Abstracting gradual references

Matías Toro *, Éric Tanter *

PLEAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile

A R T I C L E   I N F O

Article history:
Received 11 June 2019
Received in revised form 20 May 2020
Accepted 22 May 2020
Available online 8 June 2020

Keywords:
Gradual typing
Mutable references
Abstract interpretation

A B S T R A C T

Gradual typing is an effective approach to integrate static and dynamic typing, which supports the smooth transition between both extremes via the imprecision of type annotations. Gradual typing has been applied in many scenarios such as objects, subtyping, effects, ownership, typestates, information-flow typing, parametric polymorphism, etc. In particular, the combination of gradual typing and mutable references has been explored by different authors, giving rise to four different semantics—invariant, guarded, monotonic and permissive references. These semantics were specially crafted to reflect different design decisions with respect to precision and efficiency tradeoffs. Since then, progress has been made in the formulation of methodologies to systematically derive gradual counterparts of statically-typed languages, but these have not been applied to study mutable references. In this article, we explore how the Abstracting Gradual Typing (AGT) methodology, which has been shown to be effective in a variety of settings, applies to mutable references. Starting from a standard statically-typed language with references, we systematically derive with AGT a novel gradual language, called $\lambda_{\text{AGT}}$. We establish the properties of $\lambda_{\text{AGT}}$; in particular, it is the first gradual language with mutable references that is proven to satisfy the gradual guarantee. We then compare $\lambda_{\text{AGT}}$ with the main four existing approaches to gradual references, and show that the application of AGT does justify one of the proposed semantics: we formally prove that the treatment of references in $\lambda_{\text{AGT}}$ corresponds to the guarded semantics, by presenting a bisimulation with the coercion semantics of Herman et al. In the process, we uncover that any direct application of AGT yields a gradual language that is not space-efficient. We consequently adjust the dynamic semantics of $\lambda_{\text{AGT}}$ to recover space efficiency. We then show how to extend $\lambda_{\text{AGT}}$ to support both monotonic and permissive references as well. Finally, we provide the first proof of the dynamic gradual guarantee for monotonic references. As a result, this paper sheds further light on the design space of gradual languages with mutable references and contributes to deepening the understanding of the AGT methodology.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

Gradual typing supports the smooth transition between static and dynamic checking based on the (programmer-controlled) precision of type annotations [35,38]. Gradual typing relates types of different precision using consistent type relations, such as type consistency (resp. consistent subtyping), the gradual counterpart of type equality (resp. subtyping). This

* This work is partially funded by FONDECYT Projects 1190058 and 3200583.

* Corresponding authors.

E-mail addresses: mtoro@dcc.uchile.cl (M. Toro), etanter@dcc.uchile.cl (E. Tanter).
approach has been applied in a number of settings, such as objects [36], subtyping [36,20], effects [4,5], ownership [34], typestates [47,21], information-flow typing [14,18,41], session types [26], refinements [30], set-theoretic types [8], Hoare logic [3], parametric polymorphism [12,28,27,48,42], and references [35,24,39].

In particular, gradual typing for mutable references has seen the elaboration of various possible semantics: invariant references [35], guarded references [24], monotonic references [39], and permissive references [39]. Invariant references are a form of references where reference types are invariant with respect to type consistency. Guarded references admit variance thanks to systematic runtime checks on reference reads and writes; the runtime type of an allocated cell never changes during execution. Guarded references have been formulated in a space-efficient coercion calculus, which ensures that gradual programs do not accumulate unbounded pending checks during execution. Monotonic references favor efficiency over flexibility by only allowing reference cells to vary towards more precise types. This allows reference operations in statically-typed regions to safely proceed without any runtime checks. Permissive references are the most flexible approach, in which reference cells can be initialized and updated to any value of any type at any time.

These four developments reflect different design decisions with respect to gradual references: is the reference type constructor variant under consistency? Can the programmer specify a precise bound on the static type of a reference, and hence on the corresponding heap cell type? Can the heap cell type evolve its precision at runtime, and if yes, how? There is obviously no absolute answer to these questions, as they reflect different tradeoffs. This work explores the semantics that results from the application of a systematic methodology to gradualize static type systems. Currently we can find in the literature two methodologies to gradualize statically-typed languages: Abstracting Gradual Typing (AGT) [20], and the Gradualizer [10]. In this work, we consider the AGT methodology as it naturally scales to auxiliary structures such as a mutable heap.

The AGT methodology helps to systematically construct gradually-typed languages by using abstract interpretation [12] at the type level. In brief, AGT interprets gradual types as an abstraction of sets of possible static types, formally captured through a Galois connection. The static semantics of a gradual language are then derived by lifting the semantics of a statically-typed language through this connection, and the dynamic semantics follow by Curry-Howard from proof normalization of the type safety argument. The AGT methodology has been shown to be effective in many contexts: records and subtyping [20], type-and-effects [4,5], refinement types [30,45], set-theoretic and union types [8,44], information-flow typing [41], and parametric polymorphism [42]. However, this methodology has never been applied to mutable references in isolation. Although Toro et al. [41] apply AGT to a language with references, they only gradualize security levels of types (e.g., Ref Int), not whole types (e.g., Ref ? is not supported). In this article we answer the following open questions: Which semantics for gradually-type references follows by systematically applying AGT? Does AGT justify one of the existing approaches, or does it suggest yet another design? Can we recover other semantics for gradual references, if yes, how?

