Gradual types for effect handlers

LI-YAO XIA, University of Edinburgh, United Kingdom
PHILIP WADLER, University of Edinburgh, United Kingdom

ACM Reference Format:
Li-yao Xia and Philip Wadler. 2023. Gradual types for effect handlers. 1, 1 (July 2023), 49 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computational effects are everywhere: state, concurrency, probability, nondeterminism, input-output, exception handling. Algebraic effects with handlers, introduced by Plotkin and Pretnar [2009], have seen rapid development in recent years as a way to support a wide range of computational effects. They have inspired numerous libraries, experimental programming languages including Links, Eff, Koka, and Frank, and features in programming languages including WebAssembly, OCaml, and Haskell. (We give citations for these later.)

Type systems for tracking algebraic effects with handlers are a subject of intense study, with prototypes appearing in Links, Eff, Koka, and Frank. Meanwhile, the new features adopted in WebAssembly, OCaml, and Haskell use algebraic effects and handlers without reflecting effects in types. Further, virtually every program in existence makes some use of computational effects. In the future, it will become vital to have some way for legacy code with effects not reflected in the types to interact soundly with new code that does have effects reflected in the types.

Gradual types, introduced by Siek and Taha [2006], provide a model of how code with less precise types can interface soundly with code with more precise types. Gradual types have been extensively studied for a wide range of features, including some forms of computational effects. However, until now noone has studied the combination of algebraic effects with handlers and gradual types. We show that in fact this combination is straightforward.

2 EXAMPLES

2.1 State

From "Handlers in Action".

The type of state is (currently) hard-coded as the type of natural numbers.

```
St : Type
St = $ ℕ
```

The state effect consists of "get" and "put" operations.

```
state : List Op
state = ("get" :: "put" :: [])
```

Authors' addresses: Li-yao Xia, ly.xia@ed.ac.uk, University of Edinburgh, Edinburgh, United Kingdom; Philip Wadler, wadler@inf.ed.ac.uk, University of Edinburgh, Edinburgh, United Kingdom.

XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The state handler interprets a stateful computation as a function `St ⇒ ⟨ ε ⟩ A`. The return clause returns the result `x ∶ A`, ignoring the state. The operation clause for `"get"` passes the current state to the continuation, whereas the operation clause for `"put"` discards the current state and continues with the value that the operation was called with.

```
state-handler ∶ ∀ {Γ A}
  → Γ ⊢ ⟨ ¡ state ⟩ A ⇒ʰ ⟨ ε ⟩ (St ⇒ ⟨ ε ⟩ A)
state-handler = record
  { Hooks = "get" ∷ "put" ∷ []
  ; Hooks-handled = refl
  ; on-return = return¦ x ⇒ fun _ ⇒ x
  ; on-perform
      = handle¦ "get" ⇒ (λ _ k → fun s ⇒ k ∶ s ∶ s)
      | handle¦ "put" ⇒ (λ s k → fun _ ⇒ k ∶ () ∶ s)
      | []
  }

state-handler☆ ∶ ∀ {Γ A} → Γ ⊢ ⟨ ☆ ⟩ A ⇒ʰ ⟨ ☆ ⟩ (St ⇒ ⟨ ☆ ⟩ A)
state-handler☆ = record
  { Hooks-handled = refl
  ; on-return = return¦ x ⇒ fun _ ⇒ x
  ; on-perform
      = handle¦ "get" ⇒ (λ _ k → fun s ⇒ k ∶ s ∶ s)
      | handle¦ "put" ⇒ (λ s k → fun _ ⇒ k ∶ () ∶ s)
      | []
  }
```

We wrap the handler in the following `run-state` function which initializes the state to 0.

```
run-state ∶ ∀ {Γ A}
  → Γ ⊢ ⟨ ¡ state ⟩ A
  → Γ ⊢ ⟨ ε ⟩ A
run-state M =
  handle state-handler M ∶ $ 0

run-state☆ ∶ ∀ {Γ A}
  → Γ ⊢ ⟨ ☆ ⟩ A
  → Γ ⊢ ⟨ ☆ ⟩ A
run-state☆ M =
  handle state-handler☆ M ∶ $ 0
```

An example computation that uses state:

```
infixl 4 _|>_
pattern _|>_ N M = M ∶ N

incr-state ∶ ∀ {Γ} → Γ ⊢ ⟨ ¡ state ⟩ $ℕ
incr-state =
  Let x ∶= perform¦ "get" () In
  Let _ ∶= perform¦ "put" (x + $ 1) In
  perform¦ "get" ()

incr-state☆ ∶ ∀ {Γ} → Γ ⊢ ⟨ ☆ ⟩ $ℕ
```

```
incr-state☆ =
  Let x ⦂= perform! "get" () In
  Let _ ⦂= perform! "put" (x + $ 1) In
  perform! "get" ()
```

Apply `run-state` to `incr-state`

```
state-example ⦂ ∀ {Γ} → Γ ⊢ ⟨ ε ⟩ $ℕ
state-example = run-state incr-state

state-example☆ ⦂ ∀ {Γ} → Γ ⊢ ⟨ ☆ ⟩ $ℕ
state-example☆ = run-state☆ incr-state☆

state-example☆ˡ ⦂ ∀ {Γ} → Γ ⊢ ⟨ ☆ ⟩ $ℕ
state-example☆ˡ = run-state☆ (castᵉ (+ (¡≤☆ {E = state})) incr-state)

state-example☆ʳ ⦂ ∀ {Γ} → Γ ⊢ ⟨ ε ⟩ $ℕ
state-example☆ʳ = run-state (castᵉ (- (¡≤☆ {E = state})) incr-state☆)
```

`state-example` reduces to the constant `$ 4`.

```
eval-state-example ⦂ ∃[ M⟶↠N ]
    eval (gas 25) state-example
  ≡ steps {⟨ ¡ [] ⟩ $ℕ} M⟶↠N (done ($ 1))
eval-state-example = _ , refl

eval-state-example☆ ⦂ ∃[ M⟶↠N ]
    eval (gas 25) state-example☆
  ≡ steps {⟨ ☆ ⟩ $ℕ} M⟶↠N (done ($ 1))
eval-state-example☆ = _ , refl

eval-state-example☆ˡ ⦂ ∃[ M⟶↠N ]
    eval (gas 25) state-example☆ˡ
  ≡ steps {⟨ ☆ ⟩ $ℕ} M⟶↠N (done ($ 1))
eval-state-example☆ˡ = _ , refl

eval-state-example☆ʳ ⦂ ∃[ M⟶↠N ]
    eval (gas 25) state-example☆ʳ
  ≡ steps {⟨ ¡ [] ⟩ $ℕ} M⟶↠N (done ($ 1))
eval-state-example☆ʳ = _ , refl
```

2.2 Nondeterminism

Also from Handlers in Action. A drunk tosses a coin: they may flip head or tails, or they may drop the coin and it falls in the gutter.

```
nondet ⦂ Effect
nondet = ¡ ("choose" ⁞⁞ "fail" ⁞⁞ [])

fail ⦂ ∀ {Γ} → Γ ⊢ ⟨ nondet ⟩ $𝔹
fail =
  Let _ ⦂= perform! "fail" () In
  ($ true) {- unreachable -}

drunkToss ⦂ ∅ ⊢ ⟨ nondet ⟩ $𝔹
drunkToss =
```

```
  Let catch-coin ꞉= perform! "choose" () In
  if catch-coin
  ( Let coin-flip ꞉= perform! "choose" () In
    if coin-flip ($ true) ($ false)
  )
  ( fail )
```

Handle a non-deterministic computation of type $\mathbb{B}$, returning `true` when at least one execution returns `true`.

```
nondet-handler ꞉
  ∅ ⊢ ⟨ nondet ⟩ $𝔹 ⇒ʰ ⟨ ε ⟩ $𝔹
nondet-handler = record
  { Hooks = "choose" ∷ "fail" ∷ []
  ; Hooks-handled = refl
  ; on-return = ` Z
  ; on-perform
      = handle! "choose" ⇒ (λ _ k → (k ꞉ tru) ⟪ _v_ ⟫ (k ꞉ fls))
      | handle! "fail" ⇒ (λ _ k → $ false)
      | []
  }

nondet-example ꞉ ∅ ⊢ ⟨ ε ⟩ $𝔹
nondet-example =
  handle nondet-handler drunkToss
```

`nondet-example` reduces to the constant `$ true`.

```
from-steps ꞉ ∀ {P} {M ꞉ ∅ ⊢ P} → Steps M → Maybe (∅ ⊢ P)
from-steps (steps _ (done v)) = just (value v)
from-steps _ = nothing

eval-nondet-example ꞉ ∃[ M⟹N ]
     from-steps (eval (gas 1000) nondet-example)
  ≡  just ($ true)
eval-nondet-example = _ , refl
```

3 TYPES AND EFFECTS

We define types, effects, and the *precision* relation on types.

The module `Utils` reexports the standard library and exports some additional general lemmas.

3.1 Base types

Base types are primitive data types such as numbers and booleans.

```
data Base ꞉ Set where
  ′ℕ ꞉ Base
  ′𝔹 ꞉ Base
  ′𝕌 ꞉ Base
```

The `rep` function interprets base types into Agda types.

```
rep ꞉ Base → Set
rep ′ℕ = ℕ
```

```
rep ′𝔹 = 𝔹
rep ′U = ⊤
```

3.2 Effects

Algebraic effects are names that a program may call, submitting a request with some arguments, expecting some result in response.

We represent those names simply as strings.

```
Op : Set
Op = String
```

A type-and-effect system keeps track of the operations that a computation may perform. A *gradual effect* `E : Effect` may be either static or dynamic. A static effect is a list of operations that a computation may perform. The dynamic effect ☆ allows a computation to perform any operations.

```
StaticEffect : Set
StaticEffect = List Op

data Effect : Set where
  ¡_ : StaticEffect → Effect
  ☆ : Effect
```

Pattern synonym for the empty effect (a computation which calls no operations).

```
pattern ε = ¡ []
```

Consistent membership lifts the membership relation `_∈_` from lists (static effects) to gradual effects. The dynamic effect statically accepts any effect `e` as a member.

```
data _∈☆_ (e : Op) : Effect → Set where
  ¡_ : ∀ {E} → e ∈ E → e ∈☆ ¡ E
  ☆ : e ∈☆ ☆
```

List concatenation `_++_` is similarly lifted to gradual effects: extending the dynamic effect yields back the dynamic effect.

```
_++☆_ : List Op → Effect → Effect
E ++☆ ☆ = ☆
E ++☆ (¡ F) = ¡ (E ++ F)
```

3.3 Types

We distinguish computations from the values they return, assigning them different notions of types. Computation types `CType` are pairs of effects `Effect` and value types `Type`. Computation types and value types are defined mutually recursively, so we declare both of their type signatures before giving their definitions.

```
record CType : Set
data Type : Set
```

A value type can be the dynamic type ⋆ for values whose type will be known at run time. The base type `$_` is for primitives. And the function type has a domain which is a value type and a codomain which is a computation type: when a function is applied, it may perform effects.

```
data Type where
  ⋆ : Type
```

```
  $_ : (ι : Base) → Type
  _⇒_ : (A : Type) → (P : CType) → Type
```

Computation types are pairs of an effect and a value type, respectively describing the operations that a computation may perform, and the values that it may return.

```
record CType where
  inductive
  constructor ⟨_⟩_
  field
    effects : Effect
    returns : Type
```

Having defined types, we can assign signatures to operations, which are their input and output types, also called requests and responses.

```
Op-sig : Op → Type × Type
Op-sig "get"    = ($ ′𝕌 , $ ′ℕ)
Op-sig "put"    = ($ ′ℕ , $ ′𝕌)
Op-sig "choose" = ($ ′𝕌 , 𝟚)     -- TODO: conditionals (eliminate bool)
Op-sig "fail"   = ($ ′𝕌 , $ ′𝕌) -- TODO: empty type
Op-sig _        = (⋆ , ⋆)

request : Op → Type
request e = proj₁ (Op-sig e)

response : Op → Type
response e = proj₂ (Op-sig e)
```

Gradual types let us control how much information about the program's behavior we want to keep track of at compile time or at run time. There is an ordering of types, called *precision*, with $\star$ at the top and completely static types at the bottom, with no occurrences of $\star$. Intuitively, more precise types provide more static information, while less precise types give more flexibility in exchange for more run-time checks. We define precision in the rest of this section.

3.4 Precision

Types are ordered by *precision*—also known as *imprecision* (which better fits our notation), *materialization* [Castagna et al., 2019], or *dynamism* [New and Ahmed, 2018].

3.4.1 Ground types. One early dimension to consider in designing a gradual type system is whether to compare types at run time *deeply* or *shallowly*. Deep type comparisons are known to break the gradual guarantee [Boyland, 2014], so we will go with shallow type comparisons. *Ground types* are those that reflect exactly the information learned from such a shallow comparison. We only look at the first type constructor of a type, so the type is either a base type `$_` or a function type `_⇒_`, and in the latter case we don't learn anything about the domain or codomain, so the most precise type describing what we know is $\star \Rightarrow \langle\ \star\ \rangle\ \star$.

```
data Ground : Type → Set where
  $_
    : (ι : Base)
      ------------
    → Ground ($ ι)

  ⋆⇒⋆
```

```
  ⁞ --------------
    Ground (★ ⇒ ⟨ ☆ ⟩ ★)
```

3.4.2 The precision relation. Precision orders types by how much static information they tell us about their values.

The dynamic effect ☆ is less precise than any static effect `¦ E`.

```
data _≼ᵉ_ ⁞ (_ _ ⁞ Effect) → Set where
  id ⁞ ∀ {E} → E ≼ᵉ E
  ¦≼☆ ⁞ ∀ {E} → ¦ E ≼ᵉ ☆
```

☆ is the least precise element in `Effect`.

```
≼☆ ⁞ ∀ {E} → E ≼ᵉ ☆
≼☆ {☆} = id
≼☆ {¦ _} = ¦≼☆
```

Since computation types and value types are mutually recursive, their respective precision relations are also mutually recursive. We declare the signature of one before defining the other.

```
record _≼ᶜ_ (_ _ ⁞ CType) ⁞ Set
```

A staple of gradual typing is that the function type is covariant in both domain and codomain with respect to precision.

```
data _≼_ ⁞ Type → Type → Set where

  _⇒_ ⁞ ∀ {A P A′ P′}
    → A ≼ A′
    → P ≼ᶜ P′
      ---------------
    → A ⇒ P ≼ A′ ⇒ P′
```

The dynamic type ⋆ is less precise than all types. However, following the principle that run-time type comparisons will be shallow, when we compare an arbitrary type `A` with ⋆, we look at the first constructor, represented by a ground type `G`, and further comparisons are done by comparing the components of `A` with those of `G` (which are necessarily ⋆ or ☆).

```
_⇑_ ⁞ ∀ {A G}
  → A ≼ G
  → Ground G
    -----
  → A ≼ ★
```

The reflexivity of `_≼_` includes the fact that base types `$_` are related only to themselves. In fact, we could ensure that `A ≼ B` is a singleton by restricting the `id` rule to base types. Although this would simplify some proofs, we view this uniqueness as an artifact of the simple type system being formalized. It is generally useful for coercions (which we will represent as proofs of precision) to have non-trivial structure, for purposes both practical—an identity coercion which can be immediately discarded enables better performance—and theoretical—with polymorphism, derivations of precisions tend to not be unique.

```
id ⁞ ∀ {A}
     -----
  → A ≼ A
```

Precision between computation types composes precision between their effect and value components.

```
record _≤ᶜ_ P Q where
  inductive
  constructor ⟨_⟩_
  field
    effects : CType.effects P ≤ᵉ CType.effects Q
    returns : CType.returns P ≤ CType.returns Q
```

Domain and codomain of function precision.

```
split⇒ : ∀ {A A′ E E′ B B′} (p : A ⇒ ⟨ E ⟩ B ≤ A′ ⇒ ⟨ E′ ⟩ B′) → (A ≤ A′) × (E ≤ᵉ E′)
split⇒ id = id , id , id
split⇒ (a ⇒ ⟨ e ⟩ b) = a , e , b

dom : ∀ {A B A′ B′} → A ⇒ B ≤ A′ ⇒ B′ → A ≤ A′
dom a = proj₁ (split⇒ a)

cod : ∀ {A B E A′ B′ E′} → A ⇒ ⟨ E ⟩ B ≤ A′ ⇒ ⟨ E′ ⟩ B′ → B ≤ B′
cod a = proj₂ (proj₂ (split⇒ a))

eff : ∀ {A B E A′ B′ E′} → A ⇒ ⟨ E ⟩ B ≤ A′ ⇒ ⟨ E′ ⟩ B′ → E ≤ᵉ E′
eff a = proj₁ (proj₂ (split⇒ a))
```

The use of these two functions is reminiscent of some gradually-typed source languages, where one defines

```
dom ⋆       =  ⋆
dom (A ⇒ B) =  A

cod ⋆       =  ⋆
cod (A ⇒ B) =  B
```

and has a typing rule resembling

```
Γ ⊢ L : A
Γ ⊢ M : dom A
-----------------
Γ ⊢ L ⟨ M ⟩ cod A
```

Our `dom` and `cod` will play a similar role when we define the precedence rules for abstraction and application.

