Up-to Techniques Using Sized Types

NILS ANDERS DANIELSSON, University of Gothenburg

Up-to techniques are used to make it easier—or feasible—to construct, for instance, proofs of bisimilarity. This text shows how many up-to techniques can be framed as *size-preserving functions*, using sized types to keep track of sizes. Through a number of examples it is argued that this approach to up-to techniques is convenient to use in practice.

On the more theoretical side a class of up-to techniques intended to capture a natural mode of use of such size-preserving functions is defined. This class turns out to correspond closely to "functions below the companion", a notion recently introduced by Pous.

Additional Key Words and Phrases: coinduction, up-to techniques, sized types, companion

1 INTRODUCTION

Coinductively defined types or sets can be used to represent values or proofs that are potentially infinite. In a constructive type theory the basic method for defining values in a coinductively defined type is often some notion of *guarded corecursion*. This method ensures that the definitions are productive by requiring that every corecursive call is guarded by constructors. However, guarded corecursion can be cumbersome to use in practice: if every corecursive call has to be guarded by constructors, then it is hard to write code in a modular way.

Some type theories support a more flexible variant of corecursion based on *sized types* [Hughes et al. 1996; Amadio and Coupet-Grimal 1998; Giménez 1998; Xi 2002; Blanqui 2004, 2005; Barthe et al. 2004, 2006; Grégoire and Sacchini 2010; Abel 2012; Sacchini 2013, 2014; Abel and Pientka 2016; Abel et al. 2017]. Sized types tend to make the type theory more complicated, but my experience—based on using what is perhaps the most mature implementation of type theory with sized types, Agda [Agda Team 2017]—is that they make it much easier to write corecursive programs. The goal of this text is to show how easy it is to use *up-to techniques* in a setting with sized types, and to compare with the state of the art in up-to techniques.

Up-to techniques are often used to simplify (or enable) proofs of bisimilarity, but are more general than that. Pous and Sangiorgi [2011] provide an overview of the state of the art in up-to techniques up to roughly 2011, but there has been plenty of work in this field in recent years [Hur et al. 2013; Parrow and Weber 2016; Pous 2016; Pous and Rot 2017; Schäfer and Smolka 2017; Basold et al. 2017].

The connection between sized types and up-to techniques was pointed out by Danielsson [2012], who observed in a side remark that some uses of size-preserving proofs are similar to Sangiorgi and Milner's "expansions up to $\precsim$" [1992]. This work puts this observation on more solid ground:

- A class of up-to techniques—*size-preserving functions*—intended to capture a natural mode of use of sized types is introduced (see Section 6).
- This class turns out to be closely related to an important class of up-to techniques introduced by Pous [2016]: functions that are below the *companion*, the greatest compatible function (see Section 7). This can perhaps provide some explanation of why sized types, at least in my opinion, work so well in practice.

Through a number of examples in Sections 4, 5, 8 and 9 I also try to convey a less technical message: when sized types are used for up-to techniques things tend to work rather straightforwardly, without need for heavy technical machinery (ignoring the machinery hidden inside, say, the proof assistant that supports the use of sized types). For instance, Section 8 contains a proof showing that the CCS replication operator preserves strong bisimilarity. Pous [2016] states that replication "is quite challenging as far as up-to techniques are concerned" (and presents a solution). The proof presented below is size-preserving, which implies that it can easily be combined with other size-preserving proofs. Furthermore the proof is rather straightforward: a complete formal proof (depending on general lemmas) is presented in Figure 3.

1.1 Formalisation

All the main results and examples in this paper have been formalised using Agda,[1] and the code is available to inspect. Given that Agda is perhaps the only fairly mature proof assistant with support for sized types, and that I want it to be easy for readers to start using sized types in practice, I will also use Agda code in the text below. There are differences between the accompanying code and the presentation below, but they are minor. For instance, the accompanying code is more general, because it uses much more universe polymorphism. It also includes a number of examples and results that are not discussed in the text below.

2 SIZED TYPES

Let us begin with a short introduction to sized types. Sized types can be used both for inductive and coinductive types, but here I will only use them for the latter kind.

The delay monad, representing possibly non-terminating computations of values of type A, is defined as the greatest fixpoint $\nu X.A + X$ [Capretta 2005]. This type can be defined in the following somewhat verbose way in Agda, using sized types:

```
mutual

  data Delay (A : Set) (i : Size) : Set where
    now  : A                → Delay A i
    later : Delay′ A i → Delay A i

  record Delay′ (A : Set) (i : Size) : Set where
    coinductive
    field force : {j : Size< i} → Delay A j
```

(Here *Set* is the type of small types.) The application now x stands for a computation that terminates immediately with the value x, and later x stands for a computation with at least one step left to run.

The sizes are used to aid the termination checker: one cannot match on them. They can be thought of as ordinals, and $j : \mathit{Size}< i$ means that j is a size that is strictly smaller than i (with an exception for ∞, see below). *Delay′* can be thought of as being defined by *deflationary iteration* [Abel 2012],

$$\mathit{Delay}'\ A\ i = \bigcap_{j<i} D\ (\mathit{Delay}'\ A\ j),$$

[1] With the K rule turned off. Sometimes it is assumed that equality of functions is extensional.

where D takes X to $A + X$ (ignoring the constructors now and later). A value of type $Delay\ A\ i$ should be seen as a partially defined value of size i. A fully defined value has size ∞, where ∞ corresponds to a closure ordinal for which $Delay'\ A\ \infty$ and $D\ (Delay'\ A\ \infty)$ are isomorphic. Note that $i : Size< \infty$ holds for all sizes i, including ∞.

The Agda implementation also includes a notion of subtyping. If $j : Size< i$, then values of type $Delay'\ A\ i$ can be used where values of type $Delay'\ A\ j$ are expected.

We can define values in the delay monad corecursively, using *copatterns* [Abel et al. 2013]. A basic copattern specifies how a record value computes when a certain projection function, corresponding to a record field, is applied to it. Here is a definition of the non-terminating computation *never*:

```
mutual

  never : ∀ {A i} → Delay A i
  never = later never′

  never′ : ∀ {A i} → Delay′ A i
  force never′ = never
```

The expression *never* unfolds directly to later $never'$, but $never'$ is a value that only unfolds if the projection force is applied to it.

Notation like $\{j : Size< i\} \to ...$ or $\forall\ \{A\ i\} \to ...$ is used to mark function arguments as being *implicit*, which means that they do not need to be given explicitly as long as Agda can infer them. We could also have given the sizes explicitly in the definition of *never*:

```
mutual

  never : ∀ {A i} → Delay A i
  never {i = i} = later (never′ {i = i})

  never′ : ∀ {A i} → Delay′ A i
  force (never′ {i = i}) {j = j} = never {i = j}
```

Note that the corecursive call to *never* uses a size j that is strictly smaller than i (if we ignore ∞). If every cycle in the call graph, like $never' \to never \to never'$, involves a strict decrease in some size, then Agda's termination checker accepts the code.

As an example of a coinductively defined relation, let us define strong bisimilarity—basically equality—for the delay monad. This is a relation between fully defined delay monad values. Two such values are strongly bisimilar if they are either both now x for some value x, or if they both have an outer later constructor and the corresponding arguments, once forced, are strongly bisimilar. In order to ensure that, say, *never* is strongly bisimilar to itself, this explanation should be read coinductively:

```
mutual

  data [_]_∼_ {A} (i : Size) : (x y : Delay A ∞) → Set where
    now   : ∀ {x}   →                          [ i ] now x ∼ now x
    later : ∀ {x y} → [ i ] force x ∼′ force y → [ i ] later x ∼ later y

  record [_]_∼′_ {A} (i : Size) (x y : Delay A ∞) : Set where
    coinductive
    field force : {j : Size< i} → [ j ] x ∼ y
```

Note that the now and later constructors, as well as the force destructor, have been overloaded. One of the properties satisfied by this relation is transitivity. This property can be proved corecursively:

$$
\begin{aligned}
&\mathit{transitive}^s : \forall\ \{i\ A\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \rightarrow \\
&\qquad [\ i\ ]\ x \sim_D y \rightarrow [\ i\ ]\ y \sim_D z \rightarrow [\ i\ ]\ x \sim_D z \\
&\mathit{transitive}^s\ \mathsf{now}\ \ \mathsf{now}\ \ = \mathsf{now} \\
&\mathit{transitive}^s\ (\mathsf{later}\ p)\ (\mathsf{later}\ q) = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \rightarrow \mathit{transitive}^s\ (\mathsf{force}\ p)\ (\mathsf{force}\ q)\ \}
\end{aligned}
$$

Instead of using an auxiliary definition (like never') this proof uses a local copattern: $\lambda\ \{\ .\mathsf{force} \rightarrow \ldots\ \}$.

Note that the transitivity proof above is *size-preserving*: if the argument proofs both have size i, then the resulting proof also has this size. This means that the proof is well-suited for use inside other corecursively defined proofs.

