Kent Academic Repository

Full text document (pdf)

Citation for published version

Møgelberg, Rasmus and Paviotti, Marco (2018) Denotational semantics of recursive types in synthetic guarded domain theory. Mathematical Structures in Computer Science . ISSN 0960-1295.

DOI

https://doi.org/10.1017/S0960129518000087

Link to record in KAR

https://kar.kent.ac.uk/69685/

Document Version

Author's Accepted Manuscript

Under consideration for publication in Math. Struct. in Comp. Science

Denotational semantics of recursive types in synthetic guarded domain theory

RASMUS E. MØGELBERG[1] [†] and MARCO PAVIOTTI[2][‡]

[1] *IT University of Copenhagen, Copenhagen, Denmark.*
[2] *University of Kent, Canterbury, United Kingdom.*

Received 10 April 2018

Just like any other branch of mathematics, denotational semantics of programming languages should be formalised in type theory, but adapting traditional domain theoretic semantics, as originally formulated in classical set theory to type theory has proven challenging. This paper is part of a project on formulating denotational semantics in type theories with guarded recursion. This should have the benefit of not only giving simpler semantics and proofs of properties such as adequacy, but also hopefully in the future to scale to languages with advanced features, such as general references, outside the reach of traditional domain theoretic techniques.

Working in *Guarded Dependent Type Theory* (GDTT), we develop denotational semantics for FPC, the simply typed lambda calculus extended with recursive types, modelling the recursive types of FPC using the guarded recursive types of GDTT. We prove soundness and computational adequacy of the model in GDTT using a logical relation between syntax and semantics constructed also using guarded recursive types. The denotational semantics is intensional in the sense that it counts the number of unfold-fold reductions needed to compute the value of a term, but we construct a relation relating the denotations of extensionally equal terms, i.e., pairs of terms that compute the same value in a different number of steps. Finally we show how the denotational semantics of terms can be executed inside type theory and prove that executing the denotation of a boolean term computes the same value as the operational semantics of FPC.

Contents

[†] This research was supported by The Danish Council for Independent Research for the Natural Sciences (FNU), Grant no. 4002-00442.
[‡] Marco Paviotti was funded in part by EPSRC grant EP/M017176/1.

1. Introduction

Recent years have seen great advances in formalisation of mathematics in type theory, in particular with the development of homotopy type theory [Uni13]. Such formalisations are an important step towards machine assisted verification of mathematical proofs. Rather than adapting classical set theory-based mathematics to type theory, new synthetic approaches sometimes offer simpler and clearer presentations in type theory. As an example of the synthetic approach, consider synthetic homotopy theory [Uni13], which formalises homotopy theory in type theory, not by formalising a topological space as a type with structure, but rather by thinking of types as topological spaces directly. Particular spaces such as the circle can then be constructed as types using higher inductive types. Synthetic homotopy theory can be formally related to classical homotopy theory via the simplicial sets interpretation of homotopy type theory [KL12], interpreting types essentially as topological spaces.

Just like any other branch of mathematics, domain theory and denotational semantics for programming languages with recursion should be formalised in type theory and, as was the case of homotopy theory, synthetic approaches can provide clearer and more abstract proofs. In the case of domain theory, the synthetic approach means treating types as domains, rather than constructing domains internally in type theory as types with an order relation. The result of this should be a considerable simplification of denotational semantics when expressed in type theory. For example, function types of a higher-order object language can be modelled simply as the function types of type theory, rather than as some type of Scott continuous maps. To model recursion, some form of fixed point construction must be added to type theory, but, as is well known, an unrestricted fixed point combinator makes the logical reading of type theory inconsistent.

1.1. *Synthetic guarded domain theory*

In this paper we follow the approach of guarded recursion [Nak00], which introduces a new type constructor $\triangleright$, pronounced "later". Elements of $\triangleright A$ are to be thought of as elements of type A available only one time step from now, and the introduction form $\mathsf{next}\colon A \to \triangleright A$ makes anything available now, also available later. The fixed point operator has type

$$\mathsf{fix}\colon (\triangleright A \to A) \to A$$

and maps an f to a fixed point of $f \circ \mathsf{next}$. Guarded recursion also assumes solutions to all guarded recursive type equations, i.e., equations where all occurences of the type variable are under a $\triangleright$, as for example in the equation

$$LA \cong A + \triangleright LA \tag{1}$$

used to define the lifting monad L below, but guarded recursive equations can also have negative or even non-functorial occurences.

One application of guarded recursion is for programming with coinductive types. This requires a notion of clocks used to index delays. For example, if κ is a clock and A is a type then $\triangleright_\kappa A$ is a type. If κ is a clock variable not free in A and $LA \cong A + \triangleright_\kappa LA$, then κ can be universally quantified in LA to give the type $\forall\kappa.LA$ which can be shown to be a coinductive solution to $\forall\kappa.LA \cong A + \forall\kappa.LA$. Almost everything we do in this paper uses a single implicit clock variable and all uses of $\triangleright$ should be thought of as indexed by this clock. More details can be found in Section 2.

Recent work has shown how guarded recursion can be used to construct syntactic models and operational reasoning principles for (also combinations of) advanced programming language features including general references, recursive types, countable nondeterminism and concurrency [Bir+12; BBM14; SB14]. These models often require solving recursive domain equations which are beyond the reach of domain theoretic methods. When viewing these syntactic models through the topos of trees model of guarded recursion [Bir+12] one recovers step-indexing [AM01], a technique for sidestepping recursive domain equations by indexing the interpretation of types by numbers, counting the number of unfoldings of the equation. Thus guarded recursion can be more accurately described as synthetic step-indexing. Indeed, guarded recursion provides a type system for constructing step-indexed models, in which the type equations sidestepped by step-indexing can be solved using guarded recursive types.

This work is part of a programme of developing *denotational semantics* using guarded recursion with the expectation that this will not only be simpler to formalise in type theory than the classical domain theoretic semantics, but also generalise to languages with advanced features for which step-indexing has been used for operational reasoning.

This programme was initiated in previous work [PMB15], in which a model of PCF (simply typed lambda calculus with fixed points) was developed in Guarded Dependent Type Theory (GDTT) [Biz+16] an extensional type theory with guarded recursive types and terms. By aligning the fixpoint unfoldings of PCF with the steps of the metalanguage (represented by $\triangleright$), we proved a computational adequacy result for the model inside type theory. Guarded recursive types were used both in the denotational semantics (to define a lifting monad) and in the proof of computational adequacy. Likewise, the fixed point operator fix of GDTT was used both to model fixed points of PCF and as a proof principle.

1.2. *Contributions*

Here we extend our previous work in two ways. First we extend the denotational semantics and adequacy proof to languages with recursive types. Secondly, we define a relation capturing extensionally equal elements in the model.

More precisely, we consider the language FPC (simply typed lambda calculus extended with general recursive types) with a call-by-name operational semantics. Working internally in GDTT this language can be given a denotational semantics in the synthetic style discussed above. In particular, function types of FPC are interpreted simply as the function types of GDTT. Base types are interpreted using the lifting monad L satisfying the isomorphism (1). In particular the unit type of FPC is interpreted as $L1$ isomorphic to $1 + \triangleright L1$, so that denotationally, a program of this type is either a value now, or a delayed computation. Recursive types are modelled as guarded recursive types satisfying the isomorphism

$$[\![\mu\alpha.\sigma]\!] \cong \triangleright [\![\sigma[\mu\alpha.\sigma/\alpha]]\!]$$

(in the case of closed types). This means that the introduction rule for recursive types (folding a term) can be interpreted as next. To interpret unfolding of terms of recursive types we construct, for every FPC type σ a map $\theta_\sigma : \triangleright [\![\sigma]\!] \to \sigma$, and interpret unfolding as $\theta_{\sigma[\mu\alpha.\sigma/\alpha]}$. As a consequence, folding followed by unfolding is interpreted as the map $\delta_{\sigma[\mu\alpha.\sigma/\alpha]}$ defined as $\theta_{\sigma[\mu\alpha.\sigma/\alpha]} \circ \mathsf{next}$. This composition is not the identity, rather the denotational semantics counts the number of fold-unfold reductions needed to evaluate a term to a value.

Thus, to state a precise soundness theorem, the operational semantics also needs to count the fold-unfold reductions. To do this, we define a judgement $M \to_*^k N$ to mean that M reduces to N in a sequence of reductions containing exactly k fold-unfold reductions, and an equivalent big-step semantics $M \Downarrow^k v$. One might hope to formulate an adequacy theorem stating that for M of type 1, $M \Downarrow^k \langle\rangle$ (where $\langle\rangle$ is the introduction form for 1) if and only if $[\![M]\!] = \delta^k\,[\![\langle\rangle]\!]$. Unfortunately this is not true. For example, if $M \Downarrow^2 \langle\rangle$ the type $M \Downarrow^1 \langle\rangle$ is empty, but the identity type $[\![M]\!] = \delta^1\,[\![\langle\rangle]\!]$ is equivalent to $\triangleright 0$, a non-standard truth value different from 0. To state an exact correspondence between the operational and denotational semantics we use the *guarded transitive closure of the small-step semantics* which *synchronises* the steps of FPC with those of GDTT. This is defined as $M \Rightarrow^{k+1} N$ if $M \to_*^0 M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^k N)$, where $M' \to^1 M''$ is a fold-unfold reduction in an evaluation context.

The adequacy theorem states that $M \Rightarrow^k \langle\rangle$ if and only if $[\![M]\!] = \delta^k\, [\![\langle\rangle]\!]$. We prove this working inside GDTT, and the proof shows an interesting aspect of guarded domain theory: It uses a logical relation between syntax and semantics defined by induction over the structure of types. The case of recursive types requires a solution to a recursive type equation. In the setting of classical domain theory, the existence of this solution requires a separate argument [Pit96], but here it is simply a guarded recursive type.

The second contribution is a relation capturing extensionally equal elements in the model. As mentioned above, the denotational semantics distinguishes between computations computing the same value in a different number of steps. In this paper we construct a relation on the denotational semantics of each type relating elements extensionally equal elements, i.e., elements that compute the same value in a different number of steps. This relation is defined on the *global interpretation of types* $[\![\sigma]\!]^{\mathrm{gl}}$ defined from $[\![\sigma]\!]$ by quantifying over the implicit clock variable (see Section 1.1 above). This is necessary, because, as can be seen from the denotational semantics of guarded recursion, any relation on $[\![1]\!]$ relating $[\![\langle\rangle]\!]$ to $\delta^n\, [\![\langle\rangle]\!]$ for any n will also necessarily relate non-termination to $[\![\langle\rangle]\!]$. On the other hand, it is possible to define such a relation on $[\![1]\!]^{\mathrm{gl}}$ which is the coinductive solution to $[\![1]\!]^{\mathrm{gl}} \cong 1 + [\![1]\!]^{\mathrm{gl}}$. This is then lifted to function types in the usual way for logical relations: Two functions are related it they map related elements to related elements, and to recursive types using a solution to a guarded recursive type equation. We prove a soundness result for this relation stating that if the (global) denotation of two terms are related, then the terms are contextually equivalent.

Finally we show that it is possible to execute the denotational semantics. Of course, FPC is a non-total programming language, so to run FPC programs in type theory, these must be given a time-out to ensure termination. We demonstrate the technique in the case of boolean typed programs and show that the denotation of a program executes to true with a time-out of n steps if and only if the program evaluates to true in less than n steps in the operational semantics.

All constructions and proofs are carried out working informally in GDTT. This work illustrates the strength of GDTT , and indeed influenced the design of the type theory.

1.3. *Related work*

Escardó constructs a model of PCF using a category of ultrametric spaces [Esc99]. Since this category can be seen as a subcategory of the topos of trees [Bir+12], our previous work on PCF is a synthetic version of Escardó's model. Escardó's model also distinguishes between computations computing the same value in a different number of steps, and captures extensional behaviour using a logical relation similar to the one constructed here. Escardó however, does not consider recursive types. Although Escardó's model was useful for intuitions, the synthetic construction in type theory presented here is very different, in particular the proof of adequacy, which here is formulated in guarded dependent type theory.

Synthetic approaches to domain theory have been developed based on a wide range of models dating back to [Hyl91; Ros86]. Indeed, the internal languages of these models can be used to construct models of FPC and prove computational adequacy [Sim02]. A more axiomatic approach was developed in Reus's work [Reu96] where an axiomatisation of domain theory is postulated a priori inside the Extended Calculus of Constructions.

There has also been work on (non-synthetic) adaptations of domain theory to type theory [BKV09; Ben+10; Doc14]. However, due to the mistmatch between set-theory and type theory "*some of the proofs and constructions are much more complex than they would classically and one does sometimes have to pay attention to which of two classically-equivalent forms of definition one works with*" [BKV09]. More recently Altenkirch et al. [ADK17] have shown how to encode the free pointed ω-cpo as a quotient inductive-inductive types (QIIT). This looks like a more promising direction for domain theory in type theory, but this has not yet been developed to models of programming languages.

The lifting monad used in this paper is a *guarded* recursive variant of Capretta's delay monad [Cap05] considered by among others [BKV09; Ben+10; Dan12; CUV15; ADK17; Vel17]. The monad $D(A)$ is coinductively generated by the constructors $\texttt{now} : A \to D(A)$ and $\texttt{later} : D(A) \to D(A)$. As reported by Danielsson [Dan12], working with the partiality monad requires convincing Agda of productivity of coinductive definitions using workarounds. In this paper productivity is ensured by the type system for guarded recursion.

In the delay monad, two computations of type $D(A)$ can be distinguished by their number of steps. To address this issue, Capretta also defines a weak bisimulation on this monad, similar to the one defined in Definition 6.2, and proves the combination of the delay monad with the weak bisimulation is a monad using setoids. Chapman et al.[CUV15; Vel17] avoid using setoids, but they crucially rely on proposition extensionality and the axiom of countable choice. Altenkirch et al. [ADK17] show that under the assumption of countable choice, their free pointed ω-cpo construction is equivalent to quotiented delay monad of Chapman et al. We work crucially with the non-quotiented delay monad when defining the denotational semantics, since the steps are necessary for guarded recursion.

This is an extended version of a conference publication [MP16]. A number of proofs that were omitted from the previous version due to space restrictions have been included in this version. There is also a slight difference in approach: the conference version defined a big-step operational semantics equivalent to the guarded transitive closure of the small-step operational semantics of Figure 2 below. This operational semantics synchronises the steps of FPC with those of the meta-language, and capturing this in a big-step semantics was quite tricky. Here, instead, we define a simpler big-step operational semantics and prove this equivalent to the "global" small-step semantics (Lemma 3.2). The results on executing the denotational semantics presented in Section 7 are also new.

Since this work was carried out, the extensional type theory GDTT that we work in in this paper has been extended in two directions towards intensionality and implementation. The first direction is Guarded Cubical Type Theory [Bir+16], extending the fragment of GDTT without universal quantification over clocks with constructions from Cubical Type Theory [Coh+16]. Guarded Cubical Type Theory even has a prototype implementation. The other direction is Clocked Type Theory [BGM17], a variant of the fragment of GDTT without identity types in which delayed substitutions (Section 5.1) are encoded using a new notion of ticks on a clock. Clocked Type Theory has a strongly normalising reduction semantics. Since neither theory is complete, we stick to GDTT as our type theory for this paper.

The paper is organized as follows. Section 2 gives a brief introduction to the most important concepts of GDTT. More advanced constructions of the type theory are introduced as needed. Section 3 defines the encoding of FPC and its operational semantics in GDTT. The denotational semantics is defined and soundness is proved in Section 4. Computational adequacy is proved in Section 5, and the relation capturing extensional equivalence is defined in Section 6. Section 7 shows how to execute the denotational semantics of boolean programs. We conclude and discuss future work in Section 8.

Acknowledgements. We thank Nick Benton, Lars Birkedal, Aleš Bizjak, and Alex Simpson for helpful discussions and suggestions.

2. Guarded recursion

In this paper we work informally within a type theory with dependent types, inductive types and guarded recursion. Although inductive types are not mentioned in [Biz+16] the ones used here can be safely added – as they can be modelled in the topos of trees model – and so the arguments of this paper can be formalised in Guarded Dependent Type Theory (GDTT) [Biz+16]. We start by recalling some core features of this theory, but postpone delayed substitutions to Section 5.1 since these are not needed for the moment.

When working in type theory, we use $\equiv$ for judgemental equality of types and terms and $=$ for propositional equality (sometimes $=_A$ when we want to be explicit about the type). We also use $=$ for (external) set theoretical equality.

The core of guarded recursion consists of the type constructor $\triangleright$ and the fixed point operator $\mathsf{fix} : (\triangleright A \to A) \to A$ satisfying

$$\mathsf{fix}\, f = f(\mathsf{next}(\mathsf{fix}(f))) \tag{2}$$

both introduced in Section 1.1. Elements of type $\triangleright A$ are intuitively elements of type A available one time step from now. To illustrate the power of the fixed point operator, consider a type of guarded streams Str_g satisfying

$$\mathsf{Str}_g \cong \mathbb{N} \times \triangleright \mathsf{Str}_g \tag{3}$$

This is a guarded recursive type in the sense that the recursion variable appears under a $\triangleright$, and its elements are to be thought of as streams, whose head is immediately available and whose tails take one time step to compute. The fixed point operator can be used to define guarded streams by recursion. For example, the constant stream of a number n can be defined as $\mathsf{fix}(\lambda x.\, \langle n, x\rangle)$, where the type isomorphism (3) is left implicit. Note that the type of the fixed point operator prevents us from defining elements like $\mathsf{fix}(\lambda x.x)$, which are not productive, in the sense that any element of the stream can be computed in finite time. In fact, the type $\triangleright \mathsf{Str}_g \to \mathsf{Str}_g$ precisely captures productive recursive stream definitions.

The type constructor $\triangleright$ is an applicative functor in the sense of [MP08], which means that there is a "later application" $\circledast \colon \triangleright(A \to B) \to \triangleright A \to \triangleright B$ written infix, satisfying

$$\mathsf{next}(f) \circledast \mathsf{next}(t) \equiv \mathsf{next}(f(t)) \tag{4}$$

among other axioms (see also [BM13]). In particular, $\triangleright$ extends to a functor mapping $f \colon A \to B$ to $\lambda x \colon \triangleright A.\, \mathsf{next}(f) \circledast x$. Moreover, the $\triangleright$ operator distributes over the identity type as follows

$$\triangleright(t =_A u) \equiv (\mathsf{next}\, t =_{\triangleright A} \mathsf{next}\, u) \tag{5}$$

Guarded dependent type theory comes with universes in the style of Tarski. In this paper, we will just use a single universe $\mathcal{U}$. Readers familiar with [Biz+16] should think of this as $\mathcal{U}_\kappa$, but since we work with a unique clock κ, we will omit the subscript. The universe comes with codes for type operations, including $\widehat{+} \colon \mathcal{U} \times \mathcal{U} \to \mathcal{U}$ for binary sum types, codes for dependent sums and products, and $\widehat{\triangleright} \colon \triangleright\mathcal{U} \to \mathcal{U}$ satisfying

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}(A))) \equiv \triangleright\, \mathsf{El}(A) \tag{6}$$

where we use $\mathsf{El}(A)$ for the type corresponding to an element $A \colon \mathcal{U}$. The type of $\widehat{\triangleright}$ allows us to solve recursive type equations using the fixed point combinator. For example, if A is small, i.e., has a code $\widehat{A}$ in $\mathcal{U}$, the type equation (1) can be solved by computing a code of LA as

$$\widehat{L}\, A = \mathsf{fix}(\lambda X \colon \triangleright\mathcal{U}.\, \widehat{+}(\widehat{A}, \widehat{\triangleright}X)) \tag{7}$$

and then by taking the elements using El. More precisely, defining LA as $\mathsf{El}(\widehat{L}\, A)$, LA unfolds to $\mathsf{El}(\widehat{+}(\widehat{A}, \widehat{\triangleright}(\mathsf{next}(\widehat{L}\, A))))$ which is equal to $A + \mathsf{El}(\widehat{\triangleright}(\mathsf{next}(\widehat{L}\, A)))$ which is equal to $A + \triangleright LA$. In this paper, we will only apply the monad L to small types A.

To ease presentation, we will usually not distinguish between types and type operations on the one hand, and their codes on the other. We will still refer use the notation $\widehat{\triangleright} \colon \triangleright\mathcal{U} \to \mathcal{U}$, but write $\triangleright$ for the composition $\widehat{\triangleright} \circ \mathsf{next}$. We generally leave El implicit.

2.1. *The topos of trees model*

The topos $\mathcal{S}$ of trees is the category of presheaves over ω, the first infinite ordinal. The category $\mathcal{S}$ models guarded recursion [Bir+12] and provides useful intuitions, and so we briefly recall it.

A closed type is modelled as an object of the topos of trees, i.e., as a family of sets $X(n)$ indexed by natural numbers together with *restriction maps* $r_n^X : X(n+1) \to X(n)$ as in the following diagram

$$X(1) \longleftarrow X(2) \longleftarrow X(3) \longleftarrow X(4) \longleftarrow \dots \tag{8}$$

A term of type Y in context $x : X$, for X, Y closed types, is modelled as a morphism in $\mathcal{S}$, i.e., as a family of functions $f_i : X(i) \to Y(i)$ obeying the *naturality* condition

$f_i \circ r_i^X = r_i^Y \circ f_{i+1}$ as in the following diagram

$$\begin{tikzcd}
X(i) \arrow[d, "f_i"] & X(i+1) \arrow[l, "r_i^X"'] \arrow[d, "f_{i+1}"] \\
Y(i) & Y(i+1) \arrow[l, "r_i^Y"']
\end{tikzcd} \tag{9}$$

The $\triangleright$ type operator is modelled as an endofunctor in $\mathcal{S}$ such that $\triangleright X(1) = 1$, $\triangleright X(n+1) = X(n)$. Intuitively, $X(n)$ is the nth approximation for computations of type X, thus $X(n)$ describes the type X as it looks if we have n computational steps to reason about it.