3.5 Casts

```
infix 6 _=>_ _=>ᶜ_ _=>ᵉ_
infix 4 +_ -_
```

We define notions of casts for the different precision relations `_≤_`, `_≤ᶜ_`, `_≤ᵉ_` uniformly with the `Cast` combinator.

```
data Cast {S : Set}
  (_<_ _⊏_ : S → S → Set) (A B : S) : Set where
```

There are three kinds of casts. Upcasts reduce precision, *e.g.*, casting from `$ ι` to ⋆,

```
  +_ : A < B
       ----------------
    → Cast _<:_ _⊑_ A B
```

Downcasts increase precision.

```
  -_ : B < A
       ----------------
    → Cast _<:_ _⊑_ A B
```

The types of casts for value types, computation types, and effects are obtaines by applying `Cast` to their respective precision and subtyping relations.

```
_⇒_ : Type → Type → Set
_⇒_ = Cast _≤_ _⊑_

_⇒ᶜ_ : CType → CType → Set
_⇒ᶜ_ = Cast _≤ᶜ_ _⊑ᶜ_

_⇒ᵉ_ : Effect → Effect → Set
_⇒ᵉ_ = Cast _≤ᵉ_ _⊑ᵉ_
```

4 SYNTAX

In this section, we define the syntax of the calculus, renaming, substitution, and prove some related lemmas.

4.1 Contexts and Variables

The context is represented as a snoc list of types, and variables are de Bruijn indices, indexing into that list.

```
data Context : Set where
  ∅   : Context
  _▷_ : Context → Type → Context

data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
      ----------
    → Γ ▷ A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ A
      ---------
    → Γ ▷ B ∋ A
```

4.2 Terms

The type of terms `_⊢_` is defined recursively with the type of handlers `_⊢_⇒ʰ_`. In order to streamline this presentation, we will interleave term constructor declarations with related definitions. This is allowed in an **`interleaved mutual`** block, which Agda desugars by deinterleaving them.

```
interleaved mutual

data _⊢_ : Context → CType → Set
record _⊢_⇒ʰ_ (Γ : Context) (P Q : CType) : Set
```

The data type `Γ ⊢ P` represents a term of computation type `P` in context `Γ`. The first five typing rules are standard (variables, abstraction, application, constants, and primitive operators). Then three typing rules extend the calculus with gradual typing (casts, boxes, and blame), and two more with algebraic effects (operations and handlers).

Within an `interleaved mutual` block, constructors can be declared in an anonymous `data _ where` block, which is not associated to any one data type. The data type that a constructor belongs to is inferred from its result type.

```
data _ where
```

Variables are to be substituted with values (hence they have value types), so they perform no operations.

There are several possible ways to formulate the typing rules for variables and values (abstractions, constants, and boxes). Here, we make them effect polymorphic, which leads to simpler operational semantics. An alternative is to give them only the empty effect `ε`. Values would then always be wrapped with the subsumption rule.

```
`_ : ∀ {Γ E A}
  → Γ ∋ A
    -----------
  → Γ ⊢ ⟨ E ⟩ A
```

The typing rule for abstractions extends the standard rule from the simply-typed calculus with additional effects annotations. We must distinguish the effect `E` when the function is constructed (no operations are performed, so `E` is unconstrained), from the effect `F` when the function is applied (whatever effects are performed by the body of the function).

```
ƛ_ : ∀ {Γ E F B A}
  → Γ ▷ A ⊢ ⟨ F ⟩ B
    ----------------------
  → Γ ⊢ ⟨ E ⟩ (A ⇒ ⟨ F ⟩ B)
```

Applications ensure that the effect of the function matches the ambient one `E`.

```
_·_ : ∀ {Γ E A B}
  → Γ ⊢ ⟨ E ⟩ (A ⇒ ⟨ E ⟩ B)
  → Γ ⊢ ⟨ E ⟩ A
    ----------------------
  → Γ ⊢ ⟨ E ⟩ B
```

Primitive constants (`true`, `false`, `n ∈ ℕ`) and operators (`_||_`, `_+_`).

```
$_ : ∀ {Γ E ι}
  → rep ι
    -------
  → Γ ⊢ ⟨ E ⟩ ($ ι)

_(_)_ : ∀ {Γ E ι ι′ ι″}
  → Γ ⊢ ⟨ E ⟩ ($ ι)
  → (rep ι → rep ι′ → rep ι″)
  → Γ ⊢ ⟨ E ⟩ ($ ι′)
    ------------------------
  → Γ ⊢ ⟨ E ⟩ ($ ι″)
```

A cast between computation types `P` to `Q` checks during run time that the inner computation, of type `P`, behaves like a computation of type `Q`: the operations performed by the inner computation must belong to the effect of `Q`, and the resulting value is to be wrapped or unwrapped according to the return type of `Q`.

Note how casts looks similar to handlers (defined below).

```
cast : ∀ {Γ E A B}
  → A => B
  → Γ ⊢ ⟨ E ⟩ A
    -------
  → Γ ⊢ ⟨ E ⟩ B

cast^e : ∀ {Γ E F A}
  → E =>^e F
  → Γ ⊢ ⟨ E ⟩ A
    -----------
  → Γ ⊢ ⟨ F ⟩ A
```

A *box* `(M ⇑ g)` constructs a value of the dynamic type ⋆: it is a pair of a typed term `M` and a "tag" `g` which is to be inspected by run-time downcasts. It is generated by casts to ⋆.

```
_⇑_ : ∀ {Γ G E}
  → Γ ⊢ ⟨ E ⟩ G
  → Ground G
    -----------
  → Γ ⊢ ⟨ E ⟩ ⋆
```

When a cast fails, it raises `blame`.

```
blame : ∀ {Γ A}
    -----
  → Γ ⊢ A
```

A computation `perform e M` performs an operation `e` with arguments (request) `M`, returning a response of type `response e`.

We use the standard technique of *fording* [McBride, 2000], where we replace `response e` as an index in the result type of `perform-` with a fresh variable `A` and a propositional equality `response e ≡ A`.

```
perform- : ∀ {Γ E e A}
  → e ∈☆ E
  → Γ ⊢ ⟨ E ⟩ request e
  → response e ≡ A
    ------------------
  → Γ ⊢ ⟨ E ⟩ A
```

A pattern synonym to hide the equality argument in `perform-`.

```
pattern perform e∈E M = perform- e∈E M refl
```

Whereas functions are essentially maps between value types, handlers are maps between computation types `P ⇒^h Q`, whose syntax is defined below.

```
data _ where

  handle : ∀ {Γ P Q}
```

```
      → Γ ⊢ P ⇒ʰ Q
      → Γ ⊢ P
        -----------
      → Γ ⊢ Q
```

A handler `H ⦂ Γ ⊢ P ⇒ʰ Q` consists of: a list of operations it handles (`Hooks`), which will be subtracted from the effects of the inner computation (`Hooks-handled`); a *return clause* (`on-return`), which is a continuation to be called when the inner computation returns a value; and a list of *operation clauses* (`on-perform`), one for every operation in `Hooks`. We collect those components in a record.

The type of operation clauses is given by the auxiliary definition `On-Perform`. `All` is the type of list indexed by lists.

```
On-Perform ⦂ Context → CType → List Op → Set
On-Perform Γ Q Hooks =
  All (λ e →
      Γ ▷ request e ▷ (response e ⇒ Q) ⊢ Q)
    Hooks

record _⊢_⇒ʰ_ Γ P Q where
  inductive
  open CType
  field
    Hooks ⦂ List Op
    Hooks-handled ⦂
      P ,effects ≡ (Hooks ++☆ Q ,effects)
    on-return ⦂ Γ ▷ P ,returns ⊢ Q
    on-perform ⦂ On-Perform Γ Q Hooks

open _⊢_⇒ʰ_ public
```

4.3 Values

The values of our calculus are abstractions, constants, and boxes, as defined by the `Value` predicate.

```
data Value {Γ E} ⦂ ∀ {A} → Γ ⊢ ⟨ E ⟩ A → Set where

  ƛ_ ⦂ ∀ {F A B}
    → (N ⦂ Γ ▷ A ⊢ ⟨ F ⟩ B)
      --------------
    → Value (ƛ N)

  $_ ⦂ ∀ {ι}
    → (k ⦂ rep ι)
      ------------------
    → Value ($ k)

  _⇑_ ⦂ ∀{G}{V ⦂ Γ ⊢ ⟨ E ⟩ G}
    → (v ⦂ Value V)
    → (g ⦂ Ground G)
      ------------------
    → Value (V ⇑ g)
```

Extract term from evidence that it is a value.

```
value : ∀ {Γ P} {V : Γ ⊢ P} → Value V → Γ ⊢ P
value {V = V} _ = V
```

Values won't reduce. In particular, they don't perform any effect. The `gvalue` function generalizes a value to any effect.

```
gvalue : ∀ {Γ E A} {V : Γ ⊢ ⟨ E ⟩ A}
  → (v : Value V)
    -------------
  → ∀ {F} → Γ ⊢ ⟨ F ⟩ A
gvalue (ƛ N) = ƛ N
gvalue ($ k) = $ k
gvalue (v ⇑ g) = gvalue v ⇑ g

gvalue′ : ∀ {Γ E A} {V : Γ ⊢ ⟨ E ⟩ A}
  → (v : Value V)
  → ∀ {F} → Value (gvalue v {F = F})
gvalue′ (ƛ N) = ƛ N
gvalue′ ($ k) = $ k
gvalue′ (v ⇑ g) = gvalue′ v ⇑ g
```

5 OPERATIONAL SEMANTICS

In this section, we define the operational semantics as a small-step reduction relation. We prove progress, and since the proof is constructive, it doubles as an evaluation function which we can apply on examples.

5.1 Frames

Frames are "terms with a hole". Frames are also known as evaluation contexts, but the identifier `Context` is already taken in our development. They are used to define a congruence rule for reduction, *i.e.,* the contexts under which reduction may happen, as well as to represent continuations for effect handlers.

```
data Frame (Γ : Context) (C : CType) :
  CType → Set where
```

The base case is the empty frame.

```
  □ : Frame Γ C C
```

There are two frame constructors for applications: one where the hole is on the left of the application `_·_`, and one where the hole is on the right. To make the semantics deterministic, we require that we can only focus on the right operand once the left one is a value.

```
  [_]·_ :
      (ℰ : Frame Γ C (⟨ E ⟩ (A ⇒ ⟨ E ⟩ B)))
    → (M : Γ ⊢ ⟨ E ⟩ A)
      ---------------
    → Frame Γ C (⟨ E ⟩ B)

  _·[_] : {V : Γ ⊢ ⟨ E ⟩ (A ⇒ ⟨ E ⟩ B)}
    → (v : Value V)
    → (ℰ : Frame Γ C (⟨ E ⟩ A))
```

```
    ---------------
  → Frame Γ C (⟨ E ⟩ B)
```

Primitive operators follow the same logic, requiring the left operand to be a value before reducing the right operand.

```
[_](_)_ : ∀ {ι ι′ ι″}
  → (ℰ : Frame Γ C (⟨ E ⟩ ($ ι)))
  → (_⊕_ : rep ι → rep ι′ → rep ι″)
  → (N : Γ ⊢ ⟨ E ⟩ ($ ι′))
    ------------------
  → Frame Γ C (⟨ E ⟩ ($ ι″))

_(_)[_] : ∀ {ι ι′ ι″} {V : Γ ⊢ ⟨ E ⟩ $ ι}
  → (v : Value V)
  → (_⊕_ : rep ι → rep ι′ → rep ι″)
  → (ℰ : Frame Γ C (⟨ E ⟩ ($ ι′)))
    ------------------
  → Frame Γ C (⟨ E ⟩ ($ ι″))
```

The other constructors represent term constructors with only one immediate subterm.

```
[_]⇑_ : ∀ {E G}
  → (ℰ : Frame Γ C (⟨ E ⟩ G))
  → (g : Ground G)
    --------------
  → Frame Γ C (⟨ E ⟩ ⋆)

`cast_[_] : ∀ {E A B}
  → (±a : A => B)
  → (ℰ : Frame Γ C (⟨ E ⟩ A))
    -------------
  → Frame Γ C (⟨ E ⟩ B)

`cast⁽ᵉ⁾_[_] : ∀ {E F A}
  → (±e : E =>ᵉ F)
  → (ℰ : Frame Γ C (⟨ E ⟩ A))
    -------------
  → Frame Γ C (⟨ F ⟩ A)

″perform_[_]_ : ∀ {E e}
  → e ∈☆ E
  → Frame Γ C (⟨ E ⟩ request e)
  → ∀ {A}
  → response e ≡ A
  → Frame Γ C (⟨ E ⟩ A)

′handle_[_] : ∀ {P Q}
  → Γ ⊢ P ⇒ʰ Q
  → Frame Γ C P
    -----------
  → Frame Γ C Q

pattern ′perform_[_] e ℰ = ″perform e [ ℰ ] refl
```

The plug function inserts an expression into the hole of a frame.

```
_⟦_⟧ : ∀{Γ P B} → Frame Γ P B → Γ ⊢ P → Γ ⊢ B
```

Composition of two frames

```
_⨟_ : Frame Γ Q R → Frame Γ P Q → Frame Γ P R
```

Composition and plugging

```
⨟-lemma : ∀ {Γ P B C}
  → (ℰ : Frame Γ B C)
  → (F : Frame Γ P B)
  → (M : Γ ⊢ P)
    ----------------------------
  → ℰ ⟦ F ⟦ M ⟧ ⟧ ≡ (ℰ ⨟ F) ⟦ M ⟧

bound^e : E =>^e F → Op → Set
bound^e (+ e) = λ _ → ⊥
bound^e (- id) = λ _ → ⊥
bound^e (- ¡≤☆ {E = F}) = λ e → ¬ e ∈ F

∈-bound^e : {op : Op} → op ∈☆ F → (±e : E =>^e F) → ¬ bound^e ±e op
∈-bound^e (¡ op∈F) (- ¡≤☆) ¬op∈F = ¬op∈F op∈F

-- Set of operations bound by a frame -- notation from Shallow Effect Handlers
bound : Frame Γ P Q → Op → Set
bound □ = λ _ → ⊥ -- Empty set
bound ([ ℰ ]· M) = bound ℰ
bound (v ·[ ℰ ]) = bound ℰ
bound ([ ℰ ]( _⊕_ ) N) = bound ℰ
bound (v ( _⊕_ )[ ℰ ]) = bound ℰ
bound ([ ℰ ]↑ g) = bound ℰ
bound (″perform x [ ℰ ] x₁) = bound ℰ
bound (′handle H [ ℰ ]) = (_∈ H ·Hooks) ∪ bound ℰ
bound (`cast ±a [ ℰ ]) = bound ℰ
bound (`cast^e ±e [ ℰ ]) = bound^e ±e ∪ bound ℰ
```

`handled e ℰ` means that the operation `e` is handled by the evaluation context $\mathscr{E}$: either $\mathscr{E}$ contains a handler where `e` is one of its hooks, or $\mathscr{E}$ contains a cast where `e` is not allowed by the codomain of the cast.

```
handled : ∀ e → Frame Γ P Q → Set
handled e ℰ = bound ℰ e
```

For casts, this definition unconditionally checks whether `e` is in the codomain of the cast. Those checks are actually only necessary for downcasts (from ☆ to a static effeect).