For more details about sized types and deflationary iteration, including normalisation proofs (in one case sketched), see Abel and Pientka [2016] and Sacchini [2015]. Note that Abel and Pientka discuss a language based on System F_ω, and that Sacchini's unpublished draft treats a dependently typed language that differs from Agda. Furthermore the current Agda implementation contains bugs related to sized types. This paper is based on the assumption that meta-theoretic properties like consistency can be proved for a system that is sufficiently close to the language outlined in this section.

3 TRADITIONAL COINDUCTION USING SIZED TYPES

Let us now compare corecursion, as presented above, with the traditional proof technique of coinduction. Coinduction is typically presented in roughly the following way: If we have a monotone function F on some complete lattice, then the Knaster-Tarski theorem implies that there is a greatest post-fixpoint νF, and the proof technique of coinduction states that for any post-fixpoint $X \leq F\ X$ we have $X \leq \nu F$.

We can repeat this argument in Agda for strictly positive functors F, as captured by *indexed containers* [Altenkirch et al. 2015]. For an index type X there is a type of containers $\mathit{Container}\ X$, which comes with an interpretation function

$$[\![\_]\!] : \forall\ \{X\} \rightarrow \mathit{Container}\ X \rightarrow (X \rightarrow \mathit{Set}) \rightarrow (X \rightarrow \mathit{Set})$$

and a map function:

$$
\begin{aligned}
&\mathit{map} : \forall\ \{X\}\ (C : \mathit{Container}\ X)\ \{A\ B\} \rightarrow \\
&\qquad A \subseteq B \rightarrow [\![\ C\ ]\!]\ A \subseteq [\![\ C\ ]\!]\ B
\end{aligned}
$$

Here $\_\subseteq\_$ is a type of index-preserving functions (defined universe-polymorphically so that it can be used with a large definition in Section 7):

$$
\begin{aligned}
&\_\subseteq\_ : \forall\ \{a\ b\}\ \{X : \mathit{Set}\} \rightarrow (X \rightarrow \mathit{Set}\ a) \rightarrow (X \rightarrow \mathit{Set}\ b) \rightarrow \mathit{Set}\ (a \sqcup b) \\
&R \subseteq S = \forall\ \{x\} \rightarrow R\ x \rightarrow S\ x
\end{aligned}
$$

Note that the map function shows that $[\![\ C\ ]\!]$ is monotone with respect to $\_\subseteq\_$.

The exact details of how indexed containers are defined are not very relevant to this paper, so they are not presented here, but for interested readers there is a brief summary in Appendix A.

Given an indexed container one can define its greatest fixpoint coinductively. The omitted details of the definition of indexed containers ensure that this definition is strictly positive:

```
mutual

  ν : ∀ {X} → Container X → Size → (X → Set)
  ν C i = ⟦ C ⟧ (ν′ C i)

  record ν′ {X} (C : Container X) (i : Size) (x : X) : Set where
    coinductive
    field force : {j : Size< i} → ν C j x
```

The following lemma implies that $\nu\ C\ \infty$ is a post-fixpoint of $[\![\ C\ ]\!]$:

```
ν-out : ∀ {X i} (C : Container X) {j : Size< i} →
        ν C i ⊆ ⟦ C ⟧ (ν C j)
ν-out C = map C (λ x → force x)
```

Furthermore every instance $\nu\ C\ i$ is a pre-fixpoint:

```
ν-in : ∀ {X i} (C : Container X) →
       ⟦ C ⟧ (ν C i) ⊆ ν C i
ν-in C = map C (λ x → λ { .force → x })
```

Finally we can show that $\nu\ C\ i$ is greater than every post-fixpoint:

```
unfold : ∀ {X A i} (C : Container X) →
         A ⊆ ⟦ C ⟧ A → A ⊆ ν C i
unfold C f = map C (λ a → λ { .force → unfold C f a }) ∘ f
```

Note that this corecursive proof corresponds to the technique traditionally called coinduction. However, it is not phrased for an arbitrary complete lattice. In fact, functions of type $X \to Set$, ordered by $\_\subseteq\_$, do not even form a poset (if X is inhabited), because $\_\subseteq\_$ is not antisymmetric.

4 LABELLED TRANSITION SYSTEMS

Let us now define labelled transition systems, using ν to define strong and weak bisimilarity.

A labelled transition system (LTS) consists of a type of processes Proc, a type of labels *Label*, and a transition relation $\_[\_]\!\to\_$, relating two processes with a label. I have also chosen to include a function silent that decides whether a label should be counted as silent or not:

```
record LTS : Set₁ where
  field
    Proc     : Set
    Label    : Set
    _[_]→_ : Proc → Label → Proc → Set
    silent   : Label → Bool
```

(Here Set_1 is a type of large types.)

The traditional definition of bisimilarity states that two processes are bisimilar if there is some bisimulation that relates them. A binary relation R on processes is a bisimulation if, whenever P and Q are related by R and P can make a μ-transition to P', then Q can make a μ-transition to some process Q' such that P' and Q' are related by R (and symmetrically, starting with a transition from Q):

These diagrams are captured by $R \subseteq B\ R$, where B is the following record type:

```
record B (R : Proc × Proc → Set) (PQ : Proc × Proc) : Set where
  private
    P = proj₁ PQ
    Q = proj₂ PQ

  field
    left-to-right : ∀ {μ P′} → P [ μ ]→ P′ → ∃ λ Q′ → Q [ μ ]→ Q′ × R (P′ , Q′)
    right-to-left : ∀ {μ Q′} → Q [ μ ]→ Q′ → ∃ λ P′ → P [ μ ]→ P′ × R (P′ , Q′)
```

(If B has type $A \to \mathit{Set}$, then the type $\exists\ B$ consists of dependent pairs $(x\ ,\ y)$, where x has type A and y has type $B\ x$. The type $A \times B$ is the non-dependent product of A and B.) This type can also be defined as a container (details omitted):

```
B : Container (Proc × Proc)
```

There are now at least two ways to define bisimilarity. One is the traditional definition:

```
Bisimilar : Proc → Proc → Set₁
Bisimilar P Q = ∃ λ R → (R ⊆ ⟦ B ⟧ R) × R (P , Q)
```

Note that this definition is large. (Agda is predicative.) An alternative is to use ν:

```
[_]_∼_ : Size → Proc → Proc → Set
[ i ] P ∼ Q = ν B i (P , Q)
```

This definition is small, and it is easy to show that $\mathit{Bisimilar}\ P\ Q$ and $[\ \infty\ ]\ P \sim Q$ are logically equivalent. Let us also introduce the following definition, which uses the primed variant of ν:

```
[_]_∼′_ : Size → Proc → Proc → Set
[ i ] P ∼′ Q = ν′ B i (P , Q)
```

One can define weak bisimilarity in a similar way, by first defining what a weak transition is, and then adapting the container B. See the accompanying code for details.

Let us now see how the delay monad can be turned into a labelled transition system. The type of processes is $\mathit{Delay}\ A\ \infty$, and the type of labels is $\mathit{Maybe}\ A$: either nothing (which is treated as silent), or just x (which is treated as non-silent). The transition relation is defined in the following way:

```
data _[_]→_ {A} : Delay A ∞ → Maybe A → Delay A ∞ → Set where
  now   : ∀ {x} → now x  [ just x  ]→ now  x
  later : ∀ {x} → later x [ nothing ]→ force x
```

The computation now x makes a non-silent transition to itself, with just x as the label, and later x makes a silent transition to force x. The definition of bisimilarity obtained from

```
transitive : ∀ {i P Q R} → [ i ] P ∼ Q → [ i ] Q ∼ R → [ i ] P ∼ R
transitive = λ p q → record
  { left-to-right = lr p q
  ; right-to-left = map₃ symmetric′ ∘ lr (symmetric q) (symmetric p)
  }
  where
  lr : ∀ {i P P′ Q R μ} →
       [ i ] P ∼ Q → [ i ] Q ∼ R → P [ μ ]→ P′ →
       ∃ λ R′ → R [ μ ]→ R′ × [ i ] P′ ∼′ R′
  lr p q tr = let (Q′ , tr′ , p′) = left-to-right p tr
                  (R′ , tr″ , q′) = left-to-right q tr′
              in  (R′ , tr″ , λ { .force → transitive (force p′) (force q′) })
```

Fig. 1. A size-preserving transitivity proof for strong bisimilarity.

this LTS is pointwise logically equivalent to the definition of strong bisimilarity given in Section 2 ($[\_]\_\sim_D\_$):

$$\forall\ \{A\ i\}\ \{x\ y : \mathit{Delay}\ A\ \infty\} \to [\ i\ ]\ x \sim_D y \Leftrightarrow [\ i\ ]\ x \sim y$$

Note that this proof is size-preserving in both directions. Both directions are defined using corecursion; see the accompanying code for details.

In Section 2 a size-preserving transitivity proof was given for strong bisimilarity for the delay monad. Now we can prove that there is a size-preserving transitivity proof for (strong) bisimilarity for any LTS.

