Towards Verification of a Denotational Semantics of Inheritance

Auxiliary Material

PETER D. MOSSES, Delft University of Technology, Netherlands and Swansea University, United Kingdom

This auxiliary material accompanies a paper in the festschrift for Jens Palsberg. It includes all the Agda definitions and proofs presented in the paper, but omits the explanatory text. It also presents the Agda proofs of Lemmas 2–4, which were omitted in the paper.

Familiarity with the published paper (and with Agda) is assumed.

Introduction

Most of Sections 4 and 5 of the published paper [1] are *literate* Agda specification, where Agda code is interleaved with informal text. Although the text is intended to help understand the code, it sometimes obscures the overall structure of the specification. Moreover, the indentation of module bodies has been omitted in the paper, due to the narrowness of the columns.

Here, the Agda code is presented without any interleaved text. This 'illiterate' Agda specification may be easier to read for those who have no need for the informal explanations, and when browsing the code in Agda editors.

The absence of a page limit for auxiliary material allows the inclusion of the Agda proofs of Lemmas 2–4, which were omitted in the published paper. The single-column format used here is also more convenient for browsing than the double-column format used in the paper, and module bodies have the same indentation as in the source files.

Semantic Definitions

The Agda definitions given below use the following modules from the standard library (v2.1).[1]

```
{-# OPTIONS --safe #-}              -- Agda        ~ CP89 notation
open import Data.Nat.Base
  using ( ℕ; zero; suc; _≤_ )       -- ℕ           ~ Nat
open import Data.Maybe.Base
  renaming ( Maybe to _+?;          -- A +?        ~ A + ?
             nothing to ??;         -- ??          ~ ⊥?
             maybe′ to [_,_]? )     -- [ f , x ]?  ~ [f, λ⊥?.x]
open import Data.Product.Base
  using (_×_; _,_; proj₁; proj₂)    -- A × B       ~ A × B
open import Function
  using (Inverse; _↔_; _∘_)         -- A ↔ B       ~ implicit
open Inverse {{ ... }}
  using (to; from)                  -- to from     ~ implicit
```

[1] https://agda.github.io/agda-stdlib/v2.1/

Author's Contact Information: Peter D. Mosses, Delft University of Technology, Delft, Netherlands and Swansea University, Swansea, United Kingdom, P.D.Mosses@tudelft.nl.

```
module Inheritance.Definitions
    ( Domain : Set_1 )
    ( ⟨_⟩       : Domain → Set )
    ( ⊥         : {D : Domain} → ⟨ D ⟩ )
    ( fix       : {D : Domain} → ( ⟨ D ⟩ → ⟨ D ⟩ ) → ⟨ D ⟩ )
    ( ?⊥        : Domain )
    ( _+⊥_      : Domain → Domain → Domain )
    ( inl       : {D E : Domain} → ⟨ D ⟩ → ⟨ D +⊥ E ⟩ )
    ( inr       : {D E : Domain} → ⟨ E ⟩ → ⟨ D +⊥ E ⟩ )
    ( [_,_]⊥    : {D E F : Domain} →
                    ( ⟨ D ⟩ → ⟨ F ⟩ ) → ( ⟨ E ⟩ → ⟨ F ⟩ ) →
                    ⟨ D +⊥ E ⟩ → ⟨ F ⟩ )
    ( Instance  : Set )              – objects
    ( Name      : Set )              – class names
    ( Key       : Set )              – method names
    ( Primitive : Set )              – function names
    ( Number    : Domain )           – unspecified
    ( Value     : Domain )           – a value is a behavior or a number
    ( Behavior  : Domain )           – a behavior maps keys to funs
    ( Fun       : Domain )           – a fun maps values to values
    {{ iso^v    : ⟨ Value ⟩          ↔ ⟨ Behavior +⊥ Number ⟩ }}
    {{ iso^b    : ⟨ Behavior ⟩       ↔ ( Key → ⟨ Fun +⊥ ?⊥ ⟩ ) }}
    {{ iso^f    : ⟨ Fun ⟩            ↔ ( ⟨ Value ⟩ → ⟨ Value ⟩ ) }}
    ( apply⟦_⟧  : Primitive → ⟨ Value ⟩ → ⟨ Value ⟩ )
  where
variable ρ : Instance; m : Key; f : Primitive; ν : ⟨ Number ⟩
variable α : ⟨ Value ⟩; σ π : ⟨ Behavior ⟩; ϕ : ⟨ Fun ⟩

data Class : Set where
  child  : Name → Class → Class    – a subclass
  origin : Class                   – the root class
variable κ : Class

data Exp : Set where
  self  : Exp                      – current object behavior
  super : Exp                      – superclass behavior
  arg   : Exp                      – method argument value
  call  : Exp → Key → Exp → Exp    – call method with argument
  appl  : Primitive → Exp → Exp    – apply primitive to value
variable e : Exp
```

```
module Semantics
    ( class    : Instance → Class )          – the class of an object
    ( methods′ : Class → Key → (Exp +?) )    – the methods of a class
  where
  methods : Class → Key → (Exp +?)           – no root class methods
  methods (child c κ) m = methods′ (child c κ) m
  methods origin m      = ??
```