An evaluation context $\mathscr{E}_0$ containing only an upcast may never raise blame: no effects are handled by $\mathscr{E}_0$.

```
upcast-handled : ∀ {Γ E F A P} (p : E ≤^e F) (ℰ : Frame Γ P (⟨ E ⟩ A)) {e : Op}
  → ¬ handled e ℰ → ¬ handled e (`cast^e (+ p) [ ℰ ])
upcast-handled id ℰ e//ℰ (inj₂ e∈ℰ) = e//ℰ e∈ℰ
upcast-handled ¡≤☆ ℰ e//ℰ (inj₂ e∈ℰ) = e//ℰ e∈ℰ
```

```
upcast-safety : ∀ {Γ E F A} (E≤F : E ≤ᵉ F) →
  let ℰ₀ : Frame Γ (⟨ E ⟩ A) (⟨ F ⟩ A)
      ℰ₀ = `castᵉ (+ E≤F) [ □ ] in
  ∀ (e : Op) → e ∈☆ E → ¬ handled e ℰ₀
upcast-safety ¡≤☆ e e∈E (inj₁ ¬e∈☆) = ¬e∈☆
upcast-safety id e e∈E (inj₁ ¬e∈E) = ¬e∈E
```

An operation **e** is not handled by a cast **±p** if **e** is not an element of the target effect of the cast.

```
¬handled-cast : ∀ {e}
    {±p : E =>ᵉ F}
    (ℰ : Frame Γ P (⟨ E ⟩ A))
  → e ∈☆ F
  → ¬ handled e ℰ
    ------------------------
  → ¬ handled e (`castᵉ ±p [ ℰ ])
¬handled-cast {±p = - ¡≤☆} ℰ (¡ e∈F) ¬e//ℰ (inj₁ ¬e∈F) = ¬e∈F e∈F
¬handled-cast ℰ e∈F ¬e//ℰ (inj₂ e//ℰ) = ¬e//ℰ e//ℰ
```

An operation **e** is not handled by a handler if **e** is not one of its hooks.

```
¬handled-handle : ∀ {e}
    {H : Γ ⊢ P ⇒ʰ Q} (ℰ : Frame Γ P′ P)
  → ¬ e ∈ Hooks H
  → ¬ handled e ℰ
    ----------------------------
  → ¬ handled e (′handle H [ ℰ ])
¬handled-handle ℰ ¬e∈H ¬e//ℰ (inj₁ e∈H)
  = ¬e∈H e∈H
¬handled-handle ℰ ¬e∈H ¬e//ℰ (inj₂ e//ℰ)
  = ¬e//ℰ e//ℰ
```

If a computation under a handler raises an effect **e** which is not a hook of the handler, then **e** must be in the resulting effect of the handler.

```
¬∈-handler : ∀ {e} (H : Γ ⊢ ⟨ E ⟩ A ⇒ʰ ⟨ F ⟩ B) → e ∈☆ E → ¬ e ∈ H ↓Hooks → e ∈☆ F
¬∈-handler H e∈E ¬e∈H rewrite Hooks-handled H
  with ∈☆-++☆⁻ e∈E
... | inj₁ e∈H = ⊥-elim (¬e∈H e∈H)
... | inj₂ e∈F = e∈F

¬handled-∈ : ∀ {e}
  (ℰ : Frame Γ (⟨ E ⟩ A) (⟨ F ⟩ B))
  → ¬ handled e ℰ → e ∈☆ E → e ∈☆ F
¬handled-∈ □ _ e = e
¬handled-∈ ([ ℰ ]↓ M) = ¬handled-∈ ℰ
¬handled-∈ (v ↓[ ℰ ]) = ¬handled-∈ ℰ
¬handled-∈ ([ ℰ ]⟨ _⊕_ ⟩ N) = ¬handled-∈ ℰ
¬handled-∈ (v ⟨ _⊕_ ⟩[ ℰ ]) = ¬handled-∈ ℰ
¬handled-∈ ([ ℰ ]↑ g) = ¬handled-∈ ℰ
¬handled-∈ (″perform e [ ℰ ] x₁) = ¬handled-∈ ℰ
¬handled-∈ `cast ±a [ ℰ ] = ¬handled-∈ ℰ
```

```
¬handled-∈ `castᵉ + id [ ℰ ] ¬e//ℰ = ¬handled-∈ ℰ (¬e//ℰ ∘ inj₂)
¬handled-∈ `castᵉ + ¡≤☆ [ ℰ ] ¬e//ℰ e = ☆
¬handled-∈ `castᵉ - id [ ℰ ] ¬e//ℰ = ¬handled-∈ ℰ (¬e//ℰ ∘ inj₂)
¬handled-∈ `castᵉ - ¡≤☆ [ ℰ ] ¬e//ℰ e = ¡ (¬¬-dec (_ ∈? _) (¬e//ℰ ∘ inj₁))
¬handled-∈ (′handle H [ ℰ ]) ¬e//ℰ e
  = ¬∈-handler H (¬handled-∈ ℰ (¬e//ℰ ∘ inj₂) e) (¬e//ℰ ∘ inj₁)
```

5.2 Decomposing a cast

The following construction unifies the behaviors of some casts.

```
data _==>_ ⁞ Type → Type → Set where

  id ⁞ ∀ {A}
       -------
    → A ==> A

  _⇒⟨_⟩_ ⁞ ∀ {A A′ E E′ B B′}
    → A′ => A
    → E =>ᵉ E′
    → B => B′
       -----------------
    → A ⇒ ⟨ E ⟩ B ==> A′ ⇒ ⟨ E′ ⟩ B′

  other ⁞ ∀ {A B}
       -------
    → A ==> B

split ⁞ ∀ {A B} → A => B → A ==> B
split (+ id)     = id
split (- id)     = id
split (+ s ⇒ ⟨ e ⟩ t) = (- s) ⇒⟨ + e ⟩ (+ t)
split (- s ⇒ ⟨ e ⟩ t) = (+ s) ⇒⟨ - e ⟩ (- t)
split (+ p ⇑ g) = other
split (- p ⇑ g) = other
```

5.3 Wrapping functions

```
ƛ-wrap ⁞ ∀ (∓s ⁞ A′ => A) (±t ⁞ B => B′) (±e ⁞ E =>ᵉ E′)
  → (∀ {F} → Γ ⊢ ⟨ F ⟩ (A ⇒ ⟨ E ⟩ B))
  → (∀ {F} → Γ ⊢ ⟨ F ⟩ (A′ ⇒ ⟨ E′ ⟩ B′))
ƛ-wrap ∓s ±t ±e M =
  ƛ cast ±t (castᵉ ±e (lift M ⁞ (cast ∓s (` Z))))
```

5.4 Reduction

We first define a reduction relation `_↦_` on redexes, and then close it under congruence, as `_—»_`.

```
data _↦_ {Γ} ⁞
  ∀ {P} → (_ _ ⁞ Γ ⊢ P) → Set where
```

Because there are effects in our type system, we must modify the β rule a bit from its standard formulation. In the application `(ƛ N) ⁞ W`, the value `W` is a term with some effect `E`, but when

substituting `W` in `N`, the substituted variable may occur in contexts with different effects `E`, in which case `W` would be an ill-typed replacement. Hence we generalize `W` before applying a substitution.

```
β : ∀ {N : Γ ▷ A ⊢ ⟨ E ⟩ B} {W : Γ ⊢ ⟨ E ⟩ A}
  → (w : Value W)
    --------------------
  → (ƛ N) · W ↦ N [ gvalue w ]
```

The `δ` rule reduces primitive operators applied to constants.

```
δ : ∀ {ι ι′ ι″}
    {_⊕_ : rep ι → rep ι′ → rep ι″}
    {k : rep ι} {k′ : rep ι′}
    ----------------------------------------
  → _(_)_ {E = E} ($ k) _⊕_ ($ k′) ↦ $ (k ⊕ k′)
```

The next six rules have to do with casts. The first five are based on standard cast calculus rules, describing how to cast values. The sixth is a rule related to casting effects.

The `ident` rule removes identity casts, after the casted computation returned a value.

```
ident : ∀ {V : Γ ⊢ ⟨ E ⟩ A}
    {±a : A => A}
  → split ±a ≡ id
  → (v : Value V)
    -------------
  → cast ±a V ↦ gvalue v
```

The `wrap` rule reduces casts between function types. The cast `±p` is split into two casts, `∓s` between domains and `±t` codomains; the function being cast is wrapped using `ƛ-wrap`, composing it with those two casts.

```
wrap : {N : Γ ▷ A ⊢ ⟨ E ⟩ B}
    {∓s : A′ => A} {±t : B => B′} {±e : E =>ᵉ E′}
    {±p : A ⇒ ⟨ E ⟩ B => A′ ⇒ ⟨ E′ ⟩ B′}
  → split ±p ≡ ∓s ⇒⟨ ±e ⟩ ±t
    ---------------------------------------
  → cast {E = F} ±p (ƛ N) ↦ ƛ-wrap ∓s ±t ±e (ƛ N)
```

The `expand` rule reduces an upcast to ⋆ to a box.

```
expand : ∀{V : Γ ⊢ ⟨ E ⟩ A}
    {p : A ≼ G}
  → Value V
  → (g : Ground G)
    ------------------------------
  → cast (+ (p ⇑ g)) V ↦ cast (+ p) V ⇑ g
```

The `collapse` rule reduces a downcast `(p ⇑ g)` from ⋆, in which case the value under the cast must be a box `(V ⇑ g)`, by unwrapping the box, provided the tag `g` in the box and in the cast match.

```
collapse : ∀ {V : Γ ⊢ ⟨ E ⟩ G}
    {p : A ≼ G}
  → Value V
  → (g : Ground G)
```

```
    ------------------------------
  → cast (- (p ⇑ g)) (V ⇑ g)
    ↦ cast (- p) V
```

The `collide` rule reduces a downcast `(p ⇑ h)` applied to a box `(V ⇑ g)` when the tags `g` and `h` don't match. This raises `blame`.

```
collide : ∀{G H} {V : Γ ⊢ ⟨ E ⟩ G}
    {p : A ≲ H}
  → Value V
  → (g : Ground G)
  → (h : Ground H)
  → G ≢ H
    ----------------------------
  → cast (- (p ⇑ h)) (V ⇑ g) ↦ blame
```

Casts contain both a cast on values (whose behavior is defined by the previous five rules), and a cast on effects. The next rule describes how such a cast may fail: the computation under the cast performs an effect which: is not handled by any inner handler and is not a member of the target effect `F` of the cast.

```
blame^e : ∀ {e} {e∈E′ : e ∈☆ E′} {V} {M}
    {ℰ : Frame Γ (⟨ E′ ⟩ response e) (⟨ E ⟩ A)}
    {±e : E =>^e F}
  → ¬ e ∈☆ F
  → ¬ handled e ℰ
  → Value V
  → M ≡ ℰ ⟦ perform e∈E′ V ⟧
    --------------------------
  → cast^e ±e M ↦ blame

cast^e-return : ∀ {V : Γ ⊢ ⟨ E ⟩ A} {±e : E =>^e F}
  → (v : Value V)
    ------------------------
  → cast^e ±e V ↦ gvalue v
```

Note that there is no rule for "successful effect casts". When an effect passes successfully through a cast, it simply keeps being raised until a matching handler or cast is found.

Handlers have two rules. When the handled computation returns a value, the return clause is invoked.

```
handle-value : ∀ {H : Γ ⊢ P ⇒^h Q} {V}
  → (v : Value V)
    -------------
  → handle H V ↦ on-return H [ gvalue v ]
```

When the handled computation performs an operation, the corresponding operation clause of the closest matching handler is invoked. This rule expressed in Agda looks rather complex.

In the right-hand side of the reduction, `All.lookup` finds the corresponding clause, given a proof that the operation `e` is an element of the handler's `Hooks`. Two substitutions follow, because operation clauses extend the context with two variables, one for the operation's request payload, and one for the continuation. Since the continuation variable occurs at the end of the context, it must be substituted first.

```
clause : Γ ▷ request e ▷ (response e ⇒ Q) ⊢ Q

handle-perform : ∀ {e} {e∈E : e ∈☆ E}
    {H : Γ ⊢ P ⇒ʰ Q} {V 𝒞 e∈Hooks}
  → (v : Value V)
  → ¬ handled e 𝒞
    -- ensures H is the first matching handler
  → (e ∈? Hooks H) ≡ yes e∈Hooks
    -- ensures this is the first matching clause within H
    -- TODO: a more declarative reformulation?
  → handle H (𝒞 ⟦ perform e∈E V ⟧)
    ↦ All.lookup (on-perform H) e∈Hooks
        [ ƛ (handle (liftʰ (liftʰ H)) (liftᶠ (liftᶠ 𝒞) ⟦ ` Z ⟧)) ]
        [ gvalue v ]
```

The top-level reduction relation `_—»_` allows reduction to happen under any frame. Again, we use fording to keep the frame substitution function out of the type's indices.

```
data _—→_ :
  (Γ ⊢ P) → (Γ ⊢ P) → Set where

  ξξ :
       {M N : Γ ⊢ P} {M′ N′ : Γ ⊢ Q}
    →  (𝒞 : Frame Γ P Q)
    →  M′ ≡ 𝒞 ⟦ M ⟧
    →  N′ ≡ 𝒞 ⟦ N ⟧
    →  M ↦ N
       --------
    →  M′ —→ N′
```

Notation to hide the fording indices.

```
pattern ξ 𝒞 M—→N = ξξ 𝒞 refl refl M—→N
```

That makes ξ a constructor with the following type:

```
ξ  :  (𝒞 : Frame Γ P Q)
   →  M ↦ N
      --------
   →  𝒞 ⟦ M ⟧ —→ 𝒞 ⟦ N ⟧
```

5.5 Reflexive and transitive closure of reduction

5.6 Progress

Every term that is well typed and closed either takes a reduction step or belongs to one of several well-defined classes of normal forms: `blame`, a value, or a `pending` operation in some context. The following data type lists those possible cases.

```
data Progress {P} : (∅ ⊢ P) → Set where

  step : ∀ {M N : ∅ ⊢ P}
    → M —→ N
      ----------
    → Progress M

  done : ∀ {M : ∅ ⊢ P}
```

```
    → Value M
      ----------
    → Progress M

  blame ∣ ∀ {Q}
    → (E ∣ Frame ∅ Q P)
      --------------------
    → Progress (E ⟦ blame ⟧)

  pending ∣ ∀ {e} {V} ℰ
    → (e∈E ∣ e ∈☆ E)
    → Value V
    → ¬ handled e ℰ
      -----------------
    → Progress (ℰ ⟦ perform e∈E V ⟧)
```

As one subcase of the proof of progress, we prove that a `cast` applied to a value always takes a step.

```
progress± ∣ ∀ {V ∣ ∅ ⊢ ⟨ E ⟩ A}
  → (v ∣ Value V)
  → (±a ∣ A => B)
    --------------------
  → ∃[ M ](cast ±a V ↦ M)

progress ∣
  (M ∣ ∅ ⊢ P)
  -----------
  → Progress M
```

6 PRECISION ON TERMS

The gradual guarantee says that the behavior of a program doesn't significantly change when we add or remove type annotations. Formally, if a well-typed term `M` steps to `N`, then a term `M´` which is less precise than `M` steps to a term `N´` which is less precise than `N´`.

In order to state this theorem in Agda, the first step is to define the precision relation on contexts, terms, and frames.

6.1 Precision on contexts

Viewing contexts as lists of types, context precision is the pointwise lifting of type precision.

```
data _⊑ᴳ_ ∣ Context → Context → Set where

  ∅ ∣
      ------
      ∅ ⊑ᴳ ∅

  _▷_ ∣
      Γ ⊑ᴳ Γ´
    → A ⊑ A´
      ------------------
    → Γ ▷ A ⊑ᴳ Γ´ ▷ A´
```

Context precision is reflexive.

```
idᴳ ⦂ Γ ≼ᴳ Γ
idᴳ {∅}     = ∅
idᴳ {Γ ▷ A} = idᴳ ▷ id
```

Context precision is transitive.

```
_⨟ᴳ_ ⦂ Γ ≼ᴳ Γ′ → Γ′ ≼ᴳ Γ″ → Γ ≼ᴳ Γ″
∅ ⨟ᴳ ∅                    = ∅
(Γ≼ ▷ A≼) ⨟ᴳ (Γ′≼ ▷ A′≼) = (Γ≼ ⨟ᴳ Γ′≼) ▷ (A≼ ⨟ A′≼)
```

From a proof-relevant perspective, context precision defines a category, where `idᴳ` is the identity morphism, and `_⨟ᴳ_` is morphism composition. They satisfy the following laws:

```
left-idᴳ ⦂ (Γ≼Δ ⦂ Γ ≼ᴳ Δ) → idᴳ ⨟ᴳ Γ≼Δ ≡ Γ≼Δ
left-idᴳ ∅ = refl
left-idᴳ (Γ≼Δ ▷ p) rewrite left-idᴳ Γ≼Δ
                    | left-id p = refl

right-idᴳ ⦂(Γ≼Δ ⦂ Γ ≼ᴳ Δ) → Γ≼Δ ⨟ᴳ idᴳ ≡ Γ≼Δ
right-idᴳ ∅ = refl
right-idᴳ (Γ≼Δ ▷ p) rewrite right-idᴳ Γ≼Δ
                     | right-id p = refl
```

6.2 Precision on variables

Variable precision `Γ≼ ⊢ x ≼ˣ x′ ⨾ A≼` relates variables `x` and `x′` that structurally represent the same natural number, *i.e.,* the same index in contexts of the same length.