Using the proposition-as-types principle, types like $\triangleright^3 0$ are non-standard truth values. Following the intuition that $\triangleright^3 0(n)$ is the type $\triangleright^3 0$ as it looks, if we have n steps to reason about it, $\triangleright^3 0$ is the truth value of propositions that appear true for 3 computation steps, but then are falsified after 4. In fact, in the model, $(\triangleright^3 0)(3)$ equals 1, but $(\triangleright^3 0)(4)$ equals 0 zero as depicted by the following diagram

$$\begin{tikzcd}
1 & 1 \arrow[l] & 1 \arrow[l] & 0 \arrow[l] & 0 \arrow[l] & \dots \arrow[l]
\end{tikzcd} \tag{10}$$

The *global elements* of a closed type X is the set of morphisms from the constant object 1 to X in $\mathcal{S}$. This can be thought of as the *limit* of the sequence of (8) as a diagram in **Set**. This construction gives us the *global view* of a type as it allows us to observe *all the computation at once*. For example, the global elements of $\triangleright X$ correspond to those of X simply by discarding the first component. Note that objects can have equal sets of global elements without being isomorphic. In particular 0 and $\triangleright^n 0$ are not isomorphic.

For guarded recursive type equations, $X(n)$ describes the nth unfolding of the type equation. For example, fixing an object A, the unique solution to (1) is

$$LA(n) = 1 + A(1) + \cdots + A(n)$$

with restriction maps defined using the restriction maps of A. In particular, if A is a constant presheaf, i.e., $A(n) = X$ for some fixed X and r_n^A identities, then we can think of $LA(n)$ as $\{0, \dots, n-1\} \times X + \{\bot\}$ with restriction map given by $r_n(\bot) = \bot$, $r_n(n, x) = \bot$ and $r_n(i, x) = (i, x)$ for $i < n$. The set of global elements of LA is then isomorphic to $\mathbb{N} \times X + \{\bot\}$. In particular, if $X = 1$, the set of global elements is $\bar{\omega}$, the natural numbers extended with a point at infinity.

The global elements of LA, correspond to the elements of Capretta's partiality monad [Cap05] L^{gl} defined as the coinductive solution to the type equation

$$L^{\mathrm{gl}} A \cong A + L^{\mathrm{gl}} A \tag{11}$$

Similarly, the type of Str_g can be modelled as $\mathsf{Str}_g(n) = \mathbb{N}^n \times 1$. Note that if these products associate to the right, we can even model (3) as an identity. The restriction maps of this type are projections, and the global elements of this type correspond to streams in the usual sense.

2.2. *Universal quantification over clocks*

The type of guarded streams Str_g mentioned above, is not the usual coinductive type of streams. For example, a term $t : \mathsf{Str}_g$ in context $x : \mathsf{Str}_g$ is a causal function of streams, i.e., one where the n first elements of the output depend only on the n first elements of the input. This can be seen e.g. in the topos of trees model, where such a term is modelled by a family of maps $f_n : \mathbb{N}^n \times 1 \to \mathbb{N}^n \times 1$ commuting with projections. Causality is crucial to the encoding of productivity in types mentioned above.

On the other hand, a *closed* term $t : \mathsf{Str}_g$ is modelled by a global element of Str_g and thus corresponds to a real stream of numbers. Likewise, if $t : \mathsf{Str}_g$ only depends on a variable $x : \mathbb{N}$, then t denotes a map from the set of natural numbers to that of streams, because the context is modelled as the constant topos of trees object $\mathbb{N}$, with restriction maps being identities. More generally, say a context is *independent of time* if it is modelled as a constant object, i.e, one where all restriction maps are isomorphisms. The denotation of a term $t : \mathsf{Str}_g$ in a context Γ independent of time, corresponds to a map from $\Gamma(1)$ to the set of streams.

The idea of independence of time can be captured syntactically using a notion of clocks, and universal quantification over these [AM13]. We now briefly recall this as implemented in GDTT, referring to [Biz+16] for details.

In GDTT all types and terms are typed in a clock context, i.e., a finite set of names of clocks. For each clock κ, there is a type constructor $\triangleright_\kappa$, a fixed point combinator, and so on. Each clock carries its own notion of time, and the idea of a context being independent of time mentioned above, can be captured as a clock not appearing in a context.

If A is a type in a context where κ does not appear, one can form the type $\forall\kappa.A$, binding κ. This construction behaves in many ways similarly to polymorphic quantification over types in System F. There is an associated binding introduction form $\Lambda\kappa.(-)$ (applicable to terms where κ does not appear free in the context), and elimination form $t[\kappa']$ having type $A[\kappa'/\kappa]$ whenever $t \colon \forall\kappa.A$.

Semantically, a closed type in the empty clock variable context is modelled by a set, and a type in a context of a single clock is modelled as an object in the topos of trees. In the latter case, universal quantification over the single clock is modelled by taking the set of global elements. As we saw above, these sets correspond to coinductive types, and this also holds in the type theory: If Str_g is the type of streams guarded on clock κ, i.e., satisfies $\mathsf{Str}_g \cong \mathbb{N} \times \triangleright_\kappa \mathsf{Str}_g$, then one can prove [AM13; Møg14] that the type $\forall\kappa.\, \mathsf{Str}_g$ behaves as a coinductive type of streams. Similarly, if $LA \cong A + \triangleright_\kappa LA$, and κ is not free in A, then $\forall\kappa.LA$ is a coinductive solution to $X \cong A + X$. This isomorphism arises as a composite of isomorphisms

$$\begin{aligned}
\forall\kappa.LA &\cong \forall\kappa.(A + \triangleright_\kappa LA) \\
&\cong (\forall\kappa.A) + (\forall\kappa.\triangleright_\kappa LA) \qquad (12)\\
&\cong A + \forall\kappa.\triangleright_\kappa LA \qquad (13)\\
&\cong A + \forall\kappa.LA \qquad (14)
\end{aligned}$$

the components of which we recall below. Using these encodings one can use guarded recursion to program with coinductive types in such a way that typing guarantees productivity. We refer to [BM15] for a full model of guarded recursion with clocks, in particular for how to model types with more than one free clock variable.

The isomorphism (14) arises from a general type isomorphism $\forall\kappa.\triangleright_\kappa A \cong \forall\kappa.A$ holding for all A. The direction from right to left is induced by $\mathsf{next}^\kappa : A \to \triangleright_\kappa A$. For the direction from left to right, a form of elimination for $\triangleright_\kappa$ is needed, but note that an unrestricted such of type $\triangleright_\kappa A \to A$ in combination with fixed points makes the type system inconsistent. Instead GDTT allows for a restricted elimination rule for $\triangleright_\kappa$: If t is of type $\triangleright_\kappa A$ in a context where κ does not appear free, then $\mathsf{prev}\,\kappa.t$ has type $\forall\kappa.A$. Using $\mathsf{prev}\,\kappa.$ we can define a term force:

$$\begin{aligned}&\mathsf{force} : (\forall\kappa.\triangleright_\kappa A) \to \forall\kappa.A\\ &\mathsf{force} \overset{\text{def}}{=\!=} \lambda x.\,\mathsf{prev}\,\kappa.x[\kappa]\end{aligned} \tag{15}$$

The term force can be proved to be an isomorphism by the axioms

$$\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa(t) \equiv \Lambda\kappa.t \qquad \mathsf{next}^\kappa((\mathsf{prev}\,\kappa.t)[\kappa]) \equiv t \tag{16}$$

If κ is not free in A, the type $\forall\kappa.A$ is isomorphic to A, justifying the isomorphism (13). The map $A \to \forall\kappa.A$ is simply $\lambda x\colon A.\Lambda\kappa.x$. The other direction is given by application to a clock constant κ_0, which we assume exists. These can be proved to be inverses of each other using *the clock irrelevance axiom*, which states that if $t : \forall\kappa.A$ and κ does not appear free in A, then $t[\kappa'] \equiv t[\kappa'']$ for all κ' and κ''. Using force and the isomorphism $\forall\kappa.0 \cong 0$, one can prove that $\forall\kappa.\triangleright_\kappa^n 0$ is isomorphic to 0, reflecting the fact that there are no global elements of $\triangleright^n 0$ in the model, as mentioned earlier. We refer to [Biz+16] for details.

The isomorphism (12) is a special case of an isomorphism

$$\forall\kappa.(B + C) \cong (\forall\kappa.B) + (\forall\kappa.C) \tag{17}$$

distributing $\forall\kappa$ over sums for all small types B and C. To describe this isomorphism, encode sum types as $B + C \overset{\text{def}}{=\!=} \Sigma x : (1 + 1).[B, C](x)$ where $[B, C]$ is defined by cases by $[B, C](\mathsf{inl}(\star)) \equiv B$ and $[B, C](\mathsf{inr}(\star)) \equiv C$. The result of applying the left to right direction d of the isomorphism to $x : \forall\kappa.(B + C)$ is defined by cases of $\pi_1(x[\kappa_0]) : 1 + 1$. If $\pi_1(x[\kappa_0]) = \mathsf{inl}(\star)$, note that for any κ, using the clock irrelevance axiom

$$\pi_1(x[\kappa]) = (\Lambda\kappa.\pi_1(x[\kappa]))[\kappa] = (\Lambda\kappa.\pi_1(x[\kappa]))[\kappa_0] = \pi_1(x[\kappa_0]) = \mathsf{inl}(\star)$$

and so $\Lambda\kappa.\pi_2(x[\kappa])$ has type

$$\forall\kappa.[B, C](\pi_1(x[\kappa])) = \forall\kappa.[B, C](\mathsf{inl}(\star)) = \forall\kappa.C$$

and so we can define in this case $d(x) = \Lambda\kappa.\,\mathsf{inl}(\pi_2(x[\kappa]))$. The case of $\pi_1(x[\kappa_0]) = \mathsf{inr}(\star)$ is similar. In fact, this construction generalises to an isomorphism

$$\forall\kappa.\Sigma(x : A).B \cong \Sigma(x : A).\forall\kappa.B \tag{18}$$

valid whenever κ is not free in A.

Finally we note the following extensionality rule for quantification over clocks.

$$(t =_{\forall\kappa.A} u) \equiv \forall\kappa.(t[\kappa] =_A u[\kappa]) \tag{19}$$

In most of this paper we will work in a setting of a unique implicit clock κ, and simply write $\triangleright$ for $\triangleright_\kappa$ to avoid cluttering all definitions and calculations with clocks.

For the proof of computational adequacy we will need one more construction from GDTT: The delayed substitutions. These will be recalled in Section 5.1.

3. FPC

This section defines the syntax, typing judgements and operational semantics of FPC. These are inductive types in guarded type theory, but, as mentioned earlier, we work informally in type theory, and in particular remain agnostic with respect to choice of representation of syntax with binding.

The typing judgements of FPC are defined in an entirely standard way. The grammar for terms of FPC

$$\begin{aligned} L, M, N ::= &\langle\rangle \mid x \mid \mathtt{inl}\, M \mid \mathtt{inr}\, M \mid \mathtt{case}\, L\, \mathtt{of}\, x_1.M; x_2.N \mid \langle M, N\rangle \\ &\mid \mathtt{fst}\, M \mid \mathtt{snd}\, M \mid \lambda x : \tau.M \mid MN \mid \mathtt{fold}\, M \mid \mathtt{unfold}\, N \end{aligned}$$

should be read as an inductive type of terms in the standard way. Likewise the grammars for types and contexts and the typing judgements defined in Figure 1 should be read as defining inductive types in type theory, allowing us to do proofs by induction over e.g. typing judgements.

We denote by $\mathtt{Type}_{\mathtt{FPC}}$, $\mathtt{Term}_{\mathtt{FPC}}$ and $\mathtt{Value}_{\mathtt{FPC}}$ the types of *closed* FPC types and terms, and values of FPC and by $\mathtt{OTerm}_{\mathtt{FPC}}$ the type of all (also open) terms. By a value we mean a closed term matching the grammar

$$v ::= \langle\rangle \mid \mathtt{inl}\, M \mid \mathtt{inr}\, M \mid \langle M, N\rangle \mid \lambda x : \tau.M \mid \mathtt{fold}\, M$$

3.1. *Operational semantics*

Figure 2 defines a big-step and a small-step operational semantics for FPC, as well as two transitive closures of the latter. All these definitions should be read as inductive types. Since the denotational semantics of FPC is intensional, counting reduction steps, it is necessary to also count the steps in the operational semantics in order to state the soundness and adequacy theorems precisely. More precisely, the semantics counts the number of `unfold-fold` reductions in the same fashion in which Escardó counted fix-point reduction for PCF.

The statement

$$M \Downarrow^k v \tag{20}$$

where M is a term, k a natural number, and v a value, should be read as 'M evaluates in k steps to a value v. We can define more standard big-step evaluation predicates as follows

$$M \Downarrow v \overset{\text{def}}{=\!=} \Sigma k.M \Downarrow^k v$$

We note that the semantics is trivially deterministic.

Well formed types

$$\Theta \in \text{Type Contexts} \stackrel{\text{def}}{=\!=} \langle\rangle \mid \langle \Theta, \alpha \rangle$$

$$\frac{}{\vdash \langle\rangle} \qquad \frac{\vdash \Theta}{\vdash \Theta, \alpha} \alpha \notin \Theta$$

$$\frac{\vdash \Theta}{\Theta \vdash \Theta_i} 1 \leq i \leq \mid \Theta \mid \qquad \frac{\vdash \Theta}{\Theta \vdash 1}$$

$$\frac{\Theta, \alpha \vdash \tau}{\Theta \vdash \mu\alpha.\tau} \qquad \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \mathtt{op} \ \tau_2} \text{ for } \mathtt{op} \in \{+, \times, \rightarrow\}$$

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \cdot \vdash \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : \sigma \rightarrow \tau} \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash L : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash M : \sigma \quad \Gamma, x_2 : \tau_2 \vdash N : \sigma}{\Gamma \vdash \mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N : \sigma}$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ M : \tau_1} \qquad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\ e : \tau_2} \qquad \frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash M : \mu\alpha.\tau}{\Gamma \vdash \mathtt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Gamma \vdash M : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{fold}\ M : \mu\alpha.\tau}$$

Fig. 1. Syntax of FPC

Lemma 3.1. The small-step semantics is deterministic: if $M \rightarrow^k N$ and $M \rightarrow^{k'} N'$, then $k = k'$ and $N = N'$.

Of the two transitive closures of the small-step semantics defined in Figure 2 the first is a standard one, equivalent to the big-step operational semantics. The second is a guarded version which synchronises the steps of FPC with those of the metalogic. This is needed for the statement of the soundness and adequacy theorems, and also allows for guarded recursion to be used in the proofs of these. The next lemma states the relationship between the big-step semantics and the two transitive closures of the small-step semantics

Lemma 3.2. Let M and N be FPC terms, v a value and k a natural number. Then

1 $M \Downarrow^k v$ iff $M \rightarrow^k_* v$

2 $M \rightarrow^k_* N$ iff $\forall\kappa.M \Rightarrow^k N$

Note that in particular $M \rightarrow^k_* N$ implies $M \Rightarrow^k N$. The opposite implication does not hold, as we shall see in the examples below.

Proof. The first statement is a essentially a textbook result on operational semantics, and we omit the proof.

For the second statement the proof from left to right is by induction on $M \rightarrow^k_* N$. The case of $M = N$ is trivial, so consider the case when $M \rightarrow^k M'$ and $M' \rightarrow^m_* N$. When

Big-step semantics

$$\frac{}{v \Downarrow^0 v}$$

$$\frac{L \Downarrow^k \texttt{inl}\ L' \quad M[L'/x_1] \Downarrow^m v}{\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^{m+k} v} \qquad \frac{L \Downarrow^k \texttt{inr}\ L' \quad N[L'/x_2] \Downarrow^m v}{\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^{m+k} v}$$

$$\frac{L \Downarrow^k \langle M, N\rangle \quad M \Downarrow^l v}{\texttt{fst}\ L \Downarrow^{k+l} v} \qquad \frac{L \Downarrow^k \langle M, N\rangle \quad N \Downarrow^l v}{\texttt{snd}\ L \Downarrow^{k+l} v}$$

$$\frac{M \Downarrow^k \lambda x.L \quad L[N/x] \Downarrow^l v}{MN \Downarrow^{k+l} v} \qquad \frac{M \Downarrow^k \texttt{fold}\ N \quad N \Downarrow^m v}{\texttt{unfold}\ M \Downarrow^{k+m+1} v}$$

Small-step semantics

$$(\lambda x : \sigma.M)(N) \to^0 M[N/x] \qquad \texttt{unfold}\ (\texttt{fold}\ M) \to^1 M$$

$$\texttt{case}\ (\texttt{inl}\ L)\ \texttt{of}\ x_1.M; x_2.N \to^0 M[L/x_1]$$

$$\texttt{case}\ (\texttt{inr}\ L)\ \texttt{of}\ x_1.M; x_2.N \to^0 N[L/x_2]$$

$$\texttt{fst}\ \langle M, N\rangle \to^0 M \qquad \texttt{snd}\ \langle M, N\rangle \to^0 N$$

$$\frac{M_1 \to^k M_2 \quad k = 0, 1}{E[M_1] \to^k E[M_2]}$$

$$E ::= [\cdot] \mid EM \mid \texttt{case}\ E\ \texttt{of}\ x_1.M; x_2.N \mid \texttt{fst}\ E \mid \texttt{snd}\ E \mid \texttt{unfold}\ E$$

$$\frac{}{M \to_*^0 M} \qquad \frac{M \to^k M' \quad M' \to_*^m N}{M \to_*^{k+m} N}$$

Guarded transitive closure of the small-step semantics

$$\frac{M \to_*^0 N}{M \Rightarrow^0 N} \qquad \frac{M \to_*^0 M' \quad M' \to^1 M'' \quad \triangleright(M'' \Rightarrow^k N)}{M \Rightarrow^{k+1} N}$$

Fig. 2. Operational semantics for FPC.

$k = 0$, by definition $M \to_*^0 M'$, and by induction hypothesis we know that $\forall\kappa.M' \Rightarrow^m N$. Thus, $M \Rightarrow^m N$ holds for any κ, and so also $\forall\kappa.M \Rightarrow^m N$, since κ is not free in the assumption $M \to_*^k N$. When $k = 1$ by induction hypothesis $\forall\kappa.M' \Rightarrow^m N$ and thus, for any κ, $M \to^1 M'$ and $\triangleright_\kappa(M' \Rightarrow^m N)$. As before, this allows us to conclude $\forall\kappa.M \Rightarrow^{m+1} N$.

The right to left implication is proved by induction on k. When $k = 0$ the clock κ is not free in $M \Rightarrow^k N$ and so $\forall\kappa.M \Rightarrow^k N$ is isomorphic to $M \Rightarrow^k N$, which implies $M \to_*^k N$. When $k = k' + 1$ the assumption $\forall\kappa.M \Rightarrow^k N$ implies that $M \to_*^0 N'$, $N' \to^1 N''$ and $\forall\kappa.\triangleright_\kappa(N'' \Rightarrow^{k'} N)$. By the type isomorphism (15) the latter implies $\forall\kappa.(N'' \Rightarrow^{k'} N)$, which by the induction hypothesis implies $N'' \to_*^{k'} N$. Thus we conclude $M \to_*^k N$. □

3.2. *Examples*

As an example of a recursive FPC type, one can encode the natural numbers as

$$\begin{aligned} \mathbf{nat} &\stackrel{\text{def}}{=\!=} \mu\alpha.1+\alpha \\ \mathsf{zero} &\stackrel{\text{def}}{=\!=} \mathtt{fold}\,(\mathtt{inl}\ (\langle\rangle)) \\ \mathsf{succ}\ M &\stackrel{\text{def}}{=\!=} \mathtt{fold}\,(\mathtt{inr}\ (M)) \end{aligned}$$

Using this definition we can define the term ifz of PCF. If L is a term of type $\mathbf{nat}$ and M,N are terms of type σ define ifz as

$$\mathsf{ifz}\ L\ M\ N \stackrel{\text{def}}{=\!=} \mathtt{case}\ (\mathtt{unfold}\ L)\ \mathtt{of}\ x_1.M; x_2.N$$

where x_1, x_2 are fresh. It is easy to see that $\mathsf{ifz}\ \mathsf{zero}\ M\ N \Rightarrow^{k+1} v$ iff $\triangleright(M \Rightarrow^k v)$ and that $\mathsf{ifz}\ (\mathsf{succ}\ L)\ M\ N \Rightarrow^{k+1} v$ iff $\triangleright(N \Rightarrow^k v)$ for any L term of type $\mathbf{nat}$. For example, $\mathsf{ifz}\ 1\ 0\ 1 \Rightarrow^2 42$ is $\triangleright 0$. On the other hand, $\mathsf{ifz}\ 1\ 0\ 1 \to_*^2 42$ is equivalent to 0, showing that $\Rightarrow$ and $\to_*$ are not equivalent.