Method Lookup Semantics

```
  D^g = ( Instance → ⟨ Behavior ⟩ ) ×
        ( Class → Instance → ⟨ Behavior ⟩ ) ×
        ( Exp → Instance → Class → ⟨ Fun ⟩ )

  module _
      { G^g : Domain }
      {{ iso^g : ⟨ G^g ⟩ ↔ D^g }}
    where
    g : D^g → D^g
    g (s , l , d⟦_⟧) = (send , lookup , do⟦_⟧) where

      send : Instance → ⟨ Behavior ⟩
      send ρ = l (class ρ) ρ

      lookup : Class → Instance → ⟨ Behavior ⟩
      lookup (child c κ) ρ =
        from λ m → [ ( λ e → inl (d⟦ e ⟧ ρ (child c κ)) ) ,
                     ( to (l κ ρ) m )
                   ]? (methods (child c κ) m)
      lookup origin ρ = ⊥

      do⟦_⟧ : Exp → Instance → Class → ⟨ Fun ⟩
      do⟦ self         ⟧ ρ κ           = from λ α → from (inl (s ρ))
      do⟦ super        ⟧ ρ (child c κ) = from λ α → from (inl (l κ ρ))
      do⟦ super        ⟧ ρ origin      = from λ α → ⊥
      do⟦ arg          ⟧ ρ κ           = from λ α → α
      do⟦ call e1 m e2 ⟧ ρ κ =
        from λ α → [ ( λ σ → [ ( λ ϕ → to ϕ (to (d⟦ e2 ⟧ ρ κ) α) ) ,
                               ( λ _ → ⊥ )
                             ]⊥ (to σ m) ) ,
                     ( λ ν → ⊥ )
                   ]⊥ (to (to (d⟦ e1 ⟧ ρ κ) α))
      do⟦ appl f e1    ⟧ ρ κ =
        from λ α → apply⟦ f ⟧ (to (d⟦ e1 ⟧ ρ κ) α)
```

$$
\begin{array}{ll}
\gamma : \langle\, G^{g} \,\rangle \to \langle\, G^{g} \,\rangle & \\
\gamma = \mathrm{from} \circ \mathrm{g} \circ \mathrm{to} & \\
\mathrm{send} & = \mathrm{proj}_1\ (\mathrm{to}\ (\mathrm{fix}\ \gamma)) \\
\mathrm{lookup} & = \mathrm{proj}_1\ (\mathrm{proj}_2\ (\mathrm{to}\ (\mathrm{fix}\ \gamma))) \\
\mathrm{do}[\![\_]\!] & = \mathrm{proj}_2\ (\mathrm{proj}_2\ (\mathrm{to}\ (\mathrm{fix}\ \gamma)))
\end{array}
$$