Viewed in a proof-relevant manner, context precision is a type of heterogeneous lists of type precision proofs, and variable precision is the corresponding type of indices.

```
data _⊢_≼ˣ_⨾_ ⦂ Γ ≼ᴳ Γ′
              → Γ ∋ A
              → Γ′ ∋ A′
              → A ≼ A′
              → Set where

  Z≼Z ⦂ {Γ≼ ⦂ Γ ≼ᴳ Γ′} {A≼ ⦂ A ≼ A′}
      ----------------------
    → Γ≼ ▷ A≼ ⊢ Z ≼ˣ Z ⨾ A≼

  S≼S ⦂ ∀ {x x′} {Γ≼ ⦂ Γ ≼ᴳ Γ′}
      {A≼ ⦂ A ≼ A′} {B≼ ⦂ B ≼ B′}
    → Γ≼ ⊢ x ≼ˣ x′ ⨾ A≼
      --------------------------
    → Γ≼ ▷ B≼ ⊢ S x ≼ˣ S x′ ⨾ A≼
```

6.3 Commuting diagram

When defining term precision, the key rules are those that relate casts. If (1) `M ⦂ Γ ⊢ P` is more precise than `M′ ⦂ Γ′ ⊢ P′`, (2) there is a cast `P =>ᶜ Q`, and (3) `Q` is more precise than `P′`, then `cast ±p M ⦂ ⊢ Γ ⊢ Q` is more precise than `M′ ⦂ Γ′ ⊢ P′`. (And similarly for a cast on the right.)

In addition, the cast `P =>ᶜ Q` and the precision relations `P ≼ᶜ P′` and `Q ≼ᶜ Q′` should form a commutative triangle.

```
commute≤ᶜ : (±p : P =>ᶜ Q) (q : Q ≤ᶜ R)
            (r : P ≤ᶜ R) → Set
commute≤ᶜ (+ p) q r = p ⨟ᶜ q ≡ r
commute≤ᶜ (- p) q r = p ⨟ᶜ r ≡ q
```

A similar commutative triangle arises for casts on the right of term precision.

```
≤commuteᶜ : (p : P ≤ᶜ Q) (±q : Q =>ᶜ R)
            (r : P ≤ᶜ R) → Set
≤commuteᶜ p (+ q) r = p ⨟ᶜ q ≡ r
≤commuteᶜ p (- q) r = r ⨟ᶜ q ≡ p
```

The same commutative triangles can be defined on value type precision.

```
commute≤ : (±p : A => B) (q : B ≤ C)
           (r : A ≤ C) → Set
commute≤ (+ p) q r = p ⨟ q ≡ r
commute≤ (- p) q r = p ⨟ r ≡ q

≤commute : (p : A ≤ B) (±q : B => C)
           (r : A ≤ C) → Set
≤commute p (+ q) r = p ⨟ q ≡ r
≤commute p (- q) r = r ⨟ q ≡ p
```

6.4 Precision on terms

Term precision `_⊢_≤ᴹ_⦂_` and handler precision `_⊢_≤_⦂_⇒ʰ_` are defined mutually recursively.

```
data _⊢_≤ᴹ_⦂_ {Γ Γ′} (Γ≤ : Γ ≤ᴳ Γ′)
  : ∀ {A A′} → Γ ⊢ A → Γ′ ⊢ A′ → A ≤ᶜ A′ → Set
record _⊢_≤_⦂_⇒ʰ_ {Γ Γ′} (Γ≤ : Γ ≤ᴳ Γ′)
  {P P′ Q Q′}
  (H : Γ ⊢ P ⇒ʰ Q)
  (H′ : Γ′ ⊢ P′ ⇒ʰ Q′)
  (P≤ : P ≤ᶜ P′)
  (Q≤ : Q ≤ᶜ Q′) : Set
```

Start by defining term precision. For constructs other than casts, the general rule is "a term `M` is more precise than `M′` if the subterms of `M` are more precise than the subterms of of `M′`".

```
data _⊢_≤ᴹ_⦂_ {Γ Γ′} Γ≤ where
```

We defined variable precision `_⊢_≤ˣ_⦂_` previously. Note that the effects on both sides may be arbitrary effects `E` and `E′` satisfying `E ≤ᵉ E′`.

```
`≤` : ∀ {x x′} {pᵉ : E ≤ᵉ E′} {p : A ≤ A′}
  → Γ≤ ⊢ x ≤ˣ x′ ⦂ p
    ----------------------
  → Γ≤ ⊢ ` x ≤ᴹ ` x′ ⦂ ⟨ pᵉ ⟩ p
```

The rules for abstraction and application are quantified over precision witnesses between function types `p : A ⇒ P ≤ A′ ⇒ P′`, which can be projected to precision witnesses between their domains `dom p : A ≤ A′` and codomains `P ≤ᶜ P′`. This allows `p` to be either `_⇒_` or `id`. This lets us use `id` uniformly in the proof of reflexivity for term precision.

```
ƛ≤ƛ : ∀ {N N′} {e : E ≤ᵉ E′} {p : A ⇒ P ≤ A′ ⇒ P′} {a f b}
  → split⇒ p ≡ (a , f , b)
```

```
    → Γ≤ ▷ a ⊢ N ≤ᴹ N′ ⨟ ⟨ f ⟩ b
      ---------------------------
    → Γ≤ ⊢ ✗ N ≤ᴹ ✗ N′ ⨟ ⟨ e ⟩ p

  '≤' : ∀ {L L′ M M′} {p : A ⇒ P ≤ A′ ⇒ P′} {a e b}
    → split⇒ p ≡ (a , e , b)
    → Γ≤ ⊢ L ≤ᴹ L′ ⨟ ⟨ e ⟩ p
    → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ a
      ---------------------------
    → Γ≤ ⊢ L ' M ≤ᴹ L′ ' M′ ⨟ ⟨ e ⟩ b
```

Base types are only related by `id`, which thus serves as the index for constants and primitive operators.

```
  $≤$ : ∀ {ι} {pᵉ : E ≤ᵉ E′}
    → (k : rep ι)
      ----------------------
    → Γ≤ ⊢ $ k ≤ᴹ $ k ⨟ ⟨ pᵉ ⟩ id

  ()≤() : ∀ {ι ι′ ι″ M M′ N N′} {pᵉ : E ≤ᵉ E′}
    → (_⊕_ : rep ι → rep ι′ → rep ι″)
    → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ pᵉ ⟩ id
    → Γ≤ ⊢ N ≤ᴹ N′ ⨟ ⟨ pᵉ ⟩ id
      ---------------------------------
    → Γ≤ ⊢ M ⟪ _⊕_ ⟫ N
         ≤ᴹ M′ ⟪ _⊕_ ⟫ N′ ⨟ ⟨ pᵉ ⟩ id
```

Handlers and effects also follow the same pattern of relating subterms. Precision between `handle` terms uses handler precision `_⊢_≤_⨟_⇒ʰ_` which will be defined below.

```
  perform≤perform : ∀ {e}
    {e∈E : e ∈☆ E} {e∈E′ : e ∈☆ E′}
    {E≤ : E ≤ᵉ E′} {M M′}
    → {eq : response e ≡ A}
    → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ E≤ ⟩ id
    → Γ≤ ⊢ perform- e∈E M eq
         ≤ᴹ perform- e∈E′ M′ eq ⨟ ⟨ E≤ ⟩ id

  handle≤handle :
    ∀ {P≤ : P ≤ᶜ P′} {Q≤ : Q ≤ᶜ Q′} {H H′ M M′}
    → Γ≤ ⊢ H ≤ H′ ⨟ P≤ ⇒ʰ Q≤
    → Γ≤ ⊢ M ≤ᴹ M′ ⨟ P≤
    → Γ≤ ⊢ handle H M ≤ᴹ handle H′ M′ ⨟ Q≤
```

Boxes have type ⋆, and their contents have ground types, which can only be related by precision if they are equal. So the relation should be witnessed by `id`.

```
  ⇑≤⇑ : ∀ {G E E′ M M′} {pᵉ : E ≤ᵉ E′}
    → (g : Ground G)
    → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ pᵉ ⟩ id
      ---------------------------
    → Γ≤ ⊢ (M ⇑ g) ≤ᴹ (M′ ⇑ g) ⨟ ⟨ pᵉ ⟩ id
```

`M` is more precise than a box `M′ ⇑ g` if `M` is more precise than the underlying term `M′`. Note the absence of a symmetric rule where the box is on the left. Intuitively, a more precisely typed term uses fewer dynamic boxes.

```
≤⇑ : ∀ {G M M′} {p : A ≤ G} {pᵉ : E ≤ᵉ E′}
  → (g : Ground G)
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ pᵉ ⟩ p
    ------------------------
  → Γ≤ ⊢ M ≤ᴹ (M′ ⇑ g) ⨟ ⟨ pᵉ ⟩ (p ⇑ g)
```

Term precision does not imply that the more precise side has fewer casts. Indeed, increasing the precision of a term may introduce more run-time checks.

For instance, consider the identity `ID = ƛ (` Z) : ∅ ⊢ ⋆ ⇒ ⟨ ☆ ⟩ ⋆`, and the term obtained from casting a monomorphic identity `ID′ = cast (+ p) ID-ℕ`, where `ID-ℕ = λ (` Z) : ∅ ⊢ ℕ ⇒ ⟨ ε ⟩ ℕ`. `ID-ℕ` is more precise than `ID`, and `ID-ℕ` contains a cast while `ID` does not.

Unlike the preceding rules, we will have separate rules for inserting casts on either side. When we insert a cast on the left with `cast≤`, the right-hand side is less precise than the term on the left-hand side before and after the cast. This results in a triangle, with vertices `P`, `Q`, `R`, where one side consists of the cast `P =>ᶜ Q`, and the other two sides are the inequalities `P ≤ᶜ R` and `Q ≤ᶜ R`. We require that triangle to commute, using the predicate `commute≤ᶜ`.

```
cast≤ : ∀ {M : Γ ⊢ ⟨ E ⟩ A} {M′ : Γ′ ⊢ ⟨ E′ ⟩ A′} {e : E ≤ᵉ E′}
    {±p : A => B} {q : B ≤ A′} {r : A ≤ A′}
  → commute≤ ±p q r
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ r
    ------------------------
  → Γ≤ ⊢ cast ±p M ≤ᴹ M′ ⨟ ⟨ e ⟩ q
```

The `≤cast` rule is symmetrical to `cast≤`.

```
≤cast : ∀ {M : Γ ⊢ ⟨ E ⟩ A} {M′ : Γ′ ⊢ ⟨ E′ ⟩ A′} {e : E ≤ᵉ E′}
    {p : A ≤ A′} {±q : A′ => B′} {r : A ≤ B′}
  → ≤commute p ±q r
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ p
    ------------------------
  → Γ≤ ⊢ M ≤ᴹ cast ±q M′ ⨟ ⟨ e ⟩ r

cast≤ : ∀ {M : Γ ⊢ ⟨ E ⟩ A} {M′ : Γ′ ⊢ ⟨ E′ ⟩ A′} {a : A ≤ A′}
    {e : E ≤ᵉ E′} {±f : E =>ᵉ F} {f′ : F ≤ᵉ E′}
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ a
    -----
  → Γ≤ ⊢ castᵉ ±f M ≤ᴹ M′ ⨟ ⟨ f′ ⟩ a

≤cast : ∀ {M : Γ ⊢ ⟨ E ⟩ A} {M′ : Γ′ ⊢ ⟨ E′ ⟩ A′} {a : A ≤ A′}
    {e : E ≤ᵉ E′} {±f : E′ =>ᵉ F′} {f′ : E ≤ᵉ F′}
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ a
    -----
  → Γ≤ ⊢ M ≤ᴹ castᵉ ±f M′ ⨟ ⟨ f′ ⟩ a

blame≤ : ∀ {A A′ M′} {p : A ≤ᶜ A′}
    --------------------
  → Γ≤ ⊢ blame ≤ᴹ M′ ⨟ p
```

A cast between function types eventually steps to a ƛ-wrap, so we add an ad-hoc rule for that construct. After a further β step, the resulting terms will related using cast≤ for wrap≤, and ≤cast for ≤wrap.

```
wrap≤ ⁝
    {N ⁝ Γ ▷ A ⊢ ⟨ E ⟩ B} {N′ ⁝ Γ′ ▷ A″ ⊢ ⟨ E″ ⟩ B″}
    {±p ⁝ A ⇒ ⟨ E ⟩ B => A′ ⇒ ⟨ E′ ⟩ B′}
    {q ⁝ A′ ⇒ ⟨ E′ ⟩ B′ ≤ A″ ⇒ ⟨ E″ ⟩ B″}
    {r ⁝ A ⇒ ⟨ E ⟩ B ≤ A″ ⇒ ⟨ E″ ⟩ B″}
    {∓s ⁝ A′ => A} {±t ⁝ B => B′} {±e ⁝ E =>ᵉ E′}
  → split ±p ≡ ∓s ⇒⟨ ±e ⟩ ±t
  → commute≤ ±p q r
  → (∀ {F F′} {f ⁝ F ≤ᵉ F′} →
      Γ≤ ⊢ ƛ N ≤ᴹ ƛ N′ ⨟ ⟨ f ⟩ r)
    ----------------------------------------
  → ∀ {F F′} {f ⁝ F ≤ᵉ F′} → Γ≤ ⊢ ƛ-wrap ∓s ±t ±e (ƛ N) ≤ᴹ ƛ N′ ⨟ ⟨ f ⟩ q
```

Here is an example reduction sequence (with some oversimplifications for conciseness) of an abstraction ƛ N under a cast applied to an argument M. Every term in this sequence is more precise than (ƛ N′) ⁝ M′ given ƛ N ≤ᴹ ƛ N′ and M ≤ᴹ M′. This is witnessed by cast≤ for the first term, wrap≤ for the second term, and a combination of cast≤ and ⁝≤⁝ for the last term.

```
                    cast ±p (ƛ N) ⁝ M
⟶              ƛ-wrap ∓s ±t (ƛ N) ⁝ M
=   (ƛ (cast ±t (lift (ƛ N)
                ⁝ cast ∓s (` Z)))) ⁝ M
⟶          cast ±t ((ƛ N) ⁝ cast ∓s M)
```

```
≤wrap ⁝
    {N ⁝ Γ ▷ A ⊢ ⟨ E ⟩ B} {N′ ⁝ Γ′ ▷ A′ ⊢ ⟨ E′ ⟩ B′}
    {p ⁝ A ⇒ ⟨ E ⟩ B ≤ A′ ⇒ ⟨ E′ ⟩ B′}
    {±q ⁝ A′ ⇒ ⟨ E′ ⟩ B′ => A″ ⇒ ⟨ E″ ⟩ B″}
    {r ⁝ A ⇒ ⟨ E ⟩ B ≤ A″ ⇒ ⟨ E″ ⟩ B″}
    {∓s ⁝ A″ => A′} {±t ⁝ B′ => B″} {±e ⁝ E′ =>ᵉ E″}
  → split ±q ≡ ∓s ⇒⟨ ±e ⟩ ±t
  → ≤commute p ±q r
  → (∀ {F F′} {f ⁝ F ≤ᵉ F′} → Γ≤ ⊢ ƛ N ≤ᴹ ƛ N′ ⨟ ⟨ f ⟩ p)
    ------------------------------------------------
  → ∀ {F F′} {f ⁝ F ≤ᵉ F′} → Γ≤ ⊢ ƛ N ≤ᴹ ƛ-wrap ∓s ±t ±e (ƛ N′) ⨟ ⟨ f ⟩ r
```

Precision between the operation clauses of handlers.

```
On-Perform ⁝
  ∀ (Γ≤ ⁝ Γ ≤ᴳ Γ′) (Q≤ ⁝ Q ≤ᶜ Q′) → ∀ {Eh Eh′}
  → Core⁝On-Perform Γ Q Eh
  → Core⁝On-Perform Γ′ Q′ Eh′ → Set
On-Perform Γ≤ Q≤
  = All₂′ λ M M′ →
      ∃[ B⇒Q≤ ] dom B⇒Q≤ ≡ id ×
                ⟨ eff B⇒Q≤ ⟩ cod B⇒Q≤ ≡ Q≤ ×
                (Γ≤ ▷ id ▷ (B⇒Q≤) ⊢ M ≤ᴹ M′ ⨟ Q≤)
```

Precision between handlers.

```
record _⊢_≤_⦂_⇒ʰ_ Γ≤ {P P′ Q Q′} H H′ P≤ Q≤ where
  inductive
  open _≤ᶜ_ using (returns)
  field
    on-return ∶
      Γ≤ ▷ returns P≤ ⊢ on-return H ≤ᴹ on-return H′ ⦂ Q≤
    on-perform ∶
      On-Perform Γ≤ Q≤ (on-perform H) (on-perform H′)

open _⊢_≤_⦂_⇒ʰ_ public
```

Term precision is reflexive. Because term precision is indexed by context precision and type precision, its reflexivity proof will be indexed by their respective reflexivity proofs.

