Up-to Techniques using Sized Types

NILS ANDERS DANIELSSON, University of Gothenburg & Chalmers University of Technology, Sweden

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 often convenient to use in practice. Some examples of functions that cannot be made size-preserving are also included, in order to illustrate the limits of the approach. On the more theoretical side a class of up-to techniques intended to capture a natural mode of use of size-preserving functions is defined. This class turns out to correspond closely to "functions below the companion", a notion recently introduced by Pous.

CCS Concepts: • **Theory of computation → Type theory**; *Type structures*; *Program verification*;

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

ACM Reference Format:
Nils Anders Danielsson. 2018. Up-to Techniques using Sized Types. *Proc. ACM Program. Lang.* 2, POPL, Article 43 (January 2018), 28 pages. https://doi.org/10.1145/3158131

1 INTRODUCTION

Coinductively defined types or sets can be used to represent values or proofs that are potentially infinite. In constructive type theories with support for coinductive types the basic method for defining values in such types is often some notion of *guarded corecursion* [Coquand 1994]. This method ensures that the values are well-defined (productive) by requiring that every corecursive call is guarded by constructors. However, the method can be awkward 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. A goal of this text is to show how easy it is to use a certain class of *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 [Milner 1983] 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]. Readers who are not familiar with up-to techniques can find a brief introduction in Section 6.

2475-1421/2018/1-ART43
https://doi.org/10.1145/3158131

The connection between sized types and up-to techniques was pointed out by Danielsson [2012], who compared some size-preserving proofs to Sangiorgi and Milner's "expansion up to $\lesssim$" [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 a useful class of up-to techniques introduced by Pous [2016]: functions below the *companion*, the greatest compatible function (see Section 7).

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.

Some of the presented size-preserving proofs are closely related to known up-to techniques. These proofs generally fall into one of two classes: preservation proofs, which are related to "up to context" techniques; and transitivity-like proofs, which are related to up-to techniques such as "weak bisimulation up to expansion".

An important event in the history of up-to techniques was the discovery of the problem of weak bisimulation up to weak bisimilarity [Sangiorgi and Milner 1992]: the problem is that weak bisimulations up to weak bisimilarity are not, in general, contained in weak bisimilarity (as previously stated by Milner [1989]). I believe it is fair to say that lots of work in this area has been devoted to finding alternatives to this technique, and that a perfect alternative has not been conclusively identified. The problem implies that transitivity of weak bisimilarity cannot, in general, be size-preserving. In Section 9.2 it is shown (using an idea due to de Vries [2009]) that a number of size-preserving transitivity-like statements for various combinations of strong and weak bisimilarity and expansion do not hold in general. These results put limits on what can be done using size-preserving functions.

1.1 Formalisation

All the main results and examples in this paper have been formalised using Agda (with the K rule turned off), 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 below.

2 SIZED TYPES

Let us begin with a short introduction to sized types, as implemented in Agda. There are multiple notions of sized types, and in fact Agda (which is used for experiments in this area) supports at least two of them. However, here I will describe only one notion. Furthermore sizes can be used with both inductive and coinductive types, but here I will only use them with 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 the special size ∞. Given a value of type $\mathit{Delay}'\ A\ i$ one can use the force projection to get a value of type $\mathit{Delay}\ A\ j$ for any $j : \mathit{Size}< i$. One can see values of type $\mathit{Delay}\ A\ i$ as values that are perhaps not fully defined. The special size ∞ has the property that $i : \mathit{Size}< \infty$ holds for all sizes i, including ∞. Thus, if one has a value of type $\mathit{Delay}'\ A\ \infty$ then one can use the force projection to get a value of type $\mathit{Delay}\ A\ \infty$. The idea is that ∞ stands for a closure ordinal for which $\mathit{Delay}'\ A\ \infty$ and $\mathit{Delay}\ A\ \infty$ are (in some sense) isomorphic. Values of type $\mathit{Delay}\ A\ \infty$ can be seen as fully defined.

The Agda implementation also includes a successor function on sizes, as well as a notion of subtyping: if $j : \mathit{Size}< i$, then values of type $\mathit{Delay}'\ A\ i$ can be used where values of type $\mathit{Delay}'\ A\ j$ are expected.

We can define values in the delay monad corecursively, using *copatterns* [Abel et al. 2013]. A copattern corresponding to a record field specifies the result of projecting the given field from the value. 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 to later *never′*, but *never′* is a value that only unfolds if the projection force is applied to it. (This projection is by default not in scope unqualified in Agda. Code used to bring names into scope is omitted from this text.)

Notation like $\{j : \mathit{Size}< i\} \to \ldots$ or $\forall\ \{A\ i\} \to \ldots$ 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*. A pattern $\{x = p\}$ binds the variable p to the implicit argument x, and an application $f\ \{x = e\}$ stands for f with the implicit argument x bound to e:

```
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 ∞). Agda's termination checker accepts this code because every cycle in the call graph (*never* $\rightarrow$ *never′* $\rightarrow$ *never* and *never′* $\rightarrow$ *never* $\rightarrow$ *never′*) involves a strict decrease in the size argument, and furthermore the strictly smaller size j is associated in a certain way with a copattern force that corresponds to a field of the *coinductive* record type *Delay′*. The idea is, roughly, that for any given size other than ∞ the result is specified via a form of transfinite recursion. Furthermore, because the definition is given for an arbitrary size i and matching on sizes is not allowed, the termination of force *never′* $\{j = j\}$ is not affected by whether j is ∞ or not.

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

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

