Denotational semantics of recursive types in synthetic guarded domain theory

Rasmus Ejlers Møgelberg Marco Paviotti
IT University of Copenhagen, Denmark
mogel@itu.dk, mpav@itu.dk

Abstract

Guarded recursion is a form of recursion where recursive calls are guarded by delay modalities. Previous work has shown how guarded recursion is useful for reasoning operationally about programming languages with advanced features including general references, recursive types, countable non-determinism and concurrency.

Guarded recursion also offers a way of adding recursion to type theory while maintaining logical consistency. In previous work we initiated a programme of denotational semantics in type theory using guarded recursion, by constructing a computationally adequate model of the language PCF (simply typed lambda calculus with fixed points). This model was intensional in that it could distinguish between computations computing the same result using a different number of fixed point unfoldings.

In this work we show how also programming languages with recursive types can be given denotational semantics in type theory with guarded recursion. More precisely, we give a computationally adequate denotational semantics to the language FPC (simply typed lambda calculus extended with recursive types), modelling recursive types using guarded recursive types. The model is intensional in the same way as was the case in previous work, but we show how to recover extensionality using a logical relation.

All constructions and reasoning in this paper, including proofs of theorems such as soundness and adequacy, are by (informal) reasoning in type theory, often using guarded recursion.

Keywords Denotational Semantics, Recursive Types, Type Theory, Guarded Recursion, Synthetic Domain Theory

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Denotational semantics; F.4.1 [*Mathematical Logic and Formal Languages*]: Lambda calculus and related systems

1. Introduction

Recent years have seen great advances in formalisation of mathematics in type theory, in particular with the development of homotopy type theory (Univalent Foundations Program 2013). 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 illustrated by the development of synthetic homotopy theory.

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.

Guarded recursion (Nakano 2000) can be seen as a synthetic form of domain theory, or, perhaps more accurately, a synthetic form of step-indexing (Birkedal et al. 2012; Appel et al. 2007). 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 (Birkedal et al. 2012; Bizjak et al. 2014; Svendsen and Birkedal 2014). The hope is that synthetic guarded domain theory can also provide denotational models of these features.

1.1 Synthetic guarded domain theory

The synthetic approach to domain theory is to assume that types are domains, rather than constructing a notion of domain as a type equipped with a certain structure. To model recursion a fixed point combinator is needed, but adding unrestricted fixed points makes the type theory inconsistent when read as a logical system. The approach of guarded recursion is to introduce 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, 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. Guarded recursion can be proved consistent with type theory using the topos of trees model and related variants (Birkedal et al. 2012; Bizjak and Møgelberg 2015; Bizjak et al. 2016). In this paper we will be working in guarded dependent type theory (gDTT) (Bizjak et al. 2016), an extensional type theory with guarded recursion.

In previous work (Paviotti et al. 2015), we initiated a study of denotational semantics inside guarded dependent type theory, constructing a model of PCF (simply typed lambda calculus with fixed points). By carefully aligning the fixpoint unfoldings of PCF

LICS '16, July 05 - 08, 2016, New York, NY, USA
 ISBN 978-1-4503-4391-6/16/07…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2933575.2934516

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 on our previous work in two ways. First we extend the denotational semantics and adequacy proof to languages with recursive types. More precisely, we consider the language FPC (simply typed lambda calculus extended with general recursive types), modelling recursive types using guarded recursive types. The proof of computational adequacy 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 (Pitts 1996), but here it is simply a guarded recursive type.

The second contribution is a relation capturing extensionally equal elements in the model. Like the model for PCF in our previous work, the model for FPC constructed here distinguishes between programs computing the same value using a different number of fixed point unfoldings. We construct a relation on the interpretation of types, that relates elements that only differ by a finite number of computation steps. The relation is proved sound, meaning that, if the denotations of two terms are related, then the terms are contextually equivalent.

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 (Escardó 1999). Since this category can be seen as a subcategory of the topos of trees (Birkedal et al. 2012), our previous work on PCF is a synthetic version of Escardó's model. Escardó's model also distinguishes between computations using 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 and axiomatisations dating back to (Rosolini 1986; Hyland 1991; Reus 1996). Indeed, the internal languages of these models can be used to construct models of FPC and prove computational adequacy (Simpson 2002). Unlike guarded synthetic domain theory, these models do not distinguish between computations using different numbers of steps. On the other hand, with the success of guarded recursion for syntactic models, we believe that the guarded approach could model languages with more advanced features.

The lifting monad used in this paper is a guarded recursive variant of the partiality monad considered by among others (Danielsson 2012; Capretta 2005; Benton et al. 2009, 2010). Danielsson also defines a weak bisimulation on this monad, similar to the one defined in Definition 10. As reported by Danielsson, working with the partiality monad requires convincing Agda of productivity of coinductive definitions using workarounds. Here, productivity is ensured by the type system for guarded recursion.

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 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. We conclude and discuss future work in Section 7.

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 (Bizjak et al. 2016) the ones used here can be safely added, and so the arguments of this paper can be formalised in gDTT. We start by recalling some core features of this theory. In fact, for the first part of the development, we will need just the features of (Birkedal and Møgelberg 2013), which corresponds to the fragment of gDTT with a single clock and no delayed substitutions. Quantification over clocks and delayed substitutions will be introduced later, when needed.

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 type constructor $\triangleright$ introduced in Section 1.1 is an applicative functor in the sense of (McBride and Paterson 2008), which means that there is a "later application" $\circledast \colon \triangleright(A \to B) \to \triangleright A \to \triangleright B$ written infix, satisfying $\mathsf{next}(f) \circledast \mathsf{next}(t) \equiv \mathsf{next}(f(t))$ among other axioms (see also (Birkedal and Møgelberg 2013)). In particular, $\triangleright$ extends to a functor mapping $f \colon A \to B$ to $\lambda x \colon \triangleright A.\, \mathsf{next}(f) \circledast x$.

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 (Bizjak et al. 2016) 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 $\mathsf{El}(\widehat{\triangleright}(\mathsf{next}(A))) \equiv \triangleright\,\mathsf{El}(A)$, 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

$$\mathsf{fix}(\lambda X : \triangleright\mathcal{U}.\, \widehat{+}(\widehat{A}, \widehat{\triangleright} X))\,.$$

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 generally leave El implicit.

