Up-to Techniques using Sized Types

Nils Anders Danielsson

POPL 2018, Los Angeles, 2018-01-11

Introduction

What is an up-to technique?

- Often used to make it easier to define bisimulations.

Introduction

What is an up-to technique?

- R is a bisimulation (for CCS):

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & R & Q \arrow[d, dashed, "\mu"] \\
P' & R & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & R & Q \arrow[d, "\mu"] \\
P' & R & Q'
\end{tikzcd}$$

Introduction

What is an up-to technique?

- Up to bisimilarity:

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & & R & & Q \arrow[d, dashed, "\mu"] \\
P' & \sim & R & \sim & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & & R & & Q \arrow[d, "\mu"] \\
P' & \sim & R & \sim & Q'
\end{tikzcd}$$

Introduction

What is an up-to technique?

- More generally:

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & R & Q \arrow[d, dashed, "\mu"] \\
P' & F\ R & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & R & Q \arrow[d, "\mu"] \\
P' & F\ R & Q'
\end{tikzcd}$$

Introduction

- In dependently typed languages:
  Bisimilarity as indexed coinductive data type.
- If sized types are used:
  A class of up-to techniques falls out naturally.

Coinductive data types

Coinduction without sized types

Potentially infinite lists, roughly $\nu X.\ 1 + A \times X$:

```
data Colist (A : Set) : Set where
  []   :                  Colist A
  _::_ : A → Colist′ A → Colist A

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

Corecursion using copatterns

A map function for colists:

$$\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{Colist}\ A \to \mathsf{Colist}\ B \\
&\mathsf{map}\ f\ [\,] \qquad\qquad = [\,] \\
&\mathsf{map}\ f\ (x :: xs) = f\ x :: \mathsf{map}'\ f\ xs
\end{aligned}$$

$$\begin{aligned}
&\mathsf{map}' : (A \to B) \to \mathsf{Colist}'\ A \to \mathsf{Colist}'\ B \\
&\mathsf{force}\ (\mathsf{map}'\ f\ xs) = \mathsf{map}\ f\ (\mathsf{force}\ xs)
\end{aligned}$$

Corecursion using copatterns

A map function for colists:

$$\begin{aligned} &\mathsf{map} : (A \to B) \to \mathsf{Colist}\ A \to \mathsf{Colist}\ B \\ &\mathsf{map}\ f\ [\,] \qquad\qquad\;\; = [\,] \\ &\mathsf{map}\ f\ (x :: xs) = f\ x :: \lambda\ \{\ .\mathsf{force} \to \mathsf{map}\ f\ (\mathsf{force}\ xs)\ \} \end{aligned}$$

Corecursion using copatterns

A map function for colists:

$$
\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{Colist}\ A \to \mathsf{Colist}\ B \\
&\mathsf{map}\ f\ [\,] &&= [\,] \\
&\mathsf{map}\ f\ (x :: xs) &&= f\ x :: \lambda\ \{\ .\mathsf{force} \to \mathsf{map}\ f\ (\mathsf{force}\ xs)\ \}
\end{aligned}
$$

Guarded, productive.

Guardedness

$0, 1, 2, \ldots$:

```
nats : Colist ℕ
nats = 0 ∷ λ { .force → map (1 +_) nats }
```

Not guarded, rejected.

Sized types

Sized types

Previous definition:

```
data Colist (A : Set) : Set where
  []   :                   Colist A
  _::_ : A → Colist′ A → Colist A

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

Sized types

With sized types:

```
data Colist (i : Size) (A : Set) : Set where
  []   :                   Colist i A
  _::_ : A → Colist′ i A → Colist i A

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

Sized types

With sized types:

```
data Colist (i : Size) (A : Set) : Set where
  []   :                       Colist i A
  _∷_  : A → Colist′ i A → Colist i A

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

Colist′ i A: Partially defined colists of depth at least i.