```
transitive^s : ∀ {A i} {x y z : Delay A ∞} →
               [ i ] x ∼D y → [ i ] y ∼D z → [ i ] x ∼D z
transitive^s now      now      = now
transitive^s (later p) (later q) = later λ { .force → transitive^s (force p) (force q) }
```

Instead of using an auxiliary definition (like *never′*) this proof uses a local copattern (λ { .force $\rightarrow$ ... }). This transitivity proof 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. Note that a size-preserving proof of this kind has to generate the output in lock-step with the consumption of the input (or "faster"): the arguments of the later constructors in the input cannot be forced until the first later constructor has been output, because only then is a suitable size j : *Size*< i available.

The approach to sized types presented above is based on *deflationary iteration* [Abel 2012]. For more details about this kind of sized types, including normalisation proofs (in one case sketched), see Abel and Pientka [2016] and Sacchini [2015]. Note that Abel and Pientka discuss a language without dependent types (based on System F_ω), and that while Sacchini's unpublished draft treats a dependently typed language, this language differs from Agda. Furthermore the current, experimental Agda implementation contains bugs related to sized types. For instance, the fact that ∞ : *Size*< ∞ has led to problems in the implementation, and a new, slightly different design is being discussed. 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 $X \leq \nu F$.

In dependent type theories like Agda and Coq inductive and coinductive data types have to be *strictly positive*. A variant of the argument above can be carried out 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 $Container\ X$, which comes with an interpretation function

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

and a map function:

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

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

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

Note that the map function shows that $[\![\ C\ ]\!]$ is monotone with respect to $\_\subseteq\_$; strict positivity is a stronger condition than monotonicity.

The definitions of *Container*, $[\![\_]\!]$ and *map* are presented at the end of this section. First let us see how, given an indexed container, one can define a greatest fixpoint for it coinductively. The details of the definition of indexed containers ensure that this definition is accepted as strictly positive by Agda:

```
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 $[\![\ C\ ]\!]$:

$$\begin{aligned} &\nu\text{-}out : \forall\ \{X\ i\}\ \{j : Size< i\}\ (C : Container\ X) \to \\ &\qquad \nu\ C\ i \subseteq [\![\ C\ ]\!]\ (\nu\ C\ j) \\ &\nu\text{-}out\ C = map\ C\ (\lambda\ x \to \mathsf{force}\ x) \end{aligned}$$

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

$$\begin{aligned} &\nu\text{-}in : \forall\ \{X\ i\}\ (C : Container\ X) \to \\ &\qquad [\![\ C\ ]\!]\ (\nu\ C\ i) \subseteq \nu\ C\ i \\ &\nu\text{-}in\ C = map\ C\ (\lambda\ x \to \lambda\ \{\ .\mathsf{force} \to x\ \}) \end{aligned}$$

Finally we can show that $\nu\ C\ i$ is greater than every post-fixpoint (and thus also every 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. The type family $\nu\ C\ \infty$ will nevertheless be referred to as *the* greatest fixpoint of C below.

Before giving a definition of indexed containers I would like to stress that containers do not have to be used to define greatest fixpoints in Agda. The reason for using containers above is that I want to define greatest fixpoints for a large class of functors. However, if one is only interested in a particular greatest fixpoint, then it may be more convenient to define it directly—as with the delay monad in Section 2—because then one can avoid noise related to the coding.

