En Garde! Unguarded Iteration for Reversible Computation in the Delay Monad*

Robin Kaarsgaard[1][0000−0002−7672−799X] and Niccolò Veltri[2][0000−0002−7230−3436]

[1] DIKU, Department of Computer Science, University of Copenhagen
robin@di.ku.dk
[2] Department of Computer Science, IT University of Copenhagen
nive@itu.dk

Abstract. Reversible computation studies computations which exhibit both forward and backward determinism. Among others, it has been studied for half a century for its applications in low-power computing, and forms the basis for quantum computing.
Though certified program equivalence is useful for a number of applications (e.g., certified compilation and optimization), little work on this topic has been carried out for reversible programming languages. As a notable exception, Carette and Sabry have studied the equivalences of the finitary fragment of Π°, a reversible combinator calculus, yielding a two-level calculus of type isomorphisms and equivalences between them. In this paper, we extend the two-level calculus of finitary Π° to one for full Π° (i.e., with both recursive types and iteration by means of a trace combinator) using the delay monad, which can be regarded as a "computability-aware" analogue of the usual maybe monad for partiality. This yields a calculus of iterative (and possibly non-terminating) reversible programs acting on user-defined dynamic data structures together with a calculus of certified program equivalences between these programs.

Keywords: reversible computation · iteration · delay monad

1 Introduction

Reversible computation is an emerging computation paradigm encompassing computations that are not just deterministic when executed the *forward* direction, but also in the *backward* direction. While this may seem initially obscure, reversible computation forms the basis for quantum computing, and has seen applications in a number of different areas such as low-power computing [28], robotics [30], discrete event simulation [33], and the simultaneous construction of parser/pretty printer pairs [32]. Like classical computing, it has its own automata [4], circuit model [40], machine architectures [35], programming languages [21, 22, 41, 34, 20], semantic metalanguages [15, 25, 24], and so on.

* Niccolò Veltri was supported by a research grant (13156) from VILLUM FONDEN.

Π° is a family of reversible combinator calculi comprising structural isomorphisms and combinators corresponding to those found in dagger-traced ω-continuous rig categories [26] (a kind of dagger category with a trace, monoidal sums $\oplus$ and products $\otimes$ such that they form a rig structure, and fixed points of the functors formed from the rig structure). Though superficially simple, Π° is expressive enough as a metalanguage to give semantics to the typed reversible functional programming language Theseus [22].

In [7], Carette and Sabry studied the equivalences of isomorphisms in the finitary fragment of Π° (i.e., without recursive types and iteration via the trace combinator), and showed that these equivalences could be adequately described by another combinator calculus of equivalences of isomorphisms, in sum yielding a two-level calculus of isomorphisms and equivalences of isomorphisms. In this paper, we build on this work to produce a (fully formalized) two-level calculus for full Π° (supporting both recursive types and iteration) via the delay monad, using insights gained from the study of its Kleisli category [37, 8, 39], as well as of join inverse categories in which reversible iteration may be modelled [25].

The full Π° calculus cannot be modelled in the same framework of [7], since Martin-Löf type theory is a total language which in particular disallows the specification of a trace operator on types. Consequently, it is necessary to move to a setting supporting the existence of partial maps, and in type theory this can be done by using monads, by considering partiality as an effect. Our choice fell on the coinductive delay monad, introduced by Capretta [6] as a way of representing general recursive functions in Martin-Löf type theory. The delay datatype has been employed in a large number of applications, ranging from operational semantics of functional languages [11] to formalization of domain theory in type theory [5] and normalization by evaluation [1]. Here it is used for giving denotational semantics to Π°. In particular, we show how to endow the delay datatype with a trace combinator, whose construction factors through the specification of a uniform iteration operator [17, 16].

The uniform iteration operator introduces a notion of feedback, typically used to model control flow operations such as while loops. In the Kleisli category of the delay monad, this operation can be intuitively described as follows: We can apply a function $f : A \to B + A$ on an input $a : A$ and either produce an element $b : B$, or produce a new element $a' : A$ which can be fed back to f. This operation can be iterated, and it either terminates returning a value in B or it goes on forever without producing any output. This form of iteration is "unguarded" because it allows the possibility of divergence. The trace operator can then be seen as a particular form of iteration where, given a function $f : A + C \to B + C$, which can be decomposed as $f_{\mathsf{L}} : A \to B + C$ and $f_{\mathsf{R}} : C \to B + C$, we first apply f_{L} on an input $a : A$, and, if the latter operation produces a value $c : C$, we continue by iterating f_{R} on c. Notice that the notion of trace can be generally defined in monoidal categories where the monoidal structure is not necessarily given by coproducts, and it has been used to model other things besides iteration, such as partial traces in vector spaces [23], though this use falls outside of the scope of this paper.

Throughout the paper, we reason constructively in Martin-Löf type theory. Classically, the delay monad (quotiented by weak bisimilarity) is isomorphic to the maybe monad $\mathsf{Maybe}\, X = X + 1$, and thus just a complication of something that can be expressed much simpler. Constructively, however, they are very different. In particular, it is impossible to define a well-behaved trace combinator for the maybe monad without assuming classical principles such as the limited principle of omniscience.

We have fully formalized the development of the paper in the dependently typed programming language Agda [31]. The code is available online at `https://github.com/niccoloveltri/pi0-agda`. The formalization uses Agda 2.6.0.

Overview In Section 2, we present the syntax of Π° as formalized in Agda, with particular emphasis on recursive types and the trace operator. In Section 3, we recall the definition of Capretta's delay datatype and weak bisimilarity. We discuss finite products and coproducts in the Kleisli category of the delay monad and we introduce the category of partial isomorphisms that serves as the denotational model of Π°. In Section 4, we build a complete Elgot monad structure on the delay datatype, that allows the encoding of a dagger trace operator in the category of partial isomorphisms. In Section 5, we formally describe the interpretation of Π° types, terms and terms equivalences. We conclude in Section 6 with some final remarks and discussion on future work.

The type-theoretical framework Our work is settled in Martin-Löf type theory with inductive and coinductive types. We write $(a : A) \to B\, a$ for dependent function spaces and $(a : A) \times B\, a$ for dependent products. We allow dependent functions to have implicit arguments and indicate implicit argument positions with curly brackets (as in Agda). We use the symbol $=$ for definitional equality of terms and $\equiv$ for propositional equality. Given $f : A \to C$ and $g : B \to C$, we write $[f, g] : A + B \to C$ for their copairing. The coproduct injections are denoted inl and inr. Given $h : C \to A$ and $k : C \to B$, we write $\langle h, k\rangle : C \to A \times B$ for their pairing. The product projections are denoted fst and snd. The empty type is 0 and the unit type is 1. We write Set for the category of types and functions between them. We also use Set to denote the universe of types. We define $A \leftrightarrow B = (A \to B) \times (B \to A)$.

We do not assume uniqueness of identity proofs (UIP), i.e. we do not consider two proofs of $x \equiv y$ necessarily equal. Agda natively supports UIP, so we have to manually switch it off using the `without-K` option.

In Section 3, we will need to quotient a certain type by an equivalence relation. Martin-Löf type theory does not support quotient types, but quotients can be simulated using setoids [3]. Alternatively, we can consider extensions of type theory with quotient types à la Hofmann [19], such as homotopy type theory [36]. Setoids and quotient types à la Hofmann are not generally equivalent approaches, but they are indeed equivalent for the constructions we develop in this work. Therefore, in the rest of the paper we assume the existence of quotient types and we refrain from technical discussions on their implementation.

2 Syntax of Π°

In this section, we present the syntax of Π°. The 1-structure of Π°, i.e. its types and terms, has originally been introduced by James and Sabry [21]. In particular, we include the presence of recursive types and a primitive trace combinator. Following Carette and Sabry's formalization of the finitary fragment of Π°, we consider a collection of equivalences between terms. Our list of axioms notably differs from theirs in that we do not require each term to be a total isomorphism, we ask only for the existence of a partial inverse.