Recursive types introduce divergent terms. For example, given a type A, the Turing fixed point combinator on A can be encoded as follows:

$$\begin{aligned} B &\stackrel{\text{def}}{=\!=} \mu\alpha.(\alpha \to (A \to A) \to A) \\ \theta &: B \to (A \to A) \to A \\ \theta &\stackrel{\text{def}}{=\!=} \lambda x\lambda y.y(\mathtt{unfold}\ x\ x\ y) \\ \mathsf{Y}_A\ &\stackrel{\text{def}}{=\!=} \theta(\mathtt{fold}\ \theta) \end{aligned}$$

An easy induction shows that $(\mathsf{Y}_\sigma\ (\lambda x.x) \Rightarrow^k v) = \triangleright^k 0$, where 0 is the empty type.

If $M \to_*^k v$ with v a value and M a term, then

— $M \Rightarrow^k v$ is true
— $M \Rightarrow^n v$ is logically equivalent to $\triangleright^{\mathtt{min}(n,k)} 0$ if $n \neq k$, where 0 is the empty type

If, on the other hand, M is divergent in the sense that for any k there exists an N such that $M \to_*^k N$, then $M \Rightarrow^n v$ is equivalent to $\triangleright^n 0$.

4. Denotational Semantics

We now define the denotational semantics of FPC. First we recall the definition of the guarded recursive version of the *lifting monad* on types from [PMB15]. This is defined as the *unique* solution to the guarded recursive type equation

$$LA \cong A + \triangleright LA$$

which exists because the recursive variable is guarded by a $\triangleright$. Recall (Section 2) that guarded recursive types are defined as fixed points of endomaps on the universe, so LA is only defined for small types A. We will only apply L to small types in this paper.

The isomorphism induces a map $\theta_{LA} : \triangleright LA \to LA$ and a map $\eta \colon A \to LA$. An element of LA is either of the form $\eta(a)$ or $\theta(r)$. We think of these cases as values "now" or

$$\begin{aligned}
\llbracket \Theta \vdash \alpha \rrbracket (\rho) &\overset{\text{def}}{=\!=} \rho(\alpha) \\
\llbracket \Theta \vdash 1 \rrbracket (\rho) &\overset{\text{def}}{=\!=} L1 \\
\llbracket \Theta \vdash \tau_1 \times \tau_2 \rrbracket (\rho) &\overset{\text{def}}{=\!=} \llbracket \Theta \vdash \tau_1 \rrbracket (\rho) \times \llbracket \Theta \vdash \tau_2 \rrbracket (\rho) \\
\llbracket \Theta \vdash \tau_1 + \tau_2 \rrbracket (\rho) &\overset{\text{def}}{=\!=} L(\llbracket \Theta \vdash \tau_1 \rrbracket (\rho) + \llbracket \Theta \vdash \tau_2 \rrbracket (\rho)) \\
\llbracket \Theta \vdash \tau_1 \to \tau_2 \rrbracket (\rho) &\overset{\text{def}}{=\!=} \llbracket \Theta \vdash \tau_1 \rrbracket (\rho) \to \llbracket \Theta \vdash \tau_2 \rrbracket (\rho) \\
\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho) &\overset{\text{def}}{=\!=} \triangleright(\llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))
\end{aligned}$$

Fig. 3. Interpretation of FPC types

computations that "tick". Moreover, given $f : A \to B$ with B a $\triangleright$-algebra (i.e., equipped with a map $\theta_B \colon \triangleright B \to B$), we can lift f to a homomorphism of $\triangleright$-algebras $\hat{f} \colon LA \to B$ as follows

$$\begin{aligned}
\hat{f}(\eta(a)) &\overset{\text{def}}{=\!=} f(a) \\
\hat{f}(\theta(r)) &\overset{\text{def}}{=\!=} \theta_B(\mathsf{next}(\hat{f}) \circledast r)
\end{aligned} \tag{21}$$

Formally $\hat{f}$ is defined as a fixed point of a term of type $\triangleright(LA \to B) \to LA \to B$. Recall that $\lambda r.\, \mathsf{next}(\hat{f}) \circledast r$ is the application of the functor $\triangleright$ to the map $\hat{f}$, thus $\hat{f}$ is an algebra homomorphism.

Intuitively LA is the type of computations possibly returning an element of A, recording the number of steps used in the computation. We can define the divergent computation as $\bot \overset{\text{def}}{=\!=} \mathsf{fix}(\theta)$ and a "delay" map δ_{LA} of type $LA \to LA$ for any A as $\delta_{LA} \overset{\text{def}}{=\!=} \theta_{LA} \circ \mathsf{next}$. The latter can be thought of as adding a step to a computation. The lifting L extends to a functor. For a map $f : A \to B$ the action on morphisms can be defined using the unique extension as $L(f) \overset{\text{def}}{=\!=} \widehat{\eta \circ f}$.

4.1. *Interpretation of types*

A type judgement $\Theta \vdash \tau$ is interpreted as a map of type $\mathcal{U}^{|\Theta|} \to \mathcal{U}$, where $|\Theta|$ is the cardinality of the set of variables in Θ. This interpretation map is defined by a combination of induction and *guarded recursion* for the case of recursive types as in Figure 3.

More precisely, the case of recursive types is defined to be the fixed point of a map from $\triangleright(\mathcal{U}^{|\Theta|} \to \mathcal{U})$ to $\mathcal{U}^{|\Theta|} \to \mathcal{U}$ defined as follows:

$$\lambda X.\lambda\rho.\widehat{\triangleright}(\mathsf{next}(\lambda Y : \mathcal{U}.\, \llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, Y)) \circledast (X \circledast \mathsf{next}(\rho))) \tag{22}$$

ensuring

$$\begin{aligned}
\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho) &\equiv \widehat{\triangleright}(\mathsf{next}(\lambda Y : \mathcal{U}.\, \llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, Y)) \circledast (\mathsf{next}(\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket) \circledast \mathsf{next}(\rho))) \\
&\equiv \widehat{\triangleright}(\mathsf{next}(\lambda Y : \mathcal{U}.\, \llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, Y)) \circledast (\mathsf{next}(\llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))) \\
&\equiv \widehat{\triangleright}(\mathsf{next}(\llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))) \\
&\equiv \triangleright(\llbracket \Theta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.\tau \rrbracket (\rho)))
\end{aligned}$$

The first equation is the application of rule (2) for the guarded fix-point combinator, whereas the second equation is derived by distributivity over the later application operator described by rule (4). Finally, the last equation is derived by the fact that the elements of the code of the later operator is the later operator on types (rule (6)).

We prove now the substitution lemma for types which states that substitution behaves as expected, namely that substituting type variables in the syntax with syntactic types corresponds to applying a dependent type $\mathcal{U} \to \mathcal{U}$ to a type $\mathcal{U}$. This can be proved using guarded recursion in the case of recursive types.

Lemma 4.1 (Substitution Lemma for Types). Let σ be a well-formed type with variables in Θ and let ρ be of type $\mathcal{U}^{|\Theta|}$. If $\Theta, \beta \vdash \tau$ then

$$\llbracket \Theta \vdash \tau[\sigma/\beta] \rrbracket (\rho) = \llbracket \Theta, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))$$

Proof. The proof is by induction on $\Theta, \beta \vdash \tau$. Most cases are straightforward, and we just show the case of $\Theta, \beta \vdash \mu\alpha.\tau$. The proof of this case is by *guarded recursion*, and thus we assume that

$$\triangleright(\llbracket \Theta \vdash (\mu\alpha.\tau)[\sigma/\beta] \rrbracket (\rho) = \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))) \tag{23}$$

Assuming (without loss of generality) that α is not β we get the following series of equalities

$$\begin{aligned}
&\llbracket \Theta \vdash (\mu\alpha.\tau)[\sigma/\beta] \rrbracket (\rho) \\
&\quad = \llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho) \\
&\quad = \triangleright(\llbracket \Theta, \alpha \vdash \tau[\sigma/\beta] \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho))) \\
&\quad = \triangleright(\llbracket \Theta, \alpha, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho), \llbracket \Theta, \alpha \vdash \sigma \rrbracket (\rho, \llbracket \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho)))) \\
&\quad = \triangleright(\llbracket \Theta, \alpha, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho), \llbracket \Theta \vdash \sigma \rrbracket (\rho)))
\end{aligned}$$

The latter equals

$$\widehat{\triangleright}(\mathsf{next}(\lambda X \lambda Y.\, \llbracket \Theta, \alpha, \beta \vdash \tau \rrbracket (\rho, X, Y)) \circledast (\mathsf{next}(\llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho))) \circledast \mathsf{next}\, \llbracket \Theta \vdash \sigma \rrbracket (\rho))$$

By (5), (23) implies

$$\mathsf{next}(\llbracket \Theta \vdash \mu\alpha.(\tau[\sigma/\beta]) \rrbracket (\rho)) = \mathsf{next}(\llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho)))$$

and so

$$\begin{aligned}
\llbracket \Theta \vdash \mu\alpha.\tau[\sigma/\beta] \rrbracket (\rho) &= \triangleright(\llbracket \Theta, \alpha, \beta \vdash \tau \rrbracket (\rho, \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho)), \llbracket \Theta \vdash \sigma \rrbracket (\rho))) \\
&= \triangleright(\llbracket \Theta, \beta, \alpha \vdash \tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho), \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho)))) \\
&= \llbracket \Theta, \beta \vdash \mu\alpha.\tau \rrbracket (\rho, \llbracket \Theta \vdash \sigma \rrbracket (\rho))
\end{aligned}$$

□

By direct use of the Substitution Lemma we can prove that the interpretation of the recursive type equals the interpretation of the unfolding of the recursive type itself, only one step later. Intuitively, this means that we need to consume one computational step to look at the data.

$$
\begin{aligned}
\theta_1 &\overset{\text{def}}{=\!=} \lambda x : \triangleright [\![1]\!] \,.\theta_{L[\![1]\!]}(x) \\
\theta_{\tau_1\times\tau_2} &\overset{\text{def}}{=\!=} \lambda x : \triangleright [\![\tau_1 \times \tau_2]\!] \,.\langle \theta_{\tau_1}(\triangleright(\pi_1)(x)), \theta_{\tau_2}(\triangleright(\pi_2)(x))\rangle \\
\theta_{\tau_1+\tau_2} &\overset{\text{def}}{=\!=} \lambda x : \triangleright [\![\tau_1 + \tau_2]\!] \,.\theta_{L[\![\tau_1+\tau_2]\!]}(x) \\
\theta_{\sigma\to\tau} &\overset{\text{def}}{=\!=} \lambda f : \triangleright([\![\sigma]\!] \to [\![\tau]\!]).\lambda x : [\![\sigma]\!] \,.\theta_\tau(f \circledast (\mathsf{next}(x))) \\
\theta_{\mu\alpha.\tau} &\overset{\text{def}}{=\!=} \lambda x \colon \triangleright [\![\mu\alpha.\tau]\!] \,.\, \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (x)
\end{aligned}
$$

Fig. 4. Definition of $\theta_\sigma : \triangleright [\![\sigma]\!] \to [\![\sigma]\!]$

Lemma 4.2. For all types τ and environments ρ of type $\mathcal{U}^{|\Theta|}$,

$$[\![\Theta \vdash \mu\alpha.\tau]\!] \,(\rho) = \triangleright [\![\Theta \vdash \tau[\mu\alpha.\tau/\alpha]]\!] \,(\rho)$$

The interpretation of every *closed* type τ carries a $\triangleright$-algebra structure, i.e., a map $\theta_\tau \colon \triangleright [\![\tau]\!] \to [\![\tau]\!]$, defined by guarded recursion and structural induction on τ as in Figure 4. The case of recursive types is welltyped by Lemma 4.2, and can be formally constructed as a fixed point of a term of type

$$G \colon \triangleright(\Pi\sigma \colon \mathtt{Type}_{\text{FPC}} \,.(\triangleright [\![\sigma]\!] \to [\![\sigma]\!])) \to \Pi\sigma.(\triangleright [\![\sigma]\!] \to [\![\sigma]\!])$$

as follows. Suppose $F \colon \triangleright(\Pi\sigma \colon \mathtt{Type}_{\text{FPC}} \,.(\triangleright [\![\sigma]\!] \to [\![\sigma]\!]))$, and define $G(F)$ essentially as in Figure 4 but with the clause $G(F)_{\mu\alpha.\tau}$ for recursive types being defined as

$$\lambda x \colon \triangleright [\![\mu\alpha.\tau]\!] \,.(F_{\tau[\mu\alpha.\tau/\alpha]} \circledast x) \tag{24}$$

Here F_σ is defined as $F \circledast \mathsf{next}(\sigma)$ using a generalisation of $\circledast$ to dependent products to be defined in Section 5.1. Define θ as the fixed point of G. Then

$$
\begin{aligned}
\theta_{\mu\alpha.\tau}(x) &\equiv G(\mathsf{next}\,(\theta))_{\mu\alpha.\tau}(x) \\
&\equiv \mathsf{next}\,(\theta)_{\tau[\mu\alpha.\tau/\alpha]} \circledast (x)
\end{aligned} \tag{25}
$$

Using the θ we define the delay operation which, intuitively, takes a computation and adds one step.

$$\delta_\sigma \overset{\text{def}}{=\!=} \theta_\sigma \circ \mathsf{next}\,.$$

4.2. *Interpretation of terms*

Figure 5 defines the interpretation of judgements $\Gamma \vdash M : \sigma$ as functions from $[\![\Gamma]\!]$ to $[\![\sigma]\!]$ where $[\![x_1 : \sigma_1, \cdots, x_n : \sigma_n]\!] \overset{\text{def}}{=\!=} [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$. In the case of `case`, the function $\widehat{f}$ is the extension of f to a homomorphism defined as in (21) above, using the fact that all types carry a $\triangleright$-algebra structure. The interpretation of `fold` is welltyped because $\mathsf{next}([\![M]\!]\,(\gamma))$ has type $\triangleright [\![\tau[\mu\alpha.\tau/\alpha]]\!]$ which by Lemma 4.2 is equal to $[\![\mu\alpha.\tau]\!]$. In the case of `unfold`, since $[\![M]\!]\,(\gamma)$ has type $[\![\mu\alpha.\tau]\!]$, which by Lemma 4.2 is equal to $\triangleright [\![\tau[\mu\alpha.\tau/\alpha]]\!]$, the type of $\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\gamma))$ is $[\![\tau[\mu\alpha.\tau/\alpha]]\!]$.

Lemma 4.3. If $\Gamma \vdash M : \tau[\mu\alpha.\tau/\alpha]$ then $[\![\mathtt{unfold}\ (\mathtt{fold}\ M)]\!]\,(\gamma) = \delta_{\tau[\mu\alpha.\tau/\alpha]}\,[\![M]\!]\,(\gamma)$.

$$\begin{aligned}
\llbracket \Gamma \vdash t : \sigma \rrbracket &: \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \\
\llbracket \Gamma \vdash x \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \gamma(x) \\
\llbracket \Gamma \vdash \langle\rangle \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \eta(*) \\
\llbracket \Gamma \vdash \langle M, N \rangle \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \langle \llbracket M \rrbracket (\gamma), \llbracket N \rrbracket (\gamma) \rangle \\
\llbracket \Gamma \vdash \texttt{fst}\ M \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \pi_1(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{snd}\ M \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \pi_2(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \lambda x.M \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \lambda x.\, \llbracket M \rrbracket (\gamma, x) \\
\llbracket \Gamma \vdash MN \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \llbracket M \rrbracket (\gamma)\, \llbracket N \rrbracket (\gamma) \\
\llbracket \Gamma \vdash \texttt{inl}\ E \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \eta(\mathsf{inl}\, \llbracket E \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{inr}\ E \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \eta(\mathsf{inl}\, \llbracket E \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \widehat{f}(\llbracket L \rrbracket (\gamma)) \\
\text{where } f(\mathsf{inl}(x_1)) &\overset{\text{def}}{=\!=} \llbracket M \rrbracket (\gamma, x_1) \\
f(\mathsf{inl}(x_2)) &\overset{\text{def}}{=\!=} \llbracket N \rrbracket (\gamma, x_2) \\
\llbracket \Gamma \vdash \texttt{fold}\ M \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \mathsf{next}(\llbracket M \rrbracket (\gamma)) \\
\llbracket \Gamma \vdash \texttt{unfold}\ M \rrbracket (\gamma) &\overset{\text{def}}{=\!=} \theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M \rrbracket (\gamma))
\end{aligned}$$

Fig. 5. Interpretation of FPC terms

Proof. Straightforward by definition of the interpretation and by the type equality from Lemma 4.2. □

Next lemma proves substitution is well-behaved for terms. The proof is standard textbook result from domain theory (e.g. [Win93; Str06]).

Lemma 4.4 (Substitution Lemma). Let $\Gamma \equiv x_1 : \sigma_1, \cdots, x_k : \sigma_k$ be a context such that $\Gamma \vdash M : \tau$, and let $\Delta \vdash N_i : \sigma_i$ be a term for each $i = 1, \ldots k$. If further $\gamma \in \llbracket \Delta \rrbracket$, then

$$\left\llbracket \Delta \vdash M[\vec{N}/x] : \tau \right\rrbracket (\gamma) = \llbracket \Gamma \vdash M : \tau \rrbracket \left(\left\llbracket \Delta \vdash \vec{N} : \vec{\sigma} \right\rrbracket (\gamma) \right)$$

Proof.

By induction on the typing judgement $\Gamma \vdash M : \tau$.

The cases for $\Gamma \vdash \langle\rangle : 1$, $\Gamma \vdash x : \tau$, $\Gamma \vdash M\ N : \tau$, $\Gamma \vdash \texttt{fst}\ M : \tau_1$, $\Gamma \vdash \texttt{snd}\ M : \tau_2$, $\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2$ are standard.

For the case $\Gamma \vdash \texttt{inl}\ M : \tau_1 + \tau_2$ we start from

$$\left\llbracket \Delta \vdash (\texttt{inl}\ M)[\vec{N}/\vec{x}] : \tau_1 + \tau_2 \right\rrbracket (\gamma)$$

By substitution $(\texttt{inl}\ M)[\vec{N}/\vec{x}]$ equals $\texttt{inl}\ (M[\vec{N}/\vec{x}])$. We also know that its denotation equals $\eta(\mathsf{inl}\left[\!\!\left[(M[\vec{N}/\vec{x}])\right]\!\!\right](\gamma))$ by induction hypothesis this is equal to

$$\eta(\mathsf{inl}\,[\![\Gamma \vdash (M) : \tau_1 + \tau_2]\!]\,(\gamma, \left[\!\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\!\right](\gamma)))$$

which is now by definition what we wanted. The case for $\Gamma \vdash \texttt{inr}\ N : \tau_1 + \tau_2$ is similar.

Now the case for $\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \sigma$. By definition we know that $\left[\!\!\left[\Delta \vdash (\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N)[\vec{N}/\vec{x}] : \tau\right]\!\!\right](\gamma)$ is equal

$$\left[\!\!\left[\Delta \vdash \texttt{case}\ L[\vec{N}/\vec{x}]\ \texttt{of}\ x_1.M[\vec{N}/\vec{x}]; x_2.N[\vec{N}/\vec{x}] : \tau\right]\!\!\right](\gamma)$$

which is by definition of the interpretation equal to

$$\widehat{f}(\lambda x_1.\left[\!\!\left[M[\vec{N}/\vec{x}]\right]\!\!\right](\gamma, x_1), \lambda x_2.\left[\!\!\left[N[\vec{N}/\vec{x}]\right]\!\!\right](\gamma, x_2))(\left[\!\!\left[L[\vec{N}/\vec{x}]\right]\!\!\right](\gamma))$$

where $\widehat{f}$ is as in Figure 5. By induction hypothesis we know that this is equal to

$$\widehat{f}(\lambda x_1.[\![M]\!](\gamma, x_1, \left[\!\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\!\right](\gamma)), (\lambda x_2.[\![N]\!](\gamma, x_2, \left[\!\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\!\right](\gamma))))$$
$$([\![L]\!](\gamma, \left[\!\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\!\right](\gamma)))$$

which is equal by definition to

$$[\![\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \tau]\!](\gamma, \left[\!\!\left[\Delta \vdash \vec{N} : \vec{\sigma}\right]\!\!\right](\gamma))$$

Now the fixed point cases. For the case $\Gamma \vdash \texttt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]$ we know that $\left[\!\!\left[\Gamma \vdash (\texttt{unfold}\ M)[\vec{N}/\vec{x}]\right]\!\!\right](\gamma)$ is equal by definition of the substitution function to

$$\left[\!\!\left[\Gamma \vdash \texttt{unfold}\ (M[\vec{N}/\vec{x}])\right]\!\!\right](\gamma)$$

which by definition of interpretation is $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\left[\!\!\left[\Gamma \vdash (M[\vec{N}/\vec{x}])\right]\!\!\right](\gamma))$. By induction hypothesis this is equal to

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![\Gamma \vdash M]\!](\left[\!\!\left[\Delta \vdash \vec{N}\right]\!\!\right](\gamma))$$

which by definition is $[\![\Gamma \vdash \texttt{unfold}\ (M)]\!](\left[\!\!\left[\Delta \vdash \vec{N}\right]\!\!\right](\gamma))$. For the case $\Gamma \vdash \texttt{fold}\ M : \mu\alpha.\tau$ we know that $\left[\!\!\left[\Gamma \vdash (\texttt{fold}\ M)[\vec{N}/\vec{x}]\right]\!\!\right](\gamma)$ is equal by defintion to $\left[\!\!\left[\Gamma \vdash \texttt{fold}\ (M[\vec{N}/\vec{x}])\right]\!\!\right](\gamma)$ which is by definition of the interpretation equal to $\mathsf{next}(\left[\!\!\left[\Gamma \vdash (M[\vec{N}/\vec{x}])\right]\!\!\right](\gamma))$. By induction hypothesis we get $\mathsf{fold}([\![\Gamma \vdash M]\!](\left[\!\!\left[\Theta \vdash \vec{N}\right]\!\!\right](\gamma))$ which is by definition

$$[\![\Gamma \vdash \texttt{fold}\ (M)]\!]\left[\!\!\left[\Theta \vdash \vec{N}\right]\!\!\right](\gamma)$$

□

We now aim to show a soundness theorem for the interpretation of FPC. We do this by first showing soundness of the single step reduction as in the next lemma. As usual in denotational semantics, this proves that the model is agnostic to operational reductions.