Denotational Semantics

$$
\begin{array}{l}
\mathrm{eval}[\![\_]\!] : \mathrm{Exp} \to \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Fun} \,\rangle \\
\mathrm{eval}[\![\ \mathrm{self}\ ]\!]\ \sigma\ \pi = \mathrm{from}\ \lambda\ \alpha \to \mathrm{from}\ (\mathrm{inl}\ \sigma) \\
\mathrm{eval}[\![\ \mathrm{super}\ ]\!]\ \sigma\ \pi = \mathrm{from}\ \lambda\ \alpha \to \mathrm{from}\ (\mathrm{inl}\ \pi\ ) \\
\mathrm{eval}[\![\ \mathrm{arg}\ ]\!]\ \sigma\ \pi = \mathrm{from}\ \lambda\ \alpha \to \alpha \\
\mathrm{eval}[\![\ \mathrm{call}\ e_1\ \mathrm{m}\ e_2\ ]\!]\ \sigma\ \pi = \\
\quad \mathrm{from}\ \lambda\ \alpha \to [\ (\ \lambda\ \sigma' \to [\quad (\ \lambda\ \phi \to \mathrm{to}\ \phi\ (\mathrm{to}\ (\mathrm{eval}[\![\ e_2\ ]\!]\ \sigma\ \pi)\ \alpha)\ )\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \lambda\ \_ \to \bot\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad]_{\bot}\ (\mathrm{to}\ \sigma'\ \mathrm{m})\ )\ , \\
\qquad\qquad\qquad\qquad (\ \lambda\ \nu \to \bot\ ) \\
\qquad\qquad\qquad]_{\bot}\ (\mathrm{to}\ (\mathrm{to}\ (\mathrm{eval}[\![\ e_1\ ]\!]\ \sigma\ \pi)\ \alpha)) \\
\mathrm{eval}[\![\ \mathrm{appl}\ \mathrm{f}\ e_1\ ]\!]\ \sigma\ \pi = \\
\quad \mathrm{from}\ \lambda\ \alpha \to \mathrm{apply}[\![\ \mathrm{f}\ ]\!]\ (\mathrm{to}\ (\mathrm{eval}[\![\ e_1\ ]\!]\ \sigma\ \pi)\ \alpha)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Generator} = \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle \\
\mathrm{Wrapper} = \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle
\end{array}
$$

$$
\begin{array}{l}
\_\oplus\_ : \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle \to \langle\, \mathrm{Behavior} \,\rangle \\
\sigma_1 \oplus \sigma_2 = \mathrm{from}\ \lambda\ \mathrm{m} \to \\
\quad [\ (\ \lambda\ \phi \to \mathrm{inl}\ \phi\ )\ , (\ \lambda\ \_ \to \mathrm{to}\ \sigma_2\ \mathrm{m}\ )\ ]_{\bot}\ (\mathrm{to}\ \sigma_1\ \mathrm{m}) \\
\_\boxed{\rhd}\_ : \mathrm{Wrapper} \to \mathrm{Generator} \to \mathrm{Generator} \\
\mathrm{w}\ \boxed{\rhd}\ \mathrm{p} = \lambda\ \sigma \to (\mathrm{w}\ \sigma\ (\mathrm{p}\ \sigma)) \oplus (\mathrm{p}\ \sigma) \\
\mathrm{wrap} : \mathrm{Class} \to \mathrm{Wrapper} \\
\mathrm{wrap}\ \kappa = \lambda\ \sigma \to \lambda\ \pi \to \mathrm{from}\ \lambda\ \mathrm{m} \to \\
\quad [\ (\ \lambda\ \mathrm{e} \to \mathrm{inl}\ (\mathrm{eval}[\![\ \mathrm{e}\ ]\!]\ \sigma\ \pi)\ ), (\ \mathrm{inr}\ \bot\ )\ ]_{?}\ (\mathrm{methods}\ \kappa\ \mathrm{m})
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{gen} : \mathrm{Class} \to \mathrm{Generator} & \\
\mathrm{gen}\ (\mathrm{child}\ \mathrm{c}\ \kappa) & = \mathrm{wrap}\ (\mathrm{child}\ \mathrm{c}\ \kappa)\ \boxed{\rhd}\ \mathrm{gen}\ \kappa \\
\mathrm{gen}\ \mathrm{origin} & = \lambda\ \sigma \to \bot \\
\mathrm{behave} : \mathrm{Instance} \to \langle\, \mathrm{Behavior} \,\rangle & \\
\mathrm{behave}\ \rho = \mathrm{fix}\ (\mathrm{gen}\ (\mathrm{class}\ \rho)) &
\end{array}
$$