2.1 The topos of trees model

The topos of trees model of guarded recursion (Birkedal et al. 2012) provides useful intuitions, and so we briefly recall it.

In the model, a closed type is modelled as a family of sets $X(n)$ indexed by natural numbers together with restriction maps $r_n^X \colon X(n+1) \to X(n)$. The $\triangleright$ type operator is modelled as $\triangleright X(1) = 1$, $\triangleright X(n+1) = X(n)$. 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 $\triangleright^{42} 0$ are non-standard truth values. Intuitively, this is the truthvalue of propositions that appear true for 42 computation steps, but then are falsified after 43.

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

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

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

$$\frac{\vdash \Theta}{\Theta \vdash \Theta_i} 1 \leq i \leq |\,\Theta\,| \quad \frac{\vdash \Theta}{\Theta \vdash 1}$$

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

Figure 1: Rules for welformed FPC types

$$\Gamma \in \text{Expression Contexts} \stackrel{\text{def}}{=\!=} \langle\rangle \mid \langle \Gamma, x : \tau \rangle$$

$$\frac{\vdash \Theta}{\Theta \vdash \langle\rangle} \quad \frac{\Theta \vdash \Gamma \quad \Theta \vdash \tau}{\Theta \vdash \Gamma, x : \tau} x \notin \Gamma$$

Figure 2: Rules for welformed FPC contexts

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^A_n identities, then we can think of $LA(n)$ as $\{0, \ldots, n-1\} \times X + \{\bot\}$. 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.

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.

Unlike the operational semantics to be defined below, the typing judgements of FPC are defined in an entirely standard way. The grammar for terms of FPC

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

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 Figures 1, 2 and 3 should read as defining inductive types in type theory, allowing us to do proofs by induction over e.g. typing judgements.

We denote by $\mathtt{Type}_{\text{FPC}}$, $\mathtt{Term}_{\text{FPC}}$ and $\mathtt{Value}_{\text{FPC}}$ the types of *closed* FPC types and terms, and values of FPC. 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 Small-step semantics

Figure 4 defines the reductions of the small-step call-by-name operational semantics. Since the denotational semantics of FPC are 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.

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

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : \sigma \to \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

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

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

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ M : \tau_1} \qquad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2}$$

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

Figure 3: Typing rules for FPC terms

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

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

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

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

$$\frac{M_1 \to^k M_2}{E[M_1] \to^k E[M_2]}$$

$$\begin{aligned} E ::= {} & [\cdot] \mid EM \mid \mathtt{case}\ E\ \mathtt{of}\ x_1.M; x_2.N \\ & \mid \mathtt{fst}\ E \mid \mathtt{snd}\ E \mid \mathtt{unfold}\ E \end{aligned}$$

Figure 4: Reductions of the small-step call-by-name operational semantics. In the last rule, k is either 0 or 1.

We next define the transitive closure of the small-step operational semantics. To ease the comparison with the big-step operational semantics, we define a generalisation of the transitive closure as a relation of the form $M \Rightarrow^k Q$ to be read as 'M reduces in k steps to a term N satisfying Q'. Here $Q \colon \mathtt{Term}_{\text{FPC}} \to \mathcal{U}$ is a (proof relevant) predicate on closed terms. The more standard big-step evaluation of terms to values can be defined as

$$M \Rightarrow^k v \stackrel{\text{def}}{=\!=} M \Rightarrow^k (\lambda N.N = v)$$

Definition 1. The transitive closure of the small-step relation is defined by induction on k as follows.

$$M \Rightarrow^0 Q \stackrel{\text{def}}{=\!=} \Sigma N \colon \mathtt{Term}_{\text{FPC}}.M \to^0_* N \text{ and } Q(N)$$

$$M \Rightarrow^{k+1} Q \stackrel{\text{def}}{=\!=} \Sigma M' M'' \colon \mathtt{Term}_{\text{FPC}}.M \to^0_* M' \text{ and } M' \to^1 M'' \text{ and } \triangleright(M'' \Rightarrow^k Q)$$

Here $\to^0_*$ is the reflexive-transitive closure of $\to^0$.

The use of $\triangleright$ in the second clause of Definition 1 synchronizes the steps of FPC with those of the metalogic. This allows guarded recursion to be used as a proof principle for operational semantics, and is also needed to get the precise relationship to the denotational semantics.

3.2 Big-step semantics

We now define a big-step call-by-name operational semantics for FPC. Big-step semantics are usually defined as relations between closed terms and values. Here, we generalize to a (proof relevant) relation of the form

$$M \Downarrow^k Q \tag{2}$$

where M is a term, k a natural number, and $Q \colon \mathtt{Value}_{\mathtt{FPC}} \times \mathbb{N} \to \mathcal{U}$ a proof relevant relation on values and natural numbers. The statement (2) should be read as 'M evaluates in $l \leq k$ steps to a value v such that $Q(v, k-l)$'. As for the small-step semantics, a step is an `unfold-fold` reduction. If $Q \colon \mathtt{Value}_{\mathtt{FPC}} \to \mathcal{U}$ we overload notation and write

$$M \Downarrow^k Q \stackrel{\text{def}}{=\!=} M \Downarrow^k (\lambda \langle v, l\rangle .l = 0 \text{ and } Q(v)) \tag{3}$$

to be read as 'M evaluates in exactly k steps to a value satisfying Q'. We can define more standard big-step evaluation predicates as follows

$$M \Downarrow^k v \stackrel{\text{def}}{=\!=} M \Downarrow^k (\lambda w.w = v)$$
$$M \Downarrow v \stackrel{\text{def}}{=\!=} \Sigma k.M \Downarrow^k v$$

The big-step relation is defined as an inductive type in Figure 5. Following the reading of the big-step predicate given above, $MN \Downarrow^k Q$ holds if M reduces in l steps (for some $l \leq k$) to a term of the form $\lambda x.L$, such that $L[N/x] \Downarrow^{k-l} Q$. The cases of projections and `case` are similar. In the case of `unfold`, once M has been reduced to $\mathtt{fold}\ N$, one time step is consumed to reduce $\mathtt{unfold}\ (\mathtt{fold}\ N)$ to N before continuing reduction. Just as was the case for the small-step semantics, the use of $\triangleright$ in this rule synchronizes the steps of FPC with those of the metalogic.