Formally, the collection of types of Π° correspond to those naturally interpreted in dagger traced ω-continuous rig categories (see [26]).

2.1 Types

The types of Π° are given by the grammar:

$$A ::= \mathsf{Z} \mid A \oplus A \mid \mathsf{I} \mid A \otimes A \mid X \mid \mu X.A$$

where X ranges over a set of variables. In Agda, we use de Bruijn indexes to deal with type variables, so the grammar above is formally realized by the rules in Figure 1. The type $\mathsf{Ty}\, n$ represents Π° types containing at most n free variables. Variables themselves are encoded as elements of $\mathsf{Fin}\, n$, the type of natural numbers strictly smaller then n. The type constructor μ binds a variable, which, for $A : \mathsf{Ty}\,(n+1)$, we consider to be $n+1$.

It is also necessary to define substitutions. In Agda, given types $A : \mathsf{Ty}\,(n+1)$ and $B : \mathsf{Ty}\, n$, we construct $\mathsf{sub}\, A\, B : \mathsf{Ty}\, n$ to represent the substituted type $A[B/X]$, where X corresponds to the $(n+1)$-th variable in context.

$$\frac{}{\mathsf{Z} : \mathsf{Ty}\, n} \qquad \frac{}{\mathsf{I} : \mathsf{Ty}\, n} \qquad \frac{i : \mathsf{Fin}\, n}{\mathsf{Var}\, i : \mathsf{Ty}\, n}$$

$$\frac{A : \mathsf{Ty}\, n \quad B : \mathsf{Ty}\, n}{A \oplus B : \mathsf{Ty}\, n} \qquad \frac{A : \mathsf{Ty}\, n \quad B : \mathsf{Ty}\, n}{A \otimes B : \mathsf{Ty}\, n} \qquad \frac{A : \mathsf{Ty}\,(n+1)}{\mu A : \mathsf{Ty}\, n}$$

Fig. 1. Types of Π°, as formalized in Agda

2.2 Terms

The terms of Π° are inductively generated by the rules in Figure 2. They include the identity programs id and sequential composition of programs $\bullet$. $(\mathsf{Z}, \oplus)$ is a symmetric monoidal structure, with terms $\lambda_\oplus$, $\alpha_\oplus$ and $\sigma_\oplus$ as structural morphisms. Similarly for $(\mathsf{I}, \otimes)$. Moreover $\otimes$ distributes over Z and $\oplus$ from the right,

as evidenced by κ and δ. Elements of $\mu X.A$ are built using the term constructor fold and destructed with unfold. Finally, we find the trace combinator.

Every Π° program is reversible. The (partial) inverse of a program is given by the function $\mathsf{dagger} : (A \longleftrightarrow B) \to (B \longleftrightarrow A)$, recursively defined as follows:

$$
\begin{array}{ll}
\mathsf{dagger}\,\mathsf{id} & = \mathsf{id} \\
\mathsf{dagger}\,(f \oplus g) & = \mathsf{dagger}\,f \oplus \mathsf{dagger}\,g \\
\mathsf{dagger}\,\lambda_\oplus^{-1} & = \lambda_\oplus \\
\mathsf{dagger}\,\sigma_\oplus & = \sigma_\oplus \\
\mathsf{dagger}\,\alpha_\oplus^{-1} & = \alpha_\oplus \\
\mathsf{dagger}\,\lambda_\otimes^{-1} & = \lambda_\otimes \\
\mathsf{dagger}\,\alpha_\otimes & = \alpha_\otimes^{-1} \\
\mathsf{dagger}\,\kappa & = \kappa^{-1} \\
\mathsf{dagger}\,\delta & = \delta^{-1} \\
\mathsf{dagger}\,\mathsf{fold} & = \mathsf{unfold} \\
\mathsf{dagger}\,(\mathsf{trace}\,f) & = \mathsf{trace}\,(\mathsf{dagger}\,f)
\end{array}
\qquad
\begin{array}{ll}
\mathsf{dagger}\,(g \bullet f) & = \mathsf{dagger}\,f \bullet \mathsf{dagger}\,g \\
\mathsf{dagger}\,(f \otimes g) & = \mathsf{dagger}\,f \otimes \mathsf{dagger}\,g \\
\mathsf{dagger}\,\lambda_\oplus & = \lambda_\oplus^{-1} \\
\mathsf{dagger}\,\alpha_\oplus & = \alpha_\oplus^{-1} \\
\mathsf{dagger}\,\lambda_\otimes & = \lambda_\otimes^{-1} \\
\mathsf{dagger}\,\sigma_\otimes & = \sigma_\otimes \\
\mathsf{dagger}\,\alpha_\otimes^{-1} & = \alpha_\otimes \\
\mathsf{dagger}\,\kappa^{-1} & = \kappa \\
\mathsf{dagger}\,\delta^{-1} & = \delta \\
\mathsf{dagger}\,\mathsf{unfold} & = \mathsf{fold}
\end{array}
$$

The dagger operation is involutive. Notice that this property holds up to propositional equality. This is proved by induction on the term f.

$$\mathsf{daggerInvol} : (f : A \longleftrightarrow B) \to \mathsf{dagger}\,(\mathsf{dagger}\,f) \equiv f$$

The right unitor for $\oplus$ is given by $\rho_\oplus = \lambda_\oplus \bullet \sigma_\oplus : A \oplus \mathsf{Z} \longleftrightarrow A$, and $\rho_\otimes$ is defined similarly. Analogously, we can derive the left distributors $\kappa' : A \otimes \mathsf{Z} \longleftrightarrow A$ and $\delta' : A \otimes (B \oplus C) \longleftrightarrow (A \otimes B) \oplus (A \otimes C)$.

2.3 Term Equivalences

A selection of term equivalences of Π° is given in Figure 3. We only include the equivalences that either differ or have not previously considered by Carette and Sabry in their formalization of the finite fragment of Π° [7]. In particular, we leave out the long list of Laplaza's coherence axioms expressing that types and terms of Π° form a rig category [29]. We also omit the equivalences stating that $\lambda_\oplus^{-1}$ is the total inverse of $\lambda_\oplus$, similarly for the other structural morphisms.

The list of term equivalences in Figure 3 contains the trace axioms, displaying that the types of Π° form a traced monoidal category wrt. the additive monoidal structure $(\mathsf{Z}, \oplus)$ [23]. Next we ask for $\mathsf{trace}\,(\mathsf{dagger}\,f)$ to be the partial inverse of $\mathsf{trace}\,f$. Remember that we have defined $\mathsf{dagger}\,(\mathsf{trace}\,f)$ to be $\mathsf{trace}\,(\mathsf{dagger}\,f)$, so the axiom tracePIso is evidence that the trace combinator of Π° is a dagger trace. Afterwards we have two equivalences stating that unfold is the total inverse of fold.

It is possible to show that every term f has $\mathsf{dagger}\,f$ as its partial inverse. The notion of partial inverse used here comes from the study of *inverse categories* (see [27]) and amounts to saying that $\mathsf{dagger}\,f$ is the *unique* map that undoes everything which f does (unicity of partial inverses follows by the final equivalence of Figure 3, see [27]). Note that this is different from requiring that

$$\mathsf{id} : A \longleftrightarrow A \qquad \frac{g : B \longleftrightarrow C \quad f : A \longleftrightarrow B}{g \bullet f : A \longleftrightarrow C}$$

$$\frac{f : A \longleftrightarrow C \quad g : B \longleftrightarrow D}{f \oplus g : A \oplus B \longleftrightarrow C \oplus D} \qquad \frac{f : A \longleftrightarrow C \quad g : B \longleftrightarrow D}{f \otimes g : A \otimes B \longleftrightarrow C \otimes D}$$

$$\lambda_{\oplus} : \mathsf{Z} \oplus A \longleftrightarrow A \qquad \lambda_{\oplus}^{-1} : A \longleftrightarrow \mathsf{Z} \oplus A$$

$$\lambda_{\otimes} : \mathsf{I} \otimes A \longleftrightarrow A \qquad \lambda_{\otimes}^{-1} : A \longleftrightarrow \mathsf{I} \otimes A$$