Equivalence

```
{-# OPTIONS --allow-unsolved-metas #-}
open import Data.Nat.Base
  using ( ℕ; zero; suc; _≤_ )      -- ℕ          ~ Nat
open import Data.Maybe.Base
  renaming ( Maybe to _+?;         -- A +?       ~ A + ?
             nothing to ??;        -- ??         ~ ⊥?
             maybe′ to [_,_]? )    -- [ f , x ]? ~ [f, λ⊥?.x]
open import Data.Product.Base
  using (_×_; _,_; proj₁; proj₂)   -- A × B      ~ A × B
open import Function
  using (Inverse; _↔_; _∘_)        -- A ↔ B      ~ implicit
open Inverse {{ ... }}
  using (to; from; inverseˡ)       -- to from    ~ implicit

module Inheritance.Equivalence
    ( Domain :   Set₁ )
    ( ⟨_⟩     :   Domain → Set )
    ( _⊑_     :   {D : Domain} → ⟨ D ⟩ → ⟨ D ⟩ → Set )
    ( ⊥       :   {D : Domain} → ⟨ D ⟩ )
    ( fix     :   {D : Domain} → ( ⟨ D ⟩ → ⟨ D ⟩ ) → ⟨ D ⟩ )
    ( ?⊥      :   Domain )
    ( _+⊥_    :   Domain → Domain → Domain )
    ( inl     :   {D E : Domain} → ⟨ D ⟩ → ⟨ D +⊥ E ⟩ )
    ( inr     :   {D E : Domain} → ⟨ E ⟩ → ⟨ D +⊥ E ⟩ )
    ( [_,_]⊥  :   {D E F : Domain} →
                    ( ⟨ D ⟩ → ⟨ F ⟩ ) → ( ⟨ E ⟩ → ⟨ F ⟩ ) →
                    ⟨ D +⊥ E ⟩ → ⟨ F ⟩ )
    ( Instance    : Set )      -- objects
    ( Name        : Set )      -- class names
    ( Key         : Set )      -- method names
    ( Primitive   : Set )      -- function names
    ( Number   : Domain )      -- unspecified
    ( Value    : Domain )      -- a value is a behavior or a number
    ( Behavior : Domain )      -- a behavior maps keys to funs
    ( Fun      : Domain )      -- a fun maps values to values
    {{ isoᵛ    : ⟨ Value ⟩    ↔ ⟨ Behavior +⊥ Number ⟩ }}
    {{ isoᵇ    : ⟨ Behavior ⟩ ↔ ( Key → ⟨ Fun +⊥ ?⊥ ⟩ ) }}
    {{ isoᶠ    : ⟨ Fun ⟩      ↔ ( ⟨ Value ⟩ → ⟨ Value ⟩ ) }}
    ( apply⟦_⟧ : Primitive → ⟨ Value ⟩ → ⟨ Value ⟩ )
  where
```

```
open import Inheritance.Definitions
    ( Domain ) ( ⟨_⟩ ) ( ⊥ ) ( fix ) ( ?⊥ )
    ( _+⊥_ ) ( inl ) ( inr ) ( [_,_]⊥ )
    ( Instance ) ( Name ) ( Key ) ( Primitive )
    ( Number ) ( Value ) ( Behavior ) ( Fun )
    {{ isoᵛ }} {{ isoᵇ }} {{ isoᶠ }} ( apply⟦_⟧ )

module _
    ( class      : Instance → Class )
    ( methods′  : Class → Key → (Exp +?) )
  where
  open Semantics ( class ) ( methods′ )
```