The use of predicates on the right hand sides of the big-step semantics is crucial for the equivalence of the small-step and big-step semantics. More precisely, it allows us to postpone existence of terms to the time they are needed. For example, if $MN \Downarrow^k v$, and M uses one step to reduce to a value, the term $\lambda x.L$ that M should reduce to is only required to exist later, rather than now, as a more direct big-step semantics would require. This makes a difference, since Σ and $\triangleright$ do not commute.

3.3 Examples

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

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

Using this definition we can define the term ifz of PCF. If L is a closed term of type **nat** and M,N are closed terms of type σ then 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 $\mathsf{ifz}\ \mathsf{zero}\ M\ N \Downarrow^k Q$ iff $\triangleright(M \Downarrow^{k-1} Q)$ and that $\mathsf{ifz}\ (\mathsf{succ}\ L)\ M\ N \Downarrow^k Q$ iff $\triangleright(N \Downarrow^{k-1} Q)$ for any L closed term of type **nat**. For example, $\mathsf{ifz}\ 1\ 0\ 1 \Downarrow^2 42$ is $\triangleright 0$.

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

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

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

To understand the relationship of the operational semantics defined in this paper to more traditional semantics defined without delays in the form of $\triangleright$, write $M \to_*^k N$ to mean that M reduces to N in the transitive closure of the reduction semantics, where k is the sum of the steps in the reduction. If $M \to_*^k v$ then

- $M \Downarrow^k v$ is true
- $M \Downarrow^n v$ is logically equivalent to $\triangleright^{\min(n,k)} 0$ 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 $M \Downarrow^n v$ is equivalent to $\triangleright^n 0$.

3.4 Equivalence of small-step and big-step semantics

We now state the equivalence of the two operational semantics given above. Since the big-step operational semantics as defined in (3) uses predicates on values, and the transitive closure of the small-step semantics (Definition 1) uses predicates on terms, we first introduce the notation

$$Q_T(N) \stackrel{\text{def}}{=\!=} \Sigma v.N = v \text{ and } Q(v)$$

such that $Q_T \colon \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}$ whenever $Q \colon \mathtt{Value}_{\mathtt{FPC}} \to \mathcal{U}$.

Proposition 2. *If $M : \mathtt{Term}_{\mathtt{FPC}}$ and $Q : \mathtt{Value}_{\mathtt{FPC}} \to \mathcal{U}$, then $M \Downarrow^k Q$ iff $M \Rightarrow^k Q_T$.*

This has an immediate corollary.

Corollary 1. $M \Downarrow^k v \Leftrightarrow M \Rightarrow^k v$

The proof Proposition 2, uses a strengthened induction hypothesis obtained by overloading the small step predicate once again. If $Q \colon \mathtt{Term}_{\mathtt{FPC}} \times \mathbb{N} \to \mathcal{U}$, define

$$M \Rightarrow^k Q \stackrel{\text{def}}{=\!=} \Sigma N \colon \mathtt{Term}_{\mathtt{FPC}}.M \to_*^0 N \text{ and } Q(N,k)$$
$$\text{or } (\Sigma k', M', M''.k = k' + 1 \text{ and } M \to_*^0 M'$$
$$\text{and } M' \to^1 M'' \text{ and } \triangleright(M'' \Rightarrow^{k'} Q))$$

One easily proves that if $Q \colon \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}$ then

$$(M \Rightarrow^k \lambda \langle N, k\rangle .(k = 0 \text{ and } Q(N))) \text{ iff } M \Rightarrow^k Q$$

We omit the inductive proof for reasons of space.

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 (Paviotti et al. 2015). This is defined as the *unique* solution to the guarded recursive type equation[1]

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

which exists because the recursive variable is guarded by a $\triangleright$. This isomorphism induces a map $\theta_{LA} : \triangleright LA \to LA$ and a map

[1] Since guarded recursive types are encoded using universes, L is strictly an operation on $\mathcal{U}$. As stated in Section 2 we will only apply L to types that have codes in $\mathcal{U}$.

$$v \Downarrow^k Q \stackrel{\text{def}}{=\!=} Q(v,k)$$

$$\texttt{case}\ L\ \texttt{of}\ x_1.M; x_2.N \Downarrow^k Q \stackrel{\text{def}}{=\!=} L \Downarrow^k Q' \quad \text{where} \begin{array}{l} Q'(\texttt{inl}\ L, l) \stackrel{\text{def}}{=\!=} M[L/x_1] \Downarrow^l Q \\ Q'(\texttt{inr}\ L, l) \stackrel{\text{def}}{=\!=} N[L/x_2] \Downarrow^l Q \end{array}$$

$$\texttt{fst}\ L \Downarrow^k Q \stackrel{\text{def}}{=\!=} L \Downarrow^k Q' \quad \text{where } Q'(\langle M, N\rangle, m) \stackrel{\text{def}}{=\!=} M \Downarrow^m Q$$

$$\texttt{snd}\ L \Downarrow^k Q \stackrel{\text{def}}{=\!=} L \Downarrow^k Q' \quad \text{where } Q'(\langle M, N\rangle, m) \stackrel{\text{def}}{=\!=} N \Downarrow^m Q$$

$$MN \Downarrow^k Q \stackrel{\text{def}}{=\!=} M \Downarrow^k Q' \quad \text{where } Q'(\lambda x.L, m) \stackrel{\text{def}}{=\!=} L[N/x] \Downarrow^m Q$$

$$\texttt{unfold}\ M \Downarrow^k Q \stackrel{\text{def}}{=\!=} M \Downarrow^k Q' \quad \text{where } Q'(\texttt{fold}\ N, m+1) \stackrel{\text{def}}{=\!=} \triangleright(N \Downarrow^m Q)$$

Figure 5: The big-step operational semantics. In the definitions of Q' only non-empty cases are given, e.g., in the case of $\texttt{unfold}\ M$, $Q'(P,n)$ is defined to be the empty type unless P is of the form $\texttt{fold}\ N$ and n is a successor.