$$\alpha_{\oplus} : (A \oplus B) \oplus C \longleftrightarrow A \oplus (B \oplus C) \qquad \alpha_{\oplus}^{-1} : A \oplus (B \oplus C) \longleftrightarrow (A \oplus B) \oplus C$$

$$\alpha_{\otimes} : (A \otimes B) \otimes C \longleftrightarrow A \otimes (B \otimes C) \qquad \alpha_{\otimes}^{-1} : A \otimes (B \otimes C) \longleftrightarrow (A \otimes B) \otimes C$$

$$\sigma_{\oplus} : A \oplus B \longleftrightarrow B \oplus A \qquad \sigma_{\otimes} : A \otimes B \longleftrightarrow B \otimes A$$

$$\kappa : \mathsf{Z} \otimes A \longleftrightarrow \mathsf{Z} \qquad \delta : (A \oplus B) \otimes C \longleftrightarrow (A \otimes C) \oplus (B \otimes C)$$

$$\kappa^{-1} : \mathsf{Z} \longleftrightarrow \mathsf{Z} \otimes A \qquad \delta^{-1} : (A \otimes C) \oplus (B \otimes C) \longleftrightarrow (A \oplus B) \otimes C$$

$$\mathsf{fold} : A[\mu X.A/X] \longleftrightarrow \mu X.A \qquad \mathsf{unfold} : \mu X.A \longleftrightarrow A[\mu X.A/X]$$

$$\frac{f : A \oplus C \longleftrightarrow B \oplus C}{\mathsf{trace}\, f : A \longleftrightarrow B}$$

Fig. 2. Terms of Π°

f is an isomorphism in the usual sense, as $\mathsf{dagger}\, f \bullet f$ is not going to be the identity when f is only partially defined, though it will behave as the identity on all points where f is defined.

The proof that every term has $\mathsf{dagger}\, f$ as its partial inverse proceeds by induction on f.

$$\mathsf{existsPIso} : (f : A \longleftrightarrow B) \to f \bullet \mathsf{dagger}\, f \bullet f \iff f$$

3 Delay Monad

The coinductive delay datatype was first introduce by Capretta for representing general recursive functions in Martin-Löf type theory [6]. Given a type A, elements of $\mathsf{Delay}\, A$ are possibly non-terminating "computations" returning a value of A whenever they terminate. Formally, $\mathsf{Delay}\, A$ is defined as a coinductive type with the following introduction rules:

$$\frac{a : A}{\mathsf{now}\, a : \mathsf{Delay}\, A} \qquad \frac{x : \mathsf{Delay}\, A}{\mathsf{later}\, x : \mathsf{Delay}\, A}$$

The constructor now embeds A into $\mathsf{Delay}\, A$, so $\mathsf{now}\, a$ represents the terminating computation returning the value a. The constructor later adds an additional

$$\mathsf{naturality_L} : f \bullet \mathsf{trace}\, g \iff \mathsf{trace}\,((f \oplus \mathsf{id}) \bullet g)$$

$$\mathsf{naturality_R} : \mathsf{trace}\, g \bullet f \iff \mathsf{trace}\,(g \bullet (f \oplus \mathsf{id}))$$

$$\mathsf{dinaturality} : \mathsf{trace}\,((\mathsf{id} \oplus f) \bullet g) \iff \mathsf{trace}\,(g \bullet (\mathsf{id} \oplus f))$$

$$\mathsf{superposing} : \mathsf{trace}\,(\alpha_{\oplus}^{-1} \bullet (\mathsf{id} \oplus f) \bullet \alpha_{\oplus}) \iff \mathsf{id} \oplus \mathsf{trace}\, f$$

$$\mathsf{vanishing}_{\oplus} : \mathsf{trace}\, f \iff \mathsf{trace}\,(\mathsf{trace}\,(\alpha_{\oplus}^{-1} \bullet f \bullet \alpha_{\oplus}))$$

$$\mathsf{vanishing_Z} : f \iff \rho_{\oplus}^{-1} \bullet \mathsf{trace}\, f \bullet \rho_{\oplus} \qquad \mathsf{yanking} : \mathsf{trace}\, \sigma_{\oplus} \iff \mathsf{id}$$

$$\mathsf{tracePIso} : \mathsf{trace}\, f \bullet \mathsf{trace}\,(\mathsf{dagger}\, f) \bullet \mathsf{trace}\, f \iff \mathsf{trace}\, f$$

$$\mathsf{foldIso} : \mathsf{fold} \bullet \mathsf{unfold} \iff \mathsf{id} \qquad \mathsf{unfoldIso} : \mathsf{unfold} \bullet \mathsf{fold} \iff \mathsf{id}$$

$$\mathsf{uniquePIso} : f \bullet \mathsf{dagger}\, f \bullet g \bullet \mathsf{dagger}\, g \iff g \bullet \mathsf{dagger}\, g \bullet f \bullet \mathsf{dagger}\, f$$

Fig. 3. Selection of term equivalences of Π°

unit of time delay to a computation. Double rule lines refer to a coinductive constructor, which can be employed an infinite number of times in the construction of a term of type $\mathsf{Delay}\, A$. E.g., the non-terminating computation never is corecursively defined as $\mathsf{never} = \mathsf{later}\, \mathsf{never}$.

The delay datatype is a monad. The unit is the constructor now, while the Kleisli extension bind is corecursively defined as follows:

$$\begin{aligned}
&\mathsf{bind} : (A \to \mathsf{Delay}\, B) \to \mathsf{Delay}\, A \to \mathsf{Delay}\, B \\
&\mathsf{bind}\, f\, (\mathsf{now}\, a) = f\, a \\
&\mathsf{bind}\, f\, (\mathsf{later}\, x) = \mathsf{later}\, (\mathsf{bind}\, f\, x)
\end{aligned}$$

The delay monad, like any other monad on Set, has a unique strength operation which we denote by $\mathsf{str} : A \times \mathsf{Delay}\, B \to \mathsf{Delay}\, (A \times B)$. Similarly, it has a unique costrength operation $\mathsf{costr} : (\mathsf{Delay}\, A) \times B \to \mathsf{Delay}\, (A \times B)$ definable using str. Moreover, the delay datatype is a commutative monad.

The Kleisli category of the delay monad, that we call $\mathbb{D}$, has types as objects and functions $f : A \to \mathsf{Delay}\, B$ as morphisms between A and B. In $\mathbb{D}$, the identity map on an object A is the constructor now, while the composition of morphisms $f : A \to \mathsf{Delay}\, B$ and $g : B \to \mathsf{Delay}\, C$ is given by $f \diamond g = \mathsf{bind}\, f \circ g$.

The delay datatype allows us to program with partial functions, but the introduced notion of partiality is intensional, in the sense that computations terminating with the same value in a different number of steps are considered different. To obtain an extensional notion of partiality, which in particular allows the specification of a well-behaved trace operator, we introduce the notion of (termination-sensitive) weak bisimilarity.