Sized types

With sized types:

```
data Colist (i : Size) (A : Set) : Set where
  []   :                    Colist i A
  _∷_  : A → Colist′ i A → Colist i A

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

`Colist′ ∞ A`: Fully defined colists.

Sized types

The map function is size-preserving:

```
map : ∀ {i} → (A → B) → Colist i A → Colist i B
map f []       = []
map f (x ∷ xs) = f x ∷ λ { .force → map f (force xs) }
```

Sized types

The size is smaller in every corecursive call:

```
nats : ∀ {i} → Colist i ℕ
nats = 0 :: λ { .force → map (1 +_) nats }
```

Sized types

The size is smaller in every corecursive call:

```
nats : ∀ i → Colist i ℕ
nats i = 0 :: λ { .force {j} → map (1 +_) (nats j) }
```

Bisimilarity

A tiny process calculus

Labels/actions:

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

A tiny process calculus

Processes:

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

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

Roughly $\nu C.\ \mu I.\ 1 + I \times I + C$.

A tiny process calculus

Transition relation:

```
data _[_]↦_ : Proc → Label → Proc → Set where
  action    : ● P [ ● ]↦ force P
  par-left  : P [ μ ]↦ P′ → P ∣ Q [ μ ]↦ P′ ∣ Q
  par-right : Q [ μ ]↦ Q′ → P ∣ Q [ μ ]↦ P  ∣ Q′
```

Bisimilarity

R is a bisimulation:

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & R & Q \arrow[d, dashed, "\mu"] \\
P' & R & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & R & Q \arrow[d, "\mu"] \\
P' & R & Q'
\end{tikzcd}$$

Bisimilarity

```
record Bisimilar (i : Size) (P Q : Proc) : Set where
  inductive
  field left-to-right :
            P [ μ ]↦ P′ → ∃ Q′ → Q [ μ ]↦ Q′ × Bisimilar′ i P′ Q′

        right-to-left :
            Q [ μ ]↦ Q′ → ∃ P′ → P [ μ ]↦ P′ × Bisimilar′ i P′ Q′

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

Examples

Bisimilarity is transitive:

$$\begin{aligned}\textsf{trans} : {} & \textsf{Bisimilar}\ i\ P\ Q \to \\ & \textsf{Bisimilar}\ i\ Q\ R \to \\ & \textsf{Bisimilar}\ i\ P\ R\end{aligned}$$

Note that the proof is size-preserving.

Examples

Parallel composition preserves bisimilarity:

$$
\begin{aligned}
\mid\text{-cong} : {} & \text{Bisimilar}\ i\ P\ P' \to \\
& \text{Bisimilar}\ i\ Q\ Q' \to \\
& \text{Bisimilar}\ i\ (P \mid Q)\ (P' \mid Q')
\end{aligned}
$$

Note that the proof is size-preserving.

Examples

Prefixing preserves bisimilarity:

$$\begin{aligned}\bullet\text{-cong} : {} & \text{Bisimilar}'\ i\ (\text{force}\ P)\ (\text{force}\ Q) \to \\ & \text{Bisimilar}\ \ i\ (\bullet\ P)\ (\bullet\ Q)\end{aligned}$$

Note that the proof is size-preserving.

Examples

$\emptyset$ is a left identity of parallel composition:

$$\emptyset\text{-left-identity} : \text{Bisimilar}\ i\ (\emptyset \mid P)\ P$$

Examples

Two processes:

$$\begin{aligned} &\mathsf{P}\ \mathsf{Q} : \mathsf{Proc} \\ &\mathsf{P} = \emptyset \mid (\bullet\ \mathsf{P}' \mid \bullet\ \mathsf{P}') \\ &\mathsf{Q} = \bullet\ \mathsf{Q}' \mid \bullet\ \mathsf{Q}' \end{aligned} \qquad \begin{aligned} &\mathsf{P}'\ \mathsf{Q}' : \mathsf{Proc}' \\ &\mathsf{force}\ \mathsf{P}' = \mathsf{P} \\ &\mathsf{force}\ \mathsf{Q}' = \mathsf{Q} \end{aligned}$$