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

Figure 6: Interpretation of FPC types

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

Figure 7: Definition of $\theta_\sigma : \triangleright[\![\sigma]\!] \to [\![\sigma]\!]$

$\eta\colon A \to LA$. An element of LA is either of the form $\eta(a)$ or $\theta(r)$. We think of these cases as values "now" or 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

$$\begin{aligned}
\hat{f}(\eta(a)) &\stackrel{\text{def}}{=\!=} f(a) \\
\hat{f}(\theta(r)) &\stackrel{\text{def}}{=\!=} \theta_{LB}(\mathsf{next}(\hat{f}) \circledast r)
\end{aligned}$$

Formally $\hat{f}$ is defined as a fixed point of a term of type $\triangleright(LA \to B) \to LA \to B$.

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 $\bot \stackrel{\text{def}}{=\!=} \mathsf{fix}(\theta)$ and a "delay" map δ_{LA} of type $LA \to LA$ for any A as $\delta_{LA} \stackrel{\text{def}}{=\!=} \theta_{LA} \circ \mathsf{next}$. The latter can be thought of as adding a step to a computation.

4.1 Interpretation of types

The typing 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 6.

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

$$\lambda X.\lambda\rho.\widehat{\triangleright}(\mathsf{next}([\![\tau]\!]) \circledast \mathsf{next}(\rho) \circledast (X \circledast \mathsf{next}(\rho))$$

ensuring (using $\mathsf{El}(-)$ explicitly)

$$\begin{aligned}
&\mathsf{El}([\![\Theta \vdash \mu\alpha.\tau]\!]\,\rho) \\
&\equiv \mathsf{El}(\widehat{\triangleright}(\mathsf{next}([\![\tau]\!]) \circledast \mathsf{next}(\rho) \circledast (\mathsf{next}([\![\Theta \vdash \mu\alpha.\tau]\!]) \circledast \mathsf{next}(\rho)))) \\
&\equiv \mathsf{El}(\widehat{\triangleright}(\mathsf{next}([\![\tau]\!]\,(\rho, ([\![\Theta \vdash \mu\alpha.\tau]\!]\,\rho))))) \\
&\equiv \triangleright\,\mathsf{El}([\![\tau]\!]\,(\rho, ([\![\Theta \vdash \mu\alpha.\tau]\!]\,\rho)))
\end{aligned}$$

The substitution lemma for types can be proved using guarded recursion in the case of recursive types.

Lemma 3 (Substitution Lemma for Types)**.** *Let σ be a well-formed type with variables in Θ and let ρ be of type $\mathcal{U}^{|\Theta|}$, for $\Theta, \beta \vdash \tau$, $[\![\Theta \vdash \tau[\sigma/\beta]]\!]\,(\rho) = [\![\Theta, \beta \vdash \tau]\!]\,(\rho, [\![\Theta \vdash \sigma]\!]\,(\rho))$*

The following lemma follows directly from the substitution lemma.

Lemma 4. *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 $\theta_\tau : \triangleright[\![\tau]\!] \to [\![\tau]\!]$, defined by guarded recursion and structural induction on τ as in Figure 7. The case of products uses the functorial action of $\triangleright$ as described in Section 2. The $\triangleright$-algebra for the unit type and for the sum type exists due to them being interpreted using the lifting monad. The case of recursive types is welltyped by Lemma 4. More formally, θ can be defined using a fix, but since this is most easily done using delayed substitutions, we postpone this to Section 5.2.

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

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

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

Figure 8: Interpretation of FPC terms

4.2 Interpretation of terms

Figure 8 defines the interpretation of judgements $\Gamma \vdash M : \sigma$ as functions from $[\![\Gamma]\!]$ to $[\![\sigma]\!]$ where $[\![x_1 : \sigma_1, \cdots, x_n : \sigma_n]\!] \stackrel{\text{def}}{=\!=} [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$. In the case of `case`, $\widehat{f}$ refers to the extension of functions to homomorphisms defined 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 is equal to $[\![\mu\alpha.\tau]\!]$. In the case of `unfold`, since $[\![M]\!]\,(\gamma)$ has type $[\![\mu\alpha.\tau]\!]$, which by Lemma 4 is equal to $\triangleright [\![\tau[\mu\alpha.\tau/\alpha]]\!]$, the type of $\theta_{\tau[\mu\alpha.\tau/\alpha]}([\![M]\!]\,(\gamma))$ is $[\![\tau[\mu\alpha.\tau/\alpha]]\!]$.

Clearly, $[\![\mathtt{unfold}\ (\mathtt{fold}\ M)]\!]\,(\gamma) = \delta_\sigma([\![M]\!])$ for every M of type $\tau[\mu\alpha.\tau/\alpha]$. This is used to prove the soundness theorem.

Theorem 5 (Soundness)**.** *Let M be a closed term of type τ, if $M \Downarrow^k v$ then $[\![M]\!]\,(*) = \delta^k [\![v]\!]\,(*)$*

5. Computational Adequacy

Computational adequacy is opposite implication of Theorem 5 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 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 (Pitts 1996), 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.

$$\mathsf{next}\,\xi\,[x \leftarrow \mathsf{next}\,\xi.t]\,.B \equiv \mathsf{next}\,\xi.(B[t/x]) \tag{4}$$
$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.x \equiv t \tag{5}$$
$$\mathsf{next}\,\xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}\,\xi.u \tag{6}$$
$$\mathsf{next}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v \tag{7}$$
$$\mathsf{next}\,\xi.\,\mathsf{next}\,\xi'.u \equiv \mathsf{next}\,\xi'.\,\mathsf{next}\,\xi.u \tag{8}$$
$$(\mathsf{next}\,\xi.t =_{\triangleright\xi.A} \mathsf{next}\,\xi.s) \equiv \triangleright\xi.(t =_A s) \tag{9}$$
$$\forall\kappa.(x[\kappa] =_A y[\kappa]) \equiv (x =_{\forall\kappa.A} y) \tag{10}$$
$$\mathsf{El}(\widehat{\triangleright}(\mathsf{next}\,\xi.A)) \equiv \triangleright\xi.\,\mathsf{El}(A) \tag{11}$$

Figure 9: The notation $\xi\,[x \leftarrow t]$ means the extension of the delayed substitution ξ with $[x \leftarrow t]$. Rule (6) requires x not free in u. Rule (8) 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.