First note that symmetry is size-preserving. This can be proved by corecursively replacing left-to-right with right-to-left, and vice versa:

```
mutual

  symmetric : ∀ {i P Q} → [ i ] P ∼ Q → [ i ] Q ∼ P
  left-to-right (symmetric p) = map₃ symmetric′ ∘ right-to-left p
  right-to-left (symmetric p) = map₃ symmetric′ ∘ left-to-right p

  symmetric′ : ∀ {i P Q} → [ i ] P ∼′ Q → [ i ] Q ∼′ P
  force (symmetric′ p) = symmetric (force p)
```

The proof above uses map_3, which maps a function over the third component of a triple. Here and below I pretend that bisimilarity is defined using the record type B, rather than the corresponding container, to avoid noise related to the coding.

Now we can prove that transitivity is size-preserving, using the size-preserving symmetry proof to avoid having to spell out the argument for both left-to-right and right-to-left, see Figure 1. The following section includes an example illustrating how one can make use of the fact that transitivity is size-preserving.

5 CCS

This section contains a description of a CCS-like LTS, included in order to support some more complicated examples. The presentation is based on the one due to Pous and Sangiorgi [2011], with the difference that processes are allowed to be infinite.

Let us assume that there is a type $Name$ of names. These names are combined with booleans; (a , true) stands for "a" and (a , false) stands for "$\overline{a}$":

```
Name-with-kind : Set
Name-with-kind = Name × Bool
```

Given a name of one kind we can turn it into a name of the other kind:

```
co : Name-with-kind → Name-with-kind
co (a , kind) = (a , not kind)
```

The labels, or actions, are either names (with kinds) or the silent action τ:

```
data Action : Set where
  name : Name-with-kind → Action
  τ    : Action

silent : Action → Bool
silent (name _) = false
silent τ        = true
```

The following process constructors are included: an inactive process ($\emptyset$), parallel composition ($\_|\_$), sum ($\_\oplus\_$), prefixing ($\_\cdot\_$), restriction ($\langle\nu\_\rangle$), and replication (!):

```
mutual

  data Proc (i : Size) : Set where
    ∅          : Proc i
    _|_ _⊕_    : Proc i → Proc i → Proc i
    _·_        : Action → Proc′ i → Proc i
    ⟨ν_⟩       : Name → Proc i → Proc i
    !          : Proc i → Proc i

  record Proc′ (i : Size) : Set where
    coinductive
    field force : {j : Size< i} → Proc j
```

This definition uses a mixture of induction and coinduction. It should be read as an inductive definition nested inside a coinductive one. Note that all constructors are "inductive", with the exception of prefixing: infinite processes can only be constructed if cycles in the call graph are broken by prefixes.

The transition relation is defined in the following, hopefully unsurprising way:

```
data _[_]→_ : Proc ∞ → Action → Proc ∞ → Set where
  par-left    : ∀ {P Q P′ μ} → P [ μ ]→ P′ → P | Q [ μ ]→ P′ | Q
  par-right   : ∀ {P Q Q′ μ} → Q [ μ ]→ Q′ → P | Q [ μ ]→ P  | Q′
  par-τ       : ∀ {P P′ Q Q′ a} →
                P [ name a ]→ P′ → Q [ name (co a) ]→ Q′ → P | Q [ τ ]→ P′ | Q′
  sum-left    : ∀ {P Q P′ μ} → P [ μ ]→ P′ → P ⊕ Q [ μ ]→ P′
  sum-right   : ∀ {P Q Q′ μ} → Q [ μ ]→ Q′ → P ⊕ Q [ μ ]→ Q′
  action      : ∀ {P μ} → μ · P [ μ ]→ force P
  restriction : ∀ {P P′ a μ} → a ∉ μ → P [ μ ]→ P′ → ⟨ν a ⟩ P [ μ ]→ ⟨ν a ⟩ P′
  replication : ∀ {P P′ μ} → ! P | P [ μ ]→ P′ → ! P [ μ ]→ P′
```

The rule for restriction includes the requirement that a must not be the underlying name of μ:

```
_∉_ : Name → Action → Set
a ∉ name (b , _) = ¬ a ≡ b
a ∉ τ            = ⊤
```

Here $\top$ is the unit type, and $\neg\ A$ is defined as the type of functions from A to the empty type, $\bot$.

Let us now see how one can prove properties about CCS processes. We start with a very simple lemma that states that processes of the following form are strongly bisimilar to $\emptyset$:

```
Restricted : Name → Proc ∞
Restricted a = ⟨ν a⟩ (name (a , true) · λ { .force → ∅ })
```

This is easy to see: due to the restriction the process cannot make any transitions. The proof can be performed using simple case analysis:

```
Restricted∼∅ : ∀ {a} → [ ∞ ] Restricted a ∼ ∅
left-to-right Restricted∼∅ (restriction x≢x action) = ⊥-elim (x≢x refl)
right-to-left Restricted∼∅ ()
```

Here () is an "impossible pattern", marking that there is no constructor that has the right type. The function $\bot$-*elim* maps a value in the empty type to an arbitrary type, and refl is the identity type's reflexivity constructor.

As a slightly more involved example, let us now prove that $\emptyset$ is a left identity for parallel composition. This proof is corecursive, using a simple case analysis in the left-to-right direction:

```
∅| : ∀ {i P} → [ i ] ∅ | P ∼ P
left-to-right ∅| (par-right tr) = _ , tr , λ { .force → ∅| }
left-to-right ∅| (par-left ())
left-to-right ∅| (par-τ () _)
right-to-left ∅| tr             = _ , par-right tr , λ { .force → ∅| }
```

It is also straightforward to prove that all the process constructors preserve strong bisimilarity. The most complicated proof is the one for replication. This proof can be found in Section 8. The second most complicated proof is perhaps the one for parallel composition. Here symmetry is again used to avoid repeating the same argument twice, and otherwise the proof is entirely straightforward, see Figure 2. Note that this proof is size-preserving. All the other preservation proofs are also size-preserving. This means that they can be safely combined when used in corecursive proofs. In fact, for _·_ we get a stronger result, where the argument's type uses the primed variant of bisimilarity:

```
·-cong : ∀ {i μ P Q} → [ i ] force P ∼′ force Q → [ i ] μ · P ∼ μ · Q
·-cong p = record
  { left-to-right = λ { action → _ , action , p }
  ; right-to-left = λ { action → _ , action , p }
  }
```

```
_|-cong_ : ∀ {i P P′ Q Q′} →
           [ i ] P ∼ Q → [ i ] P′ ∼ Q′ → [ i ] P | P′ ∼ Q | Q′
_|-cong_ = λ p q → record
  { left-to-right = lr p q
  ; right-to-left = map₃ symmetric′ ∘ lr (symmetric p) (symmetric q)
  }
  where
  lr : ∀ {i P P′ P″ Q Q′ μ} →
       [ i ] P ∼ Q → [ i ] P′ ∼ Q′ → P | P′ [ μ ]→ P″ →
       ∃ λ Q″ → Q | Q′ [ μ ]→ Q″ × [ i ] P″ ∼′ Q″
  lr p q (par-left tr)     = let (_ , tr′          , p′) = left-to-right p tr
                             in  (_ , par-left tr′  , λ { .force → force p′ |-cong q })
  lr p q (par-right tr)    = let (_ , tr′          , q′) = left-to-right q tr
                             in  (_ , par-right tr′ , λ { .force → p |-cong force q′ })
  lr p q (par-τ tr₁ tr₂) =
    let (_ , tr′₁           , p′) = left-to-right p tr₁
        (_ , tr′₂           , q′) = left-to-right q tr₂
    in  (_ , par-τ tr′₁ tr′₂ , λ { .force → force p′ |-cong force q′ })
```

Fig. 2. Parallel composition preserves strong bisimilarity in a size-preserving way.

The type of ·-*cong* makes it easy to construct certain cyclic bisimilarity proofs corecursively. Consider the following two cyclic processes, defined using a name a and an action μ:

```
P : ∀ {i} → Proc i
P = Restricted a | (μ · λ { .force → P })

Q : ∀ {i} → Proc i
Q = μ · λ { .force → Q }
```

Using the combinators introduced above it is easy to prove that these processes are strongly bisimilar:

```
P∼Q : ∀ {i} → [ i ] P ∼ Q
P∼Q = transitive (Restricted∼∅ |-cong (·-cong λ { .force → P∼Q })) ∅|
```

In order to make proofs like this easier to read I make use of equational reasoning combinators [Norell 2007]:

```
_■       : ∀ {i} P → [ i ] P ∼ P
_∼⟨_⟩_ : ∀ {i} P {Q R} → [ i ] P ∼ Q → [ i ] Q ∼ R → [ i ] P ∼ R
```

We get the following proof instead:

```
P∼Q : ∀ {i} → [ i ] P ∼ Q
P∼Q = P      ∼⟨ Restricted∼∅ |-cong (·-cong λ { .force → P∼Q }) ⟩
      ∅ | Q  ∼⟨ ∅| ⟩
      Q      ■
```

There are several observations to make about this proof. First, note that if transitivity had not been size-preserving, and the first argument had instead been required to be fully defined (have size ∞), then the proof would not have worked. Similarly, if *_|-cong_* had required its second argument to be fully defined, or *·-cong* had required its argument to be fully defined, then the proof would have been rejected by the type-checker.