Intermediate Semantics

```
  send′    : ℕ → Instance → ⟨ Behavior ⟩
  lookup′ : ℕ → Class → Instance → ⟨ Behavior ⟩
  do′_⟦_⟧ : ℕ → Exp → Instance → Class → ⟨ Fun ⟩

  send′ n ρ = lookup′ n (class ρ) ρ

  lookup′ zero κ ρ = ⊥
  lookup′ n (child c κ) ρ =
    from λ m → [ ( λ e → inl (do′ n ⟦ e ⟧ ρ (child c κ)) ) ,
                      ( to (lookup′ n κ ρ ) m )
                    ]? (methods (child c κ) m)
  lookup′ n origin ρ = ⊥

  do′ zero         ⟦ e     ⟧ ρ κ = ⊥

  do′ (suc n)      ⟦ self ⟧ ρ κ = from λ α → from (inl (send′ n ρ))

  do′ n ⟦ super ⟧ ρ (child c κ) =
    from λ α → from (inl (lookup′ n κ ρ))
  do′ n ⟦ super ⟧ ρ origin = from λ α → ⊥
  do′ n ⟦ arg     ⟧ ρ κ = from λ α → α
  do′ n ⟦ call e₁ m e₂ ⟧ ρ κ =
    from λ α → [ ( λ σ → [ ( λ ϕ → to ϕ (to (do′ n ⟦ e₂ ⟧ ρ κ) α) ) ,
                                ( λ _ → ⊥ )
                              ]⊥ (to σ m) ) ,
                      ( λ ν → ⊥ )
                    ]⊥ (to (to (do′ n ⟦ e₁ ⟧ ρ κ ) α))
  do′ n ⟦ appl f e₁ ⟧ ρ κ =
    from λ α → apply⟦ f ⟧ (to (do′ n ⟦ e₁ ⟧ ρ κ) α)
```

Proofs

```
open import Relation.Binary.PropositionalEquality.Core
  using (_≡_; refl; cong; sym)
open import Relation.Binary.PropositionalEquality.Properties
open import Relation.Binary.Reasoning.Syntax
open ≡-Reasoning
open import Axiom.Extensionality.Propositional
  using (Extensionality)
open import Level
  renaming (zero to lzero) hiding (suc)

module _ ( ext : Extensionality lzero lzero )
  where
```

Lemma 1

```
  lemma-1 : ∀ n e ρ c κ →
    do′ (suc n) ⟦ e ⟧ ρ (child c κ) ≡
    eval⟦ e ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)

  lemma-1 n self ρ c κ =
    begin   do′ (suc n) ⟦ self ⟧ ρ (child c κ)
    ≡⟨⟩     ( from λ α → from (inl (send′ n ρ)) )
    ≡⟨⟩     eval⟦ self ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
    ∎
  lemma-1 n super ρ c (child c′ κ) =
    begin   do′ (suc n) ⟦ super ⟧ ρ (child c (child c′ κ))
    ≡⟨⟩     ( from λ α → from (inl (lookup′ (suc n) (child c′ κ) ρ)) )
    ≡⟨⟩     eval⟦ super ⟧ (send′ n ρ) (lookup′ (suc n) (child c′ κ) ρ)
    ∎
  lemma-1 n super ρ c origin =
    begin   do′ (suc n) ⟦ super ⟧ ρ (child c origin)
    ≡⟨⟩     ( from λ α → from (inl ⊥ ) )
    ≡⟨⟩     eval⟦ super ⟧ (send′ n ρ) (lookup′ (suc n) origin ρ)
    ∎
  lemma-1 n arg ρ c κ =
    begin   do′ (suc n) ⟦ arg ⟧ ρ (child c κ)
    ≡⟨⟩     ( from λ α → α )
    ≡⟨⟩     eval⟦ arg ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
    ∎

  lemma-1 n (call e₁ m e₂) ρ c κ
    rewrite (lemma-1 n e₁ ρ c κ)
    rewrite (lemma-1 n e₂ ρ c κ) = refl
```

```
lemma-1 n (appl f e₁) ρ c κ =
  begin
    do′ (suc n) ⟦ appl f e₁ ⟧ ρ (child c κ)
  ≡⟨⟩
    ( from λ α →
        apply⟦ f ⟧
          (to (do′ (suc n) ⟦ e₁ ⟧ ρ (child c κ)) α) )
  ≡⟨ use-induction ⟩
    ( from λ α →
        apply⟦ f ⟧
          (to (eval⟦ e₁ ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)) α) )
  ≡⟨⟩
    eval⟦ appl f e₁ ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
  □
  where
    use-induction =
      cong from (ext λ α →
        cong (λ X →
          apply⟦ f ⟧ ((to X) α)) (lemma-1 n e₁ ρ c κ))
```