5.1 Delayed substitutions

In gDTT, if $\Gamma, x\colon A \vdash B\ \mathsf{type}$ is a well formed type and t has type $\triangleright A$ in context Γ, one can form the type $\triangleright\,[x \leftarrow t]\,.B$. One motivation for this 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 if 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 $\Gamma, x_1\colon A_1 \ldots x_n\colon A_n \vdash$ is a wellformed context, and all A_i are independent, i.e., no x_j appears in an A_i. A delayed substitution $\xi\colon \Gamma \twoheadrightarrow x_1\colon A_1 \ldots x_n\colon A_n$ is a vector of terms $\xi = [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ such that $\Gamma \vdash t_i : A_i$. (Bizjak et al. 2016) 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 $\xi\colon \Gamma \twoheadrightarrow \Gamma'$ is a delayed substitution and $\Gamma, \Gamma' \vdash B\ \mathsf{type}$ 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 9 we recall some rules from (Bizjak et al. 2016) needed below. Of these, (4) and (5) can be considered β and η laws, and (6) is a weakening principle. Rules (4), (6) and (7) also have obvious versions for types, e.g.,

$$\triangleright\xi\,[x \leftarrow \mathsf{next}\,\xi.t]\,.B \equiv \triangleright\xi.(B[t/x]) \tag{12}$$

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

$$g \circledast y \stackrel{\text{def}}{=\!=} \mathsf{next}\,[f \leftarrow g, x \leftarrow y]\,.f(x) \tag{13}$$

Note that with this definition, the rule $\mathsf{next}(f(t)) \equiv \mathsf{next}\,f \circledast \mathsf{next}\,t$ from Section 2 generalises to

$$\mathsf{next}\,\xi.(f\,t) \equiv (\mathsf{next}\,\xi.f) \circledast (\mathsf{next}\,\xi.t) \tag{14}$$

which follows from (4).

Rules (5), (6) and (8) imply the rule

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

which by (9) gives an inhabitant of

$$\triangleright\xi\,[x \leftarrow t]\,.(\mathsf{next}\,x = t) \tag{15}$$

Rule (10) is simply clock extensionality.

$$\eta(*)\ \mathcal{R}_1\ M \stackrel{\text{def}}{=\!=} M \Downarrow^0 \langle\rangle$$

$$\theta_1(x)\ \mathcal{R}_1\ M \stackrel{\text{def}}{=\!=} \Sigma M', M'' \colon \mathtt{Term}_{\mathtt{FPC}}. M \to_*^0 M' \to^1 M'' \text{ and } x \triangleright \mathcal{R}_1\ \mathsf{next}(M'')$$

$$x\ \mathcal{R}_{\tau_1 \times \tau_2}\ M \stackrel{\text{def}}{=\!=} \pi_1(x)\ \mathcal{R}_{\tau_1}\ \mathtt{fst}\ (M) \text{ and } \pi_2(x)\ \mathcal{R}_{\tau_2}\ \mathtt{snd}\ (M)$$

$$\eta(\mathsf{inl}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M \stackrel{\text{def}}{=\!=} \Sigma L. M \Downarrow^0 \mathtt{inl}\ L \text{ s.t. } x\ \mathcal{R}_{\tau_1}\ L$$

$$\eta(\mathsf{inr}(x))\ \mathcal{R}_{\tau_1+\tau_2}\ M \stackrel{\text{def}}{=\!=} \Sigma L. M \Downarrow^0 \mathtt{inr}\ L \text{ s.t. } x\ \mathcal{R}_{\tau_2}\ L$$

$$\theta_{\tau_1+\tau_2}(x)\ \mathcal{R}_{\tau_1+\tau_2}\ L \stackrel{\text{def}}{=\!=} \Sigma M', M'' \colon \mathtt{Term}_{\mathtt{FPC}}. M \to_*^0 M' \to^1 M'' \text{ and } x \triangleright \mathcal{R}_{\tau_1+\tau_2}\ \mathsf{next}(M'')$$

$$f\ \mathcal{R}_{\tau\to\sigma}\ M \stackrel{\text{def}}{=\!=} \Pi x \colon [\![\tau]\!], N \colon \mathtt{Term}_{\mathtt{FPC}}. x\ \mathcal{R}_\tau\ N \to f(x)\ \mathcal{R}_\sigma\ (MN)$$

$$x\ \mathcal{R}_{\mu\alpha.\tau}\ M \stackrel{\text{def}}{=\!=} \Sigma M' M''. \mathtt{unfold}\ M \to_*^0 M' \to^1 M'' \text{ and } x \triangleright \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$$

Figure 10: The logical relation $\mathcal{R}_\tau : [\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}$.

5.2 Well-definedness of θ

As advertised above, we now show how θ of Figure 7 can be formally constructed as a fixed point of a term of type

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

Suppose $F \colon \triangleright(\Pi\sigma \colon \mathtt{Type}_{\mathtt{FPC}}\ .(\triangleright [\![\sigma]\!] \to [\![\sigma]\!]))$, and define $G(F)$ essentially as in Figure 7 but with the clause $G(F)_{\tau[\mu\alpha.\tau/\alpha]}$ for recursive types being defined as

$$\lambda x \colon \triangleright [\![\mu\alpha.\tau]\!]\ .\ \mathsf{next}\left[F' \leftarrow F, x' \leftarrow x\right].(F'_{\tau[\mu\alpha.\tau/\alpha]}(x'))$$

Define θ as the fixed point of G. Then

$$\begin{aligned}\theta_{\mu\alpha.\tau}(x) &\equiv G(\mathsf{next}\,(\theta))_{\mu\alpha.\tau}(x)\\ &\equiv \mathsf{next}\left[F' \leftarrow \mathsf{next}\,(\theta), x' \leftarrow x\right].(F'_{\tau[\mu\alpha.\tau/\alpha]}(x'))\\ &\equiv \mathsf{next}\left[x' \leftarrow x\right].(\theta_{\tau[\mu\alpha.\tau/\alpha]}(x'))\\ &\equiv \mathsf{next}(\theta_{\tau[\mu\alpha.\tau/\alpha]}) \circledast (x)\end{aligned}$$