Examples

P and Q are bisimilar:

```
sim : ∀ {i} → Bisimilar i P Q
sim = trans ∅-left-identity
            (|-cong (•-cong (λ { .force → sim }))
                    (•-cong (λ { .force → sim })))
```

```
P Q : Proc                 P′ Q′ : Proc′
P = ∅ | (• P′ | • P′)      force P′ = P
Q = • Q′ | • Q′            force Q′ = Q
```

Examples

P and Q are bisimilar:

```
sim : ∀ {i} → Bisimilar i P Q
sim = trans ∅-left-identity
                (∣-cong (•-cong (λ { .force → sim }))
                        (•-cong (λ { .force → sim })))
```

Compare trans and up to bisimilarity:

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & R & Q \arrow[d, dashed, "\mu"] \\
P' \arrow[r, phantom, "\sim"] & R \arrow[r, phantom, "\sim"] & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & R & Q \arrow[d, "\mu"] \\
P' \arrow[r, phantom, "\sim"] & R \arrow[r, phantom, "\sim"] & Q'
\end{tikzcd}$$

Examples

P and Q are bisimilar:

```
sim : ∀ {i} → Bisimilar i P Q
sim = trans ∅-left-identity
              (|-cong (•-cong (λ { .force → sim }))
                      (•-cong (λ { .force → sim })))
```

Compare |-cong/•-cong and up to context:

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & R & Q \arrow[d, dashed, "\mu"] \\
P' & C[R] & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[d, dashed, "\mu"'] & R & Q \arrow[d, "\mu"] \\
P' & C[R] & Q'
\end{tikzcd}$$

Examples

P and Q are bisimilar:

```
sim : ∀ {i} → Bisimilar i P Q
sim = trans ∅-left-identity
            (∣-cong (•-cong (λ { .force → sim }))
                    (•-cong (λ { .force → sim })))
```

- The proofs are size-preserving, so they can be combined freely.
- No extra work is required to show that the proofs are size-preserving.

Weak bisimilarity

- For full CCS: *Weak* bisimilarity.
- Size-preserving preservation lemmas.
- Transitivity:

  $$\begin{array}{l}\text{Weakly-bisimilar } \infty\ P\ \ Q \to \\ \text{Weakly-bisimilar } \infty\ Q\ \ R \to \\ \text{Weakly-bisimilar } \infty\ P\ \ R\end{array}$$

  Cannot be proved in a size-preserving way.
- The problem of "weak bisimulation up to".

Generalisation

- A general notion of (sound) up-to technique can be defined.
- Every size-preserving predicate transformer is an up-to technique.
- Closed under composition.
- Closely related to a class of up-to techniques identified by Pous: Functions below the “companion”.

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

- Containers (well-behaved functors):

$$\begin{aligned} \mathsf{Container} &: \mathsf{Set} \to \mathsf{Set}_1 \\ [\![\_]\!] &: \mathsf{Container}\ X \to (X \to \mathsf{Set}) \to (X \to \mathsf{Set}) \end{aligned}$$

- Greatest fixpoints of containers:

$$\nu : \mathsf{Container}\ X \to \mathsf{Size} \to (X \to \mathsf{Set})$$

Up-to techniques

Up-to techniques (sound):

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

Size-preserving

- Size-preserving predicate transformers:

$$
\begin{aligned}
&\textsf{Size-preserving} : \\
&\quad \textsf{Container}\ X \to ((X \to \textsf{Set}) \to (X \to \textsf{Set})) \to \textsf{Set}_1 \\
&\textsf{Size-preserving}\ C\ F = \\
&\quad \forall\ R\ i \to R \subseteq \nu\ C\ i \to F\ R \subseteq \nu\ C\ i
\end{aligned}
$$

- Every size-preserving predicate transformer is an up-to technique.