Let us now see how containers can be defined. The definition presented here corresponds to the doubly indexed containers of 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
```

(Here Set_1 is a type of large types.) 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. (If Y has type $X \to Set$, then the type $\exists\ Y$ consists of dependent pairs $(x\ ,\ y)$, where—at least if we ignore subtyping—x has type X and y has type $Y\ x$.) Given the definitions above it is easy to define the *map* function by using composition:

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

As an example, let us now define the delay monad $Delay\ A\ \infty$ as the greatest fixpoint of a (trivially indexed) container. The container should represent the (unindexed) functor taking X to the binary sum of A and X.

An anonymous reviewer once informed me that one can get a suitable type of shapes by applying the functor to the unit type. This follows from the definition of the interpretation function $\llbracket\_\rrbracket$. (In the indexed case one can apply the functor to the constant function yielding the unit type.) In this case we can use the type $Maybe\ A$, a type with two constructors, nothing : $Maybe\ A$ and just : $A \to Maybe\ A$.

The shape just x corresponds to the constructor application now x. The now constructor does not have any recursive arguments, so the type of positions for this shape can be empty. The shape nothing corresponds to the constructor later, which takes a single recursive argument, so we can

associate this shape with a single position. We end up with the following container, where $\bot$ is the empty type and $\top$ the unit type (with the single inhabitant tt):

$$\begin{aligned}&\mathit{Delay}^{\mathrm{C}} : \mathit{Set} \to \mathit{Container}\ \top\\&\mathit{Delay}^{\mathrm{C}}\ A = (\lambda\ \_ \to \mathit{Maybe}\ A) \lhd (\lambda\ \{\ (\mathsf{just}\ \_)\ \_ \to \bot;\ \mathsf{nothing}\ \_ \to \top\ \})\end{aligned}$$

One can prove that the greatest fixpoint of this container (applied to tt) is logically equivalent to the direct definition of the delay monad given in Section 2:

$$\forall\ \{i\} \to \mathit{Delay}\ A\ i \Leftrightarrow \nu\ (\mathit{Delay}^{\mathrm{C}}\ A)\ i\ \mathsf{tt}$$

Note that this proof is size-preserving. For i equal to ∞ one can also prove that the two components of this result are inverses up to (strong) bisimilarity, with bisimilarity for ν defined in a certain way; see the accompanying code for details.

The container above is indexed in a trivial way, with the unit type as the index type. For an example of a container with (in general) non-trivial indexing, see the following section.

4 LABELLED TRANSITION SYSTEMS

Let us now define labelled transition systems, using ν to define strong and weak bisimilarity as well as the expansion relation.

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

$$\begin{aligned}&\textbf{record}\ \mathit{LTS} : \mathit{Set}_1\ \textbf{where}\\&\quad\textbf{field}\\&\qquad\mathsf{Proc}\ \ \ \ \ \ : \mathit{Set}\\&\qquad\mathsf{Label}\ \ \ \ : \mathit{Set}\\&\qquad\_[\_]\!\mapsto\!\_\ \ : \mathsf{Proc} \to \mathsf{Label} \to \mathsf{Proc} \to \mathit{Set}\\&\qquad\mathsf{is\text{-}silent} : \mathsf{Label} \to \mathit{Bool}\end{aligned}$$

(Here *Bool* is the type of booleans.)

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:

$$\begin{aligned}&\textbf{record}\ B\ (R : \mathsf{Proc} \times \mathsf{Proc} \to \mathit{Set})\ (PQ : \mathsf{Proc} \times \mathsf{Proc}) : \mathit{Set}\ \textbf{where}\\&\quad\textbf{private}\ P = \mathsf{proj}_1\ PQ;\ Q = \mathsf{proj}_2\ PQ\\&\quad\textbf{field}\\&\qquad\mathsf{left\text{-}to\text{-}right} : \forall\ \{P'\ \mu\} \to P\ [\ \mu\ ]\!\mapsto P' \to \exists\ \lambda\ Q' \to Q\ [\ \mu\ ]\!\mapsto Q' \times R\ (P'\ ,\ Q')\\&\qquad\mathsf{right\text{-}to\text{-}left} : \forall\ \{Q'\ \mu\} \to Q\ [\ \mu\ ]\!\mapsto Q' \to \exists\ \lambda\ P' \to P\ [\ \mu\ ]\!\mapsto P' \times R\ (P'\ ,\ Q')\end{aligned}$$

(The type $X \times Y$ is the non-dependent product of X and Y, and the functions proj_1 and proj_2 give the first and second components of a pair, respectively.)

This type can also be defined as a container. The details of such definitions are not central to the paper, but a definition is included here as another example of how containers can be defined:

$$
\begin{aligned}
&B^{\mathrm{C}} : \mathit{Container}\ (\mathsf{Proc} \times \mathsf{Proc}) \\
&B^{\mathrm{C}} = (\lambda\ \{\ (P\ ,\ Q) \to (\forall\ \{P'\ \mu\} \to P\ [\ \mu\ ]\!\!\rightarrowtail P' \to \exists\ \lambda\ Q' \to Q\ [\ \mu\ ]\!\!\rightarrowtail Q') \times \\
&\qquad\qquad\qquad\qquad (\forall\ \{Q'\ \mu\} \to Q\ [\ \mu\ ]\!\!\rightarrowtail Q' \to \exists\ \lambda\ P' \to P\ [\ \mu\ ]\!\!\rightarrowtail P')\ \}) \\
&\qquad \lhd \\
&\quad (\lambda\ \{\ \{(P\ ,\ Q)\}\ (lr\ ,\ rl)\ (P'\ ,\ Q') \to \\
&\qquad (\exists\ \lambda\ \mu \to \exists\ \lambda\ (P{\to}P' \ : P\ [\ \mu\ ]\!\!\rightarrowtail P') \to \mathsf{proj}_1\ (lr\ P{\to}P')\ \equiv Q') \uplus \\
&\qquad (\exists\ \lambda\ \mu \to \exists\ \lambda\ (Q{\to}Q' : Q\ [\ \mu\ ]\!\!\rightarrowtail Q') \to \mathsf{proj}_1\ (rl\ Q{\to}Q') \equiv P')\ \})
\end{aligned}
$$

Here $\_\uplus\_$ is binary sum, $\_\equiv\_$ is equality, and $P{\to}P'$ and $Q{\to}Q'$ are both single tokens with suggestive names. The shapes consist of pairs of functions $(lr\ ,\ rl)$ corresponding to the downward arrows in the diagrams above. If such a shape is given for the pair $(P\ ,\ Q)$, then the type of positions for the pair $(P'\ ,\ Q')$ is inhabited if Q' is equal to the process at the end of the dashed arrow corresponding to the application of lr to some transition from P to P', or similarly for P', rl, Q and Q'.

The interpretation of the container is pointwise logically equivalent to the record type B:

$$\forall\ \{R\ PQ\} \to B\ R\ PQ \Leftrightarrow [\![\ B^{\mathrm{C}}\ ]\!]\ R\ PQ$$

Furthermore, in the presence of extensionality for dependent functions these logical equivalences can be strengthened to bijective correspondences (see the accompanying code for details).

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

$$
\begin{aligned}
&\mathit{Bisimilar} : \mathsf{Proc} \to \mathsf{Proc} \to \mathit{Set}_1 \\
&\mathit{Bisimilar}\ P\ Q = \exists\ \lambda\ R \to (R \subseteq [\![\ B^{\mathrm{C}}\ ]\!]\ R) \times R\ (P\ ,\ Q)
\end{aligned}
$$

Note that this definition is large: it targets Set_1 rather than Set. (Agda is predicative.) An alternative is to use ν:

$$
\begin{aligned}
&[\_]\_{\sim}\_ : \mathit{Size} \to \mathsf{Proc} \to \mathsf{Proc} \to \mathit{Set} \\
&[\ i\ ]\ P \sim Q = \nu\ B^{\mathrm{C}}\ i\ (P\ ,\ Q)
\end{aligned}
$$

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

$$
\begin{aligned}
&[\_]\_{\sim'}\_ : \mathit{Size} \to \mathsf{Proc} \to \mathsf{Proc} \to \mathit{Set} \\
&[\ i\ ]\ P \sim' Q = \nu'\ B^{\mathrm{C}}\ i\ (P\ ,\ Q)
\end{aligned}
$$

One can define weak bisimilarity, and the expansion relation [Arun-Kumar and Hennessy 1992; Sangiorgi and Milner 1992], in similar ways. The is-silent field of the *LTS* record identifies labels that should be seen as silent (like the τ label used for CCS, see Section 5). One can then define several derived transition relations:

- $P \Rightarrow Q$, which stands for a sequence of zero or more silent transitions leading from P to Q.
- $P\ [\ \mu\ ]\!\!\Rightarrow Q$, which stands for a sequence of transitions leading from P to Q, with one μ-transition and the rest silent.
- $P\ [\ \mu\ ]\!\!\stackrel{\hat{}}{\Rightarrow} Q$ (often written with $\hat{\mu}$ above an arrow), which means the same as $P\ [\ \mu\ ]\!\!\Rightarrow Q$ if μ is not silent, and otherwise means the same as $P \Rightarrow Q$.
- $P\ [\ \mu\ ]\!\!\stackrel{\hat{}}{\rightarrowtail} Q$, which holds if there is a μ-transition from P to Q, or if μ is silent and P is equal to Q.

We get weak bisimilarity by replacing the dashed arrows in the diagrams above by $\_[\_]\hat{\Rightarrow}\_$. Similarly, we get the expansion relation $[\_]\_\gtrsim\_$, with the expansion to the left, if the left-hand dashed arrow is replaced by $\_[\_]\hat{\rightarrow}\_$ and the right-hand dashed arrow is replaced by $\_[\_]\Rightarrow\_$. The accompanying code contains a general definition that can be instantiated with different transition relations to yield strong or weak bisimilarity or the expansion relation.

Let us now define a labelled transition system for the delay monad. The type of processes is *Delay A* ∞, and the type of labels is *Maybe A*; nothing is treated as silent, and just x 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 this LTS is pointwise logically equivalent to the definition of strong bisimilarity given in Section 2 ($[\_]\_\sim_{\mathrm{D}}\_$):

```
∀ {A i} {x y : Delay A ∞} → [ i ] x ∼D y ⇔ [ i ] x ∼ 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 interpretation of the container B^{C}, 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
```

```
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.