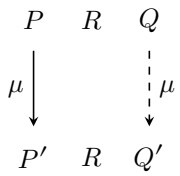
Secondly, if the argument of *·-cong* had not been of the primed variant, then the corecursive call to $P \sim Q$ would not have taken place under a copattern, the definition would not have been strongly normalising, and the code would yet again have been rejected. This aspect of *·-cong* is related to the fact that, for (at least some forms of) CCS, equations of the form $[\ \infty\ ]\ P \sim (C\ [\ P\ ])$ have unique solutions for contexts C where every hole is under a prefix [Milner 1988].

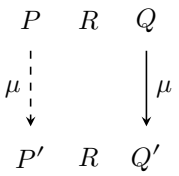
Finally, those who are familiar with up-to techniques may recognise a connection between the use of transitivity and the "up to bisimilarity" technique. Similarly, one may notice a connection between the use of the preservation lemmas and to the "up to context" technique.

6 UP-TO TECHNIQUES AND SIZE-PRESERVING FUNCTIONS

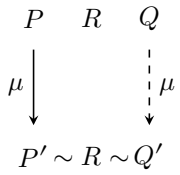
Let us now make the connection between size-preserving functions and up-to techniques more precise.

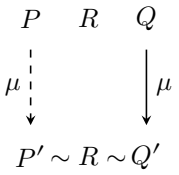
Up-to techniques are often motivated in roughly the following way: One can prove that $X \leq \nu B$ by finding some R such that $X \leq R \leq B\ R$. For instance, one can prove that two processes are bisimilar by finding some bisimulation R that relates them:





However, constructing such an R may be complicated, and it may be easier to construct some R satisfying $X \leq R \leq B\ (F\ R)$ for some function F. For instance, it may be easier to find a bisimulation up to bisimilarity:





A monotone function F is called an up-to technique for B if, for any R, if $R \leq B\ (F\ R)$, then $R \leq \nu B$. In the setting used in this paper I choose to use the following definition, where I have dropped the requirement that F should be monotone:

$$\begin{aligned}&\mathit{Up\text{-}to} : \mathit{Trans} \to \mathit{Set}_1\\ &\mathit{Up\text{-}to}\ F = \forall\ \{R\} \to R \subseteq [\![\ C\ ]\!]\ (F\ R) \to R \subseteq \nu\ C\ \infty\end{aligned}$$

Here I have fixed an index type X and a container $C : \mathit{Container}\ X$. The abbreviation *Trans*, standing for "relation transformer", is defined in the following way:

$$\begin{aligned}&\mathit{Trans} : \mathit{Set}_1\\ &\mathit{Trans} = (X \to \mathit{Set}) \to (X \to \mathit{Set})\end{aligned}$$

The example at the end of the previous section suggests that, when sized types are used, a certain class of up-to techniques is basically built-in. I have tried to capture this observation as a class of relation transformers:

```
Size-preserving : Trans → Set₁
Size-preserving F = ∀ {R i} → R ⊆ ν C i → F R ⊆ ν C i
```

The idea is that if we are in the process of defining an element in a certain stage $\nu\ C\ i$ of the greatest fixpoint of C, and we have constructed some family R that is below this stage, then it is safe to apply F to R, because $F\ R$ is also below this stage.

Let us see how the size-preserving transitivity proof fits into this framework. First note that composition of "binary relations" can be defined in the following way:

```
_⊙_ : {X : Set} → (X × X → Set) → (X × X → Set) → (X × X → Set)
(R ⊙ S) (x , z) = ∃ λ y → R (x , y) × S (y , z)
```

The technique of "up to bisimilarity" can then be defined as follows (for an arbitrary LTS):

```
Up-to-bisimilarity : (Proc × Proc → Set) → (Proc × Proc → Set)
Up-to-bisimilarity R = ν B ∞ ⊙ R ⊙ ν B ∞
```

This technique is size-preserving:

```
up-to-bisimilarity-size-preserving : Size-preserving Up-to-bisimilarity
up-to-bisimilarity-size-preserving
  R⊆νBi {P₁ , P₄} (P₂ , P₁∼P₂ , P₃ , P₂RP₃ , P₃∼P₄) =
    P₁  ∼⟨ P₁∼P₂ ⟩
    P₂  ∼⟨ R⊆νBi P₂RP₃ ⟩
    P₃  ∼⟨ P₃∼P₄ ⟩
    P₄  ■
```

Note the three uses of transitivity. The proofs $P_1{\sim}P_2$ and $P_3{\sim}P_4$ have size ∞, and $R{\subseteq}\nu Bi\ P_2RP_3$ has size i, for some i. The resulting expression has size i. (Subtyping is at play in this code.)

Similarly one can define "up to context" as a transformer, and use the size-preserving preservation lemmas mentioned in Section 5 to show that it is size-preserving (see the accompanying code). Let us do this, but only for a very simple context consisting of replication applied to a single hole:

```
Up-to-! : (Proc ∞ × Proc ∞ → Set) → (Proc ∞ × Proc ∞ → Set)
Up-to-! R (P , Q) = ∃ λ P′ → ∃ λ Q′ → P ≡ ! P′ × R (P′ , Q′) × ! Q′ ≡ Q
```

In Section 8 a combinator !-*cong* showing that ! preserves strong bisimilarity in a size-preserving way is defined. This combinator can be used to show that *Up-to-*! is size-preserving:

```
up-to-!-size-preserving : Size-preserving Up-to-!
up-to-!-size-preserving R⊆νBi (P′ , Q′ , refl , P′RQ′ , refl) =
  !-cong (R⊆νBi P′RQ′)
```

The most important property of size-preserving transformers is perhaps that they are up-to techniques. In order to prove this, first note that if a transformer is size-preserving, then it satisfies a corresponding property for ν':

```
size-preserving′ : ∀ {F} →
                   Size-preserving F → ∀ {R i} → R ⊆ ν′ C i → F R ⊆ ν′ C i
force (size-preserving′ pres R⊆ν′Ci x) = pres (λ y → force (R⊆ν′Ci y)) x
```

It is then easy to complete the proof:

```
size-preserving⇒up-to : ∀ {F} → Size-preserving F → Up-to F
size-preserving⇒up-to pres {R = R} R⊆CFR = helper
  where
  helper : ∀ {i} → R ⊆ ⟦ C ⟧ (ν′ C i)
  helper =
    map C (size-preserving′ pres (λ x → λ { .force → helper x })) ∘ R⊆CFR
```

Note that $\nu\ C\ i$ is, by definition, equal to $[\![\ C\ ]\!]\ (\nu'\ C\ i)$.

A problem with the full class of up-to techniques, *Up-to*, is that it is not closed under composition: there are up-to techniques (even monotone ones) G and H such that $G \circ H$ is not an up-to technique [Pous and Sangiorgi 2011]. Compositionality can be convenient, because it allows one to construct complicated up-to techniques from simpler building blocks. For this reason several classes of up-to techniques that are closed under composition have been identified [Sangiorgi 1998; Pous and Sangiorgi 2011; Pous 2016]. It may not be very surprising that *Size-preserving* is closed under composition:

```
∘-closure : ∀ {F G} →
            Size-preserving F → Size-preserving G → Size-preserving (F ∘ G)
∘-closure F-pres G-pres = F-pres ∘ G-pres
```

Note that this implies that there are up-to techniques that are not size-preserving.

Practically useful up-to techniques are often monotone:

```
Monotone : Trans → Set₁
Monotone F = ∀ {R S} → R ⊆ S → F R ⊆ F S
```

For monotone F the definition of *Size-preserving* can be simplified. A monotone transformer F is size-preserving if and only if it preserves every approximation of the greatest fixpoint, in the following way:

```
simplification : ∀ {F} →
                 Monotone F → Size-preserving F ⇔ (∀ {i} → F (ν C i) ⊆ ν C i)
simplification mono = record
  { to   = λ pres          → pres id
  ; from = λ pres R⊆νCi → pres ∘ mono R⊆νCi
  }
```

However, note that there are size-preserving transformers that are not monotone, and there are transformers that, despite preserving every approximation of the greatest fixpoint, are not up-to techniques (see the accompanying code).

7 THE COMPANION

After I had noted the simplification above the work of Pous [2016] was pointed out to me. Pous, building on an observation due to Hur et al. [2013], introduces the *companion* of a monotone function B on a complete lattice. A monotone function F is *B-compatible* if $F \circ B \leq B \circ F$ (here the ordering is that of the lattice of monotone functions on the

underlying lattice, ordered pointwise). Pous defines the companion of B as the least upper bound of all B-compatible (and monotone) functions, and notes that this is the greatest B-compatible function.

The companion is an up-to technique. Moreover, every function below the companion is also an up-to technique, and the property of being below the companion is closed under composition. Pous goes on to list further useful properties of the companion.