Lemma 4.5. Let M be a closed term of type τ. If $M \to^k N$ then $[\![M]\!](*) = \delta^k [\![N]\!](*)$

Proof. The proof goes by induction on $M \to^k N$. The cases when $k = 0$ follow straightforwardly from the structure of the denotational model.

The case $\texttt{unfold}\ (\texttt{fold}\ M) \to^1 M$ follows directly from Lemma 4.3.

The case for $(\lambda x : \sigma.M)(N) \to^0 M[N/x]$ is straightforward from by Substitution Lemma 4.4.

The case for $\texttt{case}\ (\texttt{inl}\ L)\ \texttt{of}\ x_1.x.M; x_2.x.N \to^0 M[L/x]$ and the case for

$$\texttt{case}\ (\texttt{inr}\ L)\ \texttt{of}\ x_1.x.M; x_2.x.N \to^0 N[L/x]$$

follow directly by definition.

Also the elimination for the product, namely $\texttt{fst}\ \langle M, N\rangle \to^0 M$ and $\texttt{snd}\ \langle M, N\rangle \to^0 N$ follow directly from the definition of the interpretation.

Now we prove the inductive cases. For the case $M_1 N \to^k M_2 N$ we know that by definition $[\![M_1 N]\!](*) = [\![M_1]\!](*)\,[\![N]\!](*)$. By induction hypothesis we know that $[\![M_1]\!](*) = \delta^k_{\sigma\to\tau}([\![M_2]\!](*))$, thus $[\![M_1]\!](*)\,[\![N]\!](*) = (\delta^k_{\sigma\to\tau}([\![M_2]\!](*)))\,[\![N]\!](*)$ By definition of δ and θ this is equal to $\delta^k_\tau([\![M_2]\!](*)\,[\![N]\!](*))$.

Now the case for

$$\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \to^k \texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N$$

The induction hypothesis gives $[\![L]\!] = \delta_{\tau_1+\tau_2} \circ [\![L']\!]$, and so Lemma 4.6 applies proving the case.

The case for $\texttt{fst}\ M \to^k \texttt{fst}\ M'$ and for $\texttt{snd}\ M \to^k \texttt{snd}\ M'$ are similar to the previous case.

Finally, the case for $\texttt{unfold}\ M_1 \to^k \texttt{unfold}\ M_2$. By definition we know that

$$[\![\texttt{unfold}\ M_1]\!](*) = \theta([\![M_1]\!](*))$$

By induction hypothesis this is equal to $\theta(\delta^k_{\mu\alpha.\tau}([\![M_2]\!](*)))$ which by Lemma 4.7 is equal to $\delta^k_{\tau[\mu\alpha.\tau/\alpha]}(\theta([\![M_2]\!](*)))$ thus concluding. □

The two most complicated cases of the proof of Lemma 4.5, namely the `unfold-fold` reductions and `case`, are captured in the following two lemmas. In particular, the first of these states that the interpretation of `case` is a $\triangleright$-algebra homomorphism. In other words, case analysing over a computation that perform n ticks and then produces a result v is equal to a computation that produces n ticks and then performs case analysis over a terminating computation producing a value v.

Lemma 4.6.

1 The interpretation of `case` is a homomorphism of $\triangleright$-algebras in the first variable, i.e.,

$$\begin{aligned}&[\![\ \lambda x\colon \tau_1 + \tau_2.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma)(\theta(r))\\ =&\theta(\mathsf{next}([\![\ \lambda x\colon \tau_1 + \tau_2.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma)) \circledast r)\end{aligned}$$

2 If $[\![L]\!](\gamma) = \delta([\![L']\!](\gamma))$, then

$$[\![\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma) = \delta\,[\![\texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N]\!](\gamma)$$

Proof. For the proof of the first part, we use the notation $\widehat{f}$ as in Figure 5. Since $\widehat{f}$ is a homomorphism of $\triangleright$-algebras we get

$$\begin{aligned}\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\gamma)(\theta_{\tau_1+\tau_2}(r)) &= \widehat{f}(\theta_{\tau_1+\tau_2}(r)) \\ &= \theta_\sigma(\mathsf{next}(\widehat{f}) \circledast r) \\ &= \theta_\sigma(\mathsf{next}\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\gamma) \circledast r)\end{aligned}$$

For the second part, note that $\widehat{f}$ is $\llbracket \lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\gamma)$, so

$$\begin{aligned}\llbracket \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\gamma) &= \widehat{f}(\llbracket L \rrbracket(\gamma)) \\ &= \widehat{f}(\delta_{\tau_1+\tau_2}(\llbracket L' \rrbracket(\gamma))) \\ &= \widehat{f}(\theta_{\tau_1+\tau_2}(\mathsf{next}(\llbracket L' \rrbracket(\gamma)))) \\ &= \theta_\sigma(\mathsf{next}(\widehat{f}) \circledast (\mathsf{next}(\llbracket L' \rrbracket(\gamma)))) \\ &= \theta_\sigma(\mathsf{next}(\widehat{f}(\llbracket L' \rrbracket(\gamma)))) \\ &= \delta_\sigma(\llbracket \texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N \rrbracket(\gamma))\end{aligned}$$

□

We now prove the same for the interpretation of `unfold`. The key point here is to observe that the tick operation for a folded recursive type , namely $\theta_{\mu\alpha.\tau}$, is precisely the tick of the unfolded recursive type after one step of computation, namely $\triangleright(\theta_{\tau[\mu\alpha.\tau/\alpha]})$.

Lemma 4.7. If $\mu\alpha.\tau$ is a closed FPC type then

1 $\llbracket\ \lambda x\colon \mu\alpha.\tau.\texttt{unfold}\ x \rrbracket(\theta_{\mu\alpha.\tau}(r)) = \theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast r)$

2 If $\llbracket M \rrbracket(\gamma) = \delta_{\mu\alpha.\tau}(\llbracket M' \rrbracket(\gamma))$, then

$$\llbracket \texttt{unfold}\ M \rrbracket(\gamma) = \delta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket \texttt{unfold}\ M' \rrbracket(\gamma))$$

Proof. The interpretation for $\llbracket\ \lambda x\colon \mu\alpha.\tau.\texttt{unfold}\ x \rrbracket(\theta_{\mu\alpha.\tau}(r))$ yields $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\theta_{\mu\alpha.\tau}(r))$. This type checks as r has type $\triangleright\llbracket\mu\alpha.\tau\rrbracket$, thus $(\theta_{\mu\alpha.\tau}(r))$ has type $\llbracket\mu\alpha.\tau\rrbracket$ which – by Lemma 4.2 – is equal to $\triangleright\llbracket\tau[\mu\alpha.\tau/\alpha]\rrbracket$. Thus the term $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\theta_{\mu\alpha.\tau}(r))$ has type $\llbracket\tau[\mu\alpha.\tau/\alpha]\rrbracket$. Now by definition of $\theta_{\mu\alpha.\tau}$ this is equal to $\theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (r))$ which is what we wanted.

For the second statement, we compute

$$\begin{aligned}\llbracket \texttt{unfold}\ M \rrbracket(\gamma) &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M \rrbracket(\gamma)) \\ &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\delta_{\mu\alpha.\tau}(\llbracket M' \rrbracket(\gamma))) \\ &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\theta_{\mu\alpha.\tau}(\mathsf{next}(\llbracket M' \rrbracket(\gamma)))) \\ &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (\mathsf{next}(\llbracket M' \rrbracket(\gamma)))) && \text{(statement 1)} \\ &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M' \rrbracket(\gamma)))) && \text{(rule (4))} \\ &= \theta_{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{next}(\llbracket \texttt{unfold}\ M' \rrbracket(\gamma))) \\ &= \delta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket \texttt{unfold}\ M' \rrbracket(\gamma))\end{aligned}$$

□

We can now prove Lemma 4.5. As stated above, this is soundness of the model w.r.t. the small-step operational semantics. For the proof, it is crucial that the interpretation of every term is an homomorphism of tickθ-algebras. This falls out in many cases. For the cases of the interpretation of `unfold` and the inductive case of `case` we use the lemmas we just proved above.

Proof of Lemma 4.5 The proof is by induction on $M \to^k N$. Most of the cases are straightforward, some (β-reductions for function and sum types) using the substitution lemma (Lemma 4.4). The case $\texttt{unfold}\ (\texttt{fold}\ M) \to^1 M$ follows directly from Lemma 4.3.

Now we prove the inductive cases. For the case $M_1 N \to^k M_2 N$ we know that by definition $[\![M_1 N]\!](*) = [\![M_1]\!](*)\,[\![N]\!](*)$. By induction hypothesis we know that $[\![M_1]\!](*) = \delta^k_{\sigma\to\tau}([\![M_2]\!](*))$, thus $[\![M_1]\!](*)\,[\![N]\!](*) = (\delta^k_{\sigma\to\tau}([\![M_2]\!](*)))\,[\![N]\!](*)$ By definition of δ and θ this is equal to $\delta^k_\tau([\![M_2]\!](*)\,[\![N]\!](*))$.

In the case of

$$\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \to^k \texttt{case}\ L'\ \texttt{of}\ x_1.M; x_2.N$$

the induction hypothesis gives $[\![L]\!](*) = \delta_{\tau_1+\tau_2}\,[\![L']\!](*)$, and so Lemma 4.6 applies proving the case.

Finally, the case for $\texttt{unfold}\ M \to^k \texttt{unfold}\ M'$. If $k = 0$ the case follows trivially from the induction hypothesis. If $k = 1$, the step from the induction hypothesis to the case is exactly the second statement of Lemma 4.7. □

We now state and prove soundness of our model w.r.t. the operational semantics. We use the transitive closure over $\to^k$, namely $\Rightarrow^k$, which is *synchronised* with the $\triangleright$ operator in the type theory. Using $\Rightarrow$ (rather than $\to_*$) is not essential to prove soundness, but it is crucial to prove computational adequacy, which will be presented in the next section. On the other hand, the explicit step-indexing k in $\Rightarrow^k$ is necessary to relate the number of operational steps with the number of delays (or ticks) in the denotational semantics.

Proposition 4.8 (Soundness). Let M be a closed term of type τ. If $M \Rightarrow^k N$ then $[\![M]\!](*) = \delta^k\,[\![N]\!](*)$.

Proof. By induction on k. When $k = 0$ Lemma 4.5 applies concluding the case. When $k = n+1$ by definition we have $M \to^0_* M'$, $M' \to^1 M''$ and $\triangleright(M'' \Rightarrow^n N)$. By repeated application of Lemma 4.5 we get $[\![M]\!](*) = [\![M']\!](*)$ and $[\![M']\!](*) = \delta([\![M'']\!](*))$. By induction hypothesis we get $\triangleright([\![M'']\!](*) = \delta^n\,[\![N]\!](*))$ which implies $\mathsf{next}([\![M'']\!](*)) = \mathsf{next}(\delta^n\,[\![N]\!](*)))$ and since $\delta = \theta \circ \mathsf{next}$, this implies $\delta([\![M'']\!](*)) = \delta^k([\![N]\!](*))$. By putting together the equations we get finally $[\![M]\!](*) = \delta^k\,[\![N]\!](*)$. □

5. Computational Adequacy

Computational adequacy is the opposite implication of Proposition 4.8 in the case of terms of unit type. It is proved by constructing a (proof relevant) logical relation between syntax and semantics. The relation cannot be constructed just by induction on the structure of types, since in the case of recursive types, the unfolding can be bigger than the recursive type. Instead, the relation is constructed by guarded recursion: we

$$\mathsf{next}\,\xi\,[x \leftarrow \mathsf{next}\,\xi.t]\,.u \equiv \mathsf{next}\,\xi.(u[t/x]) \tag{26}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.x \equiv t \tag{27}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}\,\xi.u \tag{28}$$

$$\mathsf{next}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.u \tag{29}$$

$$\mathsf{next}\,\xi.\,\mathsf{next}\,\xi'.u \equiv \mathsf{next}\,\xi'.\,\mathsf{next}\,\xi.u \tag{30}$$

$$(\mathsf{next}\,\xi.t =_{\triangleright\xi.A} \mathsf{next}\,\xi.s) \equiv \triangleright\xi.(t =_A s) \tag{31}$$

$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}\,\xi.A)) \equiv \triangleright\xi.\,\mathsf{El}(A) \tag{32}$$

Fig. 6. The notation $\xi\,[x \leftarrow t]$ means the extension of the delayed substitution ξ with $[x \leftarrow t]$. Rule (28) requires x not free in u. Rule (30) requires that none of the variables in the codomains of ξ and ξ' appear in the type of u, and that the codomains of ξ and ξ' are independent.

assume the relation exists *later*, and from that assumption construct the relation *now* by structural induction on types. Thus the well-definedness of the logical relation is ensured by the type system of GDTT, more specifically by the rules for guarded recursion. This is in contrast to the classical proof in domain theory [Pit96], where existence requires a separate argument.

The logical relation uses a lifting of relations on values available now, to relations on values available later. To define this lifting, we need *delayed substitutions*, an advanced feature of GDTT.

5.1. *Delayed substitutions*

In GDTT, if $\Gamma, x\colon A \vdash B$ type is a well formed type and t has type $\triangleright A$ in context Γ, one can form the type $\triangleright\,[x \leftarrow t]\,.B$. Intuitively, one time step from now, t delivers an element in A, and $\triangleright\,[x \leftarrow t]\,.B$ is the type of elements that one time step from now delivers something in B with x substituted by the element delivered at that time by t. One motivation for this construction is to generalise $\circledast$ (described in Section 2) to a dependent version: if $f\colon \triangleright(\Pi(x : A).B)$, then $f \circledast t\colon \triangleright\,[x \leftarrow t]\,.B$. The idea is that t will eventually reduce to a term of the form $\mathsf{next}\,u$, and then $\triangleright\,[x \leftarrow t]\,.B$ will be equal to $\triangleright B[u/x]$. But if t is open, we may not be able to do this reduction yet.

More generally, we define the notion of *delayed substitution* as follows. Suppose $\Gamma, \Gamma' \vdash$ is a wellformed context, and suppose Γ' is on the form $\Gamma' = x_1\colon A_1 \dots x_n\colon A_n$ with all A_i independent, i.e., no x_j appears in an A_i. A delayed substitution $\xi\colon \Gamma \rightarrowtriangle \Gamma'$ is a vector of terms $\xi = [x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n]$ such that $\Gamma \vdash t_i : A_i$ for each i. [Biz+16] gives a more general definition of delayed substitution allowing dependencies between the A_i's, but for this paper we just need the definition above.

If $\xi\colon \Gamma \rightarrowtriangle \Gamma'$ is a delayed substitution and $\Gamma, \Gamma' \vdash B$ type is a wellformed type, then the type $\triangleright\xi.B$ is wellformed in context Γ. The introduction form states $\mathsf{next}\,\xi.u\colon \triangleright\xi.B$ if $\Gamma, \Gamma' \vdash u : B$.

In Figure 6 we recall some rules from [Biz+16] needed below. Of these, (26) and (27) can be considered β and η laws, and (28) is a weakening principle. Rules (26), (28) and (29) also have obvious versions for types, e.g.,

$$\triangleright\xi\left[x \leftarrow \mathsf{next}\,\xi.t\right].B \equiv \triangleright\xi.(B[t/x]) \tag{33}$$

Rather than be taken as primitive, later application $\circledast$ can be defined using delayed substitutions as

$$g \circledast y \stackrel{\text{def}}{=\!=} \mathsf{next}\left[f \leftarrow g, x \leftarrow y\right].f(x) \tag{34}$$

Note that if $g : \triangleright(A \to B)$ and $y : \triangleright A$, the type of $g \circledast y$ is $\triangleright[f \leftarrow g, x \leftarrow y].B$ which reduces to $\triangleright B$ since f and x do not appear in B. With this definition, the rule $\mathsf{next}(f(t)) \equiv \mathsf{next}\,f \circledast \mathsf{next}\,t$ from Section 2 generalises to

$$\mathsf{next}\,\xi.(f\,t) \equiv (\mathsf{next}\,\xi.f) \circledast (\mathsf{next}\,\xi.t) \tag{35}$$

which follows from (26). In fact, later application generalises to the setting of delayed substitutions: if $g : \triangleright\xi.\Pi x : A.B$ and $y : \triangleright\xi.A$ define

$$g \circledast y \stackrel{\text{def}}{=\!=} \mathsf{next}\,\xi\left[f \leftarrow g, x \leftarrow y\right].f(x) : \triangleright\xi\left[x \leftarrow y\right].B \tag{36}$$

Note that in the special case where $y = \mathsf{next}\,\xi.u$ we get

$$g \circledast \mathsf{next}\,\xi.u \colon \triangleright\xi.B[u/x]$$

Rules (27), (28) and (30) imply

$$\begin{aligned}\mathsf{next}\,\xi\left[x \leftarrow t\right].\,\mathsf{next}\,x &\equiv \mathsf{next}(\mathsf{next}\,\xi\left[x \leftarrow t\right].x)\\ &\equiv \mathsf{next}(t)\\ &\equiv \mathsf{next}\,\xi\left[x \leftarrow t\right].t\end{aligned}$$

which by (31) gives an inhabitant of

$$\triangleright\xi\left[x \leftarrow t\right].(\mathsf{next}\,x = t) \tag{37}$$

5.2. *A logical relation between syntax and semantics*

Our strategy to prove computational adequacy is by logical relation argument. We construct a logical relation $\mathcal{R}$ as in Figure 7 between syntax and semantics. This is done using first guarded recursion and then induction on the FPC types.

Figure 7 uses an operation lifting relations $\mathcal{R}$ from A to B to relations $\triangleright\mathcal{R}$ from $\triangleright A$ to $\triangleright B$ defined as

$$t \mathrel{\triangleright\mathcal{R}} u \stackrel{\text{def}}{=\!=} \triangleright\left[x \leftarrow t, y \leftarrow u\right].(x \mathrel{\mathcal{R}} y) \tag{38}$$

As a consequence of (33) the following statement holds:

$$(\mathsf{next}\,\xi.t) \mathrel{\triangleright\mathcal{R}} (\mathsf{next}\,\xi.u) \equiv \triangleright\xi.(t \mathrel{\mathcal{R}} u) \tag{39}$$

The lifting on relations is used, e.g., in the second case of $\mathcal{R}_1$ where x is assumed to have type $\triangleright L1$. In that case $\theta_1(x)$ is a semantic computation that takes a step, and so should only be related to M, if M can also reduce in one step to an M'', that should be

$$\begin{aligned}
\eta(*)\ \mathcal{R}_1\ M &\overset{\text{def}}{=\!=} M \Rightarrow^0 \langle\rangle \\
\theta_1(x)\ \mathcal{R}_1\ M &\overset{\text{def}}{=\!=} \Sigma M', M''\colon \mathtt{Term}_{\mathtt{FPC}}.M \to^0_* M' \to^1 M'' \text{ and } x \triangleright\mathcal{R}_1\ \mathsf{next}(M'') \\
x\ \mathcal{R}_{\tau_1\times\tau_2}\ M &\overset{\text{def}}{=\!=} \pi_1(x)\ \mathcal{R}_{\tau_1}\ \mathtt{fst}\ (M) \text{ and } \pi_2(x)\ \mathcal{R}_{\tau_2}\ \mathtt{snd}\ (M) \\
\eta(\mathsf{inl}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M &\overset{\text{def}}{=\!=} \Sigma L.M \Rightarrow^0 \mathtt{inl}\ L \text{ and } x\ \mathcal{R}_{\tau_1}\ L \\
\eta(\mathsf{inr}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M &\overset{\text{def}}{=\!=} \Sigma L.M \Rightarrow^0 \mathtt{inr}\ L \text{ and } x\ \mathcal{R}_{\tau_2}\ L \\
\theta_{\tau_1+\tau_2}(x)\ \mathcal{R}_{\tau_1+\tau_2}\ M &\overset{\text{def}}{=\!=} \Sigma M', M''\colon \mathtt{Term}_{\mathtt{FPC}}.M \to^0_* M' \to^1 M'' \text{ and } x \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(M'') \\
f\ \mathcal{R}_{\tau\to\sigma}\ M &\overset{\text{def}}{=\!=} \Pi x\colon [\![\tau]\!], N\colon \mathtt{Term}_{\mathtt{FPC}}.x\ \mathcal{R}_\tau\ N \to f(x)\ \mathcal{R}_\sigma\ (MN) \\
x\ \mathcal{R}_{\mu\alpha.\tau}\ M &\overset{\text{def}}{=\!=} \Sigma M'M''.\mathtt{unfold}\ M \to^0_* M' \to^1 M'' \text{ and } x \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')
\end{aligned}$$

Fig. 7. The logical relation $\mathcal{R}_\tau : [\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}$.

later related to x. Note that x is not necessarily of the form $\mathsf{next}(y)$ for some y, but we can still related x to $\mathsf{next}(M'')$ using delayed substitutions as in the definition of $\triangleright\mathcal{R}_1$.

