Up-to Techniques Using Sized Types

Nils Anders Danielsson

IFIP WG 2.1 Meeting #76,
Lesbos, 2017-10-17

When using a type theory with sized types to define bisimilarity a useful class of up-to techniques falls out naturally.

The traditional approach

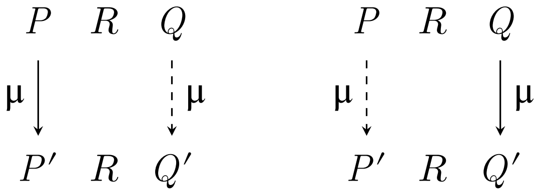
The traditional approach

Traditional coinduction:

- F: A monotone function on a complete lattice.
- νF: Its greatest post-fixpoint.
- Coinduction: $R \leq F\,R$ implies $R \leq \nu F$.

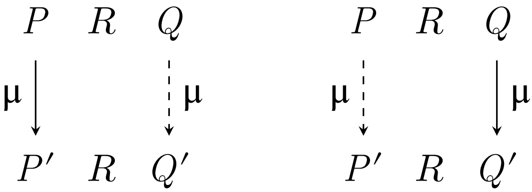
The traditional approach

R is a bisimulation:



The traditional approach

R is a bisimulation:



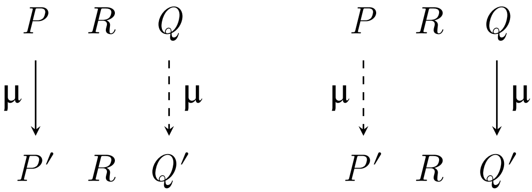
Can be turned into a monotone function:

$$B\,R = \{\ (P, Q) \mid \ldots\ \}$$

R is a bisimulation iff $R \subseteq B\,R$.

The traditional approach

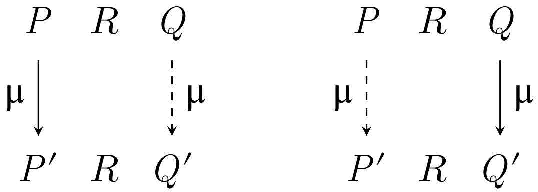
R is a bisimulation:



Bisimilarity: $P \sim Q$ if $(P, Q) \in \nu B$.

The traditional approach

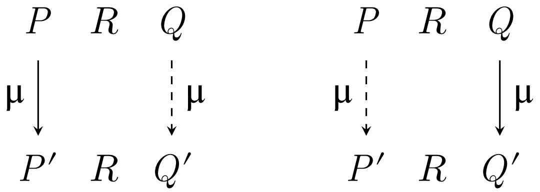
R is a bisimulation:



Coinduction: $R \subseteq B\,R$ implies $R \subseteq \nu B$.

The traditional approach

R is a bisimulation:



Coinduction: $R \subseteq B\,R$ implies $R \subseteq \nu B$.

Up-to techniques are used to make proofs easier.

G is an up-to technique if
$R \subseteq B\,(G\,R)$ implies $R \subseteq \nu B$ (for all R).

The traditional approach

R is a bisimulation:

$$
\begin{array}{ccc}
P & R & Q \\
\mu \downarrow & & \downarrow \mu \\
P' & R & Q'
\end{array}
\qquad\qquad
\begin{array}{ccc}
P & R & Q \\
\mu \downarrow & & \downarrow \mu \\
P' & R & Q'
\end{array}
$$

R is a bisimulation up to bisimilarity:

$$
\begin{array}{ccc}
P & R & Q \\
\mu \downarrow & & \downarrow \mu \\
P' \sim & R & \sim Q'
\end{array}
\qquad\qquad
\begin{array}{ccc}
P & R & Q \\
\mu \downarrow & & \downarrow \mu \\
P' \sim & R & \sim Q'
\end{array}
$$

Coinductive data types

Coinduction without sized types

The delay monad, roughly $\nu X.\ A + X$:

```
mutual

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

  record Delay′ (A : Set) : Set where
    coinductive
    field force : Delay A
```

Corecursion using copatterns

never $\approx$ later (later (later (...))):

mutual

never : $\forall$ $\{A\}$ $\rightarrow$ Delay A
never = later never′

never′ : $\forall$ $\{A\}$ $\rightarrow$ Delay′ A
force never′ = never

Corecursion using copatterns

never ≈ later (later (later (...))):

```
never : ∀ {A} → Delay A
never = later (λ { .force → never })
```

Corecursion using copatterns

never $\approx$ later (later (later (...))):

```
never : ∀ {A} → Delay A
never = later (λ { .force → never })
```

Guarded, productive.

