{def}}{=} Q(a) \qquad\qquad \hat{\mathsf{L}^\kappa} Q(\mathsf{step}_\mathsf{L}\, a) \stackrel{\mathsf{def}}{=} \triangleright(\alpha : \kappa).\hat{\mathsf{L}^\kappa} Q(a\,[\alpha])$$

and note that this also satisfies items 1 and 4 of Definition 1. Define $M \Downarrow^\kappa_\square Q \stackrel{\mathsf{def}}{=} \hat{\mathsf{L}^\kappa} Q(\mathsf{eval}(M))$.

The relationship between these two operational semantics is similar to the one between $\Downarrow_\Diamond$ and $\Downarrow^\kappa_\Diamond$. First define, for $m : \forall\kappa.\mathsf{L}^\kappa A$ and $a : A$ a termination predicate $m \Downarrow^\forall_\square a$ as an inductive family in Prop like so:

$$\frac{m \Downarrow^\forall_\square a}{(\lambda\,\kappa.\,\mathsf{step}_\mathsf{L}\,(\lambda\,(\_ : \kappa).m\,\kappa)) \Downarrow^\forall_\square a} \qquad\qquad \frac{}{(\lambda\_.\,\mathsf{now}_\mathsf{L}\, a) \Downarrow^\forall_\square a}$$

Proposition 3. *The statements $M \Downarrow_\square V$ and $(\lambda\,\kappa.\,\mathsf{eval}\, M) \Downarrow^\forall_\square V$ are logically equivalent.*

Lemma 7. *Let A be clock irrelevant, Q^κ a family over A, and $m : \forall\kappa.\,\mathsf{L}^\kappa A$. The statements $\forall\kappa.\hat{\mathsf{L}^\kappa} Q^\kappa(m\,\kappa)$ and $m \Downarrow^\forall_\square a \to \forall\kappa.Q^\kappa(a)$ are logically equivalent. As a consequence the statements $M \Downarrow_\square V \to \forall\kappa.Q^\kappa(V)$ and $\forall\kappa.M \Downarrow^\kappa_\square Q^\kappa$ are equivalent.*

Say that a relation R on closed terms is an applicative must-simulation if MRN implies

$$M \Downarrow_\Box U \to \exists V.N \Downarrow_\Box V \wedge \forall(\boldsymbol{\lambda}\, x.N' \in V).\, \exists(\boldsymbol{\lambda}\, x.M' \in U).\, (\forall(W : \mathsf{Val}).M'[W/x]\, R\, N'[W/x])$$

Define $M \leq^\kappa_\Box N$ by guarded recursion to be

$$M \Downarrow^\kappa_\Box \lambda U.\, \exists V.N \Downarrow_\Box V \wedge \forall(\boldsymbol{\lambda}\, y.N' \in V).\, \exists(\boldsymbol{\lambda}\, x.M' \in U).\, \triangleright(\alpha : \kappa).(\forall W.M'[W/x] \leq^\kappa_\Box N'[W/x])$$

This is extended to open terms by defining $M \leq^\kappa_\Box N$ to mean $M\sigma \leq^\kappa_\Box N\sigma$ for all substitutions σ mapping all free variables in M and N to closed terms. Write $M \leq_\Box N$ for $\forall\kappa.M \leq^\kappa_\Box N$.

Lemma 8. *$\leq_\Box$ is the greatest applicative must-simulation.*

Also in the case of the denotational semantics the general case described in Section 5.2 specialises to $\mathsf{P}^\kappa_\Box$. None of the proofs or constructions rely on associativity of the bind operation, so also the soundness result holds. For our applications of the denotational semantics, however, we need a variant of Corollary 1 which applies to predicates on sets of values rather than on values themselves. This uses an infix notation $m \downarrow^\kappa Q$ for $\hat{\mathsf{L}^\kappa} Q(m)$.

Corollary 2. *The statements $[\![M]\!]^\kappa \downarrow^\kappa Q$ and $M \Downarrow^\kappa_\Box Q \circ \mathsf{P}_\mathsf{f}([\![-]\!]^\kappa_{\mathsf{Val}})$ are equivalent.*

7.1 Relating syntax and semantics

As in the case of may-convergence we now construct a relation between syntax and semantics. To simplify syntax we introduce the lifting of a relation $R : X \times Y \to \mathsf{Prop}$ to a relation on powersets $\overline{\mathsf{P}}_\mathsf{f}(R) : \mathsf{P}_\mathsf{f}X \times \mathsf{P}_\mathsf{f}Y \to \mathsf{Prop}$ defined as

$$\overline{\mathsf{P}}_\mathsf{f}(R)(A,B) = \forall b \in B \exists a \in A.R(a,b)$$