Most of the definition of the logical relation is standard, e.g., in the case of function types, where related functions are required to map related input to related output. The case of recursive type deserves some attention. On the right hand side, we have x of type $[\![\mu\alpha.\tau]\!]$, which means it is a piece of data which later will be unfolded and therefore available. More precisely, it has also type $\triangleright [\![\tau[\mu\alpha.\tau/\alpha]]\!]$. This semantic program is related to a syntactic program M if and only if the unfolding of M reduces in one computational step to an M'' which is later related to x.

The logical relation is an example of a guarded recursive definition. To see this, note first that the lifting operation can be expressed on codes mapping $\mathcal{R} : A \to B \to \mathcal{U}$ to

$$\lambda x\colon \triangleright A, y\colon \triangleright B.\widehat{\triangleright}(\mathsf{next}\,[x' \leftarrow x, y' \leftarrow y]\,.(x'\ \mathcal{R}\ y'))$$

and this operation factors as $F \circ \mathsf{next}$, for $F\colon \triangleright(A \to B \to \mathcal{U}) \to A \to B \to \mathcal{U}$ defined as

$$\lambda S.\lambda x\colon \triangleright A, y\colon \triangleright B.\widehat{\triangleright}(\mathsf{next}\,[x' \leftarrow x, y' \leftarrow y,\ \mathcal{R}\ \leftarrow S]\,.(x'\ \mathcal{R}\ y'))$$

Using this, one can formally define the logical relation as a fixed point of a function of type

$$\triangleright(\Pi(\tau : \mathtt{Type}_{\mathtt{FPC}}).\,[\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}) \to (\Pi(\tau : \mathtt{Type}_{\mathtt{FPC}}).\,[\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U})$$

similarly to the formal definition of θ in the equation (24).

5.3. *Proof of computational adequacy*

Before proving computational adequacy we need to show some key properties about the logical relation $\mathcal{R}$. The first of these is that the relation respects the *applicative* structure of the $\triangleright$ operator which is that we can apply an argument that will be available later to a function that will also be available later.

Lemma 5.1. If $f \rhd\mathcal{R}_{\tau\to\sigma}$ $\mathsf{next}(M)$ and $r \rhd\mathcal{R}_{\tau}$ $\mathsf{next}(L)$ then

$$(f \circledast r) \rhd\mathcal{R}_{\sigma}\ \mathsf{next}(ML)$$

Proof. By definition $f \rhd\mathcal{R}_{\tau\to\sigma}$ $\mathsf{next}(M)$ is type equal to

$$\rhd [x \leftarrow f] .(x\ \mathcal{R}_{\tau\to\sigma}\ M)$$

which by definition is

$$\rhd [x \leftarrow f] .(\Pi(y : [\![\tau]\!])(L : \mathtt{Term}_{\mathtt{FPC}}).y\ \mathcal{R}_{\tau}\ L \to x(y)\ \mathcal{R}_{\sigma}\ ML)$$

By applying the latter to r and $\mathsf{next}\, L$ using the generalised later application of (36) we get an element of

$$\begin{aligned} &\rhd [x \leftarrow f, y \leftarrow r, L \leftarrow \mathsf{next}\, L] .(y\ \mathcal{R}_{\tau}\ L \to x(y)\ \mathcal{R}_{\sigma}\ ML) \\ &\equiv \rhd [x \leftarrow f, y \leftarrow r] .(y\ \mathcal{R}_{\tau}\ L \to x(y)\ \mathcal{R}_{\sigma}\ ML) \end{aligned}$$

By further applying this to the hypothesis $r \rhd\mathcal{R}_{\tau}$ $\mathsf{next}(L) \equiv \rhd [y \leftarrow r] .(y\ \mathcal{R}_{\tau}\ L)$ we get

$$\rhd [x \leftarrow f, y \leftarrow r] .(x(y)\ \mathcal{R}_{\sigma}\ ML)$$

which is equivalent to $(f \circledast r) \rhd\mathcal{R}_{\sigma}$ $\mathsf{next}(ML)$, thus concluding the case. □

Next we show that the relation is agnostic to 0-step reduction in the operational semantics.

Lemma 5.2. If $M \to^0 N$ then $x\ \mathcal{R}_{\sigma}\ M$ iff $x\ \mathcal{R}_{\sigma}\ N$.

Proof. We prove first the left to right implication by induction on σ, and show just a few cases.

In the case of coproducts, we proceed by case analysis on x. In the case of $x = \eta(\mathsf{inl}(y))$, by the assumption we have that $M \to^0_* \mathtt{inl}\ (N')$ and $y\ \mathcal{R}_{\tau_1}\ N'$. If $M = \mathtt{inl}\ (N')$, then by N must be of the form $\mathtt{inl}\ (N'')$ for some N'', such that $N' \to^0 N''$. In this case, by induction hypothesis $y\ \mathcal{R}_{\tau_1}\ N''$ and so $x\ \mathcal{R}_{\tau_1+\tau_2}\ N$. If the reduction $M \to^0_* \mathtt{inl}\ (N')$ has positive length, by determinancy of the operational semantics (Lemma 3.1) we get $N \to^0_* \mathtt{inl}\ N'$, and thus $x\ \mathcal{R}_{\tau_1+\tau_2}\ N$. The case where $x = \eta(\mathsf{inl}(y))$ is similar. When $x = \theta_{\tau_1+\tau_2}(y)$, by the assumption $x\ \mathcal{R}_{\tau_1+\tau_2}\ M$ there exist M' and M'' such that $M \to^0_* M'$ and $M' \to^1 M''$ and $y \rhd\mathcal{R}_1$ $\mathsf{next}(M'')$. Again by determinancy of the operational semantics, $N \to^0_* M'$ and thus we conclude $x\ \mathcal{R}_{\tau_1+\tau_2}\ N$.

Now we consider the case for recursive types. By assumption we know there exists M' and M'' such that $\mathtt{unfold}\ M \to^0_* M'$ and $M' \to^1 M''$ and $x \rhd\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}$ $\mathsf{next}(M'')$. Since $M \to^0 N$ then also $\mathtt{unfold}\ M \to^0 \mathtt{unfold}\ N$. Therefore, from the assumption and the fact that the operational semantics is deterministic (Lemma 3.1) we get $\mathtt{unfold}\ N \to^0_* M'$. By definition of the logical relation we get $x\ \mathcal{R}_{\mu\alpha.\tau}\ N$, which concludes the proof.

The proof of the right to left implication is also by induction on the structure of σ. Again we just show a few cases.

In the case of the unit type, we proceed by case analysis on x. When $x = \eta(*)$ we have that $N \to^0_* \langle\rangle$. Since $M \to^0 N$ we get $M \to^0_* \langle\rangle$ as required. When x is $\theta_1(x')$

by assumption $x \ \mathcal{R}_1 \ N$ implies that there exists N' and N'' such that $N \rightarrow^0_* N'$ and $N' \rightarrow^1 N''$ and $x' \rhd\mathcal{R}_1 \ \mathsf{next}(N'')$. Since also $M \rightarrow^0_* N'$ this implies $x \ \mathcal{R}_1 \ M$.

In the case of recursive types, by assumption we have that $x \ \mathcal{R}_{\mu\alpha.\tau} \ N$ and $M \rightarrow^0 N$. From the former we derive that there exists M' and M'' such that $\mathtt{unfold}\ N \rightarrow^0_* M'$,$M' \rightarrow^1 M''$ and $x \ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]} \ \mathsf{next}(M'')$. Since $M \rightarrow^0 N$ then also $\mathtt{unfold}\ M \rightarrow^0 \mathtt{unfold}\ N$. Therefore, we know that $\mathtt{unfold}\ M \rightarrow^0_* M'$, thus by definition of the logical relation we conclude. □

Now we show a key property of the logical relation. This states that for programs that are related later after a 1-step operational reduction are related now. Note that in the interpretation of the unit type we used the lifting monad. This was not strictly necessary to get a "tick" algebra structure, but it is crucial to make the following lemma to work.

Lemma 5.3. If $x \rhd\mathcal{R}_\tau \ \mathsf{next}(M)$ and $M' \rightarrow^1 M$ then $\theta_\tau(x) \ \mathcal{R}_\tau \ M'$.

Proof. The proof is by guarded recursion, so we assume that the lemma is "later true", i.e., that we have an inhabitant of the type obtained by applying $\rhd$ to the statement of the lemma. We proceed by induction on τ.

The cases for the unit type and for the coproduct are straightforward by definition. In the case for products, by assumption we have

$$y \rhd\mathcal{R}_{\tau_1\times\tau_2} \ \mathsf{next}(M)\,.$$

Unfolding definitions we get

$$\rhd\left[x \leftarrow y\right].(\pi_1(x) \ \mathcal{R}_{\tau_1} \ (\mathtt{fst}\ M)) \text{ and } (\pi_2(x) \ \mathcal{R}_{\tau_2} \ \mathtt{fst}\ (M))$$

which implies

$$(\pi_1(y)) \rhd\mathcal{R}_{\tau_1} \ \mathsf{next}(\mathtt{fst}\ M) \qquad \text{and} \qquad \pi_2(y) \rhd\mathcal{R}_{\tau_2} \ \mathsf{next}(\mathtt{snd}\ M)$$

Since $M' \rightarrow^1 M$ then also $\mathtt{fst}\ M' \rightarrow^1 \mathtt{fst}\ M$ and $\mathtt{snd}\ M' \rightarrow^1 \mathtt{snd}\ M$, thus we can use the induction hypothesis on τ_1 and τ_2 and get

$$\theta_{\tau_1}(\pi_1(y)) \ \mathcal{R}_{\tau_1} \ \mathtt{fst}\ M' \qquad \text{and} \qquad \theta_{\tau_2}(\pi_2(y)) \ \mathcal{R}_{\tau_2} \ \mathtt{snd}\ M'$$

by definition $\theta_{\tau_1\times\tau_2}$ commutes with π_1 and π_2. Thus, we obtain

$$\pi_1(\theta_{\tau_1\times\tau_2}(y)) \ \mathcal{R}_{\tau_1} \ \mathtt{fst}\ M' \qquad \text{and} \qquad \pi_2(\theta_{\tau_1\times\tau_2}(y)) \ \mathcal{R}_{\tau_2} \ \mathtt{snd}\ M'$$

which is by definition what we wanted.

Now the case for the function space. Assume $f \rhd\mathcal{R}_{\tau_1\rightarrow\tau_2} \ \mathsf{next}(M)$ and $M' \rightarrow^1 M$. We must show that if $y : [\![\tau_1]\!]$, $N : \mathtt{Term}_{\mathtt{FPC}}$ and $y \ \mathcal{R}_{\tau_1} \ N$ then $(\theta_{\tau_1\rightarrow\tau_2}(f))(y) \ \mathcal{R}_{\tau_2} \ (MN)$. So suppose $y \ \mathcal{R}_{\tau_1} \ N$, and thus also $\rhd(y \ \mathcal{R}_{\tau_1} \ N)$ which is equal to $\mathsf{next}(y) \rhd\mathcal{R}_{\tau_1} \ \mathsf{next}(N)$. By applying Lemma 5.1 to this and $f \rhd\mathcal{R}_{\tau_1\rightarrow\tau_2} \ \mathsf{next}(M)$ we get

$$f \circledast (\mathsf{next}(y)) \rhd\mathcal{R}_{\tau_2} \ \mathsf{next}(MN)$$

Since $M' \rightarrow^1 M$ also $M'N \rightarrow^1 MN$, and thus, by the induction hypothesis for τ_2, $\theta_{\tau_2}(f \circledast (\mathsf{next}(y))) \ \mathcal{R}_{\tau_2} \ M'N$. Since by definition $\theta_{\tau_1\rightarrow\tau_2}(f)(y) = \theta_{\tau_2}(f \circledast \mathsf{next}(y))$, this proves the case.

The interesting case is the one of $\mu\alpha.\tau$. Assume $x \triangleright\mathcal{R}_{\mu\alpha.\tau}$ $\mathsf{next}(M)$ and $M' \to^1 M$. By definition of $\triangleright\mathcal{R}$ this implies $\triangleright [y \leftarrow x] .(y\ \mathcal{R}_{\mu\alpha.\tau}\ M)$ which by definition of $\mathcal{R}_{\mu\alpha.\tau}$ is

$$\triangleright [y \leftarrow x] .\Sigma N' N''.\mathtt{unfold}\, M \to^0_* N' \text{ and } N' \to^1 N'' \text{ and } (y \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(N''))$$

Since zero-step reductions cannot eliminate outer $\mathtt{unfold}$'s, N' must be on the form $\mathtt{unfold}\, N$ for some N, such that $M \to^0_* N$. Thus, we can apply the guarded induction hypothesis to get

$$\triangleright [y \leftarrow x] .(\Sigma N.M \to^0_* N \text{ and } (\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathtt{unfold}\, N))$$

Since $\mathtt{unfold}\, M \to^0_* \mathtt{unfold}\, N$, by Lemma 5.2 we get

$$\triangleright [y \leftarrow x] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathtt{unfold}\, M)$$

which by (39) is

$$\mathsf{next}\, [y \leftarrow x] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)) \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\mathtt{unfold}\, M)$$

By (34) this implies

$$\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\mathtt{unfold}\, M)$$

Since by assumption $M' \to^1 M$ also $\mathtt{unfold}\, M' \to^1 \mathtt{unfold}\, M$ thus, by definition of the logical relation

$$\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x\ \mathcal{R}_{\mu\alpha.\tau}\ M'$$

By definition $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast x$ is equal to $\theta_{\mu\alpha.\tau}(x)$ thus we can derive

$$\theta_{\mu\alpha.\tau}(x)\ \mathcal{R}_{\mu\alpha.\tau}\ M'$$

as we wanted. □

We can now finally state and prove the fundamental lemma stating that any term is related to its denotation in the logical relation of Figure 7. As we shall see below, this will imply computational adequacy.

Lemma 5.4 (Fundamental Lemma). Suppose $\Gamma \vdash M : \tau$, for $\Gamma \equiv x_1 : \tau_1, \cdots, x_n : \tau_n$ and $\vdash N_i : \tau_i$, $\gamma_i : [\![\tau_i]\!]$ and $\gamma_i\ \mathcal{R}_{[\![\tau_i]\!]}\ N_i$ for $i \in \{1, \ldots, n\}$, then $[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ M[\vec{N}/\vec{x}]$

Proof. The proof is by guarded recursion, and so we assume $\triangleright$ applied to the statement of the lemma. This implies that for all well-typed terms M with context Γ and type τ the following holds:

$$\triangleright([\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_\tau\ M[\vec{N}/\vec{x}])$$

Then we proceed by induction on the typing derivation $\Gamma \vdash M : \tau$, showing only the interesting cases.

Consider first the case of $\Gamma \vdash \lambda x.M : \sigma \to \tau$. Assuming $\gamma_{n+1}\ \mathcal{R}_\sigma\ M_{n+1}$, we must show $[\![\lambda x.M]\!]\,(\vec{\gamma})(\gamma_{n+1})\ \mathcal{R}_\tau\ [\![M]\!]\,(\vec{\gamma}, \gamma_{n+1})$. Since

$$[\![\lambda x.M]\!]\,(\vec{\gamma})(\gamma_{n+1}) = [\![M]\!]\,(\vec{\gamma}, \gamma_{n+1})$$
$$(\lambda x.M)[\vec{M}/\vec{x}](M_{n+1}) = \lambda x.(M[\vec{M}/\vec{x}])(M_{n+1})$$

and $\lambda x.(M[\vec{M}/\vec{x}])(M_{n+1}) \to^0 (M[\vec{M}/\vec{x}])[M_{n+1}/x]$, by Lemma 5.2 it suffices to prove

$$[\![M]\!]\,(\vec{\gamma}, \gamma_{n+1})\ \mathcal{R}_\tau\ M[\vec{M}/\vec{x}][M_{n+1}/x]$$

which follows from the induction hypothesis.

For the case $\Gamma \vdash \texttt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]$ we must show that

$$[\![\texttt{unfold}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ (\texttt{unfold}\ M)[\vec{N}/\vec{x}]$$

By induction hypothesis we know that $[\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\mu\alpha.\tau}\ (M[\vec{N}/\vec{x}])$ which means that there exists M' and M'' such that $\texttt{unfold}\ (M[\vec{N}/\vec{x}]) \to^0_* M'$ and $M' \to^1 M''$ and $[\![M]\!]\,(\vec{\gamma})\ \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. By Lemma 5.3 then $\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\vec{\gamma}))\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ M'$ and since $\texttt{unfold}\ (M[\vec{N}/\vec{x}]) \to^0_* M'$ by repeated application of Lemma 5.2 we get

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\vec{\gamma}))\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \texttt{unfold}\ (M[\vec{N}/\vec{x}])$$

Since by definition $[\![\texttt{unfold}\ M]\!]\,(\vec{\gamma}) = \theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\vec{\gamma}))$ this finishes the proof of the case.

For the case $\Gamma \vdash \texttt{fold}\ M : \mu\alpha.\tau$ we want to show that

$$[\![\texttt{fold}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\mu\alpha.\tau}\ (\texttt{fold}\ M)[\vec{N}/\vec{x}]$$

By definition of the logical relation we have to show that there exist M' and M'' such that

$$\texttt{unfold}\ (\texttt{fold}\ (M[\vec{N}/\vec{x}])) \to^0_* M'$$

$M' \to^1 M''$ and that $[\![\texttt{fold}\ M]\!]\,(\vec{\gamma})\ \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. Setting M'' to be $(M[\vec{N}/\vec{x}])$, we are left to show that

$$[\![\texttt{fold}\ M]\!]\,(\vec{\gamma})\ \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M[\vec{N}/\vec{x}])$$

which is equal by definition of the interpretation function to

$$\mathsf{next}([\![M]\!]\,(\vec{\gamma}))\ \triangleright\mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}((M[\vec{N}/\vec{x}]))$$

The latter is equal by (39) to $\triangleright([\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ (M[\vec{N}/\vec{x}]))$ which is true by the guarded recursive hypothesis.

For the case $\Gamma \vdash \texttt{inl}\ M : \tau_1 + \tau_2$ we have to prove that

$$[\![\texttt{inl}\ M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2}\ \texttt{inl}\ M[\vec{M}/\vec{x}]$$

By definition of the interpretation function $[\![\texttt{inl}\ M]\!]\,(\vec{\gamma})$ is equal to $\eta(\mathsf{inl}([\![M]\!]\,(\vec{\gamma})))$. By definition of the logical relation we have to prove that there exists M' such that

$$(\texttt{inl}\ M)[\vec{M}/\vec{x}] \Rightarrow^0 \texttt{inl}\ M' \text{ and } [\![M]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1}\ M'.$$

The former is trivially true with $M' = M[\vec{M}/\vec{x}]$ and the latter is by induction hypothesis. The case for $\Gamma \vdash \texttt{inr}\ N : \tau_1 + \tau_2$ is similar.

For the case $\Gamma \vdash \texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N : \sigma$ we have to prove that

$$[\![\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})\ \mathcal{R}_\sigma\ (\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}]$$

For this it suffices to prove

$$[\![\lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N]\!]\,(\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2\to\sigma}\ (\lambda x.\texttt{case}\ x\ \texttt{of}\ x_1.M; x_2.N)[\vec{M}/\vec{x}] \tag{40}$$

and then applying this to $[\![L]\!](\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2}\ L[\vec{M}/\vec{x}]$. We prove (40) by guarded recursion thus assuming the statement is true later.

Assume y of type $[\![\tau_1+\tau_2]\!]$, L a term, and $y\ \mathcal{R}_{\tau_1+\tau_2}\ L$. We proceed by case analysis on y which is of type $[\![\tau_1+\tau_2]\!]$ which by definition is $L([\![\tau_1]\!]+[\![\tau_2]\!])$. In the case $y=\eta(\mathsf{inl}(z))$, where z is of type $[\![\tau_1]\!]$ we know by assumption that there exists L' s.t. $L\Rightarrow^0 \mathtt{inl}\ (L')$ and $z\ \mathcal{R}_{\tau_1}\ L'$. Since

$$[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma})(\eta(\mathsf{inl}(z)))=[\![M]\!](\vec{\gamma},z)$$

and

$$\mathtt{case}\ L\ \mathtt{of}\ x_1.M[\vec{M}/\vec{x}];x_2.N[\vec{M}/\vec{x}]\Rightarrow^0 M[\vec{M}/\vec{x}][L'/x_1]$$

by Lemma 5.2 we are left to prove

$$[\![M]\!](\vec{\gamma},\gamma)\ \mathcal{R}_\sigma\ \ M[\vec{M}/\vec{x}][L'/x_1]$$

which is true by induction hypothesis. The case $y=\eta(\mathsf{inl}(z))$ where z is of type $[\![\tau_2]\!]$ is similar.

Now consider the case of $y=\theta_{\tau_1+\tau_2}(z)$, where z is of type $\triangleright[\![\tau_1+\tau_2]\!]$. By induction hypothesis we know that $\theta_{\tau_1+\tau_2}(z)\ \mathcal{R}_{\tau_1+\tau_2}\ L$, thus there exist L' and L'' of type $\mathtt{Term}_{\mathtt{FPC}}$ such that $L\rightarrow^0_* L'$, $L'\rightarrow^1 L''$ and $z\ \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(L'')$.