6.5 Cast congruence

Term precision does not contain a rule to insert a cast on both sides of an inequation at once. The following lemmas derive such rules when both sides are casts with the same polarity.

```
data square {A A′ B B′} (a ∶ A ≤ A′) (b ∶ B ≤ B′) ∶ A => B → A′ => B′ → Set where
  square+ ∶ ∀ {p q} → p ⨟ b ≡ a ⨟ q → square a b (+ p) (+ q)
  square- ∶ ∀ {p q} → b ⨟ q ≡ p ⨟ a → square a b (- p) (- q)

square-refl ∶ ∀ (p ∶ A => B) → square id id p p
square-refl (+ p) = square+ (sym (left-id p))
square-refl (- p) = square- (left-id p)

cast≤cast ∶ ∀ {M ∶ Γ ⊢ ⟨ E ⟩ A} {M′ ∶ Γ′ ⊢ ⟨ E′ ⟩ A′} {Γ≤ ∶ Γ ≤ᴳ Γ′}
    {a ∶ A ≤ A′} {b ∶ B ≤ B′} {p q} {e}
  → square a b p q
  → Γ≤ ⊢ M ≤ᴹ M′ ⦂ ⟨ e ⟩ a
    -----
  → Γ≤ ⊢ cast p M ≤ᴹ cast q M′ ⦂ ⟨ e ⟩ b
cast≤cast (square+ comm) M≤M′ = cast≤ comm (≤cast refl M≤M′)
cast≤cast (square- comm) M≤M′ = ≤cast comm (cast≤ refl M≤M′)

data squareᵉ {E E′ F F′} ∶ E =>ᵉ F → E′ =>ᵉ F′ → Set where
  square+ ∶ ∀ {p q} → squareᵉ (+ p) (+ q)
  square- ∶ ∀ {p q} → squareᵉ (- p) (- q)

squareᵉ-refl ∶ ∀ (p ∶ E =>ᵉ F) → squareᵉ p p
squareᵉ-refl (+ p) = square+
squareᵉ-refl (- p) = square-

castᵉ≤castᵉ ∶ ∀ {M ∶ Γ ⊢ ⟨ E ⟩ A} {M′ ∶ Γ′ ⊢ ⟨ E′ ⟩ A′} {Γ≤ ∶ Γ ≤ᴳ Γ′}
    {a ∶ A ≤ A′} {e ∶ E ≤ᵉ E′} {f ∶ F ≤ᵉ F′} {p q}
  → squareᵉ p q
  → Γ≤ ⊢ M ≤ᴹ M′ ⦂ ⟨ e ⟩ a
    -----
  → Γ≤ ⊢ castᵉ p M ≤ᴹ castᵉ q M′ ⦂ ⟨ f ⟩ a
castᵉ≤castᵉ {e = e} (square+ {q = q}) M≤M′ = castᵉ≤ {e = e ⨟ᵉ q} (≤castᵉ M≤M′)
castᵉ≤castᵉ {e = e} (square- {p = p}) M≤M′ = ≤castᵉ {e = p ⨟ᵉ e} (castᵉ≤ M≤M′)
```

6.6 Reflexivity of term precision

Every term is at least as precise as itself. Term precision is indexed by a type precision proof, and to relate a term with itself, a natural choice is to index it by the reflexivity proof `id` (or `⟨ id ⟩ id` which is the reflexivity proof for computation type precision). Reflexivity depends crucially on the fact that the rules for abstraction `ƛ≼ƛ`, application `·≼·`, and handlers `handle≼handle` are parameterized by proofs for type precision on functions, instead of constructing them using `_⇒_` which is distinct from `id`.

```
reflᴹ : ∀ {Γ P}
  → (M : Γ ⊢ P)
    ------------------
  → idᴳ ⊢ M ≼ᴹ M ⨟ ⟨ id ⟩ id
```

6.7 Precision on frames

In the graduality proof, in the case for the rule `handle-perform`, we have a relation `M ≼ᴹ M′` where `M` is of the form `handle H (ℰ [ perform e V ])`. This implies that `M′` is also of that form `handle H′ (ℰ′ [ perform e V′ ])`, with `H ≼ʰ H′`, `ℰ ≼ ℰ′`, and `V ≼ᴹ V′`. Thus, we need to define precision on frames `ℰ ≼ ℰ′`, whose rules can be derived from the rules of precision on terms.

```
infix 3 _⊢_⇒ᶠ_∋_≼_

data _⊢_⇒ᶠ_∋_≼_ {Γ Γ′} (Γ≼ : Γ ≼ᴳ Γ′)
                {P P′} (P≼ : P ≼ᶜ P′)
       : ∀ {Q Q′} (Q≼ : Q ≼ᶜ Q′)
       → Frame Γ P Q
       → Frame Γ′ P′ Q′
       → Set where
  □ : Γ≼ ⊢ P≼ ⇒ᶠ P≼ ∋ □ ≼ □

  by_[_]·_ : ∀ {ℰ ℰ′} {M M′}
      {p : A ⇒ ⟨ E ⟩ B ≼ A′ ⇒ ⟨ E′ ⟩ B′} {e a b}
    → split⇒ p ≡ (a , e , b)
    → (ℰ≼ : Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ p ∋ ℰ ≼ ℰ′)
    → (M≼ : Γ≼ ⊢ M ≼ᴹ M′ ⨟ ⟨ e ⟩ a)
      ---------------------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ b ∋ [ ℰ ]· M ≼ [ ℰ′ ]· M′

  by_,_·[_] : ∀ {V V′} {ℰ ℰ′}
      {p : A ⇒ ⟨ E ⟩ B ≼ A′ ⇒ ⟨ E′ ⟩ B′} {e a b}
    → split⇒ p ≡ (a , e , b)
    → ((v , v′ , _) :
        Value V × Value V′ ×
        (Γ≼ ⊢ V ≼ᴹ V′ ⨟ ⟨ e ⟩ p))
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ a ∋ ℰ ≼ ℰ′
      ---------------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ b ∋ v ·[ ℰ ] ≼ v′ ·[ ℰ′ ]

  [_](_)_ : ∀ {ι ι′ ι″} {ℰ ℰ′} {M M′}
    → (ℰ≼ : Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id ∋ ℰ ≼ ℰ′)
    → (f : rep ι → rep ι′ → rep ι″)
```

```
    → (M≼ ⦂ Γ≼ ⊢ M ≼ᴹ M′ ⨟ ⟨ E≼ ⟩ id)
      ----------------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id ∋ [ ℰ ]( f ) M ≼ [ ℰ′ ]( f ) M′

  _(_)[_] ⦂ ∀ {ι ι′ ι″} {V V′} {ℰ ℰ′}
    → ((v , v′ , _) ⦂
        Value V × Value V′ ×
        (Γ≼ ⊢ V ≼ᴹ V′ ⨟ ⟨ E≼ ⟩ id))
    → (f ⦂ rep ι → rep ι′ → rep ι″)
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id ∋ ℰ ≼ ℰ′
      ----------------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id ∋ v ⟪ f ⟫[ ℰ ] ≼ v′ ⟪ f ⟫[ ℰ′ ]

  [_]↑_ ⦂ ∀ {G ℰ ℰ′}
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id ∋ ℰ ≼ ℰ′
    → (g ⦂ Ground G)
      ------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ id
        ∋ [ ℰ ]↑ g ≼ [ ℰ′ ]↑ g

  ≼↑ ⦂ ∀ {G} {A≼G ⦂ A′ ≼ G}
         {g ⦂ Ground G} {ℰ ℰ′}
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ A≼G ∋ ℰ ≼ ℰ′
      ------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ E≼ ⟩ (A≼G ↑ g)
        ∋ ℰ ≼ [ ℰ′ ]↑ g

  cast≼ ⦂ ∀ {ℰ ℰ′} {e ⦂ E ≼ᵉ E′}
      {±p ⦂ A ⇒ B} {q ⦂ B ≼ A′} {r ⦂ A ≼ A′}
    → commute≼ ±p q r
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ r ∋ ℰ ≼ ℰ′
      ------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ q ∋ `cast ±p [ ℰ ] ≼ ℰ′

  ≼cast ⦂ ∀ {ℰ ℰ′} {e ⦂ E ≼ᵉ E′}
      {p ⦂ A ≼ A′} {±q ⦂ A′ ⇒ B′} {r ⦂ A ≼ B′}
    → ≼commute p ±q r
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ p ∋ ℰ ≼ ℰ′
      ------------------------
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ r ∋ ℰ ≼ `cast ±q [ ℰ′ ]

  castᵉ≼ ⦂ ∀ {ℰ ℰ′} {a ⦂ A ≼ A′}
      {e ⦂ E ≼ᵉ E′} {±f ⦂ E ⇒ᵉ F} {f′ ⦂ F ≼ᵉ E′}
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ a ∋ ℰ ≼ ℰ′
      -----
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ f′ ⟩ a ∋ `castᵉ ±f [ ℰ ] ≼ ℰ′

  ≼castᵉ ⦂ ∀ {ℰ ℰ′} {a ⦂ A ≼ A′}
      {e ⦂ E ≼ᵉ E′} {±f ⦂ E′ ⇒ᵉ F′} {f′ ⦂ E ≼ᵉ F′}
    → Γ≼ ⊢ P≼ ⇒ᶠ ⟨ e ⟩ a ∋ ℰ ≼ ℰ′
      -----
```

```
    → Γ≤ ⊢ P≤ ⇒ᶠ ⟨ f′ ⟩ a ∋ ℰ ≤ `castᵉ ±f [ ℰ′ ]

  ″perform_[_]_ ⦂ ∀ {e} {ℰ ℰ′}
    → ((e∈E , e∈E′) ⦂ e ∈☆ E × e ∈☆ E′)
    → Γ≤ ⊢ P≤ ⇒ᶠ ⟨ E≤ ⟩ id ∋ ℰ ≤ ℰ′
    → ∀ {A}
    → (eq ⦂ response e ≡ A)
      --------------------------------
    → Γ≤ ⊢ P≤ ⇒ᶠ ⟨ E≤ ⟩ id
        ∋ ″perform e∈E [ ℰ ] eq
        ≤ ″perform e∈E′ [ ℰ′ ] eq

  ′handle_[_] ⦂ ∀ {Q₁ Q₁′ Q₂ Q₂′}
      {Q₁≤ ⦂ Q₁ ≤ᶜ Q₁′}
      {Q₂≤ ⦂ Q₂ ≤ᶜ Q₂′}
      {H H′} {ℰ ℰ′}
    → Γ≤ ⊢ H ≤ H′ ⨟ Q₁≤ ⇒ʰ Q₂≤
    → Γ≤ ⊢ P≤ ⇒ᶠ Q₁≤ ∋ ℰ ≤ ℰ′
      ------------------------
    → Γ≤ ⊢ P≤ ⇒ᶠ Q₂≤
        ∋ ′handle H [ ℰ ]
        ≤ ′handle H′ [ ℰ′ ]
```

6.8 Precision on values

Values are a subset of terms, so we don't need to define a separate relation for them. The following lemmas state that value precision is preserved by generalization (`gvalue`).

```
gvalue≤gvalue ⦂
    {V ⦂ Γ ⊢ ⟨ E ⟩ A}
    {V′ ⦂ Γ′ ⊢ ⟨ E′ ⟩ A′}
  → (v ⦂ Value V)
  → (v′ ⦂ Value V′)
  → Γ≤ ⊢ V ≤ᴹ V′ ⨟ ⟨ E≤ ⟩ A≤
  → ∀ {F F′} {F≤ ⦂ F ≤ᵉ F′}
    ------------------------------------
  → Γ≤ ⊢ gvalue v ≤ᴹ gvalue v′ ⨟ ⟨ F≤ ⟩ A≤
```

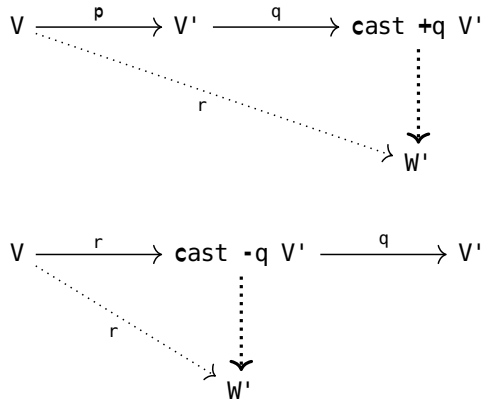
7 GRADUAL GUARANTEE

We now prove the gradual guarantee: if `M ≤ᴹ M′` and `M —→ N`, then `M′ —→ N′` and `N ≤ᴹ N′`.

We decompose the proof into several intermediate lemmas.

7.1 Right cast lemma

The right cast lemma says that when the term on the left of `≤ᴹ` is a value, reducing a cast on the right preserves precision. If `V ≤ᴹ V′`, then `cast ±q V′ —↠ W` and `V ≤ᴹ W`. The situation is represented by the following diagrams, depending on whether `±q` is an upcast `+q` or downcast `-q`.

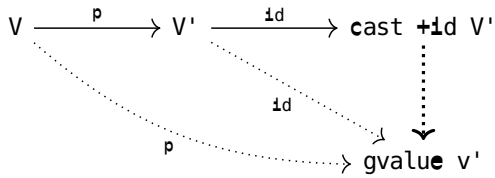



```
cast-lemma : ∀ {Γ Γ′ A B C E E′}
    {Γ≲ : Γ ≲ᴳ Γ′}
    {p : A ≲ B} {r : A ≲ C} {e : E ≲ᵉ E′}
    {V : Γ ⊢ ⟨ E ⟩ A} {V′ : Γ′ ⊢ ⟨ E′ ⟩ B}
  → Value V
  → Value V′
  → (±q : B => C)
  → ≲commute p ±q r
  → Γ≲ ⊢ V ≲ᴹ V′ ⨟ ⟨ e ⟩ p
    ------------------------
  → ∃[ W ] Value W
        × (cast ±q V′ —↠ W)
        × (Γ≲ ⊢ V ≲ᴹ W ⨟ ⟨ e ⟩ r)
```

Mirroring `progress±`, this proof proceeds by case analysis on the shape of the cast.

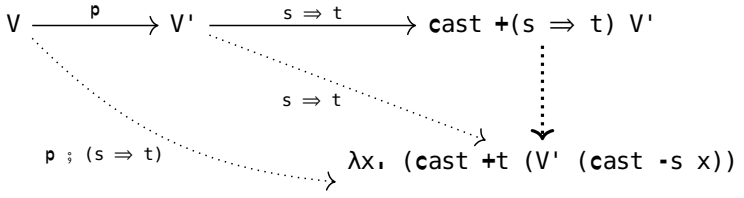
```
cast-lemma v v′ ±q e V≲V′ with split ±q in e′
```

If `±q` is the identity cast, then the cast is removed.



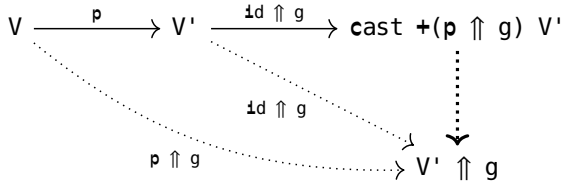

```
cast-lemma {p = _} v v′ ±q e V≲V′ | id
  rewrite ≲ident ±q e′ e
  = _ , gValue v′
      , unit (ident e′ v′)
      , ≲gvalue v v′ V≲V′
```

If ±q is a function cast, the value is wrapped.