Lemma 2

```
module _
    ( [,]⊥-elim : – [,]⊥-elim eliminates an application of [ f , g ]⊥
        {D E F : Domain} {A : Set}
        {f : ⟨ D ⟩ → ⟨ F ⟩} {g : ⟨ E ⟩ → ⟨ F ⟩}
        {x : A → ⟨ D ⟩} {y : ⟨ E ⟩} {z : A +?} →
          [ f , g ]⊥ ( [ ( inl ∘ x ) , ( inr y ) ]? z ) ≡ [ ( f ∘ x ) , ( g y ) ]? z )
  where

  lemma-2 : ∀ κ n ρ → gen κ (send′ n ρ) ≡ lookup′ (suc n) κ ρ

  lemma-2 origin n ρ =
    begin
      gen origin (send′ n ρ)
    ≡⟨⟩
      ⊥
    ≡⟨⟩
      lookup′ (suc n) origin ρ
    □
  lemma-2 (child c κ) n ρ =
    let π = lookup′ (suc n) κ ρ in
    begin
      gen (child c κ) (send′ n ρ)
    ≡⟨⟩ – use definition of gen
      (wrap (child c κ) ⊵ gen κ) (send′ n ρ)
```

```
≡⟨⟩ – use definition of _⊳_
  (wrap (child c κ) (send′ n ρ) (gen κ (send′ n ρ))) ⊕ (gen κ (send′ n ρ))
≡⟨ use-lemma-2 ⟩
  (wrap (child c κ) (send′ n ρ) π) ⊕ π
≡⟨⟩ – use definition of _⊕_
  (from λ m →
    [ ( λ ϕ → inl ϕ ) , ( λ _ → to π m ) ]⊥
    (to (wrap (child c κ) (send′ n ρ) π) m))
≡⟨⟩ – use definition of wrap
  (from λ m →
    [ ( λ ϕ → inl ϕ ) , ( λ _ → to π m ) ]⊥
    (to (from (λ m →
      [ ( λ e → inl(eval⟦ e ⟧ (send′ n ρ) π) ) , ( inr ⊥ ) ]?
      (methods (child c κ) m))) m))
≡⟨ use-to∘from-inverse ⟩
  (from λ m →
    [ ( λ ϕ → inl ϕ ) , ( λ _ → to π m ) ]⊥
    ( [ ( λ e → inl (eval⟦ e ⟧ (send′ n ρ) π) ) , ( inr ⊥ ) ]? (methods (child c κ) m) ))
≡⟨ use-[,]⊥-elim ⟩
  (from λ m →
    [ ( λ e → inl (eval⟦ e ⟧ (send′ n ρ) π) ) ,
      ( to π m ) ]? (methods (child c κ) m))
≡⟨⟩ – use definition of π
  (from λ m →
    [ ( λ e → inl (eval⟦ e ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)) ) ,
      ( to (lookup′ (suc n) κ ρ) m ) ]? (methods (child c κ) m))
≡⟨ use-lemma-1 ⟩
  (from λ m →
    [ ( λ e → inl (do′ (suc n) ⟦ e ⟧ ρ (child c κ)) ) ,
      ( to (lookup′ (suc n) κ ρ) m ) ]? (methods (child c κ) m))
≡⟨⟩ – use definition of lookup′
  lookup′ (suc n) (child c κ) ρ
□
where
  π′ = lookup′(suc n) κ ρ
  use-lemma-2 =
    cong (λ X → wrap (child c κ) (send′ n ρ) X ⊕ X) (lemma-2 κ n ρ)
  use-to∘from-inverse =
    cong from (ext λ x →
      cong (λ X → [ _ , ( λ _ → to π′ x ) ]⊥ (X x)) (inverse^l refl))
  use-[,]⊥-elim =
    cong from (ext λ x →
      [,]⊥-elim {A = Exp}
        {x = λ e → eval⟦ e ⟧ (send′ n ρ) π′} {y = ⊥} {z = methods (child c κ) x})
```

```
    use-lemma-1 =
      cong from (ext λ m →
        cong (λ X → [ X , ( to (lookup′ (suc n) κ ρ) m ) ]? (methods (child c κ) m))
          (ext λ e → cong inl (sym (lemma-1 n e _ _ _))))
```