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

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

The following process constructors are included: an inactive process (∅), parallel composition (_|_), sum (_⊕_), prefixing (_·_), restriction (⟨ν_⟩), 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
```

Following Giménez [1996] this definition uses a mixture of induction and coinduction to restrict infinite processes. The two mutually defined types 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. Infinite

processes are included instead of the recursive process constants used in other variants of CCS, and, if there is exactly one constant X, then this restriction corresponds to only allowing the definition $X \triangleq P$ if every occurrence of X in P is *weakly guarded* [Milner 1989], i.e. under a prefix. This restriction is motivated below. Readers who prefer finite processes may want to note that replication is available, and that the transition relation below takes finite processes to finite processes.

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 (if any):

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

Here $\neg A$ 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 a≢a action) = ⊥-elim (a≢a 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 of parallel composition. This proof is corecursive, using a simple case analysis in the left-to-right direction:

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

```
_|-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.

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
left-to-right (·-cong p) action = (_ , action , p)
right-to-left (·-cong p) action = (_ , action , p)
```

The reason for only making the _·_ constructor "coinductive" is that this is the only recursive constructor for which I have proved a preservation lemma where the arguments' types use the primed variant of bisimilarity, in the way above. Note that, even though the _·_ constructor is coinductive also for the silent action τ, preservation lemmas corresponding to ·-cong, with a primed argument, can also be proved for expansion and weak bisimilarity (see the accompanying code).

The type of ·-cong makes it easy to construct certain infinite bisimilarity proofs corecursively. Consider the following definitions, which both make use of an action μ (names are here taken to be natural numbers):

```
P : ∀ {i} → ℕ → Proc i
P n = Restricted n | (μ · λ { .force → P (1 + n) })
```

$$Q : \forall\ \{i\} \to \mathit{Proc}\ i$$
$$Q = \mu \cdot \lambda\ \{\ .\mathsf{force} \to Q\ \}$$

(As an aside, note that every member of the process family P is *irregular*, i.e. containing infinitely many distinct subprocesses.) Using the combinators introduced above it is easy to prove that any member of the process family P is strongly bisimilar to Q:

$$P{\sim}Q : \forall\ \{i\ n\} \to [\ i\ ]\ P\ n \sim Q$$
$$P{\sim}Q = \mathit{transitive}\ (\mathit{Restricted{\sim}\emptyset}\ \mathit{|\text{-}cong}\ (\cdot\mathit{\text{-}cong}\ \lambda\ \{\ .\mathsf{force} \to P{\sim}Q\ \}))\ \mathit{|\text{-}left\text{-}identity}$$

One can use equational reasoning combinators [Norell 2007] to (perhaps) make proofs like this easier to read:

$$\_\blacksquare \quad : \forall\ \{i\}\ P \to [\ i\ ]\ P \sim P$$
$$\_{\sim}\langle\_\rangle\_ : \forall\ \{i\}\ P\ \{Q\ R\} \to [\ i\ ]\ P \sim Q \to [\ i\ ]\ Q \sim R \to [\ i\ ]\ P \sim R$$

We get the following proof instead (_■ binds tighter than _~⟨_⟩_, which associates to the right):

$$P{\sim}Q : \forall\ \{i\ n\} \to [\ i\ ]\ P\ n \sim Q$$
$$P{\sim}Q\ \{n = n\} =$$
$$\begin{array}{ll} P\ n & {\sim}\langle\ \mathit{Restricted{\sim}\emptyset}\ \mathit{|\text{-}cong}\ (\cdot\mathit{\text{-}cong}\ \lambda\ \{\ .\mathsf{force} \to P{\sim}Q\ \})\ \rangle \\ \emptyset \mid Q & {\sim}\langle\ \mathit{|\text{-}left\text{-}identity}\ \rangle \\ Q & \blacksquare \end{array}$$

There are several observations to make about this proof. First, note that if transitivity had not been size-preserving in its first argument, i.e. if the first argument had 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.

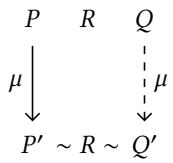
Secondly, if the argument of ·-*cong* had not been of the primed variant, then the corecursive call to $P{\sim}Q$ would not have taken place under a copattern, the definition would not have been strongly normalising, and the code would yet again have been rejected. This aspect of ·-*cong* is related to the fact that, for (at least some forms of) CCS, equations of the form $[\ \infty\ ]\ P \sim (C\ [\ P\ ])$ have unique solutions up to bisimilarity for contexts C where every hole is weakly guarded [Milner 1989].

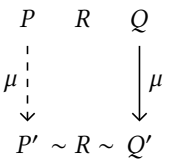
Finally, those who are familiar with up-to techniques may recognise a connection between the use of transitivity and "up to bisimilarity". Similarly, one may notice a connection between the use of the preservation lemmas and "up to context".

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 two processes are bisimilar by finding some bisimulation R that relates them, as discussed in Section 4. However, it may be easier to find, say, a bisimulation up to bisimilarity (here $\sim R \sim$ stands for the composition of three binary relations):





It turns out that this suffices: if two processes are related by a bisimulation up to bisimilarity, then they are bisimilar. Furthermore a bisimulation up to bisimilarity can be considerably smaller than a corresponding bisimulation: Pous and Sangiorgi [2011] present two processes that can be proved to be bisimilar using a bisimulation up to bisimilarity with a single pair, whereas a plain bisimulation would have to be infinitely large.

This argument can be made more abstract. For a monotone function C on a complete lattice one can prove that $X \leq \nu C$ by finding some R such that $X \leq R \leq C\ R$ and then using coinduction. However, constructing such an R may be complicated, and it may be easier to construct some R satisfying $X \leq R \leq C\ (F\ R)$ for some function F. If this always implies that $X \leq \nu C$, i.e. if $R \leq C\ (F\ R)$ implies $R \leq \nu C$ for any R, then F is a (sound) up-to technique. (I do not require F to be monotone.)

In the setting used in this paper I choose to use the following definition of up-to technique, where I have fixed an index type X and a container $C : \mathit{Container}\ X$:

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

Here the abbreviation *Trans*, standing for "relation transformer", is defined in the following way:

$$\mathit{Trans} : \mathit{Set}_1$$
$$\mathit{Trans} = (X \to \mathit{Set}) \to (X \to \mathit{Set})$$

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 at least parts of this class in the following way:

$$\mathit{Size\text{-}preserving} : \mathit{Trans} \to \mathit{Set}_1$$
$$\mathit{Size\text{-}preserving}\ F = \forall\ \{R\ i\} \to R \subseteq \nu\ C\ i \to F\ R \subseteq \nu\ C\ i$$

The idea is that if we are in the process of showing that a family is below a certain stage $\nu\ C\ i$ of the greatest fixpoint of C, and we have showed that R 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:

$$\_\odot\_ : \{X : \mathit{Set}\} \to (X \times X \to \mathit{Set}) \to (X \times X \to \mathit{Set}) \to (X \times X \to \mathit{Set})$$
$$(R \odot S)\ (x\ ,\ z) = \exists\ \lambda\ y \to R\ (x\ ,\ y) \times S\ (y\ ,\ z)$$

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

$$\mathit{Up\text{-}to\text{-}bisimilarity} : (\mathsf{Proc} \times \mathsf{Proc} \to \mathit{Set}) \to (\mathsf{Proc} \times \mathsf{Proc} \to \mathit{Set})$$
$$\mathit{Up\text{-}to\text{-}bisimilarity}\ R = \nu\ B^{\mathrm{C}}\ \infty \odot R \odot \nu\ B^{\mathrm{C}}\ \infty$$

This technique is size-preserving:

$$\begin{aligned}
&\mathit{up\text{-}to\text{-}bisimilarity\text{-}size\text{-}preserving} : \mathit{Size\text{-}preserving}\ \mathit{Up\text{-}to\text{-}bisimilarity}\\
&\mathit{up\text{-}to\text{-}bisimilarity\text{-}size\text{-}preserving}\ R{\subseteq}\nu B^{\mathrm{C}} i\ \{(P_1\ ,\ P_4)\}\ (P_2\ ,\ P_1{\sim}P_2\ ,\ P_3\ ,\ P_2RP_3\ ,\ P_3{\sim}P_4) =\\
&\quad P_1\ \ {\sim}\langle\ P_1{\sim}P_2\ \rangle\\
&\quad P_2\ \ {\sim}\langle\ R{\subseteq}\nu B^{\mathrm{C}} i\ P_2RP_3\ \rangle\\
&\quad P_3\ \ {\sim}\langle\ P_3{\sim}P_4\ \rangle\\
&\quad P_4\ \ \blacksquare
\end{aligned}$$

Note the three uses of transitivity. The proofs $P_1{\sim}P_2$ and $P_3{\sim}P_4$ have size ∞, and $R{\subseteq}\nu B^{\mathrm{C}} i\ P_2RP_3$ 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 for CCS 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 the 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⊆νB^Ci (P′ , Q′ , refl , P′RQ′ , refl) =
  !-cong (R⊆νB^Ci 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
```