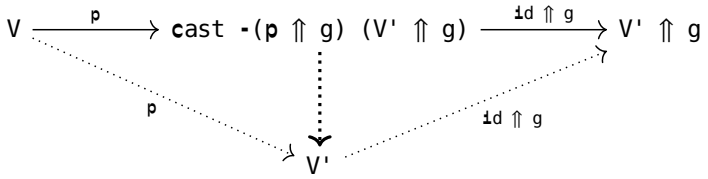

```
cast-lemma (ƛ _) (ƛ _) ±q e ƛN≤ƛN′ | ∓s ⇒⟨ ±e ⟩ ±t
  = ƛ _ , ƛ _ ,
    unit (wrap e′) ,
    ≤wrap e′ e
      (gvalue≤gvalue (ƛ _) (ƛ _) ƛN≤ƛN′)
```

If ±q is a box upcast q ⇑ g, the value is boxed, and that box is related to the left-hand side using ≤⇑.




```
cast-lemma v v′ (+ (q ⇑ g)) refl V≤V′ | other
    with cast-lemma v v′ (+ q) refl V≤V′
... | W′ , w , V′—»W′ , V≤W′
    = (W′ ⇑ g) , (w ⇑ g)
        , (unit (expand v′ g)
            ++» ξ* ([ □ ]⇑ g) V′—»W′)
        , ≤⇑ g V≤W′
```

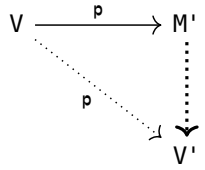
For a box downcast (- (q ⇑ g)), the cast value must be a box (v′ ⇑ g). The commutative diagram ensures that the tag g in the cast matches the tag in the box.




```
cast-lemma v (v′ ⇑ g) (- (q ⇑ ₍g)) refl (≤⇑ ₍g V≤V′) | other
    with cast-lemma v v′ (- q) refl V≤V′
... | W′ , w′ , V′—»W′ , V≤W′
    = W′ , w′ ,
      (unit (collapse v′ g) ++» V′—»W′) ,
      V≤W′
```

7.2 Catch up lemma

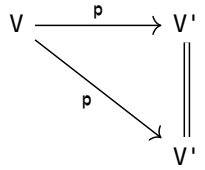
The catch up lemma says that once the left side is a value, the right side must also step to a value. If `V ≼ᴹ M′` then `M′ —↠ V′` and `V ≼ᴹ V′` for some `V′`.




```
catchup : ∀ {Γ Γ′ A A′}
    {Γ≼ : Γ ≼ᴳ Γ′}
    {V : Γ ⊢ A} {M′ : Γ′ ⊢ A′}
    {p : A ≼ᶜ A′}
  → Value V
  → Γ≼ ⊢ V ≼ᴹ M′ ⨟ p
    ------------------------------------------------
  → ∃[ V′ ](Value V′ × (M′ —↠ V′) × (Γ≼ ⊢ V ≼ᴹ V′ ⨟ p))
```

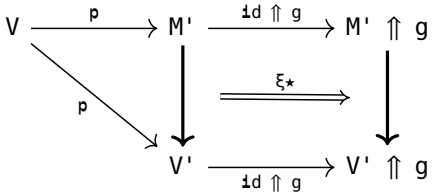
When the right side is already a value, we are done.
`ǩ-wrap` is also a value.

```
catchup (ǩ _) (wrap≼ e′ e ǩN≼ǩN′)
  = _ , ǩ _ , (_ ∎) , wrap≼ e′ e ǩN≼ǩN′
catchup (ǩ _) (≼wrap e′ e ǩN≼ǩN′)
  = _ , ǩ _ , (_ ∎) , ≼wrap e′ e ǩN≼ǩN′
```




```
catchup (ǩ _) (ǩ≼ǩ {N′ = N′} split≡ ǩN≼ǩN′)
  = ǩ N′ , ǩ N′ , (ǩ N′ ∎) , ǩ≼ǩ split≡ ǩN≼ǩN′
catchup ($ _) ($≼$ k)
  = $ k , $ k , ($ k ∎) , ($≼$ k)
```

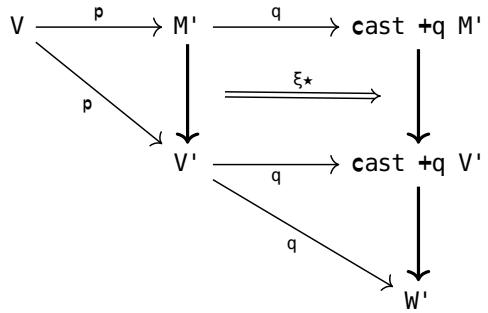
When the right side is a box (whether via `⇑≼⇑` or `≼⇑`), we reduce its contents, and a boxed value is a value.




```
catchup (v ⇑ g) (⇑≼⇑ {M = V} {M′ = M′} ,g V≼M′)
    with catchup v V≼M′
```

```
... | V′ , v′ , M′—»V′ , V≲V′
    = V′ ⇑ g
    , v′ ⇑ g
    , ξ* ([ □ ]⇑ g) M′—»V′
    , ⇑≲⇑ g V≲V′
catchup v (≲⇑ h V≲M′)
    with catchup v V≲M′
... | V′ , v′ , M′—»V′ , V≲V′
    = V′ ⇑ h
    , v′ ⇑ h
    , ξ* ([ □ ]⇑ h) M′—»V′
    , ≲⇑ h V≲V′
```

When the right side is a cast, we reduce the cast computation, and call `cast-lemma` to reduce the cast itself. In the following diagram, the applications of `catchup` and `cast-lemma` are respectively represented by the left and bottom commutative triangles.




```
catchup v (≲cast {M′ = M′} {±q = ±q} e V≲M′)
    with catchup v V≲M′
... | V′ , v′ , M′—»V′ , V≲V′
    with cast-lemma v v′ ±q e V≲V′
... | W , w , V⟨±q⟩—»W , V≲W
    = W , w
    , (ξ* (`cast ±q [ □ ]) M′—»V′ ++» V⟨±q⟩—»W)
    , V≲W

catchup v (≲castᵉ {M′ = M′} {±f = ±f} V≲M′)
    with catchup v V≲M′
... | V′ , v′ , M′—»V′ , V≲V′
    = _ , gvalue′ v′
    , (ξ* (`castᵉ ±f [ □ ]) M′—»V′ ++» unit (castᵉ-return v′))
    , ≲gvalue v v′ V≲V′
```

The following lemma formalizes the intuition that substituting for a variable which does not occur in a term yields the same term. `lift V` extends `V` with a fresh variable in its context, which thus does not occur in `V`, and the substitution operator `_[_]` substitutes for that variable.

```
lift[] : ∀ {Γ A P}
  → (V : Γ ⊢ P)
  → (W : ∀ {E} → Γ ⊢ ⟨ E ⟩ A)
```

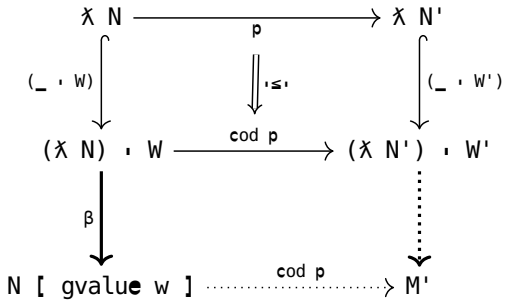
```
    --------------------
  → (lift V) [ W ] ≡ V
```

The following lemma describes β-reduction of the application of a `ƛ-wrap` value, using the lemma above to simplify the right-hand side of the reduction as required.

```
wrapβ : ∀ {Γ A B E A′ B′ E′ ∓s}
    {±t : B => B′}
    {±e : E =>ᵉ E′}
    {V : ∀ {F} → Γ ⊢ ⟨ F ⟩ (A ⇒ ⟨ E ⟩ B)}
    {W : Γ ⊢ ⟨ E′ ⟩ A′}
    {±p : A ⇒ ⟨ E ⟩ B => A′ ⇒ ⟨ E′ ⟩ B′}
  → split ±p ≡ ∓s ⇒⟨ ±e ⟩ ±t
  → (w : Value W)
    ----------------------------------
  → ƛ-wrap ∓s ±t ±e V · W
      —→ cast ±t (castᵉ ±e (V · cast ∓s (gvalue w)))
wrapβ {∓s = ∓s}{±t = ±t}{±e = ±e}{V = V}{W} e w =
  subst
    (λ X → ƛ-wrap ∓s ±t ±e V · W
      —→ cast ±t (castᵉ ±e (X · (cast ∓s (gvalue w)))))
    (lift[] V (gvalue w))
    (ξ □ (β w))
```

7.3 β-reduction is a simulation

Given a β-reduction step `(ƛ N) W ↦ N [ gvalue w ]` on the left of an inequality `(ƛ N) · W ≼ᴹ (ƛ N′) · W′`, we construct a reduction sequence on the right, `(ƛ N′) · W′ —↠ M′` such that the reducts are related `N [ gvalue w] ≼ᴹ M′`.




```
simβ : ∀ {Γ Γ′ A A′ B B′ E E′}
    {Γ≤ : Γ ≤ᴳ Γ′}
    {p : A ⇒ ⟨ E ⟩ B ≤ A′ ⇒ ⟨ E′ ⟩ B′}
    {a e b}
    {N : Γ ▷ A ⊢ ⟨ E ⟩ B}
    {N′ : Γ′ ▷ A′ ⊢ ⟨ E′ ⟩ B′}
    {W : Γ ⊢ ⟨ E ⟩ A} {W′ : Γ′ ⊢ ⟨ E′ ⟩ A′}
  → split⇒ p ≡ (a , e , b)
  → (w : Value W)
```

$$\frac{W \leq W' \quad W \leq W' \quad N \leq N' \quad \mathsf{gvalue} \leq \mathsf{gvalue}\ N \leq N' \quad \mathsf{gvalue}\ w \leq \mathsf{gvalue}\ w'}{[] \leq []\ N\ [\ \mathsf{gvalue}\ w\ ] \leq \boxtimes\ N'\ [\ \mathsf{gvalue}\ w'\ ]}$$

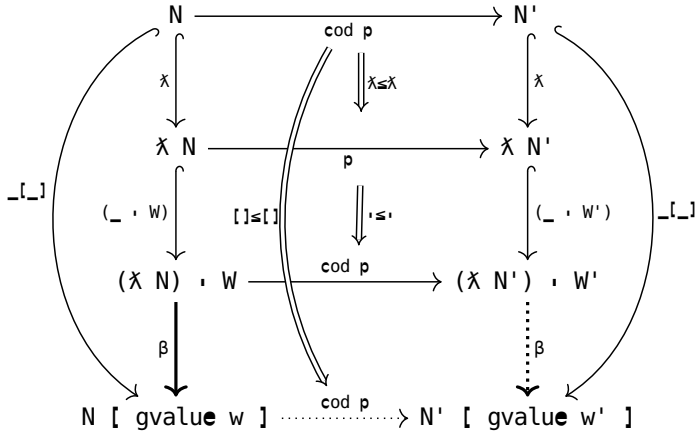


Fig. 1. Derivation and diagram for case ‘ƛ≤ƛ‘ of ‘simβ‘

```
→ (w′ ⁝ Value W′)
→ Γ≤ ⊢ ƛ N ≤ᴹ ƛ N′ ⨟ ⟨ e ⟩ p
→ Γ≤ ⊢ W ≤ᴹ W′ ⨟ ⟨ e ⟩ a
  ----------------------------------------
→ ∃[ M′ ] ((ƛ N′) · W′ —↠ M′)
        × Γ≤ ⊢ N [ gvalue w ] ≤ᴹ M′ ⨟ ⟨ e ⟩ b
```

Proceed by induction on the derivation of `ƛ N ≤ᴹ ƛ N′`. There are three rules to consider: `ƛ≤ƛ`, `wrap≤`, and `≤wrap`.

In the `ƛ≤ƛ` case, the bodies of the abstractions are related `N ≤ᴹ N′`, so we can take a β step on the right, and conclude with the derivation in Figure 1

```
simβ {W′ = W′} refl w w′ (ƛ≤ƛ {N′ = N′} refl N≤N′) W≤W′
  = _ ,
    (begin
      (ƛ N′) · W′ —→⟨ ξ □ (β w′) ⟩ N′ [ gvalue w′ ]
      ∎) ,
    []≤[] N≤N′ (gvalue≤gvalue w w′ W≤W′)
```

In the `wrap≤` case, the reduct on the left is an application interspersed with casts.

```
ƛ-wrap ∓s ±t (ƛ N) · W                —↠⟨ ξ □ (β w) ⟩
cast ±t ((ƛ N) · cast ∓s (gvalue w))
```

We take no step on the right by giving the empty reduction sequence `_ ∎`, and we construct the precision derivation in Figure 2 using the rule for applications `·≤·` and for casts on the left `cast≤`.

```
simβ {W = W}{W′} refl w w′ (wrap≤ {E = E} {N = N}{N′}{r = r} e′ e ƛN≤ƛN′) W≤W′
    rewrite lift[] {P = ⟨ E ⟩ _} (ƛ N) (gvalue w)
    = (ƛ N′) · W′ , (_ ∎) , deriv
  where
    deriv =
```

—— W≤W′ W ≤ W′ ————- gvalue≤ gvalue w ≤ W′ ———- ƛN≤ƛN′ ————————- cast≤ ƛ N ≤ ƛ N′ cast ∓s (gvalue w) ≤ W′ ———————————————— ·≤· (ƛ N) · cast ∓s (gvalue w) ≤ (ƛ N′) · W′ ————————————————- cast≤ cast ±t ((ƛ N) · cast ∓s (gvalue w)) ≤ (ƛ N′) · W′

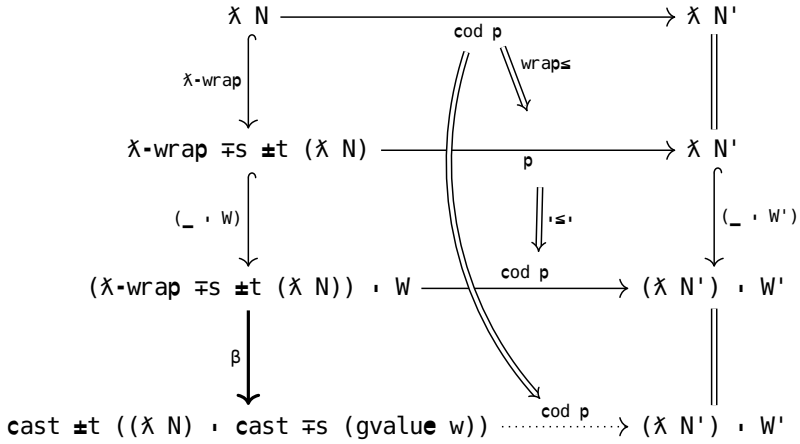


Fig. 2. Derivation and diagram for case ‘wrap≤‘ of ‘simβ‘

```
      cast≤ (cod≤ e′ e)
        (cast^e≤
          (,≤, refl ƛN≤ƛN′
               (cast≤ (dom≤ e′ e) (gvalue≤ w w′ W≤W′))))
```

In the **≤wrap** case, the reduction sequence on the right is displayed in Figure 3. We first take a β step for the application of **ƛ-wrap** using the **wrapβ** lemma. We reduce the subsequent value cast using **cast-lemma**. The last step reduces an application by the induction hypothesis **simβ**. There remains a cast that was introduced by **ƛ-wrap**, and which is covered by the **≤wrap** rule.

```
simβ {W = W}{W′} refl w w′
    (≤wrap {E′ = E′} {N = N}{N′}
           {p = p}{r = r}{∓s = ∓s}{±t = ±t}{±e = ±e}
           e′ e ƛN≤ƛN′) W≤W′
    with cast-lemma w (gValue w′) ∓s (≤dom e′ e) (≤gvalue w w′ W≤W′)
... | W″ , w″ , W′—»W″ , W≤W″
    with simβ refl w w″ ƛN≤ƛN′ W≤W″
... | M′ , [ƛN′],W″—»M′ , N[W]≤M′
    = _ ,
      (begin
        ƛ-wrap ∓s ±t ±e (ƛ N′) , W′
          —→⟨ wrapβ {V = ƛ N′} e′ w′ ⟩
        cast ±t (cast^e ±e ((ƛ N′) , cast ∓s
                                  (gvalue w′)))
          —»⟨ ξ* (`cast _ [ `cast^e _ [ (ƛ _) ,[ □ ] ] ])
                 W′—»W″ ⟩
        cast ±t (cast^e ±e ((ƛ N′) , W″))
```