Weak bisimilarity is defined in terms of convergence. A computation $x : \mathsf{Delay}\, A$ converges to $a : A$ if it terminates in a finite number of steps returning

the value a. When this happens, we write $x \downarrow a$. The relation $\downarrow$ is inductively defined by the rules:

$$\frac{}{\mathsf{now}\, a \downarrow a} \qquad \frac{x \downarrow a}{\mathsf{later}\, x \downarrow a}$$

Two computations in $\mathsf{Delay}\, A$ are weakly bisimilar if they differ by a finite number of applications of the constructor later. Alternatively, we can say that two computations x and y are weakly bisimilar if, whenever x terminates returning a value a, then y also terminates returning a, and vice versa. This informal statement can be formalized in several different but logically equivalent ways [8, 39]. Here we consider a coinductive formulation employed e.g. in [12].

$$\begin{array}{cc} \dfrac{}{\mathsf{now}_{\approx} : \mathsf{now}\, a \approx \mathsf{now}\, a} & \dfrac{p : x_1 \approx x_2}{\mathsf{later}_{\approx}\, p : \mathsf{later}\, x_1 \approx \mathsf{later}\, x_2} \\[2ex] \dfrac{p : x \approx \mathsf{now}\, a}{\mathsf{laterL}_{\approx}\, p : \mathsf{later}\, x \approx \mathsf{now}\, a} & \dfrac{p : \mathsf{now}\, a \approx x}{\mathsf{laterR}_{\approx}\, p : \mathsf{now}\, a \approx \mathsf{later}\, x} \end{array} \tag{1}$$

Notice that the constructor $\mathsf{later}_{\approx}$ is coinductive. This allows us to prove $\mathsf{never} \approx \mathsf{never}$. Weak bisimilarity is an equivalence relation and it is a congruence w.r.t. the later operation. For example, here is a proof that weak bisimilarity is reflexive.

$$\begin{aligned} &\mathsf{refl}_{\approx} : \{x : \mathsf{Delay}\, A\} \to x \approx x \\ &\mathsf{refl}_{\approx}\, \{\mathsf{now}\, a\} = \mathsf{now}_{\approx} \\ &\mathsf{refl}_{\approx}\, \{\mathsf{later}\, x\} = \mathsf{later}_{\approx}\, (\mathsf{refl}_{\approx}\, \{x\}) \end{aligned}$$

We call $\mathbb{D}_{\approx}$ the category $\mathbb{D}$ with homsets quotiented by pointwise weak bisimilarity. This means that in $\mathbb{D}_{\approx}$ two morphisms f and g are considered equal whenever $f\, a \approx g\, a$, for all inputs a. When this is the case, we also write $f \approx g$. The operation bind is compatible with weak bisimilarity, in the sense that $\mathsf{bind}\, f_1\, x_1 \approx \mathsf{bind}\, f_2\, x_2$ whenever $f_1 \approx f_2$ and $x_1 \approx x_2$.

As an alternative to quotienting the homsets of $\mathbb{D}$, we could have quotiented the delay datatype by weak bisimilarity: $\mathsf{Delay}_{\approx}\, A = \mathsf{Delay}\, A/{\approx}$. In previous work [8], we showed that this construction has problematic consequences if we employ Hofmann's approach to quotient types [19]. For example, it does not seem possible to lift the monad structure of Delay to $\mathsf{Delay}_{\approx}$ without postulating additional principles such as the axiom of countable choice. More fundamentally for this work, countable choice would be needed for modelling the trace operator of Π° in the Kleisli category of $\mathsf{Delay}_{\approx}$. Notice that, if the setoid approach to quotienting is employed, the latter constructions go through without the need for additional assumptions. In order to keep an agnostic perspective on quotient types and avoid the need for disputable semi-classical choice principles, we decided to quotient the homsets of $\mathbb{D}$ by (pointwise) weak bisimilarity instead of the objects of $\mathbb{D}$.

3.1 Finite Products and Coproducts

Colimits in $\mathbb{D}_{\approx}$ are inherited from Set. This means that 0 is also the initial object of $\mathbb{D}_{\approx}$, similarly $A + B$ is the binary coproduct of A and B in $\mathbb{D}_{\approx}$. Given $f : A \to \mathsf{Delay}\, C$ and $g : B \to \mathsf{Delay}\, C$, their copairing is $[f, g]_{\mathbb{D}} = [f, g] : A+B \to \mathsf{Delay}\, C$. The operation $[-, -]_{\mathbb{D}}$ is compatible with weak bisimilarity, in the sense that $[f_1, g_1]_{\mathbb{D}} \approx [f_2, g_2]_{\mathbb{D}}$ whenever $f_1 \approx f_2$ and $g_1 \approx g_2$. The coproduct injections are given by $\mathsf{inl}_{\mathbb{D}} = \mathsf{now} \circ \mathsf{inl} : A \to \mathsf{Delay}\,(A + B)$ and $\mathsf{inr}_{\mathbb{D}} = \mathsf{now} \circ \mathsf{inr} : B \to \mathsf{Delay}\,(A + B)$.

Just as limits in Set do not lift to limits in the category Par of sets and partial functions, they do not lift to $\mathbb{D}_{\approx}$ either. This is not an issue with these concrete formulations of partiality, but rather with the interaction of partiality (in the sense of restriction categories, a kind of categories of partial maps) and limits in general (see [10, Section 4.4]). In particular, 1 is not the terminal object and $A \times B$ is not the binary product of A and B in $\mathbb{D}_{\approx}$. In fact, 0 is (also) the terminal object, with $\lambda\_.\, \mathsf{never} : A \to \mathsf{Delay}\, 0$ as the terminal morphism. Nevertheless, it is possible to prove that 1 and $\times$ are partial terminal object and partial binary products respectively, in the sense of Cockett and Lack's restriction categories [9, 10]. Here we refrain from making the latter statement formal. We only show the construction of the partial pairing operation, which we employ in the interpretation of Π°. Given $f : C \to \mathsf{Delay}\, A$ and $g : C \to \mathsf{Delay}\, B$, we define:

$$\langle f, g\rangle_{\mathbb{D}} : C \to \mathsf{Delay}\,(A \times B)$$
$$\langle f, g\rangle_{\mathbb{D}} = \mathsf{costr} \diamond (\mathsf{str} \circ \langle f, g\rangle)$$

Since the delay monad is commutative, the function $\langle f, g\rangle_{\mathbb{D}}$ is equal to $\mathsf{str}\diamond(\mathsf{costr}\circ \langle f, g\rangle)$. The operation $\langle -, -\rangle_{\mathbb{D}}$ is compatible with weak bisimilarity.

3.2 Partial Isomorphisms

In order to model the reversible programs of Π°, we need to consider reversible computations in $\mathbb{D}_{\approx}$. Given a morphism $f : A \to \mathsf{Delay}\, B$, we say that it is a partial isomorphism if the following type is inhabited:

$$\mathsf{isPartialIso}\, f = (g : B \to \mathsf{Delay}\, A) \times ((a : A)(b : B) \to f\, a \downarrow b \leftrightarrow g\, b \downarrow a)$$

In other words, f is a partial isomorphism if there exists a morphism $g : B \to \mathsf{Delay}\, A$ such that, if $f\, a$ terminates returning a value b, then $g\, b$ terminates returning a, and vice versa. Given a partial isomorphism f, we denote its partial inverse by $\mathsf{dagger}_{\mathbb{D}}\, f$.

In $\mathbb{D}_{\approx}$, our definition of partial isomorphisms is equivalent to the standard categorical one [27] (see also [9]), which, translated in our type-theoretical setting, is

$$\mathsf{isPartialIsoCat}\, f = (g : B \to \mathsf{Delay}\, A) \times f \diamond g \diamond f \approx f \times g \diamond f \diamond g \approx g$$

We denote $A \simeq B$ the type of partial isomorphisms between A and B:

$$A \simeq B = (f : A \to \mathsf{Delay}\, B) \times \mathsf{isPartialIso}\, f$$

We call $\mathsf{Inv}\mathbb{D}_\approx$ the subcategory of $\mathbb{D}_\approx$ consisting of (equivalence classes of) partial isomorphisms. Note that $\mathsf{Inv}\mathbb{D}_\approx$ inherits neither partial products nor coproducts of $\mathbb{D}_\approx$, as the universal mapping property fails in both cases. However, it can be shown that in the category $\mathsf{Inv}\mathbb{D}_\approx$, 0 is a zero object, $A + B$ is the disjointness tensor product of A and B (in the sense of Giles [15]) with unit 0, and $A \times B$ a monoidal product of A and B with unit 1 (though it is *not* an inverse product in the sense of Giles [15], as that would imply decidable equality on all objects). In particular, we can derive the following operations, modelling the Π° term constructors $\oplus$ and $\otimes$:

$$\times_{\mathbb{D}\simeq} : A \simeq C \to B \simeq D \to A \times B \simeq C \times D$$
$$+_{\mathbb{D}\simeq} : A \simeq C \to B \simeq D \to A + B \simeq C + D$$