Guardedness

Not guarded, rejected:

$$\text{unfold} : \forall \{X\ A\} \to (X \to A + X) \to X \to \text{Delay}\ A$$
$$\text{unfold}\ f = \text{in}_{\text{D}} \circ \text{map}\ (\lambda\ x \to \lambda\ \{\ .\text{force} \to \text{unfold}\ f\ x\ \}) \circ f$$

$$\text{in}_{\text{D}} : \forall \{A\} \to A + \text{Delay}'\ A \to \text{Delay}\ A$$

$$\text{map} : \{X\ Y\ A : \text{Set}\} \to (X \to Y) \to A + X \to A + Y$$

Sized types

Sized types

The delay monad:

```
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
```

Sized types

- Sizes can be thought of as ordinals.
- $\mathsf{Delay}'\ A\ i$: Partially defined values.
- Deflationary iteration:

$$\mathsf{Delay}'\ A\ i \approx \bigcap_{j<i} A + \mathsf{Delay}'\ A\ j$$

- ∞: Closure ordinal.
- $\mathsf{Delay}'\ A\ \infty$: Fully defined values.

Sized types

The size is smaller in every corecursive call:

$$
\begin{aligned}
&\textsf{unfold} : \forall\ \{X\ \ A\ \ i\} \rightarrow \\
&\qquad (X \rightarrow A + X) \rightarrow X \rightarrow \textsf{Delay}\ A\ \ i \\
&\textsf{unfold}\ f\ = \\
&\quad \textsf{in}_{\textsf{D}} \circ \textsf{map}\ (\lambda\ x \rightarrow \lambda\ \{\ .\textsf{force} \rightarrow \textsf{unfold}\ f\ \ x\ \}) \circ f
\end{aligned}
$$

Sized types

The size is smaller in every corecursive call:

$$
\begin{aligned}
&\mathsf{unfold} : \forall\ \{X\ A\ i\} \to \\
&\qquad (X \to A + X) \to X \to \mathsf{Delay}\ A\ i \\
&\mathsf{unfold}\ f = \\
&\quad \mathsf{in}_{\mathsf{D}} \circ \\
&\quad \mathsf{map}\ (\lambda\ x \to \lambda\ \{\ .\mathsf{force} \to \\
&\qquad\qquad\qquad \mathsf{unfold}\ f\ x\ \}) \circ \\
&\quad f
\end{aligned}
$$

Sized types

The size is smaller in every corecursive call:

```
unfold : ∀ {X A i} →
           (X → A + X) → X → Delay A i
unfold {i = i} f =
  in_D ∘
  map (λ x → λ { .force {j = j} →
                        unfold {i = j} f x }) ∘
  f
```

Greatest post-fixpoints

Index-preserving functions

Functions that preserve the index:

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

Containers

- Indexed containers,
  representing strictly positive functors:

  $$\text{Container} : \text{Set} \to \text{Set}_1$$

- Interpretation:

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

- Map function:

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

Greatest post-fixpoints

mutual

$\nu : \forall \; \{X\} \to \mathsf{Container}\; X \to \mathsf{Size} \to (X \to \mathsf{Set})$
$\nu \; C \; i = [\![\; C \;]\!] \; (\nu' \; C \; i)$

record ν' $\{X\} \; (C : \mathsf{Container}\; X) \; (i : \mathsf{Size})$
$(x : X) : \mathsf{Set}$ where
coinductive
field force : $\{j : \mathsf{Size}\!< \; i\} \to \nu \; C \; j \; x$

Greatest post-fixpoints

$$
\begin{aligned}
&\mathsf{out} : \forall\ \{X\}\ (C : \mathsf{Container}\ X) \rightarrow \\
&\qquad \boldsymbol{\nu}\ C\ \infty \subseteq [\![\ C\ ]\!]\ (\boldsymbol{\nu}\ C\ \infty) \\
&\mathsf{out}\ C = \mathsf{map}\ C\ (\lambda\ x \rightarrow \mathsf{force}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unfold} : \forall\ \{X\ A\ i\}\ (C : \mathsf{Container}\ X) \rightarrow \\
&\qquad A \subseteq [\![\ C\ ]\!]\ A \rightarrow A \subseteq \boldsymbol{\nu}\ C\ i \\
&\mathsf{unfold}\ C\ f = \\
&\quad \mathsf{map}\ C\ (\lambda\ a \rightarrow \lambda\ \{\ .\mathsf{force} \rightarrow \mathsf{unfold}\ C\ f\ a\ \})\circ f
\end{aligned}
$$

CCS

CCS

A variant of a fragment of CCS:

```
data Label : Set where
  • : Label
```

CCS

A variant of a fragment of CCS:

```
mutual

  data Proc : Set where
    ∅   : Proc
    _|_ : Proc → Proc → Proc
    •   : Proc′ →         Proc

  record Proc′ : Set where
    coinductive
    field force : Proc
```

CCS

A variant of a fragment of CCS:

$$\mathsf{data}\ \_[\_]\!\rightarrow\!\_ : \mathsf{Proc} \rightarrow \mathsf{Label} \rightarrow \mathsf{Proc} \rightarrow \mathsf{Set}\ \mathsf{where}$$

$$\mathsf{action} : \forall\ \{P\} \rightarrow \bullet\ P\ [\ \bullet\ ]\!\rightarrow \mathsf{force}\ P$$

$$\mathsf{par\text{-}left} : \forall\ \{P\ P'\ Q\ \mu\} \rightarrow P\ [\ \mu\ ]\!\rightarrow P' \rightarrow P \mid Q\ [\ \mu\ ]\!\rightarrow P' \mid Q$$

$$\mathsf{par\text{-}right} : \forall\ \{P\ Q\ Q'\ \mu\} \rightarrow Q\ [\ \mu\ ]\!\rightarrow Q' \rightarrow P \mid Q\ [\ \mu\ ]\!\rightarrow P \mid Q'$$