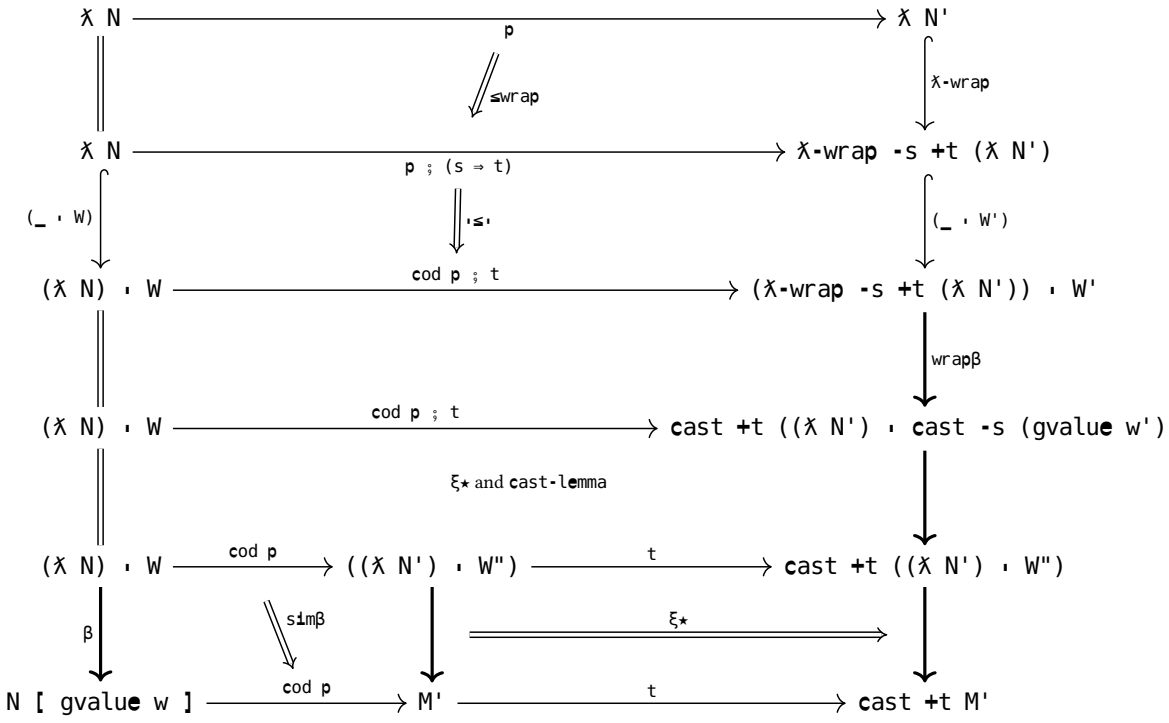


Fig. 3. Diagram for case ‘≤wrap‘ of ‘simβ‘

```
    —↠⟨ ξ* (`cast _ [ `castᵉ _ [ □ ] ])
           [ƛN′]⨾W″—↠M′ ⟩
  cast ±t (castᵉ ±e M′)
  ∎) ,
(≤cast (≤cod e′ e) (≤castᵉ N[W]≤M′))
```

7.4 Left cast lemma

The left cast lemma completes the simulation diagram for a step from a `cast` term.

```
cast≤-lemma : ∀ {Γ Γ′} {A A′ B E E′} {e : E ≤ᵉ E′}
  {Γ≤ : Γ ≤ᴳ Γ′} {±p : A => B} {b : B ≤ A′} {a : A ≤ A′} {M N M′}
  → commute≤ ±p b a
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ e ⟩ a
  → cast ±p M ↦ N
  → ∃[ N′ ] (M′ —↠ N′)
          × Γ≤ ⊢ N ≤ᴹ N′ ⨟ ⟨ e ⟩ b
```

Proof by case analysis on the derivation of `cast ±p M ↦ N`. In each of those cases except the last one, `M` is actually a value `V`. We apply `catchup` to reduce `M′` to a value `V′` such that `V ≤ V′`, which we can wrap into an inequality between the reduct of `cast ± V` and `V′`.

The application of `catchup` in this lemma is made necessary by the presence of effects in our type system. Otherwise, the `ident` rule would reduce `cast ±p V` to `V`, and we could conclude immediately with `V≲M′`.

```
cast≲-lemma comm V≲M′ (ident eq v) with catchup v V≲M′
... | V′ , v′ , M′—↠V′ , V≲V′
    rewrite ident≲ _ eq comm
    = V′ , M′—↠V′ , gvalue≲ v v′ V≲V′
```

For `wrap`, we have two subcases depending on whether the right-hand side reduces to an abstraction or to a box.

```
cast≲-lemma {b = b} comm V≲M′
    (wrap eq)
    with b | catchup (ƛ _) V≲M′
... | B≲
    | V′ , ƛ _ , M′—↠V′ , ƛN≲ƛN′
    = V′ , M′—↠V′
    , wrap≲ eq comm
        (gvalue≲gvalue (ƛ _) (ƛ _) ƛN≲ƛN′)
... | B≲ ↑ ★⇒★
    | V′ ↑ ★⇒★
      , (ƛ _) ↑ ★⇒★
      , M′—↠V′↑ , ≲↑ ★⇒★ ƛN≲ƛN′
    = V′ ↑ ★⇒★ , M′—↠V′↑ ,
      ≲↑ ★⇒★
        (wrap≲ eq (drop↑ comm)
          (gvalue≲gvalue (ƛ _) (ƛ _) ƛN≲ƛN′))

cast≲-lemma {±p = + (p ↑ ⨟g)} {b = id} refl V≲M′ (expand v g)
    with catchup v V≲M′
... | V′ ↑ ⨟g , v′ ↑ ⨟g
      , M′—↠V′↑ , ≲↑ ⨟g V≲V′
    = V′ ↑ g
    , M′—↠V′↑
    , ↑≲↑ g (cast≲ refl V≲V′)

cast≲-lemma {±p = + (p ↑ ⨟g)} {b = (q ↑ h)} refl V≲M′ (expand v g)
  = ⊥-elim (¬★≲G h q)

cast≲-lemma {±p = - (p ↑ ⨟g)} {a = id} {M = W ↑ ⨟g} refl V≲M′ (collapse w g)
  with catchup (w ↑ g) V≲M′
... | W′ ↑ ⨟g , w′ ↑ ⨟g , M′—↠W′↑ , ↑≲↑ ⨟g W≲W′
    = W′ ↑ g , M′—↠W′↑ , ≲↑ g (cast≲ refl W≲W′)

cast≲-lemma {±p = - (p ↑ ⨟g)} {a = (r ↑ h)} {M = W ↑ ⨟g} refl V≲M′ (collapse v g)
  = ⊥-elim (¬★≲G h r)
```

The two rules for raising blame are `collide` and `blame`^{e}. When the left-hand side of an inequality raises blame, there are no guarantees about the right-hand side.

```
cast≤-lemma refl V≤M′ (collide v g h G≠H)
  = _ , (_ ∎) , blame≤
```

7.5 Catchup lemma for operations

The following catchup lemma applies when the left side of an inequality is a pending operation, of the form $\mathscr{E}\ [\![\ \texttt{perform}\ \texttt{e∈E}\ \texttt{V}\ ]\!]$. In contrast the previous `catchup` lemma applies when the left side of an inequality is a value.

The conclusion of the lemma is a fairly large conjunction. We declare a data type for it, to hide existential witnesses which can be inferred from the other conjuncts.

```
data CatchupPerform {Γ Γ′} (Γ≤ ⦂ Γ ≤ᴳ Γ′)
       {P P′} (P≤ ⦂ P ≤ᶜ P′) {E} e
       (ℰ ⦂ Frame Γ (⟨ E ⟩ response e) P)
       (V ⦂ Γ ⊢ ⟨ E ⟩ request e)
       (M′ ⦂ Γ′ ⊢ P′) ⦂ Set where
  Mk ⦂ ∀ {E′} {E≤ ⦂ E ≤ᵉ E′}
       {e∈E′ ⦂ e ∈☆ E′} {V′}
       {ℰ′ ⦂ Frame Γ′ (⟨ E′ ⟩ response e) P′}
     → Value V′
     → Γ≤ ⊢ V ≤ᴹ V′ ⨟ ⟨ E≤ ⟩ id
     → Γ≤ ⊢ ⟨ E≤ ⟩ id ⇒ᶠ P≤ ∋ ℰ ≤ ℰ′
     → M′ —↠ ℰ′ [\![ perform e∈E′ V′ ]\!]
     → CatchupPerform Γ≤ P≤ e ℰ V M′
```

The term on the right side of the inequality $\mathscr{E}\ [\![\ \texttt{perform}\ \texttt{e∈E}\ \texttt{V}\ ]\!] \le \texttt{M}'$ must step to a pending operation $\mathscr{E}'\ [\![\ \texttt{perform}\ \texttt{e∈E}'\ \texttt{V}'\ ]\!]$, where each subterm is related to the corresponding one on the left. The performed operation `e` must be the same on both sides (definitionally), and it must not be handled by the frame $\mathscr{E}'$.

```
catchup-⟦perform⟧≤ ⦂ ∀ {Γ Γ′ E P P′ e}
    {e∈E ⦂ e ∈☆ E}
    {Γ≤ ⦂ Γ ≤ᴳ Γ′} {P≤ ⦂ P ≤ᶜ P′} {V M′}
    (v ⦂ Value V)
    (ℰ ⦂ Frame Γ (⟨ E ⟩ _) P)
  → Γ≤ ⊢ ℰ [\![ perform e∈E V ]\!] ≤ᴹ M′ ⨟ P≤
  → CatchupPerform Γ≤ P≤ e ℰ V M′
catchup-⟦perform⟧≤ v □ (perform≤perform V≤M′) with catchup v V≤M′
... | V′ , v′ , M′—↠V′ , V≤V′
    = Mk v′ V≤V′ □ (ξ* (′perform _ [ □ ]) M′—↠V′)
catchup-⟦perform⟧≤ v ℰ (≤↑ g M≤) with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—↠ℰV′
    = Mk v′ V≤V′ (≤↑ ℰ≤ℰ′) (ξ* ([ □ ]↑ g) M′—↠ℰV′)
catchup-⟦perform⟧≤ {e∈E = e∈E} {P≤ = P≤} v ℰ
  (≤cast {±q = ±q} comm M≤)
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk {ℰ′ = ℰ′} v′ V≤V′ ℰ≤ℰ′ M′—↠ℰV′
    = Mk v′ V≤V′ (≤cast comm ℰ≤ℰ′) (ξ* (`cast _ [ □ ]) M′—↠ℰV′)
catchup-⟦perform⟧≤ {e∈E = e∈E} {P≤ = P≤} v ℰ
  (≤castᵉ {±f = ±f} M≤)
```

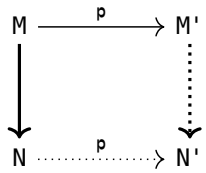
```
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk {ℰ′ = ℰ′} v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
    = Mk v′ V≤V′ (≤cast^e ℰ≤ℰ′) (ξ* (`cast^e _ [ □ ]) M′—»ℰV′)
catchup-⟦perform⟧≤ v ([ ℰ ]· M)
                     (·≤· split≡ N≤ M≤)
  with catchup-⟦perform⟧≤ v ℰ N≤
... | Mk v′ V≤V′ ℰ≤ℰ′ N′—»ℰV′
    = Mk v′ V≤V′ (by split≡ [ ℰ≤ℰ′ ]· M≤) (ξ* ([ □ ]· _) N′—»ℰV′)
catchup-⟦perform⟧≤ v (w ·[ ℰ ])
                     (·≤· split≡ N≤ M≤)
  with catchup w N≤
... | W′ , w′ , N′—»W′ , W≤
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
    = Mk v′ V≤V′
         (by split≡ , (w , w′ , W≤) ·[ ℰ≤ℰ′ ])
         (ξ* ([ □ ]· _) N′—»W′
           ++» ξ* (w′ ·[ □ ]) M′—»ℰV′)
catchup-⟦perform⟧≤ v ([ ℰ ]( f ) N)
                     (()≤() ·f M≤ N≤)
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
    = Mk v′ V≤V′ ([ ℰ≤ℰ′ ]( f ) N≤)
         (ξ* ([ □ ]( f ) _) M′—»ℰV′)
catchup-⟦perform⟧≤ v (w ( f )[ ℰ ])
                     (()≤() ·f M≤ N≤)
  with catchup w M≤
... | W′ , w′ , M′—»W′ , W≤
  with catchup-⟦perform⟧≤ v ℰ N≤
... | Mk v′ V≤V′ ℰ≤ℰ′ N′—»ℰV′
    = Mk v′ V≤V′
         ((w , w′ , W≤) ( f )[ ℰ≤ℰ′ ])
         (ξ* ([ □ ]( f ) _) M′—»W′
           ++» ξ* (w′ ( f )[ □ ]) N′—»ℰV′)
catchup-⟦perform⟧≤ v ([ ℰ ]⇑ g) (⇑≤⇑ ·g M≤)
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
    = Mk v′ V≤V′ ([ ℰ≤ℰ′ ]⇑ g)
         (ξ* ([ □ ]⇑ g) M′—»ℰV′)
catchup-⟦perform⟧≤ v (`cast ±p [ ℰ ])
                     (cast≤ comm M≤)
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
    = Mk v′ V≤V′ (cast≤ comm ℰ≤ℰ′) M′—»ℰV′
catchup-⟦perform⟧≤ v (`cast^e ±p [ ℰ ])
                     (cast^e≤ M≤)
  with catchup-⟦perform⟧≤ v ℰ M≤
... | Mk v′ V≤V′ ℰ≤ℰ′ M′—»ℰV′
```

```
    = Mk v′ V≼V′ (cast[e]≼ &≼&′) M′—»𝒱′
catchup-⟦perform⟧≼ v (″perform e∈E [ & ] eq)
                     (perform≼perform M≼)
  with catchup-⟦perform⟧≼ v & M≼
... | Mk v′ V≼V′ &≼&′ M′—»𝒱′
    = Mk v′ V≼V′ (″perform _ [ &≼&′ ] _)
         (ξ* (″perform _ [ □ ] _) M′—»𝒱′)
catchup-⟦perform⟧≼ v (′handle H [ & ])
  (handle≼handle {H′ = H′} H≼ M≼)
  with catchup-⟦perform⟧≼ v & M≼
... | Mk {&′ = &′} v′ V≼V′ &≼&′ M′—»𝒱′
    = Mk v′ V≼V′ (′handle H≼ [ &≼&′ ])
         (ξ* (′handle _ [ □ ]) M′—»𝒱′)


≼-handled : ∀ {Γ Γ′ E E′ A A′ P P′}
      {Γ≼ : Γ ≼[G] Γ′} {e : E ≼[e] E′} {a : A ≼ A′} {p : P ≼[c] P′}
      {& &′} {op : _}
    → Γ≼ ⊢ ⟨ e ⟩ a ⇒[f] p ∋ & ≼ &′
    → op ∈☆ E
    → ¬ handled op &
    → ¬ handled op &′
≼-handled □ op∈E op//& = op//&
≼-handled (by x [ &≼ ]· M≼) = ≼-handled &≼
≼-handled (by x , v ·[ &≼ ]) = ≼-handled &≼
≼-handled ([ &≼ ]( f ) M≼) = ≼-handled &≼
≼-handled (v ⟪ f )[ &≼ ]) = ≼-handled &≼
≼-handled ([ &≼ ]⇑ g) = ≼-handled &≼
≼-handled (≼⇑ &≼) = ≼-handled &≼
≼-handled (cast≼ x &≼) = ≼-handled &≼
≼-handled (≼cast x &≼) = ≼-handled &≼
≼-handled (cast[e]≼ &≼) op∈E op//& = ≼-handled &≼ op∈E (op//& ∘ inj₂)
≼-handled {& = &} (≼cast[e] {±f = ±f} {f′ = f′} &≼) op∈E op//& = [ ∈-bound[e] (∈-≼ f′ (¬h
≼-handled (″perform op′ [ &≼ ] eq) = ≼-handled &≼
≼-handled ′handle H≼ [ &≼ ] op∈E op//& = [ op//& ∘ inj₁ ∘ subst (_ ∈_) (sym (Hooks-≼
  where
    Hooks-≼ : ∀ {Γ Γ′} {Γ≼ : Γ ≼[G] Γ′}
                {P P′} {P≼ : P ≼[c] P′}
                {Q Q′} {Q≼ : Q ≼[c] Q′} {H H′}
      → Γ≼ ⊢ H ≼ H′ ⦂ P≼ ⇒[h] Q≼
      → Hooks H ≡ Hooks H′
    Hooks-≼ H≼ = All₂′-≡ (on-perform H≼)
```

7.6 Term precision is a simulation (Gradual Guarantee)

The main lemma towards proving gradual guarantee is the following simulation proof. If `M ≼ M′` and `M` takes a step `M —→ N`, then `M′` takes some sequence of steps `M′ —↠ N′` to a less precise reduct `N ≼ N′`




```
sim : ∀ {Γ Γ′ A A′ E E′ M M′ N}
    {Γ≤ : Γ ≤ᴳ Γ′}
    {p : A ≤ A′}
    {E≤ : E ≤ᵉ E′}
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ ⟨ E≤ ⟩ p
  → M —→ N
    ------------------------------
  → ∃[ N′ ] (M′ —↠ N′)
          × Γ≤ ⊢ N ≤ᴹ N′ ⨟ ⟨ E≤ ⟩ p
```

Proof by induction on the derivation of `M ≤ M′`.
Cases where the left term `M` is a value are vacuous, because values are irreducible.