We define two relations between syntax and semantics by mutual guarded recursion (overwriting notation from Section 5.3):

$$\begin{aligned}
\preceq^\kappa &: D^\kappa \times \Lambda \to \mathsf{Prop} \\
\preceq^\kappa_{\mathsf{Val}} &: \mathsf{SVal}^\kappa \times \mathsf{Val} \to \mathsf{Prop} \\
d \preceq^\kappa M &\stackrel{\mathsf{def}}{=} d \downarrow^\kappa \lambda A.\exists B.M \Downarrow_\Diamond B \wedge \overline{\mathsf{P}}_\mathsf{f}(\preceq^\kappa_{\mathsf{Val}})(A,B) \\
v \preceq^\kappa_{\mathsf{Val}} \boldsymbol{\lambda}\, x.M &\stackrel{\mathsf{def}}{=} \triangleright(\alpha : \kappa).\forall v', V'.v' \preceq^\kappa_{\mathsf{Val}} V' \to (v[\alpha](v')) \preceq^\kappa M[V'/x]))
\end{aligned}$$

If $\rho : (\mathsf{SVal}^\kappa)^n$ and $\sigma : \mathsf{Val}^n$ write $\rho \preceq^\kappa \sigma$ to mean $\rho_1 \preceq^\kappa \sigma_1 \wedge \ldots \wedge \rho_n \preceq^\kappa \sigma_n$.

Lemma 9 (Fundamental lemma)**.** *If $\Gamma \vdash M$ and $\rho \preceq^\kappa \sigma$ then $[\![M]\!]^\kappa \rho \preceq^\kappa M\sigma$.*

The proof of Lemma 9 is by induction on M. In particular the case of application requires some work, and relies on the fact that $\overline{\mathsf{P}}_\mathsf{f}$ respects the monad structure of P_f in the sense that if $f : X \to \mathsf{P}_\mathsf{f}(X')$ and $g : Y \to \mathsf{P}_\mathsf{f}(Y')$ map pairs related in $R : X \times Y \to \mathsf{Prop}$ to pairs related in $\overline{\mathsf{P}}_\mathsf{f}(S)$, then the extensions $\overline{f} : \mathsf{P}_\mathsf{f}(X) \to \mathsf{P}_\mathsf{f}(X')$ and $\overline{g} : \mathsf{P}_\mathsf{f}(Y) \to \mathsf{P}_\mathsf{f}(Y')$ map pairs related in $\overline{\mathsf{P}}_\mathsf{f}(R)$ to pairs related in $\mathsf{P}_\mathsf{f}(S)$.

Lemma 10. *$M \leq_\Box N$ iff $\forall\kappa.[\![M]\!]^\kappa \preceq^\kappa N$.*

Similarly to the case of may-convergence, this implies that applicative may-similarity is a reflexive relation. From this it follows that it is a congruence exactly as in the proof of Theorem 2.

Theorem 3. *$\leq_\Box$ is a congruence, i.e., if $M \leq_\Box N$ and $C[-]$ is a context then also $C[M] \leq_\Box C[N]$.*

8 Conclusion

The constructions of this paper illustrate how the combination of guarded recursion with higher inductive types and univalence in Clocked Cubical Type Theory gives an expressive type theory for reasoning about programming languages. In particular, this combination allows arguments known from domain theory involving constructions such as recursive types to be represented in type theory. Moreover, the abstract setting of synthetic guarded domain theory allows for these tools to be used in a much more elementary setting, far from the mathematical complexity of domain theory. This is particularly clear in the construction of the relation $\leq^{\kappa}_{\Diamond}$ which in ordinary domain theory requires a non-trivial existence argument [34, 33]. It also appears in our definitions of the guarded powerdomains, which we define much more directly than the standard constructions in domain theory [2].

It is unfortunate that the bind rule for $\mathsf{P}^{\kappa}_{\Box}$ is not associative. As mentioned, this does not affect our constructions, and we conjecture that it is associative up to weak bisimilarity, and that this is enough for most purposes. We believe the reason for the failure of associativity is that the equality (6) is not algebraic in the sense that it only applies when one side is a value. One way to avoid this is to replace (5) and (6) by an equation of the form

$$\mathsf{step}_{\Box}(x) \cup y = \mathsf{step}_{\Box}(\lambda(\alpha : \kappa).(x[\alpha] \cup y))$$

which means that to evaluate $x \cup y$ takes as many steps as the sum of steps used to evaluate x and y respectively, rather than the maximum. In particular, this means that idempotency is lost (but may hold up to weak bisimilarity) and one essentially works with finite multisets rather than the standard powerset.