Recall that we have assumed $\triangleright$ of (40), i.e.,

$$\triangleright([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma})\ \mathcal{R}_{\tau_1+\tau_2\to\sigma}\ (\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N)[\vec{M}/\vec{x}])$$

which is type equal to

$$\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma}))\ \triangleright\mathcal{R}_{\tau_1+\tau_2\to\sigma}\ \mathsf{next}((\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N)[\vec{M}/\vec{x}])$$

By Lemma 5.1 we can apply this to the assumption $z\ \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(L'')$ thus getting

$$\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma}))\circledast z\ \triangleright\mathcal{R}_\sigma\ \mathsf{next}(((\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N)[\vec{M}/\vec{x}])(L''))$$

Since $L'\rightarrow^1 L''$ we can apply Lemma 5.3 and obtain

$$\theta_\sigma(\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma}))\circledast z)\ \mathcal{R}_\sigma\ \mathtt{case}\ L'\ \mathtt{of}\ x_1.M[\vec{M}/\vec{x}];x_2.N[\vec{M}/\vec{x}]$$

By Lemma 5.2 with the fact that $L\rightarrow^0_* L'$ we get

$$\theta_\sigma(\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma}))\circledast z)\ \mathcal{R}_\sigma\ \mathtt{case}\ L\ \mathtt{of}\ x_1.M[\vec{M}/\vec{x}];x_2.N[\vec{M}/\vec{x}]$$

And finally by simplifying the left-hand side using Lemma 4.6:

$$\theta_\sigma(\mathsf{next}([\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma}))\circledast z)=[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma})(y)$$

thus getting

$$[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N]\!](\vec{\gamma})(y)\ \mathcal{R}_\sigma\ (\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M;x_2.N)[\vec{M}/\vec{x}](L)$$

as we wanted. □

From the Fundamental lemma we can now prove computational adequacy.

Theorem 5.5 (Intensional Computational Adequacy). If $M\colon 1$ is a closed term then $M\Rightarrow^k\langle\rangle$ iff $[\![M]\!](*)=\delta^k(\eta(*))$.

Proof. The left to right implication is soundness (Proposition 4.8). For the right to left implication note first that the Fundamental Lemma (Lemma 5.4) implies $\delta^k(\eta(*))\ \mathcal{R}_1\ M$. To complete the proof it suffices to show that $\delta_1^k(\eta(*))\ \mathcal{R}_1\ M$ implies $M \Rightarrow^k \langle\rangle$.

This is proved by guarded recursion and induction on k: the case of $k = 0$ is immediate by definition of $\mathcal{R}_1$. If $k = k' + 1$ first assume $\delta_1^k(\eta(*))\ \mathcal{R}_1\ M$. By definition of $\mathcal{R}$ there exist M' and M'' such that $M \to_*^0 M'$, $M' \to^1 M''$ and $\mathsf{next}(\delta_1^{k'}(\eta(*)))\ \triangleright\mathcal{R}_1\ \mathsf{next}(M'')$ which is type equal to $\triangleright(\delta_1^{k'}(\eta(*))\ \mathcal{R}_1\ M'')$. By the guarded recursion assumption we get $\triangleright(M'' \Rightarrow^{k'} \langle\rangle)$ which by definition implies $M \Rightarrow^k \langle\rangle$. □

From Theorem 5.5 one can deduce that whenever two terms have equal denotations they are contextually equivalent in a very intensional way, as we now describe. By a context, we mean a term $C[-]$ with a hole, and we say that $C[-]$ has type $\Gamma, \tau \to (-, 1)$ if $C[M]$ is a closed term of type 1, whenever $\Gamma \vdash M : \tau$.

Corollary 5.6. Suppose $\Gamma \vdash M : \tau$ and $[\![M]\!] = [\![N]\!]$. If $C[-]$ has type $\Gamma, \tau \to (-, 1)$ and $C[M] \Rightarrow^k \langle\rangle$ also $C[N] \Rightarrow^k \langle\rangle$.

As stated above, this is a very intensional result in the sense that whenever two FPC-denotable programs are equal we can derive that, under any context, they reduce to the same value with the same number of computational steps. This means that our model distinguishes programs whose input-output behaviour is the same, but the way in which the result is computed is computationally different. More specifically, two different algorithms implementing the same specification, but with a different computational complexity, will be considered different in the model. We explain how to recover this extensionality via a logical relation in the next section.

6. Extensional Computational Adequacy

Our model of FPC is intensional in the sense that it distinguishes between computations computing the same value in a different number of steps. In this section we define a logical relation which relates elements of the model if they differ only by a finite number of computation steps. In particular, this also means relating $\bot$ to $\bot$.

Such a relation must be defined on the types of the form $\forall\kappa. [\![\sigma]\!]$ rather than directly on the types $[\![\sigma]\!]$. To see why, consider the case of $\sigma = 1$, in which case $[\![\sigma]\!] = L1$. Recall from Section 2.1 that in the topos of trees model $L1$ is interpreted as the family of sets

$$L1(n) = \{\bot, 0, 1, \ldots, n-1\}$$

which describes computations terminating in at most $n-1$ steps or using at least n steps (corresponding to $\bot$). It cannot distinguish between termination in more than $n-1$ steps and real divergence. Our relation should relate a terminating value x in $L1(n)$ to any other terminating value, but not real divergence, which is impossible, if divergence cannot be distinguished from slow termination. Another, more semantic, way to phrase the problem is that termination as described by the subsets $\{0, 1, \ldots, n-1\}$ of $L1(n)$ for each n does not form a subobject of $L1$.

On the other hand, if $L1 \cong 1 + \triangleright_\kappa 1$ then, as we saw in section the type

$$L^{\mathrm{gl}} A \overset{\mathrm{def}}{=\!=} \forall\kappa.LA$$

is a coinductive solution to the type equation

$$L^{\mathrm{gl}} 1 \cong 1 + L^{\mathrm{gl}} 1$$

Semantically $L^{\mathrm{gl}}1$ is modelled as the set $\mathbb{N}+\{\bot\}$, and termination is the subset of this corresponding to the left inclusion of $\mathbb{N}$. So on the global level we can, at least semantically, distinguish between termination and non-termination. This is reflected syntactically in Lemma 6.19.

We refer to $\forall\kappa. [\![1]\!] \equiv L^{\mathrm{gl}}1$ as the *global interpretation* of the type 1 because it captures the global behaviour (computable in *any* number of steps) of terms of type 1. We now extend this to the global interpretation of all types and terms and give the definition of the logical relation.

6.1. *Global interpretation of types and terms*

Recall that the developments above should be read as taking place in a context of an implicit clock κ. To be consistent with the notation of the previous sections, κ will remain implicit in the denotations of types and terms, although one might choose to write e.g. $[\![\sigma]\!]^\kappa$ to make the clock explicit.

We define global interpretations of types and terms as follows:

$$[\![\sigma]\!]^{\mathrm{gl}} \overset{\mathrm{def}}{=\!=} \forall\kappa. [\![\sigma]\!]$$

$$[\![M]\!]^{\mathrm{gl}} \overset{\mathrm{def}}{=\!=} \Lambda\kappa. [\![M]\!]$$

such that if $\Gamma \vdash M : \tau$, then

$$[\![M]\!]^{\mathrm{gl}} \colon \forall\kappa.([\![\Gamma]\!] \to [\![\tau]\!])$$

Note that $[\![\sigma]\!]^{\mathrm{gl}}$ is a wellformed type, because $[\![\sigma]\!]$ is a wellformed type in context $\sigma \colon \mathtt{Type}_{\mathtt{FPC}}$ and $\mathtt{Type}_{\mathtt{FPC}}$ is an inductive type formed without reference to clocks or guarded recursion, thus κ does not appear in $\mathtt{Type}_{\mathtt{FPC}}$. By a similar argument $[\![M]\!]^{\mathrm{gl}}$ is welltyped.

Define for all σ the *delay* operator $\delta^{\mathrm{gl}}_\sigma \colon [\![\sigma]\!]^{\mathrm{gl}} \to [\![\sigma]\!]^{\mathrm{gl}}$ as follows

$$\delta^{\mathrm{gl}}_\sigma(x) \overset{\mathrm{def}}{=\!=} \Lambda\kappa.\delta_\sigma(x[\kappa]) \tag{41}$$

Similarly for LA, $\delta^{\mathrm{gl}}_{LA}(x) \overset{\mathrm{def}}{=\!=} \Lambda\kappa.\delta_{LA}(x[\kappa])$.

With these definitions we can lift the adequacy theorem to the global points. To prove the denotational model is computationally adequate w.r.t. the standard big-step operational semantics $\Downarrow^n$ we take the global view points of the the denotational semantics in order to be able to remove the occurrences of the $\triangleright$ operator.

Corollary 6.1 (Computational adequacy). If $M \colon 1$ is a closed term and n is a natural number, then $M \Downarrow^n \langle\rangle$ iff $\forall\kappa. [\![M]\!](*) = \delta^n(\eta(*))$.

Proof. Since $\forall\kappa.(-)$ is functorial, Theorem 5.5 gives $\forall\kappa. [\![M]\!](*) = \delta^n(\eta(*))$ iff $\forall\kappa.M \Rightarrow^n \langle\rangle$, which by Lemma 3.2 holds iff $M \Downarrow^n \langle\rangle$. □

We have now a semantics that implies the standard operational semantics. However, we are still not able to prove that if two programs are equal they are going to be contextually equivalent w.r.t. the input-output behaviour. To achieve so, we need to lift the explicit step-indexing as well.

6.2. *A weak bisimulation relation for the lifting monad*

Before defining the logical relation on the interpretation of types, we define a relational version of the guarded recursive lifting monad L. If applied to the identity relation on a type A in which κ does not appear, we obtain a weak bisimulation relation similar to the one defined by Capretta [Cap05] for the coinductive partiality monad.

Definition 6.2. For a relation $R : A \times B \to \mathcal{U}$ define the lifting $LR \ : LA \times LB \to \mathcal{U}$ by guarded recursion and case analysis on the elements of LA and LB:

$$\begin{aligned}
\eta(x)\ LR\ \eta(y) &\overset{\text{def}}{=\!=} x\ R\ y \\
\eta(x)\ LR\ \theta_{LB}(y) &\overset{\text{def}}{=\!=} \Sigma n, y'.\theta_{LB}(y) = \delta^n_{LB}(\eta(y')) \text{ and } x\ R\ y' \\
\theta_{LA}(x)\ LR\ \eta(y) &\overset{\text{def}}{=\!=} \Sigma n, x'.\theta_{LA}(x) = \delta^n_{LA}(\eta(x')) \text{ and } x'\ R\ y \\
\theta_{LA}(x)\ LR\ \theta_{LB}(y) &\overset{\text{def}}{=\!=} x \triangleright LR\ y
\end{aligned}$$

Intuitively, LR relates two elements if they either both diverge, or both both converge to elements related in R. For example, $\perp$ as defined in Section 4 is always related to itself which can be shown by guarded recursion as follows. Suppose $\triangleright(\perp\ LR\ \perp)$. Since $\perp = \theta(\mathsf{next}(\perp))$, to prove $\perp\ LR\ \perp$, we must prove $\mathsf{next}(\perp) \triangleright LR\ \mathsf{next}(\perp)$. But, this type is equal to the assumption $\triangleright(\perp\ LR\ \perp)$ by (39).

By the intuition given for LR below, it should be possible to add or remove ticks on either side without breaking relatedness in LR. The next lemma shows half of this.

Lemma 6.3. If $R : A \times B \to \mathcal{U}$, and $x\ LR\ y$ then $x\ LR\ \delta_{LB}(y)$ and $\delta_{LA}(x)\ LR\ y$.

Proof. Assume $x\ LR\ y$. We show $x\ LR\ \delta_{LB}(y)$. The proof is by guarded recursion, hence we first assume:

$$\triangleright(\Pi x : LA, y : LB.x\ LR\ y \Rightarrow x\ LR\ \delta_{LB}(y)). \tag{42}$$

We proceed by case analysis on x and y. If $x = \eta(x')$, then, since $x\ LR\ y$, there exist n and y' such that $y = \delta^n_{LB}(\eta(y'))$ and $x'\ R\ y'$. So then $\delta_{LB}(y) = \delta^{n+1}_{LB}(\eta(y'))$, from which it follows that $x\ LR\ \delta_{LB}(y)$.

For the case where $x = \theta_{LA}(x')$ and $y = \eta(v)$, it suffices to show that $\delta^n_{LA}(\eta(w))\ LR\ \eta(v)$ implies $\delta^n_{LA}(\eta(w))\ LR\ \delta_{LB}(\eta(v))$. The case of $n = 0$ was proved above. For $n = m + 1$ we know that if $\delta^n_{LA}(\eta(w))\ LR\ \eta(v)$ also $\delta^m_{LA}(\eta(w))\ LR\ \eta(v)$ holds by definition, and this implies

$$\triangleright(\delta^m_{LA}(\eta(w))\ LR\ \eta(v))$$

But this type can be rewritten as follows

$$\begin{aligned}\triangleright(\delta^m_{LA}(\eta(w))\ LR\ \eta(v)) &\equiv \mathsf{next}(\delta^m_{LA}(\eta(w))\ \triangleright LR\ \ \mathsf{next}(\eta(v))) \\ &\equiv \theta_{LA}(\mathsf{next}(\delta^m_{LA}(\eta(w))))\ LR\ \theta_{LB}(\mathsf{next}(\eta(v)))) \\ &\equiv \delta^n_{LA}(\eta(w))\ LR\ \delta_{LB}(\eta(v))\end{aligned}$$

proving the case.

Finally, the case when $x = \theta_{LA}(x')$ and $y = \theta_{LB}(y')$. The assumption in this case is $x' \triangleright LR\, y'$, which means by (38),

$$\triangleright\, [x'' \leftarrow x', y'' \leftarrow y']\, .x''\ LR\ y''$$

By the guarded recursion hypothesis (42) we get

$$\triangleright\, [x'' \leftarrow x', y'' \leftarrow y']\, .x''\ LR\ \delta_{LB}(y'')$$

which can be rewritten to

$$\triangleright\, [x'' \leftarrow x', y'' \leftarrow y']\, .x''\ LR\ \theta_{LB}(\mathsf{next}(y'')) \tag{43}$$

By (37) there is an inhabitant of the type

$$\triangleright\, [x'' \leftarrow x', y'' \leftarrow y']\, .(\mathsf{next}(y'') = y')$$

and thus (43) implies $\triangleright\, [x'' \leftarrow x']\, .x''\ LR\ \theta_{LB}(y')$, which, by (39) and since $y = \theta_{LB}(y')$ equals $x' \triangleright LR\ \ \mathsf{next}(y)$. By definition, this is

$$\theta_{LA}(x')\ LR\ \theta_{LB}(\mathsf{next}(y))$$

which since $x = \theta_{LA}(x')$ is $x\ LR\ \delta_{LB}(y)$. □

We can lift this result to L^{gl} as follows. Suppose $R : A \times B \to \mathcal{U}$ and κ not in A or B. Define $L^{\mathrm{gl}}R\ : L^{\mathrm{gl}}A \times L^{\mathrm{gl}}B \to \mathcal{U}$ as

$$x\ L^{\mathrm{gl}}R\ y \overset{\mathrm{def}}{=\!=} \forall\kappa.x[\kappa]\ LR\ y[\kappa]$$

Lemma 6.4. Let $x : L^{\mathrm{gl}}A$ and $y : L^{\mathrm{gl}}B$. If $x\ L^{\mathrm{gl}}R\ y$ then $x\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}(y)$ and $\delta^{\mathrm{gl}}(x)\ L^{\mathrm{gl}}R\ y$.

Proof. Follows directly from Lemma 6.3. □

One might expect that $\delta_{LA}(x)\ LR\ \delta_{LB}(y)$ implies $x\ LR\ y$. This is not true, it only implies $\triangleright(x\ LR\ y)$. In the case of L^{gl}, however, we can use force to remove the $\triangleright$.

Lemma 6.5. For all $x : L^{\mathrm{gl}}A$ and $y : L^{\mathrm{gl}}B$ and for all $R : A\times B \to \mathcal{U}$, if $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$ then $x\ L^{\mathrm{gl}}R\ y$.

Proof. Assume $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$. We can rewrite this type by unfolding definitions

$$\begin{aligned}
x \ \approx_1 \ y &\overset{\text{def}}{=\!=} x \ L(=_1) \ y \\
x \ \approx_{\tau_1+\tau_2} \ y &\overset{\text{def}}{=\!=} x \ L(\ \approx_{\tau_1} \ + \ \approx_{\tau_2} \ ) \ y \\
x \ \approx_{\tau_1\times\tau_2} \ y &\overset{\text{def}}{=\!=} \pi_1(x) \ \approx_{\tau_1} \ \pi_1(y) \text{ and } \pi_2(x) \ \approx_{\tau_2} \ \pi_2(y) \\
f \ \approx_{\sigma\to\tau} \ g &\overset{\text{def}}{=\!=} \Pi(x, y : [\![\sigma]\!]).x \ \approx_\sigma \ y \to f(x) \ \approx_\tau \ g(y) \\
x \ \approx_{\mu\alpha.\tau} \ y &\overset{\text{def}}{=\!=} x \rhd \approx_{\tau[\mu\alpha.\tau/\alpha]} \ y
\end{aligned}$$

Fig. 8. The logical relation $\approx_\tau$ is a predicate over denotations of τ of type $[\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$

and (39) as follows.

$$\begin{aligned}
\delta^{\text{gl}}_{LA}(x) \ L^{\text{gl}}R \ \delta^{\text{gl}}_{LB}(y) &\equiv \forall\kappa.(\delta^{\text{gl}}_{LA}(x))[\kappa] \ LR \ (\delta^{\text{gl}}_{LB}(y))[\kappa] \\
&\equiv \forall\kappa.(\delta_{LA}(x[\kappa])) \ LR \ (\delta_{LB}(y[\kappa])) \\
&\equiv \forall\kappa.(\mathsf{next}(x[\kappa]) \rhd LR \ \mathsf{next}(y[\kappa])) \\
&\equiv \forall\kappa.\rhd(x[\kappa] \ LR \ (y[\kappa]))
\end{aligned}$$

Using force this implies $\forall\kappa.(x[\kappa] \ LR \ (y[\kappa]))$ which is equal to $x \ L^{\text{gl}}R \ y$. □

Lemma 6.6. For all x of type $L^{\text{gl}}A$ and y of type $L^{\text{gl}}B$, if $\delta^{\text{gl}}_{LA}(x) \ L^{\text{gl}}R \ y$ then $x \ L^{\text{gl}}R \ y$.

Proof. Assume $\delta^{\text{gl}}_{LA}(x) \ L^{\text{gl}}R \ y$. Then by applying Lemma 6.4 we get $\delta^{\text{gl}}_{LA}(x) \ L^{\text{gl}}R \ \delta^{\text{gl}}_{LB}(y)$ and by applying Lemma 6.5 we get $x \ L^{\text{gl}}R \ y$. □

With this machinery in place we can now define a relation on the semantics that relates programs that produce the same value (or both diverge) and that discards the information about the number of delays used.

6.3. *Relating terms up to extensional equivalence*

Figure 8 defines for each FPC type τ the logical relation $\approx_\tau : [\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$. The definition is by guarded recursion, and well-definedness can be formalised using an argument similar to that used for well-definedness of θ in equation (24). The case of recursive types is well typed by Lemma 4.2. The figure uses the following lifting of relations to sum types.

Definition 6.7. Let $R : A \times B \to \mathcal{U}$ and $R' : A' \times B' \to \mathcal{U}$. Define $(R + R') : (A + A') \times (B + B') \to \mathcal{U}$ by case analysis as follows (omitting false cases)

$$\begin{aligned}
\mathsf{inl}(x) \ (R + R') \ \mathsf{inl}(y) &\overset{\text{def}}{=\!=} x \ R \ y \\
\mathsf{inl}(x) \ (R + R') \ \mathsf{inl}(y) &\overset{\text{def}}{=\!=} x \ R' \ y
\end{aligned}$$

The logical relation can be generalised to open terms and the global interpretation of terms as in the next two definitions.

Definition 6.8. For $\Gamma \equiv x_1 : \sigma_1, \cdots, x_n : \sigma_n$ and for f, g of type $[\![\Gamma]\!] \to [\![\tau]\!]$ define

$$f \ \approx_{\Gamma,\tau} \ g \overset{\text{def}}{=\!=} \Pi(\vec{x}, \vec{y} : [\![\vec{\sigma}]\!]).\vec{x} \ \approx_{\vec{\sigma}} \ \vec{y} \to f(\vec{x}) \ \approx_{\tau} \ g(\vec{y})$$

For x, y of type $\forall\kappa.([\![\Gamma]\!] \to [\![\tau]\!])$ define

$$x \ \approx^{\text{gl}}_{\Gamma,\tau} \ y \overset{\text{def}}{=\!=} \forall\kappa.x[\kappa] \ \approx_{\Gamma,\tau} \ y[\kappa]$$

Perhaps surprisingly, this relation is not reflexive. For example the function $f : L1 \to L1$ defined by $f(\eta(*)) = \eta(*)$ and $f(\theta_{L1}(x)) = \bot$ does not satisfy $f \ \approx_{1\to 1} \ f$. On the other hand, the denotation of any term is always related to itself, as the following proposition states.

Proposition 6.9. If $\Gamma \vdash M : \sigma$, then $[\![M]\!] \ \approx_{\Gamma,\sigma} \ [\![M]\!]$.