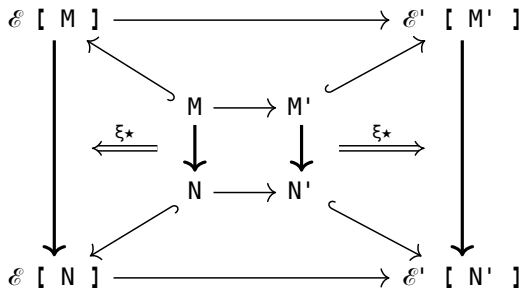
```
sim (`≤` x≤x′) M⟶N
    = ⊥-elim (variable-irreducible M⟶N)
sim ($≤$ k) M⟶N
    = ⊥-elim (value-irreducible ($ _) M⟶N)
sim (ƛ≤ƛ refl ƛN≤ƛN′) M⟶N
    = ⊥-elim (value-irreducible (ƛ _) M⟶N)
sim (wrap≤ i e V≤V′) M⟶N
    = ⊥-elim (value-irreducible (ƛ _) M⟶N)
sim (≤wrap i e V≤V′) M⟶N
    = ⊥-elim (value-irreducible (ƛ _) M⟶N)
```

`blame` is irreducible as well.

```
sim blame≤ M⟶N
    = ⊥-elim (blame-irreducible M⟶N)
```

In every other case of the derivation of `M ≤ M′`, we proceed by case analysis on the reduction step `M —→ N`.

When the evaluation context is distinct from □, the proof is relies on the induction hypothesis `sim`. The following diagram pictures this method generally: the inner square is given by the induction hypothesis, upon which we build the outer square using congruence rules. In our Agda proof, this part of the proof is spelled out for each constructor of the context $\mathscr{E}$.



For the application rule `·≤·`, this leads to three subcases.

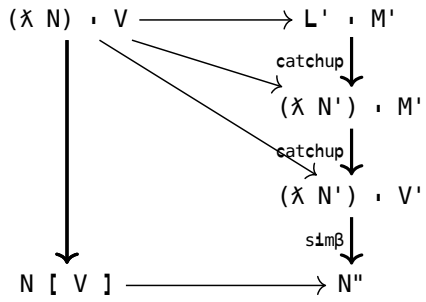
If the reduction happens under the evaluation context `[ ℰ ]· M`, we have `L ≤ L′`, `M ≤ M′` and `L —↠ N`. We apply the induction hypothesis on `L ≤ L′` to reduce `L′` to `N′`. We thus reduce `L′ · M′` to `N′ · M′` and prove `N · M′ ≤ N′ · M′` by the rule `·≤·`.

```
sim (·≤· {L′ = L′} {M′ = M′} refl L≤L′ M≤M′)
    (ξ ([ ℰ ]· M) L↦N)
    with sim L≤L′ (ξ ℰ L↦N)
... | N′ , L′—↠N′ , N≤N′
    = N′ · M′ ,
      (begin
        L′ · M′
          —↠⟨ ξ* ([ □ ]· _) L′—↠N′ ⟩
        N′ · M′
        ∎) ,
      ·≤· refl N≤N′ M≤M′
```

If the reduction happens under `V ·[ ℰ ]`, the left operand `V` must be a value. We apply the `catchup` lemma to reduce `L′ · M′` to `V′ · M′` where `V′` is a value, and the induction hypothesis `sim` to reduce `V′ · M′` to `V′ · N′`.

```
sim (·≤· {L′ = L′} {M′ = M′} refl V≤L′ M≤M′)
    (ξ (v ·[ ℰ ]) M↦N)
    with catchup v V≤L′
... | V′ , v′ , L′—↠V′ , V≤V′
    with sim M≤M′ (ξ ℰ M↦N)
... | N′ , M′—↠N′ , N≤N′
    = V′ · N′ ,
      (begin
        L′ · M′ —↠⟨ ξ* ([ □ ]· _) L′—↠V′ ⟩
        V′ · M′ —↠⟨ ξ* (v′ ·[ □ ]) M′—↠N′ ⟩
        V′ · N′
      ∎) ,
      ·≤· refl V≤V′ N≤N′
```

The last subcase for `·≤·` is β-reduction. `(ƛ N) · W ≤ L′ · M′` is inverted to `ƛ N ≤ L′` and `W ≤ M′`. By two applications of `catchup`, we reduce `L′ · M′` to `(ƛ N′) · M′` and then to `(ƛ N′) · W′`. by induction hypothesis `simβ`, we find the remaining steps to simulate the reduct `N [ gvalue W ]` on the left hand side.




```
sim (⁝≤⁝ {L′ = L′} {M′ = M′} refl ↁN≤L′ V≤M′)
    (ξ □ (β v))
    with catchup (ↁ _) ↁN≤L′
... | ↁ N′ , ↁ N′ , L′—↠ↁN′ , ↁN≤ↁN′
    with catchup v V≤M′
... | V′ , v′ , M′—↠V′ , V≤V′
    with simβ refl v v′ ↁN≤ↁN′ V≤V′
... | N″ , ↁN′⁝V′—↠N″ , N[V]≤N″
    = N″ ,
      (begin
        L′ ⁝ M′ —↠⟨ ξ* ([ □ ]⁝ _) L′—↠ↁN′ ⟩
        (ↁ N′) ⁝ M′
          —↠⟨ ξ* ((ↁ N′) ⁝[ □ ]) M′—↠V′ ⟩
        (ↁ N′) ⁝ V′ —↠⟨ ↁN′⁝V′—↠N″ ⟩
        N″
       ∎) ,
      N[V]≤N″
```

The case of ()≤() is analogous to ⁝≤⁝.

```
sim (()≤() _⊕_ L≤L′ M≤M′)
    (ξ ([ 𝓔 ]( ⁝_⊕_ ) _) L↦N)
    with sim L≤L′ (ξ 𝓔 L↦N)
... | N′ , L′—↠N′ , N≤N′
    = N′ ( _⊕_ ) _
    , ξ* ([ □ ]( _⊕_ ) _) L′—↠N′
    , ()≤() _⊕_ N≤N′ M≤M′
sim (()≤() _⊕_ V≤L′ M≤M′)
    (ξ (v ( ⁝_⊕_ )[ 𝓔 ]) M↦N)
    with catchup v V≤L′
... | V′ , v′ , L′—↠V′ , V≤V′
    with sim M≤M′ (ξ 𝓔 M↦N)
... | N′ , M′—↠N′ , N≤N′
    = V′ ( _⊕_ ) N′
    , (ξ* ([ □ ]( _⊕_ ) _) L′—↠V′
       ++↠ ξ* (v′ ( _⊕_ )[ □ ]) M′—↠N′)
    , ()≤() _⊕_ V≤V′ N≤N′
sim (()≤() _⊕_ V≤L′ W≤M′)
```

```
    (ξ □ δ)
    with catchup ($ _) V≤L′
... | $ k , $ ˽k , L′—↠V′ , ($≤$ ˽k)
    with catchup ($ _) W≤M′
... | $ k′ , $ ˽k′ , M′—↠W′ , ($≤$ ˽k′)
    = $ (k ⊕ k′)
    , (ξ* ([ □ ]( _⊕_ ) _) L′—↠V′
        ++↠ ξ* ($ k ( _⊕_ )[ □ ]) M′—↠W′ ++↠ unit δ)
    , $≤$ (k ⊕ k′)
```

For the rule ⇑≤⇑, the left-hand side is `M ⇑ g`, which is not a redex: the reduction cannot happen in the empty evaluation context □.

```
sim (⇑≤⇑ g M≤M′) (ξ □ M↦N)
    = ⊥-elim (box-irreducible g M↦N)
```

For a reduction under the context `[ ℰ ]⇑ g`, we conclude by the induction hypothesis `sim`.

```
sim (⇑≤⇑ g M≤M′) (ξ ([ ℰ ]⇑ ˽g) M↦N)
    with sim M≤M′ (ξ ℰ M↦N)
... | N′ , M′—↠N′ , N≤N′
    = N′ ⇑ g , ξ* ([ □ ]⇑ g) M′—↠N′ , ⇑≤⇑ g N≤N′
```

The two cases for ≤⇑ and ≤`cast` are also straightforward consequences of the induction hypothesis, simply wrapping right-hand side terms under `_ ⇑ g` or `cast ±q _`.

```
sim (≤⇑ g M≤M′) M—→N
    with sim M≤M′ M—→N
... | N′ , M′—↠N′ , N≤N′
    = N′ ⇑ g , ξ* ([ □ ]⇑ g) M′—↠N′ , ≤⇑ g N≤N′

sim (≤cast {±q = ±q} e M≤M′) M—→N
    with sim M≤M′ M—→N
... | N′ , M′—↠N′ , N≤N′
    = cast ±q N′
    , ξ* (`cast ±q [ □ ]) M′—↠N′
    , ≤cast e N≤N′
```

If the reduction happens under the evaluation context `‘cast ±p [ ℰ ]`, we apply the induction hypothesis.

```
sim (cast≤ e M≤M′)
    (ξ (`cast ±p [ ℰ ]) M↦N)
    with sim M≤M′ (ξ ℰ M↦N)
... | N′ , M′—↠N′ , N≤N′
    = N′ , M′—↠N′ , cast≤ e N≤N′

sim (castᵉ≤ M≤M′) (ξ (`castᵉ ±p [ ℰ ]) M↦N)
    with sim M≤M′ (ξ ℰ M↦N)
... | N′ , M′—↠N′ , N≤N′
    = N′ , M′—↠N′ , castᵉ≤ N≤N′

sim (castᵉ≤ M≤M′) (ξ □ (blameᵉ _ _ _ _))
    = _ , (_ ∎) , blame≤
```

```
sim (castᵉ≤ V≤M′) (ξ □ (castᵉ-return v))
    with catchup v V≤M′
... | V′ , v′ , M′—↠V′ , V≤V′
    = V′ , M′—↠V′ , gvalue≤ v v′ V≤V′

sim (≤castᵉ M≤M′) M↦N
    with sim M≤M′ M↦N
... | N′ , M′—↠N′ , N≤N′
    = castᵉ _ N′ , ξ* (`castᵉ _ [ □ ]) M′—↠N′ , ≤castᵉ N≤N′
```

Otherwise, if the reduction happens under the evaluation context □, the reduction rules that may apply are **ident**, wrap, **expand**, **collapse**, **collide**, or blameᵉ.

```
sim (cast≤ comm M≤M′)
    (ξ □ castM↦N)
  = cast≤-lemma comm M≤M′ castM↦N
```

Reduction under ″**perform** _ [𝒞] _.

```
sim (perform≤perform M≤M′)
    (ξ (″perform _ [ 𝒞 ] _) M↦N)
    with sim M≤M′ (ξ 𝒞 M↦N)
... | N′ , M′—↠N′ , N≤N′
    = perform- _ N′ _
    , ξ* (″perform _ [ □ ] _) M′—↠N′
    , perform≤perform N≤N′
```

perform is not a redex: reduction cannot happen under context □.

```
sim (perform≤perform M≤M′)
    (ξξ □ refl _ ())
```

Reduction under ′**handle** _ [𝒞].

```
sim (handle≤handle H≤ M≤)
    (ξ (′handle _ [ 𝒞 ]) M↦N)
    with sim M≤ (ξ 𝒞 M↦N)
... | N′ , M′—↠N′ , N≤N′
    = handle _ N′
    , ξ* (′handle _ [ □ ]) M′—↠N′
    , handle≤handle H≤ N≤N′

sim (handle≤handle H≤ V≤M′)
    (ξ □ (handle-value v))
    with catchup v V≤M′
... | V′ , v′ , M′—↠V′ , V≤V′
    = _ , (ξ* (′handle _ [ □ ]) M′—↠V′
            ++↠ unit (handle-value v′))
        , []≤[] (on-return H≤)
                (gvalue≤gvalue v v′ V≤V′)

sim (handle≤handle H≤ M≤)
    (ξ □ (handle-perform {e∈E = op∈E} {𝒞 = 𝒟} v ¬op//𝒞 eq))
    with catchup-⟦perform⟧≤ v 𝒞 M≤
```

```
      | lookup-All₂′ (on-perform H≤) eq
... | Mk v′ V≤V′ &≤ M′—↠N′
    | _ , eq′ , _ , dom≡ , cod≡ , HM′≤
    = _ , (ξ* (′handle _ [ □ ]) M′—↠N′
           ++↠ unit (handle-perform v′ (≤-handled &≤ op∈E ¬op//&) eq′))
        , []≤[]
          ([]≤[] HM′≤
            (ƛ≤ƛ refl (handle≤handle
              (lift≤ʰ (lift≤ʰ
                (subst (_ ⊢ _ ≤ _ ⨟ _ ⇒ʰ _)
                       (sym cod≡) H≤)))
              ([[]]≤[[]] (lift≤ᶠ (lift≤ᶠ &≤))
                (`≤` (subst
                       (λ A → _ ▷ A ⊢ _ ≤ˣ _ ⨟ _)
                       (sym dom≡) Z≤Z))))))
          (gvalue≤gvalue v v′ V≤V′)
```

7.7 Simulation extended to sequences

```
sim* : ∀ {Γ Γ′ P P′ M M′ N}
    {Γ≤ : Γ ≤ᴳ Γ′}
    {p : P ≤ᶜ P′}
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ p
  → M —↠ N
    ---------------------------
  → ∃[ N′ ] (M′ —↠ N′)
          × (Γ≤ ⊢ N ≤ᴹ N′ ⨟ p)
sim* M≤M′ (_ ∎)
    = _ , (_ ∎) , M≤M′
sim* L≤L′ (L —→⟨ L—→M ⟩ M—↠N)
    with sim L≤L′ L—→M
... | M′ , L′—↠M′ , M≤M′
    with sim* M≤M′ M—↠N
... | N′ , M′—↠N′ , N≤N′
    = N′ , (L′—↠M′ ++↠ M′—↠N′) , N≤N′
```

The gradual guarantee for reduction to a value.

```
gg : ∀ {Γ Γ′ P P′ M M′ V}
     {Γ≤ : Γ ≤ᴳ Γ′} {p : P ≤ᶜ P′}
  → Γ≤ ⊢ M ≤ᴹ M′ ⨟ p
  → M —↠ V
  → Value V
     ---------------------------
  → ∃[ V′ ] Value V′
          × (M′ —↠ V′)
          × Γ≤ ⊢ V ≤ᴹ V′ ⨟ p
gg M≤M′ M—↠V v
     with sim* M≤M′ M—↠V
```

```
... | N′ , M′—»N′ , V≤N′
      with catchup v V≤N′
... | V′ , v′ , N′—»V′ , V≤V′
      = V′ , v′ , (M′—»N′ ++» N′—»V′) , V≤V′
```

7.8 Example

In our running example, we showed how to increment two and its imprecise equivalent, and computed the reduction sequences for each, and also showed that the two original terms are related. Applying the gradual guarantee to the more precise reduction sequence yields the more precise reduction sequence.

Recall the example from the end of Core, where we define the following:

- `inc`, the increment function
- `inc★`, the type erasure of the increment function
- `inc★′`, the increment function upcast to type `★`
- `inc2—»3`, the reduction sequence `inc · 2 —» $ 3`
- `inc★2★—»3★`, the reduction sequence `inc★ ·★ ($★ 2) —» $★ 3`
- `inc★′2★—»3★`, the reduction sequence `inc★′ ·★ ($★ 2) —» $★ 3`

And at the example at the end of Prec we provide

- `inc2≤inc★2★`, evidence that `∅ ⊢ inc2 ≤ᴹ inc★2★ ⨟ ℕ≤★`.
- `inc2≤inc★′2★`, evidence that `∅ ⊢ inc2 ≤ᴹ inc★′2★ ⨟ ℕ≤★`.

Applying `gg` to `inc2≤inc★2★`, `inc2—»3`, and evidence that `3` yields the reduction sequence `inc★2★—»3★`, and similarly for `inc★′2★`.

```
{-
_ · gg inc2≤inc★2★ inc2—»3 ($ 3) ≡
      ($★ 3 , $ 3 ⇑ $ℕ , inc★2★—»3★ , $≤$★ 3)
_ = refl
-}
```

```
{-
_ · gg inc2≤inc★′2★ inc2—»3 ($ 3) ≡
      ($★ 3 , $ 3 ⇑ $ℕ , inc★′2★—»3★ , $≤$★ 3)
_ = refl
-}
```

REFERENCES

John Tang Boyland. 2014. The problem of structural type tests in a gradual-typed language. *Foundations of Object-Oriented Langauges* (2014).

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G Siek. 2019. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

Conor McBride. 2000. Dependently typed functional programs and their proofs. (2000).

Max S. New and Amal Ahmed. 2018. Graduality from embedding-projection pairs. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 1–30. https://doi.org/10.1145/3236768

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming.*