5.3 A logical relation between syntax and semantics

Figure 10 defines the logical relation between syntax and semantics. It uses the following operation lifting relations from A to B to relations from $\triangleright A$ to $\triangleright B$:

$$t \triangleright \mathcal{R}\ u \stackrel{\text{def}}{=\!=} \triangleright\left[x \leftarrow t, y \leftarrow u\right].(x\ \mathcal{R}\ y) \tag{16}$$

As a consequence of (12) the following statement holds:

$$(\mathsf{next}\,\xi.t) \triangleright \mathcal{R}\ (\mathsf{next}\,\xi.u) \equiv \triangleright\xi.(t\ \mathcal{R}\ u) \tag{17}$$

This lifting operation can also be expressed on codes mapping $\mathcal{R} : A \to B \to \mathcal{U}$ to

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

in fact, this operation can be shown to factor as $F \circ \mathsf{next}$, for some $F \colon \triangleright(A \to B \to \mathcal{U}) \to \triangleright A \to \triangleright B \to \mathcal{U}$. Using this, one can formally define the logical relation as a fixed point of a function of type

$$\begin{aligned}&\triangleright(\Pi(\tau : \mathtt{Type}_{\mathtt{FPC}}).\ [\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U}) \to\\ &\qquad (\Pi(\tau : \mathtt{Type}_{\mathtt{FPC}}).\ [\![\tau]\!] \times \mathtt{Term}_{\mathtt{FPC}} \to \mathcal{U})\end{aligned}$$

similarly to the formal definition of θ explained in Section 5.2.

5.4 Proof of computational adequacy

Computational adequacy follows from the fundamental lemma below, stating that all terms respect the logical relation. The proof of the fundamental lemma rests on the following two key lemmas.

Lemma 6. *If $x\ \mathcal{R}_\sigma\ N$ and $M \to_*^0 N$ then $x\ \mathcal{R}_\sigma\ M$.*

Lemma 7. *If $x \triangleright \mathcal{R}_\tau\ \mathsf{next}(M)$ and $M' \to^1 M$ 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 $\triangleright$ to the statement of the lemma. We proceed by induction on τ. 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 $\triangleright\left[y \leftarrow x\right].(y\ \mathcal{R}_{\mu\alpha.\tau}\ M)$ which by definition of $\mathcal{R}_{\mu\alpha.\tau}$ is

$$\begin{aligned}&\triangleright\left[y \leftarrow x\right].\Sigma N' N''. \mathtt{unfold}\ M \to_*^0 N' \text{ and}\\ &\qquad N' \to^1 N'' \text{ and } (y \triangleright \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(N''))\end{aligned}$$

Since zero-step reductions cannot eliminate outer `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

$$\begin{aligned}&\triangleright\left[y \leftarrow x\right].(\Sigma N. M \to_*^0 N \text{ and}\\ &\qquad (\theta_{\tau[\mu\alpha.\tau/\alpha]}(y)\ \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathtt{unfold}\ N))\end{aligned}$$

Since $\mathtt{unfold}\ M \to_*^0 \mathtt{unfold}\ N$, by Lemma 6 we get

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

which by (17) is

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

By (13) 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. □

Lemma 8 (Fundamental Lemma)**.** *Suppose $\Gamma \vdash M : \tau$, for $\Gamma \equiv x_1 : \tau_1, \cdots, x_n : \tau_n$ and $N_i : \tau_i$, $\gamma_i : [\![\tau_i]\!]$ 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 induction on the typing judgment. Here we sketch the most interesting cases, namely those of `unfold` and `fold`.

$\Gamma \vdash \mathtt{unfold}\ M : \tau[\mu\alpha.\tau/\alpha]$ we want to show that

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

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

$$\mathtt{unfold}\ (M[\vec{N}/\vec{x}]) \to_*^0 M' \text{ and } M' \to^1 M''$$

and that $[\![M]\!]\,(\vec{\gamma}) \triangleright \mathcal{R}_{\tau[\mu\alpha.\tau/\alpha]}\ \mathsf{next}(M'')$. By Lemma 7

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

and since $\mathtt{unfold}\ (M[\vec{N}/\vec{x}]) \to_*^0 M'$ by Lemma 6 we get

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

By definition of the interpretation

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

which is what we wanted.

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

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

First off, by definition of the substitution function $(\texttt{fold}\ M)[\vec{N}/\vec{x}])$ is equal to $\texttt{fold}\ \ (M[\vec{N}/\vec{x}])$. Thus, 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 (17) to

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

which is true by induction hypothesis. □

From the Fundamental lemma we can now prove computational adequacy.

Theorem 9 (Intensional Computational Adequacy)**.** *If $M\colon 1$ is a closed term then $M \Downarrow^k \langle\rangle$ iff $[\![M]\!]\,(*) = \delta^k(\eta(\star))$.*

Proof. The proof is similar to (Paviotti et al. 2015). □

From Theorem 9 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 2. Suppose $\Gamma \vdash M : \tau$ and $[\![M]\!] = [\![N]\!]$. If $C[-]$ has type $\Gamma, \tau \to (-, 1)$ and $C[M] \Downarrow^k \langle\rangle$ also $C[N] \Downarrow^k \langle\rangle$.

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 $\bot$.

To do this we need to consider global behaviour of computations, as opposed to the local (or finitely computable) behaviour captured by the guarded recursive lifting monad L. To understand what this means, consider the interpretation of $L1$ in the topos of trees as described in Section 2.1. For each number n, the set

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

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.

On the other hand, consider the partiality monad (Capretta 2005) L^{gl} defined as the coinductive solution to the type equation

$$L^{\mathrm{gl}}A \cong A + L^{\mathrm{gl}}A \tag{18}$$

When interpreted in **Set**, $L^{\mathrm{gl}}1$ is $\bar{\omega}$, i.e., describes the set of all possible behaviors of a computation of unit type.