The rest of this section is devoted to the proof of Proposition 6.9 which is important for the proof of the extensional computational adequacy theorem. To prove the proposition we first establish some basic properties of the logical relation. The first lemma states that delayed application $\circledast$ respects the logical relation.

Lemma 6.10. For all f, g of type $\triangleright [\![\tau \to \sigma]\!]$ and x, y of type $\triangleright [\![\tau]\!]$, if $f \triangleright \approx_{\tau\to\sigma} \ g$ and $x \triangleright \approx_{\tau} \ y$ then $(f \circledast x) \triangleright \approx_{\sigma} \ (g \circledast y)$.

Proof. Assume $f \ \triangleright \ \approx_{\tau\to\sigma} \ g$ and $x \ \triangleright \ \approx_{\tau} \ y$. By Definition 38 $f \ \triangleright \ \approx_{\tau\to\sigma} \ g$ is $\triangleright [f' \leftarrow f, g' \leftarrow g] .(f' \ \approx_{\tau\to\sigma} \ g')$ which by unfolding the definition of $\approx_{\tau\to\sigma}$ is

$$\triangleright [f' \leftarrow f, g' \leftarrow g] .(\Pi(x, y : [\![\sigma]\!]).x \ \approx_{\tau} \ y \to f'(x) \ \approx_{\sigma} \ g'(y))$$

By applying this to x, y and $x \triangleright \approx_{\tau} \ y$ using the dependent version of $\circledast$ defined in (36) we get

$$\triangleright [f' \leftarrow f, g' \leftarrow g, a \leftarrow x, b \leftarrow y] .(f'(a) \ \approx_{\sigma} \ g'(b))$$

By (39) this is equal to

$$\mathsf{next}\,[f' \leftarrow f, a \leftarrow x] .(f'(a)) \triangleright \approx_{\sigma} \ \mathsf{next}\,[g' \leftarrow g, b \leftarrow y] .(g'(b))$$

which by rule (34) is equal to

$$(f \circledast x) \triangleright \approx_{\sigma} \ (g \circledast y)$$

□

Next we show that θ respects the logical relation.

Lemma 6.11. Let x, y of type $\triangleright [\![\sigma]\!]$, if $(x \triangleright \approx_{\sigma} \ y)$ then $\theta_{\sigma}(x) \ \approx_{\sigma} \ \theta_{\sigma}(y)$

Proof. We prove the statement by guarded recursion. Thus, we assume the statement holds "later" and we proceed by induction on σ. All the cases for the types that are interpreted using the lifting – namely the unit type and the sum type – in Definition 6.2 hold by definition of the lifting relation.

First the case for the function types: Assume $\sigma = \tau_1 \to \tau_2$ and assume f and g of type $\triangleright [\![\tau_1 \to \tau_2]\!]$ such that $f \triangleright \approx_{\tau_1\to\tau_2} \ g$. We must show that if $x, y : [\![\tau_1]\!]^{\kappa}$ and $x \ \approx_{\tau_1} \ y$ then $(\theta_{\tau_1\to\tau_2}(f))(x) \ \approx_{\tau_2} \ (\theta_{\tau_1\to\tau_2}(g))(y))$.

So suppose $x \approx_{\tau_1} y$, then also $\triangleright(x \approx_{\tau_1} y)$, which by (39) is equal to $\mathsf{next}(x) \triangleright \approx_{\tau_1} \mathsf{next}(y)$. By applying Lemma 6.10 to this and $f \triangleright \approx_{\tau_1 \to \tau_2} g$ we get

$$f \circledast (\mathsf{next}\, x) \triangleright \approx_{\tau_2} g \circledast \mathsf{next}\, y$$

By induction hypothesis on τ_2, we get $\theta_{\tau_2}(f \circledast (\mathsf{next}\, x)) \approx_{\tau_2} \theta_{\tau_2}(g \circledast (\mathsf{next}\, y))$. We conclude by observing that by definition of θ, $\theta_{\tau_1 \to \tau_2}(f)(x) = \theta_{\tau_2}(f \circledast \mathsf{next}(x))$.

The case of the product is straightforward.

For the case of recursive types, assume $\phi \triangleright \approx_{\mu\alpha.\tau} \psi$. This is type equal to

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi] .(x \approx_{\mu\alpha.\tau} y)$$

By definition this is equal to

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi] .(x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} y)$$

By the guarded recursion hypothesis we get

$$\triangleright [x \leftarrow \phi, y \leftarrow \psi] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(x) \approx_{\tau[\mu\alpha.\tau/\alpha]} \theta_{\tau[\mu\alpha.\tau/\alpha]}(y))$$

By (39) this is equal to

$$(\mathsf{next}\, [x \leftarrow \phi] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(x))) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} (\mathsf{next}\, [y \leftarrow \psi] .(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)))$$

This equals

$$(\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \phi \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} (\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \psi$$

By definition $\mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \phi$ is equal to $\theta_{\mu\alpha.\tau}(\phi)$ thus we can derive

$$\theta_{\mu\alpha.\tau}(\phi) \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \theta_{\mu\alpha.\tau}(\psi)$$

which by definition of $\approx_{\mu\alpha.\tau}$ is

$$\theta_{\mu\alpha.\tau}(\phi) \approx_{\mu\alpha.\tau} \theta_{\mu\alpha.\tau}(\psi)$$

□

Next we generalise Lemma 6.3 to hold for $\approx_\sigma$ for all σ.

Lemma 6.12. Let σ be a closed FPC type and let x and y of type $[\![\sigma]\!]$, if $x \approx_\sigma y$ then $\delta_\sigma(x) \approx_\sigma y$ and $x \approx_\sigma \delta_\sigma(y)$.

Proof. The proof is by guarded recursion and then by induction on the type σ. Thus, assume this lemma holds "later", and proceed by induction on σ. The cases of the unit type and coproduct follow from Lemma 6.3 and the case of products follows by induction from the fact that $\delta_{\tau_i}(\pi_i(x)) = \pi_i(\delta_{\tau_1 \times \tau_2}(x))$, for $i = 1, 2$. The case of function types follows from the fact that $\delta_{\sigma \to \tau}(f)(x) = \delta_\tau(f(x))$.

For the case of recursive types assume $x \approx_{\mu\alpha.\tau} y$. Note that

$$\begin{aligned} x \approx_{\mu\alpha.\tau} y &\equiv x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} y \\ &\equiv \triangleright [x' \leftarrow x, y' \leftarrow y] .x' \approx_{\tau[\mu\alpha.\tau/\alpha]} y' \end{aligned}$$

Using the dependent version of $\circledast$ as defined in (36) we can apply the guarded recursion assumption to conclude $\triangleright [x' \leftarrow x, y' \leftarrow y] .x' \approx_{\tau[\mu\alpha.\tau/\alpha]} \delta_{\tau[\mu\alpha.\tau/\alpha]}(y')$. Note that the delay operator is the composition $\theta \circ \mathsf{next}$, thus y' appears under next. We can thus employ (37) to derive that $\triangleright [x' \leftarrow x] .x' \ \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \theta_{\tau[\mu\alpha.\tau/\alpha]}(y)$. From here we conclude by a simple computation:

$$\begin{aligned}
\triangleright [x' \leftarrow x] .x' \ \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \theta_{\tau[\mu\alpha.\tau/\alpha]}(y) &\equiv x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)) \\
&\equiv x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast \mathsf{next}(y) \\
&\equiv x \triangleright \approx_{\tau[\mu\alpha.\tau/\alpha]} \ \theta_{\mu\alpha.\tau}(\mathsf{next}(y)) \\
&\equiv x \ \approx_{\mu\alpha.\tau} \ \delta_{\mu\alpha.\tau}(y)
\end{aligned}$$

□

Lemma 6.13. Let σ be a closed FPC type and let x, y of type $[\![\sigma]\!]^{\mathrm{gl}}$. If $x \ \approx^{\mathrm{gl}}_{\sigma} \ y$ then $x \ \approx^{\mathrm{gl}}_{\sigma} \ \delta^{\mathrm{gl}}_{\sigma}(y)$ and $\delta^{\mathrm{gl}}_{\sigma}(x) \ \approx^{\mathrm{gl}}_{\sigma} \ y$

Proof. Direct from Lemma 6.12. □

Proof of Proposition 6.9 The proof is by induction on M and we just show the interesting cases. In all cases we will assume $\Gamma \equiv x_1 : \sigma_1 .., x_n : \sigma_n$ and that we are given $\vec{x}$ and $\vec{y}$ such that $\vec{x} \ \approx_{\vec{\sigma}} \ \vec{y}$.

For case expressions, to prove that

$$[\![\mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{x}) \ \approx_{\tau} \ [\![\mathtt{case}\ L\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{y})$$

it suffices to prove that

$$[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{x}) \ \approx_{\sigma\to\tau} \ [\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{y}) \tag{44}$$

Thus that for all x, y s.t. $x \ \approx_{\tau_1+\tau_2} \ y$

$$[\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{x})(x) \ \approx_{\tau} \ [\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{y})(y)$$

holds. We prove (44) by guarded recursion. Thus, we assume the statement holds "later" and we proceed by case analysis on x and y. When x is $\eta(x')$ and y is $\eta(y')$ either x' and y' are both in the left component or they are both in the right component of the sum. The former case $x' = \mathsf{inl}(x'')$ and $y' = \mathsf{inl}(y'')$ reduces to

$$[\![M]\!]\,(\vec{x}, x'') \ \approx_{\tau} \ [\![M]\!]\,(\vec{y}, y'')$$

which follows from the induction hypothesis, and the latter case is similar.

Now consider the case of $x = \theta_{\tau_1+\tau_2}(x')$ and $y = \eta(v)$. Since by assumption $x \ \approx_{\tau_1+\tau_2} \ y$ there exists n and w such that $x = \delta^n_{\tau_1+\tau_2}(\eta(w))$ and $w \ \approx_{\tau_1+\tau_2} \ v$. As before, v and w must be in the same component of the coproduct, so assume $w = \mathsf{inl}(w')$ and $v = \mathsf{inl}(v')$ such that $w' \ \approx_{\tau_1} \ v'$. By induction hypothesis we know that $[\![M]\!]\,(\vec{x}) \ \approx_{\tau_1\to\tau} \ [\![M]\!]\,(\vec{y})$ and thus that $[\![M]\!]\,(\vec{x})(w') \ \approx_{\tau} \ [\![M]\!]\,(\vec{y})(v')$. By Lemma 6.12 this implies $\delta^n_{\tau}([\![M]\!]\,(\vec{x})(w')) \ \approx_{\tau} \ [\![M]\!]\,(\vec{y})(v')$. Since

$$[\![M]\!]\,(\vec{x})(w') = [\![\lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N]\!]\,(\vec{x})(\eta(w)),$$

by Lemma 4.6 we get

$$\begin{aligned}\delta^n_\tau(\llbracket M\rrbracket(\vec{x})(w')) &= \delta^n_\tau(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x})(\eta(w))) \\ &= \llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x})(\delta^n_{\tau_1+\tau_2}(\eta(w)))\end{aligned}$$

and thus we conclude

$$\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x})(\delta^n_{\tau_1+\tau_2}(\eta(w))) \ \approx_{\tau_1+\tau_2}\ \llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x})(\eta(v))$$

which is what we wanted to show.

The last case is when x is $\theta_{\tau_1+\tau_2}(x')$ and y is $\theta_{\tau_1+\tau_2}(y')$. By guarded recursion we know that

$$\triangleright(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x})\ \approx_{\tau_1+\tau_2\to\tau}\ (\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y}))$$

By (39) we get

$$\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x}))\ \triangleright\approx_{\tau_1+\tau_2\to\tau}\ \mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{y}))$$

Since the assumption $\theta_{\tau_1+\tau_2}(x')\ \approx_{\tau_1+\tau_2}\ \theta_{\tau_1+\tau_2}(y')$, means that $x'\ \triangleright\approx_{\tau_1+\tau_2}\ y'$, by Lemma 6.10 this implies

$$\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x}) \circledast x'\ \triangleright\approx_\tau\ \mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y}) \circledast y'$$

By Lemma 6.11 this implies

$$\theta_\tau(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{x}) \circledast x')\ \approx_\tau\ \theta_\tau(\mathsf{next}(\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y}) \circledast y')$$

By Lemma 4.6 we conclude that

$$\llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket(\vec{x})(\theta_{\tau_1+\tau_2}(x'))\ \approx_\tau\ \llbracket \lambda x.\mathtt{case}\ x\ \mathtt{of}\ x_1.M; x_2.N\rrbracket)(\vec{y})(\theta_{\tau_1+\tau_2}(y'))$$

proving the case.

Finally we prove the two cases for the recursive types. We first consider the case for $\mathtt{unfold}\ M$ of type $\tau[\mu\alpha.\tau/\alpha]$. We have to show that

$$\llbracket \mathtt{unfold}\ M\rrbracket(\vec{x})\ \approx_{\tau[\mu\alpha.\tau/\alpha]}\ \llbracket \mathtt{unfold}\ M\rrbracket(\vec{y})$$

By induction hypothesis we know that $\llbracket M\rrbracket(\vec{x})\ \approx_{\mu\alpha.\tau}\ \llbracket M\rrbracket(\vec{y})$ which by definition of $\approx_{\mu\alpha.\tau}$ is $\llbracket M\rrbracket(\vec{x}) \triangleright\approx_{\tau[\mu\alpha.\tau/\alpha]}\ \llbracket M\rrbracket(\vec{y})$. By Lemma 6.11 we get

$$\theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M\rrbracket(\vec{x}))\ \approx_{\tau[\mu\alpha.\tau/\alpha]}\ \theta_{\tau[\mu\alpha.\tau/\alpha]}(\llbracket M\rrbracket(\vec{y}))$$

and by definition of the interpretation function this is what we wanted.

Now the case for $\mathtt{fold}\ M$ of type $\mu\alpha.\tau$. By induction hypothesis we know that $\llbracket M\rrbracket(\vec{x})\ \approx_{\tau[\mu\alpha.\tau/\alpha]}\ \llbracket M\rrbracket(\vec{y})$ which implies $\triangleright(\llbracket M\rrbracket(\vec{x})\ \approx_{\tau[\mu\alpha.\tau/\alpha]}\ \llbracket M\rrbracket(\vec{y}))$ which is equal to

$$\mathsf{next}(\llbracket M\rrbracket(\vec{x})) \triangleright\approx_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(\llbracket M\rrbracket(\vec{y})).$$

By definition of $\approx_{\mu\alpha.\tau}$ this is precisely $\mathsf{next}(\llbracket M\rrbracket(\vec{x}))\ \approx_{\mu\alpha.\tau}\ \mathsf{next}(\llbracket M\rrbracket(\vec{y}))$ which by definition of the interpretation function is

$$\llbracket \mathtt{fold}\ M\rrbracket(\vec{x})\ \approx_{\mu\alpha.\tau}\ \llbracket \mathtt{fold}\ M\rrbracket(\vec{y})$$

□

6.4. *Extensional computational adequacy*

Contextual equivalence of FPC is defined in the standard way by observing convergence at unit type. We first define the language of contexts. These are FPC programs with a hole $[-]$ defined inductively as in the next definition.

Definition 6.14 (Contexts).

$$\begin{aligned}\mathsf{Ctx} := {} & [-] \mid \lambda x.\,\mathsf{Ctx} \mid \mathsf{Ctx}\,N \mid M\,\mathsf{Ctx} \\ & \mid \texttt{inl Ctx} \mid \texttt{inr Ctx} \mid \langle \mathsf{Ctx}, M\rangle \mid \langle M, \mathsf{Ctx}\rangle \mid \texttt{fst Ctx} \mid \texttt{snd Ctx} \\ & \mid \texttt{case Ctx of } x_1.M; x_2.N \\ & \mid \texttt{case } L \texttt{ of } x_1.\,\mathsf{Ctx}; x_2.N \mid \texttt{case } L \texttt{ of } x_1.M; x_2.\,\mathsf{Ctx} \\ & \mid \texttt{unfold Ctx} \mid \texttt{fold Ctx}\end{aligned}$$

Intuitively, a context is a term that takes a term and returns a new term.

We define the “fill hole” function $\cdot[\cdot] : \mathsf{Ctx} \times \texttt{OTerm}_{\texttt{FPC}} \to \texttt{OTerm}_{\texttt{FPC}}$ by induction on the context in the standard way. Note that this may capture free variables in the term being substituted.

We say that a context C has type $(\Gamma, \sigma) \to (\Delta, \tau)$ if $\Delta \vdash C[M] : \tau$ whenever $\Gamma \vdash M : \sigma$. This can be captured by a typing relation on contexts as defined in Figure 9. Next we define contextual equivalence using the big-step semantics $\Downarrow$. This states that two program are contextually equivalent if no context can distinguish them. Using $\Downarrow$ (instead of $\Downarrow^k$) ensures that we capture the standard notion of contextual equivalence, thus that two programs producing the same value will be equivalent no matter how many steps they take to terminate.

Definition 6.15. Let $\Gamma \vdash M, N : \tau$. We say that M, N are contextually equivalent, written $M \approx_{\texttt{CTX}} N$, if for all contexts C of type $(\Gamma, \tau) \to (-, 1)$

$$C[M] \Downarrow \langle\rangle \iff C[N] \Downarrow \langle\rangle$$

Finally we can state the main theorem of this section. Using the global view of the logical relation $\approx$ we can prove if the denotations of two programs are related then they are contextual equivalent in the extensional sense.

Theorem 6.16 (Extensional Computational Adequacy). If $\Gamma \vdash M, N : \tau$ and $[\![M]\!]^{\mathrm{gl}} \approx^{\mathrm{gl}}_{\Gamma,\tau} [\![N]\!]^{\mathrm{gl}}$ then $M \approx_{\texttt{CTX}} N$.

To prove this theorem, we need the following lemma stating that contexts preserve the logical relation.

Lemma 6.17. Let $\Gamma \vdash M : \tau$ and $\Gamma \vdash N : \tau$ and suppose $[\![M]\!] \approx_{\Gamma,\tau} [\![N]\!]$. If C is a context such that $C : \Gamma, \tau \to \Delta, \sigma$ then $[\![C[M]]\!] \approx_{\Delta,\sigma} [\![C[N]]\!]$

Proof. The proof is by induction on C and most cases can be proved either very similarly to corresponding cases of Proposition 6.9, or by direct application of Proposition 6.9. We show how to do the latter in two cases.

For a context $\texttt{unfold}\, C$ of type $(\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])$ we have by induction that C

$$\frac{}{- : (\Gamma, \tau) \to (\Gamma, \tau)} \qquad \frac{C : (\Gamma, \tau) \to ((\Delta, x : \sigma'), \sigma)}{(\lambda x.C) : (\Gamma, \tau) \to (\Delta, \sigma' \to \sigma)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau' \to \sigma) \qquad \Delta \vdash N : \tau'}{CN : (\Gamma, \tau) \to (\Delta, \sigma)} \qquad \frac{C : (\Gamma, \sigma) \to (\Delta, \tau') \qquad \Delta \vdash M : \tau' \to \sigma}{MC : (\Gamma, \sigma) \to (\Delta, \sigma)}$$

$$\frac{C : (\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)}{\mathtt{unfold}\, C : (\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])} \qquad \frac{C : (\Gamma, \sigma) \to (\Delta, \tau[\mu\alpha.\tau/\alpha])}{\mathtt{fold}\, C : (\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)}{\mathtt{fst}\, C : (\Gamma, \tau) \to (\Delta, \tau_1)} \qquad \frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)}{\mathtt{snd}\, C : (\Gamma, \tau) \to (\Delta, \tau_2)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1) \qquad \Delta \vdash N : \tau_2}{\langle C, N \rangle : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)} \qquad \frac{C : (\Gamma, \tau) \to (\Delta, \tau_2) \qquad \Delta \vdash M : \tau_1}{\langle M, C \rangle : (\Gamma, \tau) \to (\Delta, \tau_1 \times \tau_2)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2) \qquad \Delta, x_1 : \tau_1 \vdash M : \sigma \qquad \Delta, x_2 : \tau_2 \vdash N : \sigma}{\mathtt{case}\, C \,\mathtt{of}\, x_1.M; x_2.N : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{\Delta \vdash L : \tau_1 + \tau_2 \qquad C : (\Gamma, \tau) \to ((\Delta, x_1 : \tau_1), \sigma) \qquad \Delta, x_2 : \tau_2 \vdash N : \sigma}{\mathtt{case}\, L \,\mathtt{of}\, x_1.C; x_2.N : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{\Delta \vdash L : \tau_1 + \tau_2 \qquad \Delta, x_1 : \tau_1 \vdash M : \sigma \qquad C : (\Gamma, \tau) \to ((\Delta, x_2 : \tau_2), \sigma)}{\mathtt{case}\, L \,\mathtt{of}\, x_1.M; x_2.C : (\Gamma, \tau) \to (\Delta, \sigma)}$$

$$\frac{C : (\Gamma, \tau) \to (\Delta, \tau_1)}{\mathtt{inl}\, C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2)} \qquad \frac{C : (\Gamma, \tau) \to (\Delta, \tau_2)}{\mathtt{inr}\, C : (\Gamma, \tau) \to (\Delta, \tau_1 + \tau_2)}$$

Fig. 9. Typing judgment for contexts

has type $(\Gamma, \sigma) \to (\Delta, \mu\alpha.\tau)$ and thus induction hypothesis we know that $[\![C[M]]\!]\,(\vec{x}) \;\approx_{\mu\alpha.\tau}\; [\![C[N]]\!]\,(\vec{y})$. By Proposition 6.9 we know that

$$[\![\lambda x.\mathtt{unfold}\; x]\!] \;\approx_{(\mu\alpha.\tau)\to(\tau[\mu\alpha.\tau/\alpha])}\; [\![\lambda x.\mathtt{unfold}\; x]\!]$$

By applying this latter fact to the induction hypothesis we obtain

$$[\![\mathtt{unfold}\; C[M]]\!]\,(\vec{x}) \;\approx_{\tau[\mu\alpha.\tau/\alpha]}\; [\![\mathtt{unfold}\; C[N]]\!]\,(\vec{y})$$

which is what we wanted.