Pous [2016] notes that the greatest compatible function coincides with the greatest "respectful" function, and Parrow and Weber [2016], working in classical logic, show that the greatest respectful function can be defined by

$$\lambda X.\ \bigsqcap_{X \leq B_\alpha} B_\alpha,$$

where B_α is defined by transfinite recursion on ordinals in the following way: $B_{\alpha+1} = B\ B_\alpha$, and $B_\lambda = \bigsqcap_{\alpha<\lambda} B_\alpha$ if λ is a limit ordinal. In later work Pous and Rot [2017], seemingly also working in classical logic, observe that monotone functions F are below the companion if and only if $F\ B_\alpha \leq B_\alpha$ for all ordinals α.

Note the similarity between the definition of B_α and the semantic explanation of the delay monad in terms of deflationary iteration in Section 2. Note also the similarity between Pous and Rot's characterisation of functions below the companion and the characterisation of monotone, size-preserving functions presented at the end of the previous section.

What happens if we try to port Parrow and Weber's definition of the companion to the setting of this paper, using sizes instead of ordinals? If we replace B_α with $\nu\ C\ i$, then we may end up with something like the following:

```
Companion : Trans
Companion R x = ∀ {i} → R ⊆ ν C i → ν C i x
```

We can then define the class of transformers that are below this variant of the companion:

```
Below-the-companion : Trans → Set₁
Below-the-companion F = ∀ {R} → F R ⊆ Companion R
```

It turns out that a transformer (monotone or not) is below the companion if and only if it is size-preserving. This can be proved by simply reordering some arguments:

```
below-the-companion⇔size-preserving :
  ∀ {F} → Below-the-companion F ⇔ Size-preserving F
below-the-companion⇔size-preserving = record
  { to   = λ below R⊆νCi x → below x R⊆νCi
  ; from = λ pres x R⊆νCi  → pres R⊆νCi x
  }
```

Note that this implies that the companion is size-preserving:

```
companion-size-preserving : Size-preserving Companion
companion-size-preserving = to below-the-companion⇔size-preserving id
```

How does the definition of *Companion* above compare to the definition given by Pous [2016]? Let us first define compatibility of transformers:

```
Compatible : Trans → Set₁
Compatible F = ∀ {R} → F (⟦ C ⟧ R) ⊆ ⟦ C ⟧ (F R)
```

The definition of the companion as the least upper bound of all monotone, compatible functions can then be phrased in the following way:

```
Companion₁ : (X → Set) → (X → Set₁)
Companion₁ R x = ∃ λ (F : Trans) → Monotone F × Compatible F × F R x
```

Note that this definition is large. We can prove that monotone and compatible transformers are size-preserving:

```
compatible⇒size-preserving :
  ∀ {F} → Monotone F → Compatible F → Size-preserving F
compatible⇒size-preserving {F = F} mono comp =
  from (simplification mono) helper
  where
  mutual

    helper : ∀ {i} → F (⟦ C ⟧ (ν′ C i)) ⊆ ⟦ C ⟧ (ν′ C i)
    helper = map C helper′ ∘ comp

    helper′ : ∀ {i} → F (ν′ C i) ⊆ ν′ C i
    force (helper′ x) = helper (mono (λ y → force y) x)
```

It is then easy to prove that the large companion is below the small one:

```
large⊆small : ∀ {R} → Companion₁ R ⊆ Companion R
large⊆small (F , mono , comp , x) =
  from below-the-companion⇔size-preserving
    (compatible⇒size-preserving mono comp) x
```

One can also prove that the small companion is monotone:

```
companion-monotone : Monotone Companion
companion-monotone R⊆S f S⊆νCi = f (S⊆νCi ∘ R⊆S)
```

Furthermore the small companion is compatible if and only if it is below the large one:[2]

```
small-compatible⇔⊆large :
  Compatible Companion ⇔ (∀ {R} → Companion R ⊆ Companion₁ R)
small-compatible⇔⊆large = record
  { to   = λ comp f → (Companion , companion-monotone , comp , f)
  ; from = λ below f →
             let (F , mono , comp , FCR) = below f
             in map C (λ FR {_} → large⊆small (F , mono , comp , FR))
                      (comp FCR)
  }
```

However, I have not managed to prove that the small companion is compatible, nor have I managed to find a counterexample to this property. Note that Parrow and Weber [2016] work in classical logic. Pous and Rot [2017] seemingly also work in classical logic, but Pous (personal communication) suggests that the results can be ported to Coq, using its impredicative features. It may be that *Compatible Companion* is a property that can neither be proved nor disproved in predicative, constructive type theory. (Note also that, due to

[2]The code contains an unnamed, implicit pattern ({_}) because of an Agda quirk.

Girard's paradox [1972], universes that are not at the bottom of Coq's universe hierarchy are predicative.)

Fortunately a number of other properties of the companion, listed by Pous [2016], can be proved for *Companion*. The identity function as well as $[\![C]\!]$ and the companion composed with itself are below the companion:

```
id-below : Below-the-companion id
id-below x f = f x

[[]]-below : Below-the-companion [[ C ]]
[[]]-below x = ν-in C ∘ (λ f → f x) ∘ map C

companion∘companion-below : Below-the-companion (Companion ∘ Companion)
companion∘companion-below =
  from below-the-companion⇔size-preserving
    (∘-closure companion-size-preserving companion-size-preserving)
```

Pous makes use of the fact that his notion of companion is compatible to prove the last two of these properties, but this is not needed for *Companion*. These properties can be used to show that various functions are below the companion. For instance, if F is below the companion, then $[\![C]\!] \circ F$ is below *Companion* $\circ$ *Companion*, which is below the companion.

Pous also makes use of compatibility for the companion to prove that the greatest fixpoint is the companion applied to the lattice's least element. However, compatibility is not needed in the present setting (recall that $\bot$ is the empty type):

```
ν⊆companion-⊥ : ν C ∞ ⊆ Companion (λ _ → ⊥)
ν⊆companion-⊥ x _ = x

companion-⊥⊆ν : Companion (λ _ → ⊥) ⊆ ν C ∞
companion-⊥⊆ν f = f (λ ())
```

Finally one can observe that Pous' use of compatibility for the companion to prove that the companion is an up-to technique can also be avoided in this setting:

```
companion-up-to : Up-to Companion
companion-up-to = size-preserving⇒up-to companion-size-preserving
```

8 REPLICATION PRESERVES STRONG BISIMILARITY

Let us now see how one can prove that the CCS replication operator preserves strong bisimilarity, in a size-preserving way. Pous [2016] states that replication "is quite challenging as far as up-to techniques are concerned: [...] people formalising up-to techniques in proof assistants have eluded this operation so far", and uses Coq to prove that something like the transformer *Up-to-*! introduced in Section 6 is below the companion. When proving this preservation lemma I did not know that it had been seen as difficult, and using sized types it is actually rather straightforward. The accompanying code also contains lemmas showing that the replication operator preserves strong similarity, expansion and weak bisimilarity in a size-preserving way.

It may be interesting to note that the proof presented by Pous uses the fact that, in his setting, the companion is compatible. Needless to say, I do not use this property in my proof.

The proof uses the following lemma [Pous and Sangiorgi 2011, Exercise 6.1.3, part (2)], where it should be noted that replication binds tighter than parallel composition:

$$
\begin{aligned}
&\textit{6-1-3-2} :\\
&\quad \forall\ \{P\ \mu\ R\} \to\\
&\quad !\ P\ [\ \mu\ ]\!\to\ R \to\\
&\quad (\exists\ \lambda\ P' \to P\ [\ \mu\ ]\!\to\ P' \times [\ \infty\ ]\ R \sim\ !\ P \mid P')\\
&\qquad \uplus\\
&\quad (\mu \equiv \tau \times \exists\ \lambda\ P' \to \exists\ \lambda\ P'' \to \exists\ \lambda\ a \to\\
&\qquad P\ [\ \mathsf{name}\ a\ ]\!\to\ P' \times P\ [\ \mathsf{name}\ (\mathit{co}\ a)\ ]\!\to\ P'' \times [\ \infty\ ]\ R \sim (!\ P \mid P') \mid P'')
\end{aligned}
$$

The lemma states that, if $!\ P$ can make a μ-transition to R, then at least one of two properties holds ($\_\uplus\_$ is disjoint sum), and in either case we get to know that P can make one or more transitions, and that R is strongly bisimilar to some process constructed from $!\ P$ and the "reducts" of P.