4 Elgot Iteration

A complete Elgot monad [16, 17] is a monad T whose Kleisli category supports unguarded uniform iteration. More precisely[3], a monad T is Elgot if there exists an operation

$$\mathsf{iter}_\mathsf{T} : (A \to \mathsf{T}\,(B + A)) \to A \to \mathsf{T}\,B$$

satisfying the following axioms:

$$\mathsf{fixpoint} : \mathsf{iter}_\mathsf{T}\, f \equiv [\eta_\mathsf{T}, \mathsf{iter}_\mathsf{T}\, f] \diamond_\mathsf{T} f$$

$$\mathsf{naturality} : g \diamond_\mathsf{T} \mathsf{iter}_\mathsf{T}\, f \equiv \mathsf{iter}_\mathsf{T}\, ([\mathsf{T}\mathsf{inl} \circ g, \eta \circ \mathsf{inr}] \diamond_\mathsf{T} f)$$

$$\mathsf{codiagonal} : \mathsf{iter}_\mathsf{T}\, (\mathsf{iter}_\mathsf{T}\, g) \equiv \mathsf{iter}_\mathsf{T}\, (\mathsf{T}[\mathsf{id}, \mathsf{inr}] \circ g)$$

$$\frac{p : f \circ h \equiv \mathsf{T}(\mathsf{id} + h) \circ g}{\mathsf{uniformity}\, p : \mathsf{iter}_\mathsf{T}\, f \circ h \equiv \mathsf{iter}_\mathsf{T}\, g}$$

where η_T is the unit of T and $\diamond_\mathsf{T}$ denotes morphism composition in the Kleisli category of T. The standard definition of uniform iteration operator includes the dinaturality axiom, which has recently been discovered to be derivable from the other laws [14, 16].

The delay monad is a complete Elgot monad for which the axioms holds up to weak bisimilarity, not propositional equality. In other words, the category $\mathbb{D}_\approx$ can be endowed with a uniform iteration operator. The specification of the

[3] Here we give the definition of complete Elgot monad on Set, but the definition of complete Elgot monad makes sense in any category with finite coproducts.

iteration operator relies on an auxiliary function $\mathsf{iter}'_{\mathbb{D}}$ corecursively defined as follows:

$$\begin{aligned}
&\mathsf{iter}'_{\mathbb{D}} : (A \to \mathsf{Delay}\,(B + A)) \to \mathsf{Delay}\,(B + A) \to \mathsf{Delay}\,B \\
&\mathsf{iter}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inl}\,b)) &&= \mathsf{now}\,b \\
&\mathsf{iter}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inr}\,a)) &&= \mathsf{later}\,(\mathsf{iter}'_{\mathbb{D}}\; f\,(f\,a)) \\
&\mathsf{iter}'_{\mathbb{D}}\; f\,(\mathsf{later}\,x) &&= \mathsf{later}\,(\mathsf{iter}'_{\mathbb{D}}\; f\,x)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{iter}_{\mathbb{D}} : (A \to \mathsf{Delay}\,(B + A)) \to A \to \mathsf{Delay}\,B \\
&\mathsf{iter}_{\mathbb{D}}\; f\,a = \mathsf{iter}'_{\mathbb{D}}\; f\,(f\,a)
\end{aligned}$$

The definition above can be given the following intuitive explanation. If $f\,a$ does not terminate, then $\mathsf{iter}_{\mathbb{D}}\; f\,a$ does not terminate either. If $f\,a$ terminates, there are two possibilities: either $f\,a$ converges to $\mathsf{inl}\,b$, in which case $\mathsf{iter}_{\mathbb{D}}\; f\,a$ terminates returning the value b; or $f\,a$ converges to $\mathsf{inr}\,a'$, in which case we repeat the procedure by replacing a with a'. Notice that in the latter case we also add one occurrence of later to the total computation time. This addition is necessary for ensuring the productivity of the corecursively defined function $\mathsf{iter}'_{\mathbb{D}}$. In fact, by changing the second line of its specification to $\mathsf{iter}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inr}\,a)) = \mathsf{iter}'_{\mathbb{D}}\; f\,(fa)$ and taking $f = \mathsf{inr}_{\mathbb{D}}$, we would have that $\mathsf{iter}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inr}\,a))$ unfolds indefinitely without producing any output. In Agda, such a definition would be rightfully rejected by the termination checker.

The operation $\mathsf{iter}_{\mathbb{D}}$ is compatible with weak bisimilarity, which means that $\mathsf{iter}_{\mathbb{D}}\; f_1 \approx \mathsf{iter}_{\mathbb{D}}\; f_2$ whenever $f_1 \approx f_2$.

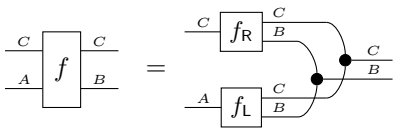
As mentioned above, $\mathsf{iter}_{\mathbb{D}}$ satisfies the Elgot iteration axioms only up to weak bisimilarity. Here we show the proof of the fixpoint axiom, which in turns relies on an auxiliary proof $\mathsf{fixpoint}'_{\mathbb{D}}$.

$$\begin{aligned}
&\mathsf{fixpoint}'_{\mathbb{D}} : (f : A \to \mathsf{Delay}\,(B + A)) \to \mathsf{bind}\,[\mathsf{now}, \mathsf{iter}_{\mathbb{D}}\; f]_{\mathbb{D}} \approx \mathsf{iter}'_{\mathbb{D}}\; f \\
&\mathsf{fixpoint}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inl}\,b)) &&= \mathsf{now}_{\approx} \\
&\mathsf{fixpoint}'_{\mathbb{D}}\; f\,(\mathsf{now}\,(\mathsf{inr}\,a)) &&= \mathsf{laterR}_{\approx}\;\mathsf{refl}_{\approx} \\
&\mathsf{fixpoint}'_{\mathbb{D}}\; f\,(\mathsf{later}\,x) &&= \mathsf{later}_{\approx}\,(\mathsf{fixpoint}'_{\mathbb{D}}\; f\,x)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{fixpoint}_{\mathbb{D}} : (f : A \to \mathsf{Delay}\,(B + A)) \to [\mathsf{now}, \mathsf{iter}_{\mathbb{D}}\; f]_{\mathbb{D}} \diamond f \approx \mathsf{iter}_{\mathbb{D}}\; f \\
&\mathsf{fixpoint}_{\mathbb{D}}\; f\,x = \mathsf{fixpoint}'_{\mathbb{D}}\; f\,(f\,x)
\end{aligned}$$

4.1 Trace

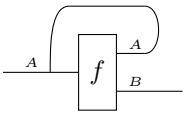
From the Elgot iteration operator it is possible to derive a trace operator. First, given $f : A+B \to C$, we introduce $f_{\mathsf{L}} = f \circ \mathsf{inl} : A \to C$ and $f_{\mathsf{R}} = f \circ \mathsf{inr} : B \to C$, so that $f = [f_{\mathsf{L}}, f_{\mathsf{R}}]$. Graphically:



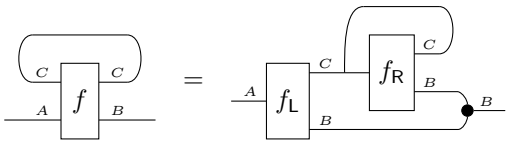
The trace operator in $\mathbb{D}_{\approx}$ is defined in terms of the iterator as follows:

$$\begin{aligned}&\mathsf{trace}_{\mathbb{D}} : (A + C \to \mathsf{Delay}\,(B + C)) \to A \to \mathsf{Delay}\,B\\&\mathsf{trace}_{\mathbb{D}}\, f = [\mathsf{now}, \mathsf{iter}_{\mathbb{D}}\, f_{\mathsf{R}}]_{\mathbb{D}} \diamond f_{\mathsf{L}}\end{aligned}$$

The operation $\mathsf{trace}_{\mathbb{D}}$ is compatible with weak bisimilarity. Graphically, we express the iterator on f as a wire looping back on the input, i.e., as



In this way, the definition of $\mathsf{trace}_{\mathbb{D}}$ may be expressed graphically as

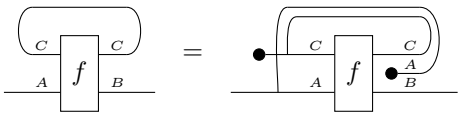


Intuitively, the function f_{L} initialises the loop. It either diverges, so that the trace of f diverges as well, or it terminates. It either terminates with an element $b : B$, in which case the loop ends immediately returning b, or it converges to a value $c : C$, and in this case we proceed by invoking the iteration of f_{R} on c.