When the context binds a variable one has to be a bit more careful. For example, for a context of the form $\mathtt{case}\; L \;\mathtt{of}\; x_1.C; x_2.N'$ of type $(\Gamma, \tau) \to (\Delta, \sigma)$ we have by induction that C has type $(\Gamma, \tau) \to ((\Delta, x_1 \colon \tau_1), \sigma)$ and thus by induction hypothesis we know by applying the context parameters that $[\![C[M]]\!]\,(\vec{x}) \;\approx_{\tau_1,\sigma}\; [\![C[N]]\!]\,(\vec{y})$. From this we also know that

$$[\![\lambda x_1.C[M]]\!]\,(\vec{x}) \;\approx_{\tau_1\to\sigma}\; [\![\lambda x_1.C[N]]\!]\,(\vec{y}). \tag{45}$$

By Proposition 6.9 we know that

$$[\![\lambda x.\texttt{case}\ L\ \texttt{of}\ x_1.x(x_1); x_2.N']\!]\,(\vec{x}) \ \approx_{(\tau_1\to\sigma)\to\sigma}\ [\![\lambda x.\texttt{case}\ L\ \texttt{of}\ x_1.x(x_1); x_2.N']\!]\,(\vec{y}).$$

By applying this to (45) we conclude. □

As a direct consequence we get the following lemma.

Lemma 6.18. If $\Gamma \vdash M, N : \tau$ and $[\![M]\!]^{\mathrm{gl}} \ \approx^{\mathrm{gl}}_{\Gamma,\tau}\ [\![N]\!]^{\mathrm{gl}}$ then for all contexts C of type $(\Gamma,\tau) \to (-,1)$, $[\![C[M]]\!]^{\mathrm{gl}} \ \approx^{\mathrm{gl}}_{(-,1)}\ [\![C[N]]\!]^{\mathrm{gl}}$

The next lemma states that if two computations of unit type are related then the first converges iff the second converges. Note that this lemma needs to be stated using the fact that the two computations are *globally related.*

Lemma 6.19. For all x, y of type $[\![1]\!]^{\mathrm{gl}}$, if $x \ \approx^{\mathrm{gl}}_{(-,1)}\ y$ then

$$\Sigma n.x = (\delta^{\mathrm{gl}}_1)^n(\eta(*)) \Leftrightarrow \Sigma m.y = (\delta^{\mathrm{gl}}_1)^m(\eta(*))$$

Proof. We show the left to right implication, so suppose $x = (\delta^{\mathrm{gl}}_1)^n(\eta(*))$. The proof proceeds by induction on n. If $n = 0$ then since by assumption $\forall\kappa.x[\kappa] \ \approx_1\ y[\kappa]$, by definition of $\ \approx_1\ $, for all κ, there exists an m such that $y[\kappa] = \delta^m_1(\eta(*))$. By type isomorphism (18), since m is a natural number, this implies there exists m such that for all κ, $y[\kappa] = \delta^m_1(\eta(*))$ which implies $y = \Lambda\kappa.y[\kappa] = (\delta^{\mathrm{gl}}_1)^m(\eta(*))$.

In the inductive case $n = n'+1$, since by Lemma 6.6 $(\delta^{\mathrm{gl}}_1)^{n'}([\![v]\!]^{\mathrm{gl}}) \ \approx^{\mathrm{gl}}_1\ y$, the induction hypothesis implies $\Sigma m.y = (\delta^{\mathrm{gl}}_1)^m(\eta(*))$. □

Proof of Theorem 6.16 Suppose $[\![M]\!]^{\mathrm{gl}} \ \approx^{\mathrm{gl}}_{\Gamma,\tau}\ [\![N]\!]^{\mathrm{gl}}$ and that C has type $(\Gamma,\tau) \to (-,1)$. We show that if $C[M] \Downarrow \langle\rangle$ also $C[N] \Downarrow \langle\rangle$. So suppose $C[M] \Downarrow \langle\rangle$. By definition this means $\Sigma n.C[M] \Downarrow^n \langle\rangle$. By Corollary 6.1 we get $\Sigma n.\forall\kappa.\,[\![C[M]]\!] = (\delta_1)^n(\eta(*))$ which is equivalent to $\Sigma n.[\![C[M]]\!]^{\mathrm{gl}} = (\delta^{\mathrm{gl}}_1)^n(\eta(*))$. From the assumption and Lemma 6.18 we know that $[\![C[M]]\!]^{\mathrm{gl}} \ \approx^{\mathrm{gl}}_1\ [\![C[N]]\!]^{\mathrm{gl}}$, so by Lemma 6.19 there exists an m such that $[\![C[N]]\!]^{\mathrm{gl}} = (\delta^{\mathrm{gl}}_1)^m(\eta(*))$. By applying the Corollary 6.1 once again we get $C[N] \Downarrow \langle\rangle$ as desired. □

7. Executing the denotational semantics

In this final section we sketch an additional benefit of the denotational semantics described in this paper: The denotational semantics can be executed. More precisely, given a closed FPC term of base type and a number n, the denotational semantics can be executed up to n steps. This will terminate if and only if the big-step operational semantics terminates in n steps or less. The time-out n is necessary since FPC programs can diverge and programs in type theory must terminate. We emphasize that at the moment there is no full implementation of GDTT and so the practical implications of this section are speculative.

We illustrate the execution of the denotational semantics in the case of programs computing booleans, i.e., closed term of type $1 + 1$. The global interpretation of such a term has type $[\![1+1]\!]^{\mathrm{gl}} = \forall\kappa.L(L1+L1)$. We first define a term

$$\mathsf{runstep} : (\forall\kappa.L(L1+L1)) \to (1+1) + (\forall\kappa.L(L1+L1))$$

running the denotation of the term for one step. We define $\mathsf{runstep}\ x$ by cases of $x[\kappa_0] : L(L1+L1)[\kappa_0/\kappa]$ where κ_0 is the clock constant. If $x[\kappa_0] = \eta(\mathsf{inl}(y))$ for some y, then $\mathsf{runstep}\ x = \mathsf{inl}(\mathsf{inl}(\star))$, and likewise if $x[\kappa_0] = \eta(\mathsf{inr}(y))$ for some y, then $\mathsf{runstep}\ x = \mathsf{inl}(\mathsf{inr}(\star))$. In case $x[\kappa_0]$ is of the form $\theta(y)$, then, as we saw in the construction of the isomorphism (17) in Section 2.2, there is a term z_κ such that $x[\kappa] = \theta(z_\kappa)$. (Precisely, $z_\kappa = \pi_2(x[\kappa])$ using the encoding of binary sums as dependent sums over $1+1$.) In that case we define $\mathsf{runstep}\, x = \mathsf{inr}(\mathsf{prev}\,\kappa.z_\kappa)$.

Using $\mathsf{runstep}$ we can define a function

$$\mathsf{exec} : \mathbb{N} \to (\forall\kappa.L(L1+L1)) \to (1+1) + (\forall\kappa.L(L1+L1))$$

such that $\mathsf{exec}\ n$ iterates $\mathsf{runstep}$ until it gets a result, or for at most $n+1$ times. Precisely, we define $\mathsf{exec}\ 0\, x = \mathsf{runstep}\ x$ and $\mathsf{exec}\,(n+1)\, x = \mathsf{runstep}\ x$ if $\mathsf{runstep}\ x$ is in the left component and $\mathsf{exec}\,(n+1)\, x = \mathsf{exec}\ n\, y$ if $\mathsf{runstep}\ x = \mathsf{inr}(y)$.

We now show that executing the denotational semantics using $\mathsf{exec}\ n$ corresponds to executing the operational semantics for up to n steps.

Proposition 7.1. Let M be a closed term of FPC of type $1+1$, and let n be a natural number. Then $\mathsf{exec}\ n\,[\![M]\!]^{\mathrm{gl}} = \mathsf{inl}(\mathsf{inl}(\star))$ iff there exists an N such that $M \Downarrow^k \texttt{inl}\ (N)$ for some $k \leq n$.

To prove Proposition 7.1 we need following two lemmas.

Lemma 7.2. If $\mathsf{exec}\, n\, x = \mathsf{inl}(\mathsf{inl}(\star))$ then there exists a $k \leq n$ and a y such that $x = (\delta^{\mathrm{gl}}_{1+1})^k(\Lambda\kappa.\eta(\mathsf{inl}(y[\kappa])))$.

Proof. The proof is by induction on n and case analysis of $x[\kappa_0]$. If $x[\kappa_0] = \eta(\mathsf{inl}(y))$ for some y, then, as above, also $x[\kappa] = \eta(\mathsf{inl}(z_\kappa))$ for some z_κ and so $x = \Lambda\kappa.\eta(\mathsf{inl}(z_\kappa))$ proving the lemma.

If $x[\kappa_0] = \eta(\mathsf{inr}(y))$, then also $\mathsf{exec}\, n\, x = \mathsf{inl}(\mathsf{inr}(\star))$. Comparing this with the assumption we get $\mathsf{inl}(\mathsf{inr}(\star)) = \mathsf{inl}(\mathsf{inl}(\star))$. Recall [Uni13, Section 2.12] that $\mathsf{inl}(\mathsf{inr}(\star)) = \mathsf{inl}(\mathsf{inl}(\star))$ is equivalent to $\mathsf{inr}(\star) = \mathsf{inl}(\star)$ which is equivalent to the empty type, so from this we conclude 0 and thus anything is provable.

Suppose finally that $x[\kappa_0] = \theta(y)$. Then $x[\kappa] = \theta(z_\kappa)$, and $\mathsf{runstep}\, x = \mathsf{inr}(\mathsf{prev}\,\kappa.z_\kappa)$. In this case n must be greater than 0, i.e., $n = m+1$, and $\mathsf{exec}\,(m+1)\, x = \mathsf{exec}\ m\,(\mathsf{prev}\,\kappa.z_\kappa)$. In this case, by induction hypothesis, $\mathsf{prev}\,\kappa.z_\kappa = (\delta^{\mathrm{gl}}_{1+1})^k(\Lambda\kappa.\eta(\mathsf{inl}(y[\kappa])))$ for some y and $k \leq n$. So then,

$$\begin{aligned}
x &= \Lambda\kappa.(x[\kappa]) \\
&= \Lambda\kappa.(\theta(z_\kappa)) \\
&= \Lambda\kappa.(\theta_{1+1}\ \mathsf{next}^\kappa((\mathsf{prev}\,\kappa.z_\kappa)[\kappa]) \\
&= \Lambda\kappa.(\delta_{1+1}((\delta^{\mathrm{gl}}_{1+1})^k(\Lambda\kappa.\eta(\mathsf{inl}(y[\kappa])))[\kappa]) \\
&= (\delta^{\mathrm{gl}}_{1+1})^{k+1}(\Lambda\kappa.\eta(\mathsf{inl}(y[\kappa]))
\end{aligned}$$

□

Lemma 7.3. Let M be a closed term of FPC of type 1+1. If $[\![M]\!]^{\mathrm{gl}} = (\delta^{\mathrm{gl}}_{1+1})^k(\Lambda\kappa.\eta(\mathsf{inl}(y[\kappa])))$ then there exists an N such that $M \Downarrow^k \texttt{inl}\ (N)$.

Proof. We prove by induction on k that if

$$\forall\kappa.\delta^k_{1+1}(\eta(\mathsf{inl}(y[\kappa])))\ \mathcal{R}_{\tau_1+\tau_2}\ M$$

then there exists an N such that $M \Downarrow^k \texttt{inl}\ (N)$. The lemma then follows from the Fundamental Lemma (Lemma 5.4). In the case of $k = 0$, by definition the assumption implies

$$\forall\kappa.\Sigma N.M \Downarrow^0 \texttt{inl}\ (N)$$

which by an application to the clock constant κ_0 implies

$$\Sigma N.M \Downarrow^0 \texttt{inl}\ (N)$$

as desired. If $k = l+1$, the assumption $\forall\kappa.\theta(\mathsf{next}^\kappa(\delta^l_{1+1}(\eta(\mathsf{inl}(y[\kappa])))))\ \mathcal{R}_{\tau_1+\tau_2}\ M$ reduces to

$$\forall\kappa.(\Sigma M', M''\colon \texttt{Term}_{\texttt{FPC}}.M \to^0_* M' \to^1 M'' \text{ and } \mathsf{next}^\kappa(\delta^l_{1+1}(\eta(\mathsf{inl}(y[\kappa])))) \triangleright\mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(M''))$$

This implies

$$\Sigma M', M''\colon \texttt{Term}_{\texttt{FPC}}.M \to^0_* M' \to^1 M'' \text{ and } \forall\kappa.\triangleright_\kappa(\delta^l_{1+1}(\eta(\mathsf{inl}(y[\kappa])))\ \mathcal{R}_{\tau_1+\tau_2}\ M'')$$

which, using force implies

$$\Sigma M', M''\colon \texttt{Term}_{\texttt{FPC}}.M \to^0_* M' \to^1 M'' \text{ and } \forall\kappa.\delta^l_{1+1}(\eta(\mathsf{inl}(y[\kappa])))\ \mathcal{R}_{\tau_1+\tau_2}\ M''$$

Now the induction hypothesis applies to give an N such that $M'' \Downarrow^l \texttt{inl}\ (N)$, which by Lemma 3.2 implies $M'' \to^l_* v$ and thus $M \to^k_* v$ which implies $M \Downarrow^k \texttt{inl}\ (N)$ again by Lemma 3.2. □

Proof of Proposition 7.1 The left to right implication follows from Lemmas 7.2 and 7.3. If $M \Downarrow^k \texttt{inl}\ (N)$ for some $k \leq n$, then $[\![M]\!]^{\mathrm{gl}} = (\delta^{\mathrm{gl}}_{1+1})^k(\Lambda\kappa.\eta(\mathsf{inl}([\![N]\!])))$. We prove that this implies that $\mathsf{exec}\ n\ [\![M]\!]^{\mathrm{gl}} = \mathsf{inl}(\mathsf{inl}(\star))$ by induction on k. The case of $k = 0$ follows directly by definition of exec. If $k = l + 1$ also $n = m + 1$ for some m. Observe now that for any $x : [\![1 + 1]\!]^{\mathrm{gl}}$

$$\begin{aligned}\mathsf{runstep}\ \delta^{\mathrm{gl}}_{1+1}(x) &= \mathsf{runstep}\ \Lambda\kappa.(\theta(\mathsf{next}^\kappa(x[\kappa])))\\ &= \mathsf{inr}(\mathsf{prev}\,\kappa.(\mathsf{next}^\kappa(x[\kappa])))\\ &= \mathsf{inr}(\Lambda\kappa.x[\kappa])\\ &= \mathsf{inr}(x)\end{aligned}$$

and so in particular

$$\mathsf{runstep}\,[\![M]\!]^{\mathrm{gl}} = \mathsf{inr}((\delta^{\mathrm{gl}}_{1+1})^l(\Lambda\kappa.\eta(\mathsf{inl}([\![N]\!]))))$$

so that

$$\mathsf{exec}\,(n+1)\,[\![M]\!]^{\mathrm{gl}} = \mathsf{exec}\ n\,(\delta^{\mathrm{gl}}_{1+1})^l(\Lambda\kappa.\eta(\mathsf{inl}([\![N]\!])))$$

which equals $\mathsf{inl}(\mathsf{inl}(\star))$ by the induction hypothesis. □

8. Conclusions and Future Work

We have shown that programming languages with recursive types can be given sound and computationally adequate denotational semantics in guarded dependent type theory. The semantics is intensional, in the sense that it can distinguish between computations computing the same result in different number of steps, but we have shown how to capture extensional equivalence in the model by constructing a logical relation on the interpretation of types.

This work can be seen as a first step towards a formalisation of domain theory in type theory. Other, more direct formalisations have been carried out in Coq, e.g. [BKV09; Ben+10; Doc14] but we believe that the synthetic viewpoint offers a more abstract and simpler presentation of the theory. Moreover, we hope that the success of guarded recursion for operational reasoning, mentioned in the introduction, can be carried over to denotational models of more advanced programming language features as, for example, to general references, for which, at the present day, no denotational model exists.

Future work also includes implementation of GDTT in a proof assistant, allowing for the theory of this paper to be machine verified. Currently, initial experiments are being carried out in this direction [Bir+16].

Finally, we have not yet investigate the possible applications of the weak bisimulation introduced in Section 6.

References

[ADK17] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. “Partiality, Revisited - The Partiality Monad as a Quotient Inductive-Inductive Type”. In: *FoSSaCS*. 2017.

[AM01] A. W. Appel and D. McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *ACM Trans. Program. Lang. Syst.* (2001).

[AM13] Robert Atkey and Conor McBride. “Productive Coprogramming with Guarded Recursion”. In: *ICFP*. 2013, pp. 197–208.

[BBM14] Aleš Bizjak, Lars Birkedal, and Marino Miculan. “A Model of Countable Nondeterminism in Guarded Type Theory”. In: *RTA-TLCA*. 2014, pp. 108–123.

[Ben+10] Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. “Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages”. 2010.

[BGM17] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. “The clocks are ticking: No more delays!” In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. 2017, pp. 1–12.

[Bir+12] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. “First steps in synthetic guarded domain theory: step-indexing in the topos of trees”. In: *LICS*. 2012.

[Bir+16] L. Birkedal, A. Bizjak, R. Clouston, H.B. Grathwohl, B. Spitters, and A. Vezzosi. “Guarded Cubical Type Theory: Path Equality for Guarded Recursion”. In: *CSL 2016*. 2016.

[Biz+16] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. “Guarded Dependent Type Theory with Coinductive Types”. In: *FoSSaCS*. 2016.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. “Some Domain Theory and Denotational Semantics in Coq”. In: *TPHOLs*. 2009.

[BM13] Lars Birkedal and Rasmus Ejlers Møgelberg. “Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes”. In: *LICS*. 2013, pp. 213–222.

[BM15] Aleš Bizjak and Rasmus Ejlers Møgelberg. “A model of guarded recursion with clock synchronisation”. In: *MFPS*. 2015.

[Cap05] Venanzio Capretta. “General recursion via coinductive types”. In: *Logical Methods in Computer Science* (2005).

[Coh+16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. “Cubical Type Theory: a constructive interpretation of the univalence axiom”. In: *CoRR* abs/1611.02108 (2016).

[CUV15] James Chapman, Tarmo Uustalu, and Niccolò Veltri. “Quotienting the Delay Monad by Weak Bisimilarity”. In: *ICTAC*. 2015.

[Dan12] Nils Anders Danielsson. “Operational semantics using the partiality monad”. In: *ICFP*. 2012, pp. 127–138.

[Doc14] Robert Dockins. “Formalized, Effective Domain Theory in Coq”. In: *ITP*. 2014.

[Esc99] M.H. Escardó. “A metric model of PCF”. Laboratory for Foundations of Computer Science, University of Edinburgh. 1999.

[Hyl91] J. Martin E. Hyland. “First steps in synthetic domain theory”. In: *Category Theory*. 1991, pp. 131–156.

[KL12] Chris Kapulkin and Peter LeFanu Lumsdaine. “The Simplicial Model of Univalent Foundations (after Voevodsky)”. In: *CoRR* abs/1211.2851 (2012). URL: `https://arxiv.org/abs/1211.2851`.

[MP08] C. McBride and R. Paterson. “Applicative Programming with Effects”. In: *Journal of Functional Programming* 18.1 (2008).

[MP16] Rasmus Ejlers Møgelberg and Marco Paviotti. “Denotational Semantics of recursive types in Synthetic Guarded Domain Theory”. In: *LICS*. 2016.

[Møg14] Rasmus Ejlers Møgelberg. “A type theory for productive coprogramming via guarded recursion”. In: *CSL-LICS*. 2014.

[Nak00] Hiroshi Nakano. “A modality for recursion”. In: *LICS*. 2000, pp. 255–266.

[Pit96] Andrew M. Pitts. “Relational Properties of Domains”. In: *Inf. Comput.* 127.2 (1996), pp. 66–90.

[PMB15] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. “A Model of PCF in Guarded Type Theory”. In: *Electr. Notes Theor. Comput. Sci.* (2015).

[Reu96] Bernhard Reus. “Synthetic Domain Theory in Type Theory: Another Logic of Computable Functions”. In: *TPHOLs*. 1996.

[Ros86] G. Rosolini. “Continuity and effectiveness in topoi”. PhD thesis. University of Oxford, 1986.

[SB14] Kasper Svendsen and Lars Birkedal. “Impredicative Concurrent Abstract Predicates”. In: *ESOP*. 2014.

[Sim02] Alex K. Simpson. “Computational Adequacy for Recursive Types in Models of Intuitionistic Set Theory”. In: *LICS*. 2002, pp. 287–298.

[Str06] Thomas Streicher. *Domain-theoretic foundations of functional programming.* World Scientific, 2006, pp. I–X, 1–120. ISBN: 978-981-270-142-8.

[Vel17] Niccolò Veltri. “A Type-Theoretical Study of Nontermination”. PhD thesis. 2017.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-23169-7.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: `http://homotopytypetheory.org/book`, 2013.