Future work includes extending to the case of countable non-determinism. This could use the countable powerset functor, which is also definable as a HIT [13]. We believe that the case of may-convergence generalises directly to the countable case, but in the case of must-convergence the definition of $\cup$ as used here requires deciding if all branches of a computation terminates. We believe this is a symptom of a much more fundamental problem, namely that the partiality monad of guarded recursion describes termination in finite steps, whereas the must-convergence predicate for countable non-determinism requires more steps to reach a fixed point. Bizjak et al. [11] observe a similar problem in the operational setting and solve it using a combination of $\top\top$-lifting and transfinite induction in the underlying step-indexing model. It would be interesting to see if such an approach also applies to type theory.

Finally, it would be interesting to develop a general theory of combinations of algebraic effects such as state, exceptions, and non-determinism (as studied here) with guarded recursion. The domain theoretic counterparts of these effects are usually described algebraically using order-enriched theories [21], but as we have seen here, in the setting of guarded recursion the intensional information of the individual steps allows us to describe the interaction of these effects with recursion in terms of ordinary equations. This theory could then give rise to a notion of guarded interaction trees [39] which would allow also equations between computations across steps as well as guarded recursive definitions.

Acknowledgements. We thank the anonymous reviewers for many useful observations and suggestions.

References

[1] Samson Abramsky (1990): *The lazy lambda calculus, Research topics in functional programming*.

[2] Samson Abramsky & Achim Jung (1994): *Domain theory*. Oxford University Press.

[3] Danel Ahman & Andrej Bauer (2020): *Runners in action*. In: *European Symposium on Programming*, Springer, Cham, pp. 29–55, doi:10.1007/978-3-030-44914-8_2.

[4] Andrew W. Appel & David McAllester (2001): *An indexed model of recursive types for foundational proof-carrying code*. *ACM Trans. Program. Lang. Syst* 23(5), pp. 657–683, doi:10.1145/504709.504712.

[5] Krzysztof R Apt & Gordon D Plotkin (1986): *Countable nondeterminism and random assignment*. *Journal of the ACM (JACM)* 33(4), pp. 724–767, doi:10.1145/6490.6494.

[6] Robert Atkey & Conor McBride (2013): *Productive coprogramming with guarded recursion*. *ACM SIGPLAN Notices* 48(9), pp. 197–208, doi:10.1145/2544174.2500597.

[7] Patrick Bahr., Hans Bugge Grathwohl & Rasmus Ejlers Møgelberg (2017): *The clocks are ticking: No more delays!* In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, pp. 1–12, doi:10.1109/LICS.2017.8005097.

[8] Henning Basold, Herman Geuvers & Niels van der Weide (2017): *Higher Inductive Types in Programming*. *J. Univers. Comput. Sci.* 23(1), pp. 63–88. Available at http://www.jucs.org/jucs_23_1/higher_inductive_types_in.

[9] Nick Benton, Andrew Kennedy & Carsten Varming (2009): *Some domain theory and denotational semantics in Coq*. In: *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 115–130, doi:10.1007/978-3-642-03359-9_10.

[10] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer & Kristian Støvring (2012): *First steps in synthetic guarded domain theory: step-indexing in the topos of trees*. *Logical Methods in Computer Science* 8(4), doi:10.2168/LMCS-8(4:1)2012.

[11] Aleš Bizjak, Lars Birkedal & Marino Miculan (2014): *A Model of Countable Nondeterminism in Guarded Type Theory*. In: *Rewriting and Typed Lambda Calculi*, Springer, pp. 108–123, doi:10.1007/978-3-319-08918-8_8.

[12] Venanzio Capretta (2005): *General Recursion via Coinductive Types*. *Logical Methods in Computer Science* Volume 1, Issue 2, doi:10.2168/LMCS-1(2:1)2005.

[13] James Chapman, Tarmo Uustalu & Niccolò Veltri (2019): *Quotienting the delay monad by weak bisimilarity*. *Mathematical Structures in Computer Science* 29(1), pp. 67–92, doi:10.1017/S0960129517000184.

[14] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2018): *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, doi:10.4230/LIPIcs.TYPES.2015.5.

[15] Ugo Dal Lago, Francesco Gavazzo & Paul Blain Levy (2017): *Effectful applicative bisimilarity: Monads, relators, and Howe's method*. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, pp. 1–12, doi:10.1109/LICS.2017.8005117.

[16] Nils Anders Danielsson (2012): *Operational semantics using the partiality monad*. In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pp. 127–138, doi:10.1145/2364527.2364546.