It is well-known that a trace operator is obtainable from an iteration operator, as shown by Hasegawa [18]. His construction, instantiated to our setting, looks as follows:

$$\begin{aligned}&\mathsf{traceH}_{\mathbb{D}} : (A + C \to \mathsf{Delay}\,(B + C)) \to A \to \mathsf{Delay}\,B\\&\mathsf{traceH}_{\mathbb{D}}\, f = \mathsf{iter}_{\mathbb{D}}(\mathsf{Delay}\,(\mathsf{id} + \mathsf{inr}) \circ f) \circ \mathsf{inl}\end{aligned}$$

or graphically



It is not difficult to prove that the two possible ways of defining a trace operator from Elgot iteration are equivalent, in the sense that $\mathsf{trace}_{\mathbb{D}}\, f \approx \mathsf{traceH}_{\mathbb{D}}\, f$ for all $f : A + C \to \mathsf{Delay}\,(B + C)$.

The trace axioms follow from the Elgot iteration axioms.

We conclude this section by remarking that the construction of a trace operator in the Kleisli category of the maybe monad is impossible without the assumption of additional classical principles. In fact, given a map $f : A+C \to B+C+1$, let xs be the possibly infinite sequence of elements of $B + C + 1$ produced by the iteration of f on a given input in A. In order to construct the trace of f, we need to decide whether xs is a finite sequence terminating with an element of $B + 1$, or xs is an infinite stream of elements of C. This decision requires the

limited principle of omniscience, an instance of the law of excluded middle not provable in Martin-Löf type theory:

$$\mathsf{LPO} = (s : \mathbb{N} \to 2) \to ((n : \mathbb{N}) \times s\, n \equiv \mathsf{true}) + ((n : \mathbb{N}) \to s\, n \equiv \mathsf{false})$$

where 2 is the type of booleans, with true and false as only inhabitants.

4.2 Dagger Trace

We now move to show that $\mathsf{trace}_{\mathbb{D}}$ is a dagger trace operator, i.e. if f is a partial isomorphism, then $\mathsf{trace}_{\mathbb{D}}\, f$ is also a partial isomorphism with partial inverse $\mathsf{trace}_{\mathbb{D}}\, (\mathsf{dagger}_{\mathbb{D}}\, f)$.

This is proved by introducing the notion of *orbit* of an element $x : A+C$ wrt. a function $f : A+C \to \mathsf{Delay}\, (B+C)$. The orbit of x consists of the terms of type $B + C$ that are obtained in a finite number of steps from repeated applications of the function f on x. Formally, a term y belongs to the orbit of f wrt. x if the type $\mathsf{Orb}\, f\, x\, y$ is inhabited, with the latter type inductively defined as:

$$\frac{p : f\, x \downarrow y}{\mathsf{done}\, p : \mathsf{Orb}\, f\, x\, y} \qquad \frac{p : f\, x \downarrow \mathsf{inr}\, c \quad q : \mathsf{Orb}\, f\, (\mathsf{inr}\, c)\, y}{\mathsf{next}\, p\, q : \mathsf{Orb}\, f\, x\, y}$$

The notion of orbit can be used to state when the iteration of a function $f : A \to \mathsf{Delay}\, (B + A)$ on a input $a : A$ terminates with value $b : B$.

$$\mathsf{iter}_{\mathbb{D}}\, f\, a \downarrow b \leftrightarrow \mathsf{Orb}\, [\mathsf{inl}_{\mathbb{D}}, f]_{\mathbb{D}}\, (\mathsf{inr}\, a)\, (\mathsf{inl}\, b)$$

We refer the interested reader to our Agda formalization for a complete proof of this logical equivalence. Similarly, the orbit can be used to state when the trace of a function $f : A + C \to \mathsf{Delay}\, (B + C)$ on a input $a : A$ terminates with value $b : B$.

$$\mathsf{trace}_{\mathbb{D}}\, f\, a \downarrow b \leftrightarrow \mathsf{Orb}\, f\, (\mathsf{inl}\, a)\, (\mathsf{inl}\, b) \tag{2}$$

Showing that $\mathsf{trace}_{\mathbb{D}}$ is a dagger trace operator requires the construction of an inhabitant of $\mathsf{trace}_{\mathbb{D}}\, f\, a \downarrow b \leftrightarrow \mathsf{trace}_{\mathbb{D}}\, (\mathsf{dagger}_{\mathbb{D}}\, f)\, b \downarrow a$. Thanks to the logical equivalence in (2), this is equivalent to prove the following statement instead:

$$\mathsf{Orb}\, f\, (\mathsf{inl}\, a)\, (\mathsf{inl}\, b) \leftrightarrow \mathsf{Orb}\, (\mathsf{dagger}_{\mathbb{D}}\, f)\, (\mathsf{inl}\, b)\, (\mathsf{inl}\, a)$$

We give a detailed proof of the left-to-right direction, the other implication is derived in an analogous way. Notice that a term $p : \mathsf{Orb}\, f\, (\mathsf{inl}\, a)\, (\mathsf{inl}\, b)$ can be seen as a finite sequence of elements of C, precisely the intermediate values produced by $\mathsf{trace}_{\mathbb{D}}\, f\, a$ before converging to b. The orbit of b wrt. the partial inverse of f can therefore be computed by reversing the sequence of elements present in p. The construction of the reverse of an orbit is very similar to the way the reverse of a list is typically defined in a functional programming language like Haskell. We first consider an intermediate value $c : C$ and we assume to have already

reversed the initial section of the orbit between $\mathsf{inl}\, a$ and $\mathsf{inr}\, c$, that is a term $p' : \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, a)$.

$$\begin{aligned}
&\mathsf{reverseOrb}' : (i : \{a : A\}\{b : B\} \to f\, a \downarrow b \to \mathsf{dagger}_{\mathbb{D}}\, f\, b \downarrow a) \to \\
&\quad \{a : A\}\{b : B\}\{c : C\} \to \\
&\quad \mathsf{Orb}\, f\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, b) \to \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, a) \to \\
&\quad \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inl}\, b)\,(\mathsf{inl}\, a) \\
&\mathsf{reverseOrb}'\, i\,(\mathsf{done}\, p)\quad p' = \mathsf{next}\,(i\, p)\, p' \\
&\mathsf{reverseOrb}'\, i\,(\mathsf{next}\, p\, q)\; p' = \mathsf{reverseOrb}'\, i\, q\,(\mathsf{next}\,(i\, p)\, p')
\end{aligned}$$

The proof of $\mathsf{reverseOrb}'$ proceeds by structural induction on the final segment of the orbit between $\mathsf{inr}\, c$ and $\mathsf{inl}\, b$ that still needs to be reversed, which is the argument of type $\mathsf{Orb}\, f\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, b)$. There are two possibilities.

- We have $p : f\,(\mathsf{inr}\, c) \downarrow \mathsf{inl}\, b$, in which case $i\, p : \mathsf{dagger}_{\mathbb{D}}\, f\,(\mathsf{inl}\, b) \downarrow \mathsf{inl}\, c$. Then we return $\mathsf{next}\,(i\, p)\, p'$.
- There exists another value $c' : C$ such that $p : f\,(\mathsf{inr}\, c) \downarrow \mathsf{inr}\, c'$ and $q : \mathsf{Orb}\, f\,(\mathsf{inr}\, c')\,(\mathsf{inl}\, b)$. Then we recursively invoke the function $\mathsf{reverseOrb}'\, i$ on arguments q and $\mathsf{next}\,(i\, p)\, p' : \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inr}\, c')\,(\mathsf{inl}\, b)$.