(The other direction also holds, see the accompanying code; this proof uses the size successor function.) It is then easy to complete the proof:

```
size-preserving→up-to : ∀ {F} → Size-preserving F → Up-to-technique 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 $[\![C]\!]\ (\nu'\ C\ i)$.

Practically useful up-to techniques are often monotone:

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

For a 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
  }
```

($X \Leftrightarrow Y$ is a record type with two fields, to : $X \to Y$ and from : $Y \to X$.) 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 (in both cases the statements apply to at least one container; see the accompanying code for details).

A problem with the full class of up-to techniques, *Up-to-technique*, is that it is not closed under composition: for at least one container C 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
```

This implies that, for at least one container, it is not the case that all up-to techniques are size-preserving. In fact, one can construct explicit examples of up-to techniques that are not size-preserving, see the accompanying code. Another class of up-to techniques, that of monotone and *compatible* transformers (see the next section), satisfies the property that if a "symmetric" transformer is in the class for a container corresponding to strong similarity for some LTS, then it is in the class for a container corresponding to strong bisimilarity [Pous and Sangiorgi 2011]. This property does not hold in general for *Size-preserving*, not even for monotone transformers (again, see the accompanying code).

7 THE COMPANION

After I had come up with the class of size-preserving functions described in the previous section 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 C on a complete lattice. A monotone function F is *C-compatible* if $F \circ C \leq C \circ F$ (here the ordering is that of the lattice of monotone functions on the underlying lattice, ordered pointwise). Pous defines the companion of C as the least upper bound of all C-compatible (and monotone) functions, and notes that this is the greatest C-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 lists further useful properties of the companion.

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

$$\lambda X. \bigwedge_{X \leq C_\alpha} C_\alpha,$$

where C_α is defined by transfinite recursion on ordinals in the following way: $C_{\alpha+1} = C\ C_\alpha$, and $C_\lambda = \bigwedge_{\alpha<\lambda} C_\alpha$ 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 $F\ C_\alpha \leq C_\alpha$ for all ordinals α.

Note the similarity between the definition of C_α and the definition of the greatest fixpoint $\nu\ C\ i$ using quantification over $Size{<}\ i$. 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 C_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:

```
companion₁⊆companion : ∀ {R} → Companion₁ R ⊆ Companion R
companion₁⊆companion (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:[1]

```
companion-compatible⇔companion⊆companion₁ :
  Compatible Companion ⇔ (∀ {R} → Companion R ⊆ Companion₁ R)
companion-compatible⇔companion⊆companion₁ = 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 {_} → companion₁⊆companion (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. By assuming that a form of excluded middle holds, as well as some properties of sizes, I could prove that the small companion is compatible following Parrow and Weber (see the accompanying code for details). However, I do not know if these assumptions are consistent with the type theory that is used.

Pous and Rot [2017] seemingly also work in classical logic, but Pous (personal communication) suggests that the results can be ported to Coq by using its impredicative features and tower induction [Schäfer and Smolka 2017]. Perhaps *Compatible Companion* is a property that—for a fixed but arbitrary pair of an index type X and a container C : *Container* X—can neither be proved nor disproved in predicative, constructive type theory. (It may be worth noting that, due to 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* ∘ *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 ⊥ is the empty type):

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

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

```
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-technique Companion
companion-up-to = size-preserving→up-to companion-size-preserving
```

8 REPLICATION PRESERVES STRONG BISIMILARITY

Let us now see how one can implement !-*cong*, a proof showing 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",[2] and uses Coq to prove that something like the transformer *Up-to-*! defined in Section 6 is below the companion. When I first implemented !-*cong* I did not know that up-to techniques for replication had been seen as challenging, and in the setting of this paper the proof 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:

```
6-1-3-2 :
  ∀ {P μ R} →
  ! P [ μ ]↦ R →
  (∃ λ P′ → P [ μ ]↦ P′ × [ ∞ ] R ∼ ! P | P′)
    ⊎
  (μ ≡ τ × ∃ λ P′ → ∃ λ P″ → ∃ λ a →
    P [ name a ]↦ P′ × P [ name (co a) ]↦ P″ × [ ∞ ] R ∼ (! P | P′) | P″)
```

The lemma states that, if ! P can make a μ-transition to R, then at least one of two properties holds, 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:

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

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

[2] However, Hirschkoff [1997] discusses a formalisation of a π-calculus with replication in Coq, including a proof showing that a variant of up to context is respectful (and thus a sound up-to technique).

```
!-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.

composition. (The **with** construction is used to pattern match on the result of the application of *6-1-3-2*, and inj_1 and inj_2 are the two constructors of the binary sum data type, _⊎_.)

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 [Capretta 2005], which identifies computations

that terminate with the same value, even if the number of later constructors differ. Let us fix the carrier type A in this section. Weak bisimilarity can then be defined by adding two constructors to the definition of strong bisimilarity [Danielsson and Altenkirch 2010; Danielsson 2012]:

```
mutual

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

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

The new constructors later$^{\text{l}}$ and later$^{\text{r}}$ 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 holds between this definition and the definition of weak bisimilarity presented by Capretta [2005, Definition 3.5], 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. Observe that these proofs are not size-preserving:

```
later⁽ʳ⁻¹⁾ : ∀ {i} {j : Size< i} {x : Delay A ∞} {y} →
              [ i ] x ≈D later y → [ j ] x ≈D force y
later⁽ʳ⁻¹⁾ (later  p) = later⁽ˡ⁾ (force p)
later⁽ʳ⁻¹⁾ (later⁽ʳ⁾ p) = p
later⁽ʳ⁻¹⁾ (later⁽ˡ⁾ p) = later⁽ˡ⁾ (later⁽ʳ⁻¹⁾ p)

later⁽ˡ⁻¹⁾ : ∀ {i} {j : Size< i} {x} {y : Delay A ∞} →
              [ i ] later x ≈D y → [ j ] force x ≈D y
later⁽ˡ⁻¹⁾ (later  p) = later⁽ʳ⁾ (force p)
later⁽ˡ⁻¹⁾ (later⁽ʳ⁾ p) = later⁽ʳ⁾ (later⁽ˡ⁻¹⁾ p)
later⁽ˡ⁻¹⁾ (later⁽ˡ⁾ p) = p

later⁻¹ : ∀ {i} {j : Size< i} {x y : Delay′ A ∞} →
            [ i ] later x ≈D later y → [ j ] force x ≈D force y
later⁻¹ (later  p) = force p
later⁻¹ (later⁽ʳ⁾ p) = later⁽ˡ⁻¹⁾ p
later⁻¹ (later⁽ˡ⁾ p) = later⁽ʳ⁻¹⁾ p
```

It is now easy to prove transitivity if the first computation is now x for some x:

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

The proof uses structural recursion. Note that the second argument is assumed to have size ∞.

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 : ∀ {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 : ∀ {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 ∞. If the proof had been size-preserving in both arguments, then we could have proved that weak bisimilarity is trivial [de Vries 2009; Danielsson and Altenkirch 2010]:

```
size-preserving→trivial :
  (∀ {i} x {y z : Delay A ∞} → [ i ] x ≈ᴰ y → [ i ] y ≈ᴰ z → [ i ] x ≈ᴰ z) →
  ∀ {i} (x y : Delay A ∞) → [ i ] x ≈ᴰ y
size-preserving→trivial _≈⟨_⟩_ x y =
  (x                          ≈⟨ later^r (x ■^w) ⟩
  (later (λ { .force → x })   ≈⟨ later (λ { .force → size-preserving→trivial _≈⟨_⟩_ x y }) ⟩
  (later (λ { .force → y }))  ≈⟨ later^l (y ■^w) ⟩
  (y                          ■^w))))
```

Here the first argument is bound to a name which is a mixfix operator (_≈⟨_⟩_), in order to enable the use of equational reasoning notation in the body of the function, and _■^w is reflexivity for weak bisimilarity. We can also prove that now x is not weakly bisimilar to *never* (at any size):

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

By combining these results we get that, if transitivity is size-preserving in both arguments, then the carrier type A is uninhabited:

```
not-size-preserving^w :
  (∀ {i} x {y z : Delay A ∞} → [ i ] x ≈ᴰ y → [ i ] y ≈ᴰ z → [ i ] x ≈ᴰ z) → ¬ A
not-size-preserving^w trans x =
  now≉never (size-preserving→trivial trans (now x) never)
```

Thus, if the carrier type is inhabited, then a fully size-preserving transitivity proof cannot be implemented. This result is related to the problem of weak bisimulation up to weak bisimilarity [Sangiorgi and Milner 1992], because a size-preserving transitivity proof could be used to prove that weak bisimulations up to weak bisimilarity are contained in weak bisimilarity (and the problem is that they are, in general, not).

Let us now generalise the counterexample above and 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
Drop-later = ∀ {i} {x : A} → [ i ] later (λ { .force → now x }) ∼ᴰ never →
                             [ i ] now x                       ≈ᴰ never
```