[17] Pietro Di Gianantonio, Furio Honsell & Gordon Plotkin (1995): *Uncountable limits and the lambda calculus*.

[18] Robert Dockins (2014): *Formalized, effective domain theory in coq*. In: *International Conference on Interactive Theorem Proving*, Springer, pp. 209–225, doi:10.1007/978-3-319-08970-6_14.

[19] Dan Frumin, Herman Geuvers, Léon Gondelman & Niels van der Weide (2018): *Finite sets in homotopy type theory*. In June Andronick & Amy P. Felty, editors: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, ACM, pp. 201–214, doi:10.1145/3167085.

[20] Douglas J Howe (1989): *Equality in lazy computation systems*. In: *LICS*, 89, pp. 198–203, doi:10.1109/LICS.1989.39174.

[21] Martin Hyland & John Power (2006): *Discrete Lawvere theories and computational effects*. Theoretical Computer Science 366(1-2), pp. 144–162, doi:10.1016/j.tcs.2006.07.007.

[22] Tom de Jong & Martín Hötzel Escardó (2021): *Domain Theory in Constructive and Predicative Univalent Foundations*. In Christel Baier & Jean Goubault-Larrecq, editors: *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, Leibniz International Proceedings in Informatics (LIPIcs) 183, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 28:1–28:18, doi:10.4230/LIPIcs.CSL.2021.28.

[23] Nicolai Kraus (2014): *The General Universal Property of the Propositional Truncation*. In Hugo Herbelin, Pierre Letouzey & Matthieu Sozeau, editors: *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*, LIPIcs 39, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 111–145, doi:10.4230/LIPIcs.TYPES.2014.111.

[24] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg & Andrea Vezzosi (2021): *Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks*. arXiv:2102.01969.

[25] Søren B Lassen & Corin S Pitcher (1998): *Similarity and bisimilarity for countable non-determinism and higher-order functions*. Electronic Notes in Theoretical Computer Science 10, pp. 246–266, doi:10.1016/S1571-0661(05)80704-2.

[26] Søren Bøgh Lassen (1998): *Relational reasoning about functions and nondeterminism*. Ph.D. thesis, University of Aarhus.

[27] Bassel Mannaa, Rasmus Ejlers Møgelberg & Niccolò Veltri (2020): *Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory*. Logical Methods in Computer Science 16, doi:10.23638/LMCS-16(4:17)2020.

[28] Rasmus Ejlers Møgelberg & Marco Paviotti (2016): *Denotational semantics of recursive types in synthetic guarded domain theory*. In: *LICS*, doi:10.1145/2933575.2934516.

[29] Rasmus Ejlers Møgelberg & Niccolò Veltri (2019): *Bisimulation as path type for guarded recursive types*. Proceedings of the ACM on Programming Languages 3(POPL), pp. 1–29, doi:10.1145/3290317.

[30] Eugenio Moggi (1991): *Notions of computation and monads*. Information and computation 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[31] C-HL Ong (1993): *Non-determinism in a functional setting*. In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 275–286, doi:10.1109/LICS.1993.287580.

[32] Marco Paviotti, Rasmus Ejlers Møgelberg & Lars Birkedal (2015): *A model of PCF in guarded type theory*. Electronic Notes in Theoretical Computer Science 319, pp. 333–349, doi:10.1016/j.entcs.2015.12.020.

[33] Andrew M Pitts (1996): *Relational properties of domains*. Information and computation 127(2), pp. 66–90, doi:10.1006/inco.1996.0052.

[34] Andrew M Pitts (1997): *A note on logical relations between semantics and syntax*. Logic Journal of the IGPL 5(4), pp. 589–601, doi:10.1093/jigpal/5.4.589.

[35] Jan Schwinghammer, Aleš Bizjak & Lars Birkedal (2013): *Step-indexed relational reasoning for countable nondeterminism*. Logical Methods in Computer Science 9, doi:10.2168/LMCS-9(4:4)2013.

[36] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[37] Tarmo Uustalu (2015): *Stateful runners of effectful computations*. Electronic Notes in Theoretical Computer Science 319, pp. 403–421, doi:10.1016/j.entcs.2015.12.024.

[38] Andrea Vezzosi, Anders Mörtberg & Andreas Abel (2019): *Cubical Agda: a dependently typed programming language with univalence and higher inductive types*. Proceedings of the ACM on Programming Languages 3(ICFP), pp. 1–29, doi:10.1145/3341691.

[39] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce & Steve Zdancewic (2019): *Interaction trees: representing recursive and impure programs in Coq*. *Proceedings of the ACM on Programming Languages* 4(POPL), pp. 1–32, doi:10.1145/3371119.