The reverse of an orbit is derivable using the auxiliary function $\mathsf{reverseOrb}'$.

$$\begin{aligned}
&\mathsf{reverseOrb} : (i : \{a : A\}\{b : B\} \to f\, a \downarrow b \to \mathsf{dagger}_{\mathbb{D}}\, f\, b \downarrow a) \to \\
&\quad \{a : A\}\{b : B\} \to \\
&\quad \mathsf{Orb}\, f\,(\mathsf{inl}\, a)\,(\mathsf{inl}\, b) \to \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inl}\, b)\,(\mathsf{inl}\, a) \\
&\mathsf{reverseOrb}\, i\,(\mathsf{done}\, p)\quad = \mathsf{done}\,(i\, p) \\
&\mathsf{reverseOrb}\, i\,(\mathsf{next}\, p\, q) = \mathsf{reverseOrb}'\, i\, q\,(\mathsf{done}\,(i\, p))
\end{aligned}$$

The proof of $\mathsf{reverseOrb}$ proceeds by structural induction on the orbit of type $\mathsf{Orb}\, f\,(\mathsf{inl}\, a)\,(\mathsf{inl}\, b)$. There are two possibilities.

- We have $p : f\,(\mathsf{inl}\, a) \downarrow \mathsf{inl}\, b$, in which case $i\, p : \mathsf{dagger}_{\mathbb{D}}\, f\,(\mathsf{inl}\, b) \downarrow \mathsf{inl}\, a$. Then we return $\mathsf{done}\,(i\, p)$.
- There exists a value $c : C$ such that $p : f\,(\mathsf{inl}\, a) \downarrow \mathsf{inr}\, c$ and $q : \mathsf{Orb}\, f\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, b)$. We conclude by invoking the function $\mathsf{reverseOrb}'\, i$ on arguments q and $\mathsf{done}\,(i\, p) : \mathsf{Orb}\,(\mathsf{dagger}_{\mathbb{D}}\, f)\,(\mathsf{inr}\, c)\,(\mathsf{inl}\, a)$.

Summing up, in this section we have showed that the $\mathsf{trace}_{\mathbb{D}}$ operator can be restricted to act on partial isomorphisms. That is, the following type is inhabited:

$$\mathsf{trace}_{\mathbb{D}\simeq} : A + C \simeq B + C \to A \simeq B$$

5 Soundness

In this section, we provide some details on the interpretation of the syntax of Π°, presented in Section 2, into the category $\mathsf{Inv}\mathbb{D}_{\approx}$. Types of Π° are modelled as

objects of $\mathsf{Inv}\mathbb{D}_{\approx}$, which are types of the metatheory. In Agda, the interpretation of types $[\![-]\!]_{\mathsf{Ty}}$ takes in input a Π° type $A : \mathsf{Ty}\, n$ and an environment $\rho : \mathsf{Fin}\, n \to \mathsf{Set}$ giving semantics to each variable in context. The interpretation is mutually inductively defined together with the operation $[\![-]\!]_{\mu}$ giving semantics to the μ type former. Remember that, given $A : \mathsf{Ty}\,(n+1)$ and $B : \mathsf{Ty}\, n$, we write $\mathsf{sub}\, A\, B$ for the substituted type $A[B/X]$, where X corresponds to the $(n+1)$-th variable in context.

$$
\begin{array}{ll}
[\![\mathsf{Z}]\!]_{\mathsf{Ty}} & \rho = 0 \\
[\![A \oplus B]\!]_{\mathsf{Ty}} & \rho = [\![A]\!]_{\mathsf{Ty}}\, \rho + [\![B]\!]_{\mathsf{Ty}}\, \rho \\
[\![\mathsf{I}]\!]_{\mathsf{Ty}} & \rho = 1 \\
[\![A \otimes B]\!]_{\mathsf{Ty}} & \rho = [\![A]\!]_{\mathsf{Ty}}\, \rho \times [\![B]\!]_{\mathsf{Ty}}\, \rho \\
[\![\mathsf{Var}\, i]\!]_{\mathsf{Ty}} & \rho = \rho\, i \\
[\![\mu A]\!]_{\mathsf{Ty}} & \rho = [\![A]\!]_{\mu}\, \rho
\end{array}
\qquad\qquad
\frac{x : [\![\mathsf{sub}\, A\, (\mu A)]\!]_{\mathsf{Ty}}\, \rho}{\mathsf{semFold}\, x : [\![A]\!]_{\mu}\, \rho}
$$

By abuse of notation, we use here $\times$ (respectively $+$) to refer to the product (respectively coproduct) in $\mathbb{D}_{\approx}$ even though it fails to be a product (respectively coproduct) in $\mathsf{Inv}\mathbb{D}_{\approx}$. However, both of these are symmetric monoidal products in $\mathsf{Inv}\mathbb{D}_{\approx}$, so their use as objects of $\mathsf{Inv}\mathbb{D}_{\approx}$ in the interpretation above is justified.

Terms of Π° are modelled as morphism of $\mathsf{Inv}\mathbb{D}_{\approx}$, i.e. partial isomorphisms. Here we only display the interpretation of a selection of programs, we refer the interested reader to our Agda formalization for a complete definition of the interpretation of terms.

$$
\begin{array}{ll}
[\![-]\!]_{\longleftrightarrow} : (A \longleftrightarrow B) \to [\![A]\!]_{\mathsf{Ty}}\, \rho \simeq [\![B]\!]_{\mathsf{Ty}}\, \rho \\
[\![f \oplus g]\!]_{\longleftrightarrow} & = [\![f]\!]_{\longleftrightarrow} +_{\mathbb{D}\simeq} [\![g]\!]_{\longleftrightarrow} \\
[\![f \otimes g]\!]_{\longleftrightarrow} & = [\![f]\!]_{\longleftrightarrow} \times_{\mathbb{D}\simeq} [\![g]\!]_{\longleftrightarrow} \\
[\![\mathsf{trace}\, f]\!]_{\longleftrightarrow} & = \mathsf{trace}_{\mathbb{D}\simeq}\, [\![f]\!]_{\longleftrightarrow}
\end{array}
$$

Term equivalences of Π° are modelled as morphism equalities in $\mathsf{Inv}\mathbb{D}_{\approx}$, i.e. proofs of weak bisimilarity between two morphisms. Formally, we define an operation:

$$[\![-]\!]_{\Longleftrightarrow} : (f \iff g) \to [\![f]\!]_{\longleftrightarrow} \approx [\![g]\!]_{\longleftrightarrow}$$

Again we refer the interested reader to our Agda formalization for a complete definition of the interpretation of term equivalences.

6 Conclusions

In this paper, we have extended the work of Carette and Sabry [7] to a (fully formalized) two-level calculus of Π° programs and program equivalences. Key in this effort was the use of the Kleisli category of the delay monad on Set under weak bisimilarity, which turned out to support iteration via a trace that preserves all partial isomorphisms, in this way giving semantics to the dagger trace of Π°. Further, the work was formalized using Agda 2.6.0.

It is natural to wonder if our work can be ported to other monads of partiality in Martin-Löf type theory. As already discussed in the end of Section

4.1, the maybe monad is not suitable for modelling a well-behaved trace combinator without the assumption of classical principles such as LPO. The partial map classifier [38, 13] $\mathsf{PMC}\,A = (P : \mathsf{Prop}) \to P \to A$, where Prop is the type of propositions (types with at most one inhabitant), supports the existence of a uniform iteration operator and therefore a trace. Nevertheless, the specification of iteration is more complicated then the one presented in Section 4 for the delay monad, which is a simple corecursive definition. The complete Elgot monad structure of PMC follows from its Kleisli category being a join restriction category, so iteration is defined in terms of least upper bounds of certain chains of morphisms. The subcategory of partial isomorphisms of the Kleisli category of PMC supports a dagger trace combinator, which can be proved following the general strategy in [25]. The exact same observations apply to the partiality monad in homotopy type theory [2, 8], to which the quotiented delay monad $\mathsf{Delay}_{\approx}\,A = \mathsf{Delay}\,A/{\approx}$ is isomorphic under the assumption of countable choice.