**Contributions.** This article first reviews the different existing gradual approaches to mutable references (§2). It then presents the semantics for gradual references that is obtained by applying AGT, and how to accommodate the other semantics. More specifically, this work makes the following contributions:

- We present λ_{\text{eff}}, a gradual language with support for mutable references (§4). We derive λ_{\text{eff}} by applying the AGT methodology to a simple language with references called λ_{\text{eff}} (§3). This is the first application of AGT that focuses on gradually-typed mutable references.
- We prove that λ_{\text{eff}} satisfies the gradual guarantee of Siek et al. [38]. We also present the first formal statement and proof of the conservative extension of the dynamic semantics of the static language [38], for a gradual language derived using AGT (§4.6).
- We prove that the derived language, λ_{\text{eff}}, corresponds to the semantics of guarded references from HCC (§5). Formally, given a λ_{\text{eff}} term and its compilation to HCC^+ (an adapted version of HCC) we prove that both terms are bisimilar, and that consequently they either both terminate, both fail, or both diverge (§5).
- We observe that λ_{\text{eff}} and HCC^+ differ in the order of combination of runtime checks. As a result, HCC is space efficient whereas λ_{\text{eff}} is not: we can write programs in λ_{\text{eff}} that may accumulate an unbounded number of checks. We formalize the changes needed in the dynamic semantics of λ_{\text{eff}} to achieve space efficiency (§5.3). This technique to recover space efficiency is in fact independent from mutable references, and is therefore applicable to other gradual languages derived with AGT.
- We formally describe how to support other gradual reference semantics in λ_{\text{eff}} by presenting λ_{\text{eff}}^\text{perm}, an extension that additionally supports both permissive and monotonic references (§6). Finally, we prove for the first time that monotonic references satisfy the dynamic gradual guarantee, a non-trivial result that requires careful consideration of updates to the store.

This article is structured as follows: §2 informally introduces the different main approaches to gradual references through examples. Then, §3 presents the fully-static simple language with references called λ_{\text{eff}} from which λ_{\text{eff}} is derived. §4 presents λ_{\text{eff}}: we start by showing how to derive λ_{\text{eff}} using AGT (§4 and §4.2), then the static and dynamic semantics (§4.3, §4.4 and §4.5), its properties (§4.6), and finally we show λ_{\text{eff}} in action through examples (§4.7). λ_{\text{eff}} is formally compared with HCC in §5: we first present the static and dynamic semantics of HCC^+ (§5.1), formally relate both languages using a bisimulation (§5.2), and then present the changes needed in the dynamic semantics of λ_{\text{eff}} to achieve
space efficiency (§5.3). §6 describes an extension to $\lambda_{\text{gif}}$ to encode both permissive and monotonic references, and explains how we prove the dynamic gradual guarantee for monotonic references. Finally, §7 discusses related work, and §8 concludes. Complete definitions and proofs of the main results can be found in the Appendix. Additionally, we implemented $\lambda_{\text{gif}}$ as an interactive prototype that displays both typing derivations and reduction traces. All the examples mentioned in this paper are readily available in the online prototype available at https://pleiad.cl/grefs.

2. Gradual typing with references

Introducing gradual typing in a language with mutable references raises a number of design and implementation challenges, which have led to different semantics. After a brief review of what mutable references are, we informally present the four approaches to gradual references from the literature. We then formulate the research question that this paper addresses.

2.1. Background: mutable references

Mutable references are memory cells whose content can vary during execution. A language with mutable references typically introduces three basic operations: allocation ref $t$ creates a new reference (i.e., heap location) initialized with the value of $t$, dereferencing $!t$ returns the value stored in the reference $t$, and assignment $t_1 := t_2$ destructively updates the reference $t_1$ with the value of $t_2$. For instance:

```
let x = ref 4
lx
x := 10
lx
```

Line 1 creates a new reference and returns a new location $o$ pointing to a mutable cell in the store whose content is 4. Line 2 reads 4 from the current stored value of $o$. Line 3 updates the stored value of $o$ to 10. And finally, line 4 reads again the current stored value of $o$, which is now 10.

An allocation term ref $t$ has type Ref $T$ where $T$ is the type of the subterm $t$. Locations $o$ are not part of the source language; they are introduced during reduction. To type locations we use a store typing $\Sigma$, a finite map from locations to types, such that $o$ has type $T$ if $\Sigma(o) = T$. One interesting particularity of reference types is that they are invariant with respect to subtyping, i.e., Ref $T_1 <:\text{Ref} T_2$ if and only if $T_1 \subseteq T_2$ [33]. This observation is key in a gradual language when considering the type consistency relation, as illustrated below.