Bisimilarity

R is a bisimulation:

$$\begin{array}{ccc} P & R & Q \\ \mu\big\downarrow & & \big\downarrow\mu \\ P' & R & Q' \end{array} \qquad\qquad \begin{array}{ccc} P & R & Q \\ \mu\big\downarrow & & \big\downarrow\mu \\ P' & R & Q' \end{array}$$

Bisimilarity

R is a bisimulation iff $R \subseteq \mathsf{B}\ R$:

```
record B (R : Proc × Proc → Set)
          (PQ : Proc × Proc) : Set where
  field
    left-to-right :
      ∀ {μ P′} → fst PQ  [ μ ]→ P′ →
      ∃ λ Q′ → snd PQ [ μ ]→ Q′ × R (P′ , Q′)

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

Bisimilarity

- B can also be defined as a container.
- Bisimilarity:

$$
\begin{array}{l}
[\_]\_\sim\_ : \mathsf{Size} \to \mathsf{Proc} \to \mathsf{Proc} \to \mathsf{Set} \\
[\ i\ ]\ P \sim Q = \nu\ \mathsf{B}\ i\ (P\ ,\ Q)
\end{array}
$$

$$
\begin{array}{l}
[\_]\_\sim'\_ : \mathsf{Size} \to \mathsf{Proc} \to \mathsf{Proc} \to \mathsf{Set} \\
[\ i\ ]\ P \sim' Q = \nu'\ \mathsf{B}\ i\ (P\ ,\ Q)
\end{array}
$$

Examples

Examples

∅ is a left and right identity of parallel composition:

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

right-to-left ∅-left-identity tr =
  (_ , par-right tr , λ { .force → ∅-left-identity })

∅-right-identity : ∀ {i P} → [ i ] P | ∅ ∼ P
-- Similarly.
```

Examples

Prefixing preserves bisimilarity:

$$
\begin{aligned}
&\bullet\text{-cong} : \forall\ \{i\ P\ Q\} \to \\
&\qquad [\ i\ ]\ \text{force}\ P \sim'\ \text{force}\ Q \to \\
&\qquad [\ i\ ]\ \bullet\ P \sim \bullet\ Q \\
&\text{left-to-right}\ (\bullet\text{-cong}\ p)\ \text{action} = (\_\ ,\ \text{action}\ ,\ p) \\
&\text{right-to-left}\ (\bullet\text{-cong}\ p)\ \text{action} = (\_\ ,\ \text{action}\ ,\ p)
\end{aligned}
$$

Note that the proof is size-preserving.

Examples

Bisimilarity is symmetric and transitive:

$$\textsf{sym} \;:\; \forall\ \{i\ P\ Q\} \rightarrow [\ i\ ]\ P \sim Q \rightarrow [\ i\ ]\ Q \sim P$$

$$\textsf{trans} \;:\; \forall\ \{i\ P\ Q\ R\} \rightarrow [\ i\ ]\ P \sim Q \rightarrow [\ i\ ]\ Q \sim R \rightarrow [\ i\ ]\ P \sim R$$

Note that the proofs are size-preserving.

Examples

Two processes:

```
P Q : Proc
P = ∅                              | • (λ { .force → P })
Q = • (λ { .force → Q }) | ∅
```

P and Q are bisimilar:

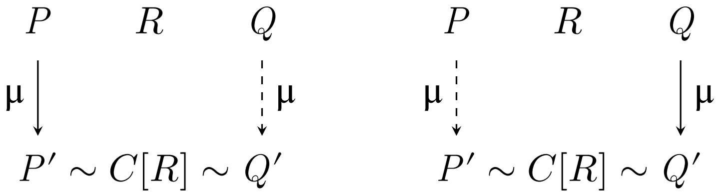
```
P∼Q : ∀ {i} → [ i ] P ∼ Q
P∼Q = trans ∅-left-identity (
        trans (•-cong λ { .force → P∼Q })
              (sym ∅-right-identity))
```

Examples

P and Q are bisimilar:

```
P∼Q : ∀ {i} → [ i ] P ∼ Q
P∼Q = trans ∅-left-identity (
        trans (•-cong λ { .force → P∼Q })
              (sym ∅-right-identity))
```

Compare to "up to context and bisimilarity":



Some further comments

- Pous has identified a useful class of up-to techniques: functions below the companion.
- This class seems to be closely related to size-preserving functions.

Some further comments

- Weak bisimulations up to weak bisimilarity are not in general contained in weak bisimilarity.
- Transitivity is not in general size-preserving for weak bisimilarity.

Conclusion

When using a type theory with sized types to define bisimilarity a useful class of up-to techniques falls out naturally.

Extra material

Containers

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

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

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