Though the Kleisli category of the delay monad on Set is well studied, comparatively less is known about this monad on other categories. It could be interesting to study under which conditions its iterator exists – e.g., whether this is still the case when Set is replaced with an arbitrary topos. Another avenue concerns the study of *time invertible* programming languages: Though not immediately clear in the current presentation, the trace on $\mathsf{Inv}\mathbb{D}_{\approx}$ is not just reversible (in the sense that it preserves partial isomorphisms) but in fact time invertible, in the sense that the number of computation steps needed to perform $\mathsf{trace}_{\mathbb{D}}\,(\mathsf{dagger}_{\mathbb{D}}\,f)$ is *precisely* the same as what is needed to perform $\mathsf{trace}_{\mathbb{D}}\,f$ on any input. Since the delay monad conveniently allows the counting of computation steps, we conjecture that this is an ideal setting in which to study such intentional semantics of reversible programming languages.

References

1. Abel, A., Chapman, J.: Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In: Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. pp. 51–67 (2014). https://doi.org/10.4204/EPTCS.153.4
2. Altenkirch, T., Danielsson, N.A., Kraus, N.: Partiality, revisited - the partiality monad as a quotient inductive-inductive type. In: Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 534–549 (2017). https://doi.org/10.1007/978-3-662-54458-7_31
3. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. J. Funct. Program. **13**(2), 261–293 (2003). https://doi.org/10.1017/S0956796802004501
4. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development **17**(6), 525–532 (1973)
5. Benton, N., Kennedy, A., Varming, C.: Some domain theory and denotational semantics in coq. In: Theorem Proving in Higher Order Logics, 22nd International

Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. pp. 115–130 (2009). https://doi.org/10.1007/978-3-642-03359-9_10
6. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science **1**(2) (2005). https://doi.org/10.2168/LMCS-1(2:1)2005
7. Carette, J., Sabry, A.: Computing with semirings and weak rig groupoids. In: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. pp. 123–148 (2016). https://doi.org/10.1007/978-3-662-49498-1_6
8. Chapman, J., Uustalu, T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. Mathematical Structures in Computer Science **29**(1), 67–92 (2019). https://doi.org/10.1017/S0960129517000184
9. Cockett, J.R.B., Lack, S.: Restriction categories I: Categories of partial maps. Theoretical Computer Science **270**(1–2), 223–259 (2002)
10. Cockett, J.R.B., Lack, S.: Restriction categories III: colimits, partial limits and extensivity. Mathematical Structures in Computer Science **17**(4), 775–817 (2007). https://doi.org/10.1017/S0960129507006056
11. Danielsson, N.A.: Operational semantics using the partiality monad. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012. pp. 127–138 (2012). https://doi.org/10.1145/2364527.2364546
12. Danielsson, N.A.: Up-to techniques using sized types. PACMPL **2**(POPL), 43:1–43:28 (2018). https://doi.org/10.1145/3158131
13. Escardó, M.H., Knapp, C.M.: Partial elements and recursion via dominances in univalent type theory. In: 26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden. pp. 21:1–21:16 (2017). https://doi.org/10.4230/LIPIcs.CSL.2017.21
14. Ésik, Z., Goncharov, S.: Some remarks on conway and iteration theories. CoRR **abs/1603.00838** (2016), http://arxiv.org/abs/1603.00838
15. Giles, B.: An investigation of some theoretical aspects of reversible computing. Ph.D. thesis, University of Calgary (2014)
16. Goncharov, S., Milius, S., Rauch, C.: Complete elgot monads and coalgebraic resumptions. Electr. Notes Theor. Comput. Sci. **325**, 147–168 (2016). https://doi.org/10.1016/j.entcs.2016.09.036
17. Goncharov, S., Schröder, L., Rauch, C., Jakob, J.: Unguarded recursion on coinductive resumptions. Logical Methods in Computer Science **14**(3) (2018). https://doi.org/10.23638/LMCS-14(3:10)2018
18. Hasegawa, M.: Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In: Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings. pp. 196–213 (1997). https://doi.org/10.1007/3-540-62688-3_37
19. Hofmann, M.: Extensional constructs in intensional type theory. CPHC/BCS distinguished dissertations, Springer (1997)
20. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: A typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation, 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11106, pp. 304–321. Springer (2018)

21. James, R.P., Sabry, A.: Information effects. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 73–84 (2012). https://doi.org/10.1145/2103656.2103667
22. James, R.P., Sabry, A.: Theseus: A high level language for reversible computing (2014), work-in-progress report at RC 2014, available at https://www.cs.indiana.edu/ sabry/papers/theseus.pdf.
23. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society **119**(3), 447–468 (1996). https://doi.org/10.1017/S0305004100074338
24. Kaarsgaard, R., Glück, R.: A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. Logical Methods in Computer Science **14**(3), 1–38 (2018)
25. Kaarsgaard, R., Axelsen, H.B., Glück, R.: Join inverse categories and reversible recursion. J. Log. Algebr. Meth. Program. **87**, 33–50 (2017). https://doi.org/10.1016/j.jlamp.2016.08.003
26. Karvonen, M.: The Way of the Dagger. Ph.D. thesis, School of Informatics, University of Edinburgh (2019)
27. Kastl, J.: Inverse categories. In: Hoehnke, H.J. (ed.) Algebraische Modelle, Kategorien und Gruppoide, Studien zur Algebra und ihre Anwendungen, vol. 7, pp. 51–60. Akademie-Verlag (1979)
28. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 261–269 (1961)
29. Laplaza, M.L.: Coherence for distributivity. In: Kelly, G.M., Laplaza, M., Lewis, G., Mac Lane, S. (eds.) Coherence in Categories. pp. 29–65. Springer Berlin Heidelberg (1972)
30. Laursen, J.S., Ellekilde, L.P., Schultz, U.P.: Modelling reversible execution of robotic assembly. Robotica **36**(5), 625–654 (2018)
31. Norell, U.: Dependently Typed Programming in Agda. In: Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009. pp. 1–2 (2009)
32. Rendel, T., Ostermann, K.: Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. ACM SIGPLAN Notices **45**(11), 1–12 (2010)
33. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.B. (eds.) RC 2015. Lecture Notes in Computer Science, vol. 9138, pp. 95–110. Springer (2015)
34. Schultz, U.P.: Reversible object-oriented programming with region-based memory management. In: Kari, J., Ulidowski, I. (eds.) Proceedings of the 10th International Conference on Reversible Computation (RC 2018). Lecture Notes in Computer Science, vol. 11106, pp. 322–328. Springer-Verlag (2018)
35. Thomsen, M.K., Axelsen, H.B., Glück, R.: A reversible processor architecture and its reversible logic design. In: De Vos, A., Wille, R. (eds.) Proceedings of the Third International Conference on Reversible Computation. Lecture Notes in Computer Science, vol. 7165, pp. 30–42. Springer-Verlag (2012)
36. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
37. Uustalu, T., Veltri, N.: The delay monad and restriction categories. In: Theoretical Aspects of Computing - ICTAC 2017 - 14th International Collo-

quium, Hanoi, Vietnam, October 23-27, 2017, Proceedings. pp. 32–50 (2017). https://doi.org/10.1007/978-3-319-67729-3_3
38. Uustalu, T., Veltri, N.: Partiality and container monads. In: Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings. pp. 406–425 (2017). https://doi.org/10.1007/978-3-319-71237-6_20
39. Veltri, N.: A Type-Theoretical Study of Nontermination. Ph.D. thesis, Tallinn University of Technology (2017), https://digi.lib.ttu.ee/i/?7631
40. de Vos, A.: Reversible Computing: Fundamentals, Quantum Computing, and Applications. Wiley-VCH Verlag (2010)
41. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. Proceedings. pp. 144–153. ACM (2007)