Coinductive types can be encoded in gDTT using guarded recursive types, following ideas of Atkey and McBride (Atkey and McBride 2013; Møgelberg 2014). The encoding uses universal quantification over clocks, which we now briefly recall, refering to (Bizjak et al. 2016) for details.

6.1 Universal quantification over clocks

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 $\overset{\kappa}{\triangleright}$, a fixed point combinator, and so on. The development of this paper so far has been in a context of a single implicit clock κ which we are going to make explicit only when necessary to avoid clutter.

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 $t\colon \forall\kappa.A$.

The type system allows for a restricted elimination rule for $\triangleright$. If t is of type $\triangleright A$ 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:

$$\begin{aligned}&\mathsf{force} : \forall\kappa.\triangleright A \to \forall\kappa.A\\ &\mathsf{force} \overset{\mathrm{def}}{=\!=} \lambda x.\ \mathsf{prev}\ \kappa.x[\kappa]\end{aligned} \tag{19}$$

The type constructor $\forall\kappa.(-)$ is modelled by taking sets of global elements. In particular, $\forall\kappa.L1$ is modelled as $\bar{\omega}$. In fact, one can prove in the type theory, that defining

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

gives a coinductive solution to (18), if κ is not free in A (Møgelberg 2014). For types A and B we say the two are type isomorphic if there exist two terms $f : A \to B$ and $g : B \to A$ such that $f(g(x)) \equiv x$ and $g(f(x)) \equiv x$. When A is a type that does not mention any free clock variables and B is free with $x : A$ and a clock variable κ, the following type isomorphism is derivable from the gDTTrules (Bizjak et al. 2016)

$$\forall\kappa.\Sigma(x : A).B \cong \Sigma(x : A).\forall\kappa.B \tag{20}$$

6.2 Global interpretation of types and terms

As said above, the model of FPC can be considered as being defined w.r.t. 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

$$[\![M]\!] : \forall\kappa.([\![\Gamma]\!] \to [\![\tau]\!])$$

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

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

$$\delta_\sigma^{\mathrm{gl}}(x) \overset{\mathrm{def}}{=\!=} \Lambda\kappa.\delta_\sigma(x[\kappa]) \tag{21}$$

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

6.3 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 Danielsson (Danielsson 2012) for the coinductive partiality monad.

Definition 10. 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:

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

The lifting of R, intuitively, captures computations that differ for a finite amount of computational steps or both diverge. For example, $\bot$ as defined in Section 4 is always related to itself which can be shown by guarded recursion as follows. Suppose $\triangleright(\bot\ LR\ \bot)$. Since $\bot = \theta(\mathsf{next}(\bot))$, to prove $\bot\ LR\ \bot$, we must prove $\mathsf{next}(\bot)\ \triangleright LR\ \mathsf{next}(\bot)$. But, this type is equal to the assumption $\triangleright(\bot\ LR\ \bot)$ by (17).

We can also prove that LR is closed under application of δ on either side.

Lemma 11. *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:

$$\triangleright(\Pi x : LA, y : LB.x\ LR\ y \Rightarrow x\ LR\ \delta_{LB}(y)). \tag{22}$$

We proceed by case analysis on x and y.

We show only 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 (16), $\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].x''\ LR\ y''$. By the guarded recursion hypothesis (22) we get

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

which can be rewritten to

$$\triangleright\left[x'' \leftarrow x', y'' \leftarrow y'\right].x''\ LR\ \theta_{LB}(\mathsf{next}(y'')) \tag{23}$$

By (15) there is an inhabitant of the type

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

and thus (23) implies $\triangleright\left[x'' \leftarrow x'\right].x''\ LR\ \theta_{LB}(y')$, which, by (17) 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

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

Lemma 12. *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 $\delta^{\mathrm{gl}}(x)\ L^{\mathrm{gl}}R\ y$.*

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 13. *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 $x\ L^{\mathrm{gl}}R\ y$.*

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

Figure 11: The logical relation $\approx_\tau$ is a predicate over denotations of τ of type $[\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$

Proof. Assume $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$. We can rewrite this type by unfolding definitions and (17) as follows.

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

Using force (19) this implies $\forall\kappa.(x[\kappa]\ LR\ (y[\kappa]))$ which is equal to $x\ L^{\mathrm{gl}}R\ y$. □

Lemma 14. *For all x of type $L^{\mathrm{gl}}A$ and y of type $L^{\mathrm{gl}}B$, if $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ y$ then $x\ L^{\mathrm{gl}}R\ y$.*

Proof. Assume $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ y$. Then by applying Lemma 12 we get $\delta^{\mathrm{gl}}_{LA}(x)\ L^{\mathrm{gl}}R\ \delta^{\mathrm{gl}}_{LB}(y)$ and by applying Lemma 13 we get $x\ L^{\mathrm{gl}}R\ y$. □

6.4 Relating terms up to extensional equivalence

Figure 11 defines for each FPC type τ the logical relation $\approx_\tau : [\![\tau]\!] \times [\![\tau]\!] \to \mathcal{U}$. The definition is by guarded recursion, and the well-definedness can be formalised using an argument similar to that used for well-definedness of θ explained in Section 5.2. The case of recursive types is well typed by Lemma 4. The figure uses the following lifting of relations to sum types.

Definition 15. 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)

$$\mathsf{inl}(x)\ (R + R')\ \mathsf{inl}(y) \stackrel{\text{def}}{=\!=} x\ R\ y$$
$$\mathsf{inr}(x)\ (R + R')\ \mathsf{inr}(y) \stackrel{\text{def}}{=\!=} x\ R'\ y$$

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

Definition 16. For $\Gamma \equiv x_1 : \sigma_1, \cdots, x_n : \sigma_n$ and for f, g of type $[\![\Gamma]\!] \to [\![\tau]\!]$ define

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

Definition 17. For f, g of type $\forall\kappa.([\![\Gamma]\!] \to [\![\tau]\!])$ define

$$f\ \approx^{\mathrm{gl}}_{\Gamma,\tau}\ g \stackrel{\text{def}}{=\!=} \forall\kappa.f[\kappa]\ \approx_{\Gamma,\tau}\ g[\kappa]$$

Contextual equivalence of FPC is defined in the standard way by observing convergence at unit type.