The proof also uses the following variants of combinators presented above, involving the primed variant of bisimilarity:

$$
\begin{aligned}
&\_\sim'\langle\_\rangle'\_ &&: \forall\ \{i\}\ P\ \{Q\ R\} \to [\ i\ ]\ P \sim' Q \to [\ i\ ]\ Q \sim'\ R \to [\ i\ ]\ P \sim'\ R\\
&\_\sim\langle\_\rangle'\_ &&: \forall\ \{i\}\ P\ \{Q\ R\} \to [\ i\ ]\ P \sim\ \ Q \to [\ i\ ]\ Q \sim'\ R \to [\ i\ ]\ P \sim'\ R\\
&\_\blacksquare' &&: \forall\ \{i\}\ P \to [\ i\ ]\ P \sim'\ P\\
&\_|\textit{-cong}'\_ &&: \forall\ \{i\ P\ P'\ Q\ Q'\} \to\\
&&&\quad [\ i\ ]\ P \sim' Q \to [\ i\ ]\ P' \sim' Q' \to [\ i\ ]\ P \mid P' \sim' Q \mid Q'
\end{aligned}
$$

With these pieces in place the proof is, as mentioned above, straightforward. A fully formal proof takes up less than one page, and is in my opinion rather readable, see Figure 3. Note that the proof uses both size-preserving transitivity and a size-preserving preservation proof for parallel composition.

9 WEAK BISIMILARITY FOR THE DELAY MONAD

In order to see some examples of functions that are provably not size-preserving, let us now consider weak bisimilarity. I choose to focus on the delay monad. Despite its simplicity there are a number of non-contrived functions involving weak bisimilarity for the delay monad that fail to be size-preserving in one or more arguments.

Strong bisimilarity for the delay monad only relates terminating computations if they terminate in the same number of steps (i.e. if they have the same number of later constructors). In many cases it is more appropriate to consider *weak* bisimilarity, which identifies computations that terminate with the same value, even if the number of later constructors differ. Weak bisimilarity can be defined by adding two constructors to the definition of strong bisimilarity [Danielsson 2012]:

mutual

$$
\begin{aligned}
&\textbf{data}\ [\_]\_\approx_D\_\ \{A\}\ (i : \mathit{Size}) : (x\ y : \mathit{Delay}\ A\ \infty) \to \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{now} &&: \forall\ \{x\} \quad \to &&&&[\ i\ ]\ \mathsf{now}\ x \approx_D \mathsf{now}\ x\\
&\quad \mathsf{later} &&: \forall\ \{x\ y\} \to [\ i\ ]\ \mathsf{force}\ x \approx'_D \mathsf{force}\ y &&\to [\ i\ ]\ \mathsf{later}\ x \approx_D \mathsf{later}\ y\\
&\quad \mathsf{later}^{\mathsf{l}} &&: \forall\ \{x\ y\} \to [\ i\ ]\ \mathsf{force}\ x \approx_D \quad y &&\to [\ i\ ]\ \mathsf{later}\ x \approx_D \quad y\\
&\quad \mathsf{later}^{\mathsf{r}} &&: \forall\ \{x\ y\} \to [\ i\ ]\ \quad x \approx_D \mathsf{force}\ y &&\to [\ i\ ]\ \quad x \approx_D \mathsf{later}\ y
\end{aligned}
$$

```
!-cong : ∀ {i P Q} → [ i ] P ∼ Q → [ i ] ! P ∼ ! Q
!-cong = λ p → record
  { left-to-right = lr p
  ; right-to-left = map₃ symmetric′ ∘ lr (symmetric p)
  }
  where
  lr : ∀ {P Q R μ i} →
       [ i ] P ∼ Q → ! P [ μ ]→ R →
       ∃ λ S → ! Q [ μ ]→ S × [ i ] R ∼′ S
  lr {P} {Q} {R} P∼Q !P→R with 6-1-3-2 !P→R
  ... | inj₁ (P′ , P→P′ , R∼!P|P′) =
    let (Q′ , Q→Q′ , P′∼′Q′) = left-to-right P∼Q P→P′
    in
    ( ! Q | Q′
    , replication (par-right Q→Q′)
    , ( R          ∼⟨ R∼!P|P′ ⟩′
        ! P | P′   ∼′⟨ (λ { .force → !-cong P∼Q }) |-cong′ P′∼′Q′ ⟩′
        ! Q | Q′   ■′
      )
    )
  ... | inj₂ (refl , P′ , P″ , a , P→P′ , P→P″ , R∼!P|P′|P″) =
    let (Q′  , Q→Q′  , P′∼′Q′)  = left-to-right P∼Q P→P′
        (Q″ , Q→Q″ , P″∼′Q″) = left-to-right P∼Q P→P″
    in
    ( (! Q | Q′) | Q″
    , replication (par-τ (replication (par-right Q→Q′)) Q→Q″)
    , ( R                  ∼⟨ R∼!P|P′|P″ ⟩′
        (! P | P′) | P″    ∼′⟨ ((λ { .force → !-cong P∼Q }) |-cong′ P′∼′Q′)
                               |-cong′ P″∼′Q″ ⟩′
        (! Q | Q′) | Q″    ■′
      )
    )
```

Fig. 3. The replication operator preserves strong bisimilarity in a size-preserving way.

```
record [_]_≈′_ {A} (i : Size) (x y : Delay A ∞) : Set where
  coinductive
  field force : {j : Size< i} → [ j ] x ≈_D y
```

The new constructors later$^{\mathsf{l}}$ and later$^{\mathsf{r}}$ make it possible to selectively add later constructors to one of the sides of a weak bisimilarity proof. Just like the definition of CCS processes in Section 5 this definition involves an inductive definition nested inside a coinductive one. The new constructors are both "inductive", so proofs that only use them and now must have finite depth.

This definition of weak bisimilarity is pointwise logically equivalent to the one obtained from the LTS for the delay monad given in Section 4, and the two directions of the proof are both size-preserving. The same kind of equivalence also holds between this definition and the definition of weak bisimilarity presented by Capretta [2005], if Capretta's definition is adapted to use sized types in a suitable way. For details of these proofs, see the accompanying code.

9.1 Transitivity

Weak bisimilarity is transitive for every LTS. A proof of this fact can be found in the accompanying code. Here I give a proof only for the delay monad, as a setup for Section 9.2, in which it is shown that this proof cannot in general be made size-preserving.

First note that if x and y are weakly bisimilar, then one gets weakly bisimilar computations also if a later constructor is removed from x and/or y. Note that these proofs are not size-preserving:

$$
\begin{aligned}
&later^{r-1} : \forall \{A\ i\}\ \{j : Size{<}\ i\}\ \{x : Delay\ A\ \infty\}\ \{y\} \to \\
&\qquad [\ i\ ]\ x \approx_D \mathsf{later}\ y \to [\ j\ ]\ x \approx_D \mathsf{force}\ y \\
&later^{r-1}\ (\mathsf{later}\ \ p) = \mathsf{later}^{\mathsf{l}}\ (\mathsf{force}\ p) \\
&later^{r-1}\ (\mathsf{later}^{\mathsf{r}}\ p) = p \\
&later^{r-1}\ (\mathsf{later}^{\mathsf{l}}\ p) = \mathsf{later}^{\mathsf{l}}\ (later^{r-1}\ p)
\end{aligned}
$$

$$
\begin{aligned}
&later^{l-1} : \forall \{A\ i\}\ \{j : Size{<}\ i\}\ \{x\}\ \{y : Delay\ A\ \infty\} \to \\
&\qquad [\ i\ ]\ \mathsf{later}\ x \approx_D y \to [\ j\ ]\ \mathsf{force}\ x \approx_D y \\
&later^{l-1}\ (\mathsf{later}\ \ p) = \mathsf{later}^{\mathsf{r}}\ (\mathsf{force}\ p) \\
&later^{l-1}\ (\mathsf{later}^{\mathsf{r}}\ p) = \mathsf{later}^{\mathsf{r}}\ (later^{l-1}\ p) \\
&later^{l-1}\ (\mathsf{later}^{\mathsf{l}}\ p) = p
\end{aligned}
$$

$$
\begin{aligned}
&later^{-1} : \forall \{A\ i\}\ \{j : Size{<}\ i\}\ \{x\ y : Delay'\ A\ \infty\} \to \\
&\qquad [\ i\ ]\ \mathsf{later}\ x \approx_D \mathsf{later}\ y \to [\ j\ ]\ \mathsf{force}\ x \approx_D \mathsf{force}\ y \\
&later^{-1}\ (\mathsf{later}\ \ p) = \mathsf{force}\ p \\
&later^{-1}\ (\mathsf{later}^{\mathsf{r}}\ p) = later^{l-1}\ p \\
&later^{-1}\ (\mathsf{later}^{\mathsf{l}}\ p) = later^{r-1}\ p
\end{aligned}
$$

It is now easy to prove transitivity if the first computation is $\mathsf{now}\ x$ for some x:

$$
\begin{aligned}
&transitive^{w}\text{-}now : \forall \{A\ i\ x\}\ \{y\ z : Delay\ A\ \infty\} \to \\
&\qquad [\ i\ ]\ \mathsf{now}\ x \approx_D y \to [\ \infty\ ]\ y \approx_D z \to [\ i\ ]\ \mathsf{now}\ x \approx_D z \\
&transitive^{w}\text{-}now\ \mathsf{now}\ \ \ \mathsf{now}\ \ \ \ \ = \mathsf{now} \\
&transitive^{w}\text{-}now\ (\mathsf{later}^{\mathsf{r}}\ p)\ q\ \ \ \ \ = transitive^{w}\text{-}now\ p\ (later^{l-1}\ q) \\
&transitive^{w}\text{-}now\ p\ \ \ \ \ (\mathsf{later}^{\mathsf{r}}\ q) = \mathsf{later}^{\mathsf{r}}\ (transitive^{w}\text{-}now\ p\ q)
\end{aligned}
$$

Note that the second argument is assumed to have size ∞. The proof is defined using recursion on the structure of the first argument (note that this argument must be entirely inductive).