Lemma 3

```
iter : {D : Domain} → ℕ → ( ⟨ D ⟩ → ⟨ D ⟩ ) → ⟨ D ⟩
iter zero g     = ⊥
iter (suc n) g = g (iter n g)

lemma-3 : ∀ n ρ → iter n (gen (class ρ)) ≡ send′ n ρ

lemma-3 zero ρ =
  begin
    iter zero (gen (class ρ))
  ≡⟨⟩
    ⊥
  ≡⟨⟩
    send′ zero ρ
  □
lemma-3 (suc n) ρ    =
  begin
    iter (suc n) (gen (class ρ))
  ≡⟨⟩
    gen (class ρ) (iter n (gen (class ρ)))
  ≡⟨ use-induction ⟩
    gen (class ρ) (send′ n ρ)
  ≡⟨ lemma-2 (class ρ) n ρ ⟩
    lookup′(suc n) (class ρ) ρ
  ≡⟨⟩
    send′ (suc n) ρ
  □
  where
    use-induction = cong (λ X → gen (class ρ) X) (lemma-3 n ρ)
```

Lemma 4

```
module _
  ( ⊥-is-least : {D : Domain} {x : ⟨ D ⟩}           → ⊥ ⊑ x )
  ( ⊑-is-reflexive : {D : Domain} {x y : ⟨ D ⟩}        → x ≡ y → x ⊑ y )
  ( ⊑-is-transitive : {D : Domain} {x y z : ⟨ D ⟩} → x ⊑ y → y ⊑ z → x ⊑ z )
  ( is-assumed-monotone :
    {D E : Domain} (f : ⟨ D ⟩ → ⟨ E ⟩) (x y : ⟨ D ⟩) →
      (x ⊑ y) → (f x ⊑ f y) )
  ( is-assumed-monotone-2 :
    {D E F : Domain} (f : ⟨ D ⟩ → ⟨ E ⟩ → ⟨ F ⟩) (x y : ⟨ D ⟩) →
      (x ⊑ y) → ({z : ⟨ E ⟩} → (f x z ⊑ f y z)) )
  where

  begin-⊑_ : {D : Domain} → {x y : ⟨ D ⟩ } → x ⊑ y → x ⊑ y
  begin-⊑ p = p
  _◻-⊑ : {D : Domain} → (x : ⟨ D ⟩ ) → x ⊑ x
  x ◻-⊑ = ⊑-is-reflexive refl
  _⊑⟨_⟩_ : {D : Domain} → (x : ⟨ D ⟩ ) → {y z : ⟨ D ⟩ } → x ⊑ y → y ⊑ z → x ⊑ z
  x ⊑⟨ p ⟩ q = ⊑-is-transitive p q
  _≡-⊑⟨_⟩_ : {D : Domain} → (x : ⟨ D ⟩ ) → {y z : ⟨ D ⟩ } → x ≡ y → y ⊑ z → x ⊑ z
  x ≡-⊑⟨ p ⟩ q = ⊑-is-transitive (⊑-is-reflexive p) q
  _⊑-≡⟨_⟩_ : {D : Domain} → (x : ⟨ D ⟩ ) → {y z : ⟨ D ⟩ } → x ⊑ y → y ≡ z → x ⊑ z
  x ⊑-≡⟨ p ⟩ q = ⊑-is-transitive p (⊑-is-reflexive q)
  _⊑⟨⟩_ : {D : Domain} → (x : ⟨ D ⟩ ) → {y : ⟨ D ⟩ } → x ⊑ y → x ⊑ y
  x ⊑⟨⟩ q = x ⊑⟨ ⊑-is-reflexive refl ⟩ q
  infix   1 begin-⊑_
  infixr  2 _⊑⟨_⟩_
  infixr  2 _⊑-≡⟨_⟩_
  infixr  2 _≡-⊑⟨_⟩_
  infixr  2 _⊑⟨⟩_
  infix   3 _◻-⊑

  -- is-chain : {D : Domain} → (δ : ℕ → ⟨ D ⟩) → Set
  -- is-chain δ = ∀ n → (δ n) ⊑ (δ (suc n))

  iter-is-chain : {D : Domain} (n : ℕ) (g : ⟨ D ⟩ → ⟨ D ⟩) → iter n g ⊑ iter (suc n) g
  iter-is-chain zero g = ⊥-is-least
  iter-is-chain (suc n) g =
    is-assumed-monotone g (iter n g) (iter (suc n) g) (iter-is-chain n g)
```

```
lemma-4-send′ : ∀ n ρ →
  send′ n ρ ⊑ send′ (suc n) ρ

lemma-4-send′ n ρ
  rewrite sym (lemma-3 n ρ)
  rewrite sym (lemma-3 (suc n) ρ) =
    iter-is-chain n (gen (class ρ))

lemma-4-lookup′ : ∀ n κ ρ →
  lookup′ n κ ρ ⊑ lookup′ (suc n) κ ρ

lemma-4-lookup′ zero κ ρ = ⊥-is-least
lemma-4-lookup′ (suc n) κ ρ
  rewrite sym (lemma-2 κ n ρ)
  rewrite sym (lemma-2 κ (suc n) ρ) =
    is-assumed-monotone (gen κ) (send′ n ρ) (send′ (suc n) ρ) (lemma-4-send′ n ρ)

lemma-4-do′ : ∀ n e ρ c κ →
  do′ (suc n) ⟦ e ⟧ ρ (child c κ) ⊑
  do′ (suc (suc n)) ⟦ e ⟧ ρ (child c κ)

lemma-4-do′ n e ρ c κ =
  begin-⊑
    do′ (suc n) ⟦ e ⟧ ρ (child c κ)
  ≡-⊑⟨ lemma-1 n e ρ c κ ⟩
    eval⟦ e ⟧ (send′ n ρ) (lookup′ (suc n) κ ρ)
  ⊑⟨ is-assumed-monotone-2 (eval⟦ e ⟧)
          (send′ n ρ) (send′ (suc n) ρ)
          (lemma-4-send′ n ρ) ⟩
    eval⟦ e ⟧ (send′ (suc n) ρ) (lookup′ (suc n) κ ρ)
  ⊑⟨ is-assumed-monotone (eval⟦ e ⟧ (send′ (suc n) ρ))
          (lookup′ (suc n) κ ρ) (lookup′ (suc (suc n)) κ ρ)
          (lemma-4-lookup′ (suc n) κ ρ) ⟩
    eval⟦ e ⟧ (send′ (suc n) ρ) (lookup′ (suc (suc n)) κ ρ)
  ≡-⊑⟨ sym (lemma-1 (suc n) e ρ c κ) ⟩
    do′ (suc (suc n)) ⟦ e ⟧ ρ (child c κ)
  □-⊑
```