If we assume that *Drop-later* and A are both inhabited, then we can derive a contradiction. 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 : Drop-later → ¬ 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:

```
not-size-preserving^ws :
  (∀ {i} {x y z : Delay A ∞} → [ ∞ ] x ≈ᴰ y → [ i ] y ∼ᴰ z → [ i ] x ≈ᴰ z) → ¬ A
not-size-preserving^ws trans = basic-counterexample (λ p → trans (later^r now) p)
```

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

```
strong-to-weak : ∀ {i} {x y : Delay A ∞} → [ i ] x ∼ᴰ y → [ i ] x ≈ᴰ y
strong-to-weak now       = now
strong-to-weak (later p) = later λ { .force → strong-to-weak (force p) }
```

```
not-size-preserving^wr :
  (∀ {i} {x y z : Delay A ∞} → [ ∞ ] x ≈D y → [ i ] y ≈D z → [ i ] x ≈D z) → ¬ A
not-size-preserving^wr trans =
  not-size-preserving^ws (λ p q → trans p (strong-to-weak q))
```

(For any LTS there are size-preserving translations from strong bisimilarity to expansion and from expansion to weak bisimilarity, see the accompanying code.) 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.

As an aside one can also prove that, if A is not inhabited, then *Drop-later* 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:

```
trivial : ∀ {i} → ¬ A → ∀ x y → [ i ] x ≈D y
trivial empty (now x) _          = ⊥-elim (empty x)
trivial empty (later x) (now y)  = ⊥-elim (empty y)
trivial empty (later x) (later y) = later λ { .force → trivial empty (force x) (force y) }
```

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:

```
transitive^sw : ∀ {i} {x y z : Delay A ∞} → [ ∞ ] x ∼D y → [ i ]  x ≈D y → [ i ] y ≈D z → [ i ] x ≈D z
transitive^ws : ∀ {i} {x y z : Delay A ∞} → [ i ]  x ≈D y → [ ∞ ] y ∼D z → [ i ] x ≈D z
```

We get stronger results by instead working with the expansion relation. Expansion for the delay monad, here called $[\_]\_\gtrsim_D\_$, can be defined by removing the later^r constructor from the definition of weak bisimilarity [Danielsson 2012]. This definition is pointwise logically equivalent (in a size-preserving way) to the definition of expansion presented in Section 4, instantiated with the LTS for the delay monad from Section 4. The following results can be proved for this definition of expansion, and in fact for expansion for an arbitrary LTS (see the accompanying code):

```
transitive^es : ∀ {i} {x y z : Delay A ∞} → [ i ]  x ≳D y → [ i ]  y ∼D z → [ i ] x ≳D z
transitive^e  : ∀ {i} {x y z : Delay A ∞} → [ ∞ ] x ≳D y → [ i ]  y ≳D z → [ i ] x ≳D z
transitive^ew : ∀ {i} {x y z : Delay A ∞} → [ ∞ ] x ≳D y → [ i ]  y ≈D z → [ i ] x ≈D z
transitive^we : ∀ {i} {x y z : Delay A ∞} → [ i ]  x ≈D y → [ ∞ ] z ≳D y → [ i ] x ≈D z
```

(Note that the last argument of *transitive*^{we} has type $[\ \infty\ ]\ z \gtrsim_D y$, with z before y.) By building on *basic-counterexample* one can also prove negative results, for instance the following ones:

```
not-size-preserving^se :
  (∀ {i} {x y z : Delay A ∞} → [ i ] x ∼D y → [ ∞ ] y ≳D z → [ i ] x ≳D z) → ¬ A

not-size-preserving^ew :
  (∀ {i} {x y z : Delay A ∞} → [ i ] x ≳D y → [ ∞ ] y ≈D z → [ i ] x ≈D z) → ¬ A

not-size-preserving^we :
  (∀ {i} {x y z : Delay A ∞} → [ i ] x ≈D y → [ ∞ ] y ≳D z → [ i ] x ≈D z) → ¬ A
```

Note that the first two of these results imply that *transitive*^{e}, *transitive*^{ew} and *transitive*^{we} cannot in general be made fully size-preserving.

Let us now compare with some up-to techniques, beginning with up to expansion [Pous and Sangiorgi 2011]:

If R satisfies the diagrams above, then R is contained in weak bisimilarity. It is easy to prove that up to expansion is size-preserving (and thus an up-to technique) by using lemmas corresponding to *transitive*$^{\text{ew}}$ and *transitive*$^{\text{we}}$ (but stated for an arbitary LTS). In applications it is perhaps preferable to use such lemmas directly, like variants of the size-preserving transitivity proof for strong bisimilarity are used in the implementation of !-*cong* in Figure 3.

Consider now instead the variant of up to expansion that you get by replacing $\gtrsim R \lesssim$ by $\gtrsim R \approx$ in the left diagram, and $\approx R \lesssim$ in the right diagram [Pous and Sangiorgi 2011]. This *non-symmetric* up-to technique does not fit easily into the framework for up-to techniques used above (see Pous and Sangiorgi [2011] for some discussion of non-symmetric up-to techniques), but it is the case that if R satisfies the modified diagrams, then R is contained in weak bisimilarity. However, it is less clear how to make use of size-preserving lemmas in this case, because there is in general no transitivity proof for weak bisimilarity that preserves the size of even one argument.

10 RELATED WORK

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

It was noted by de Vries [2009] that a certain transitive relation defined using mixed 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 *size-preserving→trivial* in Section 9.2. His observation was discussed further by Danielsson and Altenkirch [2010], and later Danielsson [2012] compared some size-preserving proofs to Sangiorgi and Milner's "expansion up to $\lesssim$" [1992]. Danielsson only discussed a small number of size-preserving functions, including proofs corresponding to *transitive*$^{\text{ew}}$ and *transitive*$^{\text{we}}$ from Section 9.2. (Danielsson's formalisation also includes a function corresponding to *transitive*$^{\text{e}}$ and a size-preserving symmetry proof for weak bisimilarity for the delay monad.)