Finally we can define the full transitivity proof mutually with a proof for the case in which the first computation starts with a later constructor:

```
mutual

  transitive^w-later : ∀ {A i x} {y z : Delay A ∞} →
                       [ ∞ ] later x ≈_D y → [ ∞ ] y ≈_D z → [ i ] later x ≈_D z
  transitive^w-later p          (later  q) = later λ { .force →
                                                       transitive^w (later^-1 p) (force q) }
  transitive^w-later p          (later^r q) = later λ { .force → transitive^w (later^l-1 p) q }
  transitive^w-later p          (later^l q) = transitive^w-later (later^r-1 p) q
  transitive^w-later (later^l p) q          = later^l (transitive^w p q)

  transitive^w : ∀ {A i} {x y z : Delay A ∞} →
                 [ ∞ ] x ≈_D y → [ ∞ ] y ≈_D z → [ i ] x ≈_D z
  transitive^w {x = now  x} p q = transitive^w-now  p q
  transitive^w {x = later x} p q = transitive^w-later p q
```

These proofs are defined corecursively (with corecursive calls under copatterns), with an inner structural recursion (either the second argument is smaller, or this argument stays unchanged and the first argument is smaller).

9.2 Transitivity is not size-preserving

Note that both arguments of the transitivity proof are assumed to have size ∞. Let us now prove that, if the carrier type A is inhabited, then transitivity cannot be made size-preserving in either argument. These results follow from a similar result for the following type:

```
Drop-later : Set → Set
Drop-later A = ∀ {i} {x : A} → [ i ] later (λ { .force → now x }) ∼_D never →
                               [ i ] now x                       ≈_D never
```

First note that now x is not weakly bisimilar to *never* (at any size):

```
now≉never : ∀ {A i} {x : A} → ¬ [ i ] now x ≈_D never
now≉never (later^r p) = now≉never p
```

Now we can derive a contradiction from the assumption that *Drop-later* A and A are both inhabited. The second assumption gives us a value x in A, and the first assumption can then be used to prove that now x and *never* are weakly bisimilar:

```
basic-counterexample : ∀ {A} → Drop-later A → ¬ A
basic-counterexample drop-later x = contradiction ∞
  where
  mutual

    now≈never : ∀ {i} → [ i ] now x ≈_D never
    now≈never = drop-later (later now∼never)

    now∼never : ∀ {i} → [ i ] now x ∼'_D never
    force now∼never {j = j} = ⊥-elim (contradiction j)

    contradiction : Size → ⊥
    contradiction i = now≉never (now≈never {i = i})
```

Using *basic-counterexample* we can now prove that, if the carrier type is inhabited, then there is no transitivity proof that preserves the size of the second argument, even if we restrict it so that it is a *strong* bisimilarity proof:

$$
\begin{aligned}
&\mathit{not\text{-}size\text{-}preserving}^{ws} :\\
&\quad \forall\ \{A\} \to\\
&\quad (\forall\ \{i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to [\ \infty\ ]\ x \approx_D y \to [\ i\ ]\ y \sim_D z \to [\ i\ ]\ x \approx_D z) \to\\
&\quad \neg\ A\\
&\mathit{not\text{-}size\text{-}preserving}^{ws}\ \mathit{trans} = \mathit{basic\text{-}counterexample}\ (\lambda\ p \to \mathit{trans}\ (\mathsf{later}^{\mathsf{r}}\ \mathsf{now})\ p)
\end{aligned}
$$

The fact that there is a size-preserving translation from strong to weak bisimilarity gives us the same result for weak bisimilarity:

$$
\begin{aligned}
&\mathit{strong\text{-}to\text{-}weak} : \forall\ \{A\ i\}\ \{x\ y : \mathit{Delay}\ A\ \infty\} \to [\ i\ ]\ x \sim_D y \to [\ i\ ]\ x \approx_D y\\
&\mathit{strong\text{-}to\text{-}weak}\ \mathsf{now} \qquad = \mathsf{now}\\
&\mathit{strong\text{-}to\text{-}weak}\ (\mathsf{later}\ p) = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \to \mathit{strong\text{-}to\text{-}weak}\ (\mathsf{force}\ p)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{not\text{-}size\text{-}preserving}^{wr} :\\
&\quad \forall\ \{A\} \to\\
&\quad (\forall\ \{i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to [\ \infty\ ]\ x \approx_D y \to [\ i\ ]\ y \approx_D z \to [\ i\ ]\ x \approx_D z) \to\\
&\quad \neg\ A\\
&\mathit{not\text{-}size\text{-}preserving}^{wr}\ \mathit{trans} =\\
&\quad \mathit{not\text{-}size\text{-}preserving}^{ws}\ (\lambda\ p\ q \to \mathit{trans}\ p\ (\mathit{strong\text{-}to\text{-}weak}\ q))
\end{aligned}
$$

Given that there are size-preserving symmetry proofs for both strong and weak bisimilarity we also get analogous results showing that, if the carrier type is inhabited, then there are no transitivity proofs that preserve the size of the *first* argument. These results are related to the problem of weak bisimulation up to weak bisimilarity [Sangiorgi and Milner 1992].

As an aside one can also prove that, if A is not inhabited, then *Drop-later* A is inhabited, and transitivity for weak bisimilarity is size-preserving in both arguments. This follows directly from the observation that in this case weak bisimilarity is trivial:

$$
\begin{aligned}
&\mathit{trivial} : \forall\ \{A\ i\} \to \neg\ A \to \forall\ x\ y \to [\ i\ ]\ x \approx_D y\\
&\mathit{trivial}\ \mathit{empty}\ (\mathsf{now}\ x)\ \_ \qquad = \bot\text{-}\mathit{elim}\ (\mathit{empty}\ x)\\
&\mathit{trivial}\ \mathit{empty}\ (\mathsf{later}\ x)\ (\mathsf{now}\ y) = \bot\text{-}\mathit{elim}\ (\mathit{empty}\ y)\\
&\mathit{trivial}\ \mathit{empty}\ (\mathsf{later}\ x)\ (\mathsf{later}\ y) = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \to \mathit{trivial}\ \mathit{empty}\ (\mathsf{force}\ x)\ (\mathsf{force}\ y)\ \}
\end{aligned}
$$

The lack of size-preserving transitivity can complicate things, but fortunately there are a number of related results that are size-preserving. For instance, fully defined strong bisimilarity can be combined with weak bisimilarity in a size-preserving way:

$$
\begin{aligned}
&\mathit{transitive}^{sw} :\ \forall\ \{A\ i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to\\
&\qquad\qquad [\ \infty\ ]\ x \sim_D y \to [\ i\ ]\ y \approx_D z \to [\ i\ ]\ x \approx_D z
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{transitive}^{ws} :\ \forall\ \{A\ i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to\\
&\qquad\qquad [\ i\ ]\ x \approx_D y \to [\ \infty\ ]\ y \sim_D z \to [\ i\ ]\ x \approx_D z
\end{aligned}
$$

These proofs correspond to weak bisimulation up to strong bisimilarity.

We get somewhat stronger results by instead working with the *expansion* relation, here called $[\_]\_ \gtrsim_D \_$, which we get if we remove the $\mathsf{later}^{\mathsf{r}}$ constructor from the definition of weak bisimilarity [Danielsson 2012]:

$$\begin{aligned}
\mathit{transitive}^{e} &: \forall\ \{A\ i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to \\
&\quad [\ \infty\ ]\ x \gtrsim_D y \to [\ i\ ]\ y \gtrsim_D z \to [\ i\ ]\ x \gtrsim_D z \\
\mathit{transitive}^{ew} &: \forall\ \{A\ i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to \\
&\quad [\ \infty\ ]\ x \gtrsim_D y \to [\ i\ ]\ y \approx_D z \to [\ i\ ]\ x \approx_D z \\
\mathit{transitive}^{we} &: \forall\ \{A\ i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to \\
&\quad [\ i\ ]\ x \approx_D y \to [\ \infty\ ]\ z \gtrsim_D y \to [\ i\ ]\ x \approx_D z
\end{aligned}$$

(Note that the last argument of $\mathit{transitive}^{we}$ has type $[\ \infty\ ]\ z \gtrsim_D y$, with z before y.) The expansion relation can be defined for any LTS [Sangiorgi and Milner 1992], and the three results above can be proved in this setting as well, see the accompanying code. One can also prove negative results, for instance the following ones:

$$\begin{aligned}
&\mathit{not\text{-}size\text{-}preserving}^{se} : \\
&\quad \forall\ \{A\} \to \\
&\quad (\forall\ \{i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to [\ i\ ]\ x \sim_D y \to [\ \infty\ ]\ y \gtrsim_D z \to [\ i\ ]\ x \gtrsim_D z) \to \\
&\quad \neg\ A
\end{aligned}$$

$$\begin{aligned}
&\mathit{not\text{-}size\text{-}preserving}^{ew} : \\
&\quad \forall\ \{A\} \to \\
&\quad (\forall\ \{i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to [\ i\ ]\ x \gtrsim_D y \to [\ \infty\ ]\ y \approx_D z \to [\ i\ ]\ x \approx_D z) \to \\
&\quad \neg\ A
\end{aligned}$$

$$\begin{aligned}
&\mathit{not\text{-}size\text{-}preserving}^{we} : \\
&\quad \forall\ \{A\} \to \\
&\quad (\forall\ \{i\}\ \{x\ y\ z : \mathit{Delay}\ A\ \infty\} \to [\ i\ ]\ x \approx_D y \to [\ \infty\ ]\ y \gtrsim_D z \to [\ i\ ]\ x \approx_D z) \to \\
&\quad \neg\ A
\end{aligned}$$

10 RELATED WORK

Quite a bit of related work has already been discussed above. This section contains some further notes.