Remaining Results

```
module _
    { Gᵍ : Domain }
    {{ isoᵍ : ⟨ Gᵍ ⟩ ↔ Dᵍ }}
    ( lub :       {D : Domain} → (δ : ℕ → ⟨ D ⟩) → ⟨ D ⟩ )
  where

  interpret : Instance → ⟨ Behavior ⟩
  interpret ρ = lub (λ n → send′ n ρ)

  proposition-1 :    ∀ ρ → interpret ρ ≡ behave ρ
  proposition-2 :    ∀ ρ → behave ρ ⊑ send ρ
  proposition-3 :    ∀ ρ → send ρ ⊑ interpret ρ
  theorem-1 :        ∀ ρ → send ρ ≡ behave ρ

  proposition-1 ρ = {!  !}
  proposition-2 ρ = {!  !}
  proposition-3 ρ = {!  !}
  theorem-1 ρ =     {!  !}
```

When Agda proofs of propositions 1–3 and theorem-1 have been completed, they are to be made available on GitHub at https://github.com/pdmosses/jensfest-agda.

References

[1] Peter D. Mosses. 2024. Towards verification of a denotational semantics of inheritance. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday (JENSFEST '24), October 22, 2024, Pasadena, CA, USA*. ACM. https://doi.org/10.1145/3694848.3694852

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.