Definition 18. Let $\Gamma \vdash M, N : \tau$. We say that M, N are contextually equivalent, written $M\ \approx_{\mathrm{CTX}}\ N$, if for all contexts C of type $(\Gamma, \tau) \to (-, 1)$

$$\forall\kappa.C[M] \Downarrow \langle\rangle \iff \forall\kappa.C[N] \Downarrow \langle\rangle$$

Finally we can state the main theorem of this section.

Theorem 19 (Extensional Computational Adequacy)**.** *If* $\Gamma \vdash M, N : \tau$ *and* $[\![M]\!]^{\mathrm{gl}} \approx^{\mathrm{gl}}_{\Gamma,\tau} [\![N]\!]^{\mathrm{gl}}$ *then* $M \approx_{CTX} N$

We now sketch a proof of Theorem 19. The first lemma needed for the proof states that the interpretation of any term is related to it self. This needs to be proved by induction over terms, as the logical relation is not reflexive, as also noted by Escardó (Escardó 1999). As a counter example, consider a function $f\colon [\![1]\!] \to [\![1]\!]$ which diverges if its input takes a step and converges otherwise. Such a function is definable in the metalanguage, but not in FPC.

Lemma 20. *If* $\Gamma \vdash M : \sigma$*, then* $[\![M]\!] \approx_{\Gamma,\sigma} [\![M]\!]$.

The global lifting of the logical relation is closed under context.

Lemma 21. *If* $\Gamma \vdash M, N : \tau$ *and* $[\![M]\!] \approx^{\mathrm{gl}}_{\Gamma,\tau} [\![N]\!]$ *then for all contexts* C *of type* $(\Gamma, \tau) \to (-, 1)$, $[\![C[M]]\!] \approx^{\mathrm{gl}}_{(-,1)} [\![C[N]]\!]$

The following 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 22. *For all* x, y *of type* $[\![1]\!]^{\mathrm{gl}}$*, if* $x \approx^{\mathrm{gl}}_{(-,1)} y$ *then*

$$\Sigma n.x = (\delta_1^{\mathrm{gl}})^n(\eta(\star)) \Leftrightarrow \Sigma m.y = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$$

Proof. (Sketch). We show the left to right implication, so suppose $x = (\delta_1^{\mathrm{gl}})^n(\eta(\star))$. 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 $y[\kappa] = \delta_1^m(\eta(\star))$. By type isomorphism (20), since m is a natural number, this implies there exists m such that for all κ, $y[\kappa] = \delta_1^m(\eta(\star))$ which implies $y = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$ by clock extensionality (10).

In the inductive case $n = n' + 1$, since by Lemma 14 $(\delta_1^{\mathrm{gl}})^{n'}([\![v]\!]^{\mathrm{gl}}) \approx^{\mathrm{gl}}_1 y$, the induction hypothesis implies $\Sigma m.y = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$. □

Proof of Theorem 19. 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 $\forall\kappa.C[M] \Downarrow \langle\rangle$ also $\forall\kappa.C[N] \Downarrow \langle\rangle$. So suppose $\forall\kappa.C[M] \Downarrow \langle\rangle$. By definition this means $\forall\kappa.\Sigma n.C[M] \Downarrow^n \langle\rangle$. Since n is a natural number, i.e. a type that does not mention any clock variable, by type isomorphism (20) we have that there exists n such that $\forall\kappa.C[M] \Downarrow^n \langle\rangle$. By Adequacy Theorem 9 we get $\forall\kappa.[\![C[M]]\!] = (\delta_1)^n(\eta(\star))$ which is equivalent to $[\![C[M]]\!]^{\mathrm{gl}} = (\delta_1^{\mathrm{gl}})^n(\eta(\star))$. We can apply Lemma 21 together with the assumption and get $[\![C[M]]\!]^{\mathrm{gl}} \approx^{\mathrm{gl}}_1 [\![C[N]]\!]^{\mathrm{gl}}$, so by Lemma 22 there exists an m such that $[\![C[N]]\!]^{\mathrm{gl}} = (\delta_1^{\mathrm{gl}})^m(\eta(\star))$ which means there exists an m such that $\forall\kappa.[\![C[N]]\!] = (\delta_1)^m(\eta(\star))$. By applying Adequacy Theorem 9 once again we get $\forall\kappa.C[N] \Downarrow \langle\rangle$ as desired. □

7. 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. (Benton et al. 2009) 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 advanced programming language features in future work.

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.

Acknowledgments

This research was supported by DFF-Research Project 1 Grant no. 4002-00442, from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

A. W. Appel, P. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, pages 109–122, 2007.

R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.

N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in coq. In *TPHOLs*, pages 115–130, 2009.

N. Benton, L. Birkedal, A. Kennedy, and C. Varming. Formalizing domains, ultrametric spaces and semantics of programming languages. 2010.

L. Birkedal and R. E. Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS*, pages 213–222, 2013.

L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS*, 8(4), 2012.

A. Bizjak and R. E. Møgelberg. A model of guarded recursion with clock synchronisation. In *MFPS*, 2015.

A. Bizjak, L. Birkedal, and M. Miculan. A model of countable nondeterminism in guarded type theory. In *RTA-TLCA*, pages 108–123, 2014.

A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *FoSSaCS*, 2016.

V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

N. A. Danielsson. Operational semantics using the partiality monad. In *ICFP*, pages 127–138, 2012.

M. Escardó. A metric model of PCF. Unpublished, 1999.

J. M. E. Hyland. First steps in synthetic domain theory. In *Category Theory*, pages 131–156, 1991.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *CSL-LICS*, 2014.

H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

M. Paviotti, R. E. Møgelberg, and L. Birkedal. A model of PCF in guarded type theory. *Electr. Notes Theor. Comput. Sci.*, 319:333–349, 2015.

A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2):66–90, 1996.

B. Reus. Synthetic domain theory in type theory: Another logic of computable functions. In *TPHOLs*, 1996.

G. Rosolini. *Continuity and effectiveness in topoi*. PhD thesis, University of Oxford, 1986.

A. K. Simpson. Computational adequacy for recursive types in models of intuitionistic set theory. In *LICS*, pages 287–298, 2002.

K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.