It was noted by de Vries [2009] that a certain transitive relation defined using nested induction and coinduction becomes trivial when its definition is extended with an inductive constructor corresponding to transitivity. He connected this to the problem of weak bisimulation up to weak bisimilarity [Sangiorgi and Milner 1992]. The proof of triviality given by de Vries is similar to the definition of *basic-counterexample* in Section 9.2. His observation was discussed further by Danielsson and Altenkirch [2010], and later Danielsson [2012] remarked that some uses of size-preserving proofs are similar to Sangiorgi and Milner's "expansions up to $\lesssim$" [1992]. Danielsson only discusses a small number of size-preserving functions, including proofs corresponding to $\mathit{transitive}^{ew}$ and $\mathit{transitive}^{we}$ from Section 9.2.

In addition to what has been mentioned above Pous proves a lemma that can be used to take advantage of symmetries when proving things [2016, Proposition 7.1]. I have chosen not to present such a lemma, because in the applications I have considered I have found it sufficient to use size-preserving symmetry proofs. For some examples, see the definitions of *transitive* in Figure 1, _|-*cong*_ in Figure 2, and !-*cong* in Figure 3.

11 CONCLUSIONS

I hope that this text gives some idea of what up-to techniques can look like when sized types are available. My experience is that things tend to "just work" when sized types are used. They are a little noisy (lots of size indices), and the Agda implementation still contains some quirks, but I can usually avoid bringing in heavy technical machinery to get my definitions past the termination checker.

The connection to the work of Pous [2016] and others on functions below the companion can perhaps be seen as an indication of why sized types, in my opinion, work rather well in practice. It can also serve as a hint about techniques to use when, say, porting code from a language with sized types to one without.

ACKNOWLEDGEMENTS

I would like to thank Andreas Abel and Damien Pous for informing me about related work, and Andreas Abel for helping me with details related to sized types.

This work has been supported by a grant from the Swedish Research Council (621-2013-4879).

A INDEXED CONTAINERS

This appendix contains a very brief definition of doubly indexed containers [Altenkirch et al. 2015], specialised so that they only have one index type.

An X-indexed container consists of an X-indexed family of shapes, and for each shape, an X-indexed family of positions:

```
record Container (X : Set) : Set₁ where
  constructor _◁_
  field
    Shape    : X → Set
    Position : ∀ {x} → Shape x → X → Set
```

The interpretation of such a container maps X-indexed type families to X-indexed type families:

```
⟦_⟧ : ∀ {X} → Container X → (X → Set) → (X → Set)
⟦ S ◁ P ⟧ A = λ x → ∃ λ (s : S x) → P s ⊆ A
```

For a given family A and index x the interpretation consists of pairs of an x-indexed shape s and an index-preserving function from positions for s to A.

There is also an associated map function:

```
map : ∀ {X} (C : Container X) {A B} →
      A ⊆ B → ⟦ C ⟧ A ⊆ ⟦ C ⟧ B
map _ f (s , g) = (s , f ∘ g)
```

REFERENCES

Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In *Proceedings 8th Workshop on Fixed Points in Computer Science*. https://doi.org/10.4204/EPTCS.77.1

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* (2016). https://doi.org/10.1017/S0956796816000022

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *POPL '13, Proceedings of 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2429069.2429075

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017). https://doi.org/10.1145/3110277

The Agda Team. 2017. The Agda Wiki. (2017). Retrieved 2017-07-02 from http://wiki.portal.chalmers.se/agda/

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* (2015). https://doi.org/10.1017/S095679681500009X

Roberto M. Amadio and Solange Coupet-Grimal. 1998. Analysis of a Guard Condition in Type Theory. In *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98*. https://doi.org/10.1007/BFb0053541

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. 2004. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science* (2004). https://doi.org/10.1017/S0960129503004122

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2006. CIC$\hat{}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006*. https://doi.org/10.1007/11916277_18

Henning Basold, Damien Pous, and Jurriaan Rot. 2017. Monoidal Company for Accessible Functors. In *7th Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*.

Frédéric Blanqui. 2004. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004*. https://doi.org/10.1007/978-3-540-25979-4_2

Frédéric Blanqui. 2005. Decidability of Type-Checking in the Calculus of Algebraic Constructions with Size Annotations. In *Computer Science Logic, 19th International Workshop, CSL 2005*. https://doi.org/10.1007/11538363_11

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *ICFP'12, Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. https://doi.org/10.1145/2364527.2364546

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, Declaratively: An Exercise in Mixed Induction and Coinduction. In *Mathematics of Program Construction, 10th International Conference, MPC 2010*. https://doi.org/10.1007/978-3-642-13321-3_8

Eduardo Giménez. 1998. Structural Recursive Definitions in Type Theory. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98*. https://doi.org/10.1007/BFb0055070

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État. Université Paris VII.

Benjamin Grégoire and Jorge Luis Sacchini. 2010. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In *Logic for Programming, Artificial Intelligence, and Reasoning, 17th International Conference, LPAR-17*. https://doi.org/10.1007/978-3-642-16242-8_24

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. https://doi.org/10.1145/237721.240882

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *POPL '13, Proceedings of 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2429069.2429093

Robin Milner. 1988. *Operational and Algebraic Semantics of Concurrent Processes*. Technical Report ECS-LFCS-88-46. LFCS, Department of Computer Science, University of Edinburgh.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology and Göteborg University.

Joachim Parrow and Tjark Weber. 2016. The Largest Respectful Function. *Logical Methods in Computer Science* (2016). https://doi.org/10.2168/LMCS-12(2:11)2016

Damien Pous. 2016. Coinduction All the Way Up. In *Proceedings of the 31st Annual ACM-IEEE Symposium on Logic in Computer Science (LICS 2016)*. https://doi.org/10.1145/2933575.2934564

Damien Pous and Jurriaan Rot. 2017. Companions, Codensity and Causality. In *Foundations of Software Science and Computation Structures, 20th International Conference, FOSSACS 2017*. https://doi.org/10.1007/978-3-662-54458-7_7

Damien Pous and Davide Sangiorgi. 2011. Enhancements of the bisimulation proof method. In *Advanced Topics in Bisimulation and Coinduction*, Davide Sangiorgi and Jan Rutten (Eds.). https://doi.org/10.1017/CBO9780511792588.007

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.2013.29

Jorge Luis Sacchini. 2014. Linear Sized Types in the Calculus of Constructions. In *Functional and Logic Programming, 12th International Symposium, FLOPS 2014*. https://doi.org/10.1007/978-3-319-07151-0_11

Jorge Luis Sacchini. 2015. Well-Founded Sized Types in the Calculus of (Co)Inductive Constructions. Draft. (2015). Retrieved 2017-07-03 from http://web.archive.org/web/20160531152811/http://www.qatar.cmu.edu:80/~sacchini/well-founded/well-founded.pdf

Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* (1998). https://doi.org/10.1017/S0960129598002527

Davide Sangiorgi and Robin Milner. 1992. The problem of "weak bisimulation up to". In *CONCUR '92, Third International Conference on Concurrency Theory*. https://doi.org/10.1007/BFb0084781

Steven Schäfer and Gert Smolka. 2017. Tower Induction and Up-to Techniques for CCS with Fixed Points. In *Relational and Algebraic Methods in Computer Science, 16th International Conference, RAMiCS 2017*. https://doi.org/10.1007/978-3-319-57418-9_17

Edsko de Vries. 2009. Re: [Coq-Club] Adding (inductive) transitivity to weak bisimilarity not sound? (was: Need help with coinductive proof). Message to the Coq-Club mailing list. (Aug. 2009).

Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *Higher-Order and Symbolic Computation* (2002). https://doi.org/10.1023/A:1019916231463