Several previous works have used coinduction in the form of coinductive data types (or similar constructions) to define bisimilarity for process calculi. Hirschkoff [1997] presents a formalisation of a π-calculus in Coq. He mentions the possibility to define bisimilarity coinductively, but seems to mainly use the traditional definition using bisimulations. Honsell et al. [2001] encode strong late bisimilarity for a π-calculus with replication coinductively in Coq. They state that some of their corecursive proofs are in some sense easier than traditional proofs using bisimulations: "One does not need to produce in advance a bisimulation containing the two processes. Such a bisimulation is gradually built interactively during the proof in a way completely transparent for the user." However, they also note that the guardedness condition imposed by Coq can be restrictive. They prove that replication preserves strong late bisimilarity, but this proof does in part go via a traditional definition of bisimilarity. Chaudhuri et al. [2015] define strong bisimilarity for a variant of CCS with replication coinductively in Abella, and prove that a variant of up to context is sound, but they do not allow holes under replication.

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 many cases one can get away with using size-preserving symmetry proofs. For some examples, see the definitions of *transitive* in Figure 1, _|-*cong*_ in Figure 2, and !-*cong* in Figure 3.

Sangiorgi [2017] presents "contraction", a relation that is coarser than expansion and finer than weak bisimilarity. He uses this relation to give an up-to technique that is more general than the variant of up to expansion presented at the end of Section 9.2. He also discusses another method for establishing weak bisimilarity, based on unique solutions of inequations involving contractions. This technique is a variant of a unique solution theorem due to Milner [1989], which states that systems of guarded and sequential equations (using weak bisimilarity) of a certain form have unique solutions up to weak bisimilarity. One of Sangiorgi's theorems states that systems of *weakly* guarded, not necessarily sequential, contractions of a certain form have unique solutions up to weak bisimilarity. (A variable is weakly guarded in a process if every occurrence is below some prefix, and guarded if every occurrence is below some *non-silent* prefix.)

As mentioned in Section 5 the proof $P{\sim}Q$ is reminiscent of uses of Milner's theorem for *strong* bisimilarity, which only requires weak guardedness. A key to making the proof $P{\sim}Q$ work was the fact that ·-*cong* takes the primed variant of bisimilarity as input. Given that proofs corresponding to ·-*cong* for expansion and weak bisimilarity also take the primed variant of bisimilarity as input it should be possible to construct proofs like $P{\sim}Q$ in those settings as well, as long as suitable size-preserving transitivity proofs are available (which may or may not be the case). Perhaps it would be useful to study the contraction relation in the setting of sized types, and work out what forms of transitivity and preservation lemmas are available for it.

The type of ·-*cong*, with the primed variant of bisimilarity, is reminiscent of Schäfer and Smolka's Lemma 17 [2017]. This lemma states (for one variant of CCS) that if P and Q are in the "open relative bisimilarity" relation for some relation R, $P \mathrel{\dot{\sim}_R} Q$, then we have $\mu.P \mathrel{\dot{\sim}_{B\ (\mathit{Companion}\ R)}} \mu.Q$, where B is the monotone function used to define strong bisimilarity, νB, and the notion of companion used is proved to match that due to Pous [2016].

Guarded recursion in the style of Nakano [2000] is an alternative to sized types: both formalisms use types to ensure that recursive definitions are well-defined. Bizjak et al. [2016], Birkedal et al. [2016], and Bahr et al. [2017] present type theories with support for guarded recursion, building on work by Atkey and McBride [2013]. I am not aware of any work investigating up-to techniques in this setting, but perhaps such an investigation would be fruitful.

11 CONCLUSIONS

I hope that this text gives some idea of what up-to techniques can look like in a setting with sized types. My experience is that things often "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 a more objective justification of the utility of sized types. It can also serve as a hint about techniques to use when porting code from a language with sized types to one without.

ACKNOWLEDGEMENTS

I would like to thank Andreas Abel, Damien Pous and Davide Sangiorgi for informing me about related work, and Andreas Abel and Andrea Vezzosi for helping me with details related to sized types. I would also like to thank the anonymous reviewers for useful feedback.

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

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 2017-11-07 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

S. Arun-Kumar and M. Hennessy. 1992. An efficiency preorder for processes. *Acta Informatica* (1992). https://doi.org/10.1007/BF01191894

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *ICFP'13, Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming*. https://doi.org/10.1145/2500365.2500597

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The Clocks Are Ticking: No More Delays! Reduction Semantics for Type Theory with Guarded Recursion. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.2017.8005097

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$^{\widehat{}}$: 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. (2017). Accepted for publication in the proceedings of the 7th Conference on Algebra and Coalgebra in Computer Science (CALCO 2017). Possible future DOI: https://doi.org/10.4230/LIPIcs.CALCO.2017.5

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *Computer Science Logic 2016, CSL 2016*. https://doi.org/10.4230/LIPIcs.CSL.2016.23

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures, 19th International Conference, FOSSACS 2016*. https://doi.org/10.1007/978-3-662-49630-5_2

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

Kaustuv Chaudhuri, Matteo Cimini, and Dale Miller. 2015. A Lightweight Formalization of the Metatheory of Bisimulation-Up-To. In *CPP'15, Proceedings of the 2015 ACM Conference on Certified Programs and Proofs*. https://doi.org/10.1145/2676724.2693170

Thierry Coquand. 1994. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES '93*. https://doi.org/10.1007/3-540-58085-9_72

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. 1996. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. Ph.D. Dissertation. Ecole Normale Supérieure de Lyon.

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

Daniel Hirschkoff. 1997. A Full Formalisation of π-Calculus Theory in the Calculus of Constructions. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs '97*. https://doi.org/10.1007/BFb0028392

Furio Honsell, Marino Miculan, and Ivan Scagnetto. 2001. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science* (2001). https://doi.org/10.1016/S0304-3975(00)00095-5

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. 1983. Calculi for synchrony and asynchrony. *Theoretical Computer Science* (1983). https://doi.org/10.1016/0304-3975(83)90114-7

Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*. https://doi.org/10.1109/LICS.2000.855774

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 2017-11-07 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. 2017. Equations, Contractions, and Unique Solutions. *ACM Transactions on Computational Logic* (2017). https://doi.org/10.1145/2971339

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