2.2. Background: gradual typing

The most important component of a gradual language is the type (im)precision relation. Type precision $\sqsubseteq$ is a partial order between gradual types, where we say that the gradual type $G_1$ is less imprecise (or, more precise) than $G_2$, notation $G_1 \sqsubseteq G_2$, if $G_1$ represents less static types than $G_2$. In most gradual languages, imprecision is introduced by adding the notion of an unknown type $\alpha$, which represents any static type whatsoever, i.e. $G \sqsubseteq \alpha$ for any $G$. Type precision helps us to define the type consistency relation $\sim$, the gradual counterpart of the equality relation between static types. For instance, any type is consistent with itself, and the $\alpha$ type is consistent with any gradual type and, vice versa, any gradual type is consistent with the $\alpha$ type. The static flexibility given by the type consistency relation is backed up dynamically by inserting casts or coercions at the boundaries of types of different precision, ensuring at runtime that no static assumptions are violated. If a static assumption is violated, then a runtime error is raised. Inserting casts may be involved in a setting with higher-order functions. It is in general impossible to immediately check if a function satisfies a particular type. Therefore, in general functions are wrapped in proxies that defer the necessary runtime checks on arguments and return values.

2.3. Existing approaches

We now briefly review the four major elaborations of gradual typing with references that have been proposed in the literature.

Invariant references. Siek and Taha [35] include a treatment of references in their original gradual typing work. However, based on the observation that “allowing variance under reference types compromises type safety”, they impose reference types to be invariant with respect to consistency. In other words, $T_1 \sim T_2$ does not imply that Ref $T_1 \sim$ Ref $T_2$. Consider Example 1 below, where :: is a type ascription operator:

```
let x = ref (4 :: ?)
let y: Ref Int = x ← type error
y := 10
```

Example 1

In this example, 4 :: ? represents an assertion that 4 has type ? . This is accepted by the type system because Int (the type of 4) is consistent with the ascribed type ?. The type of the new reference created at line 1 is inferred from the subterm,
so \( x \) has type \( \text{Ref} \, ? \). The program is rejected at line 2 because \( \text{Ref} \, ? \) (the type of \( x \)) is not consistent with \( \text{Ref} \, \text{Int} \) (the type of \( y \)) under the invariant semantics.

**Guaranteed references.** Herman et al. [24] develop a space-efficient approach to gradual typing based on coercions [23]. Hereafter, we refer to this language as HCC. HCC includes references, albeit with a different semantics from the one proposed by Siek and Taha [35]. In particular, the type system allows consistency variance for reference types, i.e. \( T_1 \sim T_2 \) implies \( \text{Ref} \, T_1 \sim \text{Ref} \, T_2 \). The dynamic semantics of the language is given by translation to a language with coercions. Coercions can be normalized in order to avoid accumulation of wrappers that compromise space efficiency. This normalization **eagerly** combine coercions, detecting some errors immediately—e.g. during the reduction of a function—in contrast to lazier approaches that accumulate wrappers and errors are not detected until the casted function is applied. The resulting semantics is called “guarded” because each reference cell assignment (resp. dereference) is guarded with a coercion from the type of the assigned values (resp. expected type of the read value) to/from the actual type of the reference cell. In other words the runtime type of an allocated cell never changes during execution. The approach is intuitively justified by analogy with how first-class languages can be used at different (consistent) types, provided that the appropriate guards check arguments and return values.

Examples 2, 3 and 4 illustrate the use of guarded references:

```
1  let x = ref (4 :: ?)
2  let y: Ref Bool = x
3  ty ← runtime error
```

**Example 2**

Example 2 raises a runtime error at line 3 because it is trying to read a \( \text{Bool} \) where an \( \text{Int} \) is stored. Example 3 fixes example 2 by updating the location with an actual boolean before the dereference operation. This is possible because the location is created at type \( \text{Ref} \, ? \), meaning that it can store any value of any type (any type is consistent with \( ? \)). In Example 4, because the content of the reference is the (unascribed) number 4, the created reference has type \( \text{Ref} \, \text{Int} \). A runtime error is raised at line 3 because the coercion injecting \( \text{true} \) from \( \text{Bool} \) to \( ? \) cannot be combined with the coercion projecting \( y \) from \( ? \) to \( \text{Int} \).

**Monotonic references.** Siek et al. [39] propose a design for gradually-typed references called **monotonic references**. The design is driven by efficiency considerations, namely allowing statically-typed code to be compiled with direct memory access instructions—without coercions or wrappers. Like guarded references, monotonic references are variant under consistency. In order to avoid using reference wrappers in statically-typed code, the runtime type of reference cells is allowed to vary but only towards more **precise** types. This monotonicity restriction ensures that direct reference accesses from statically-typed code are safe. Importantly, casts on references are performed directly on the heap.

Examples 3 and 5 illustrate the use of monotonic references:

```
1  let x = ref (4 :: ?)
2  let y: Ref Bool = x
3  y := true
4  ty
```

**Example 3**

In example 3, when variable \( x \) is cast to \( \text{Ref} \, \text{Bool} \) at line 2, the cast is performed directly on the heap: the cast fails as the stored value has type \( \text{Int} \) instead of \( \text{Bool} \). In example 5, when variable \( x \) is cast to \( \text{Ref} \, \text{Int} \), the runtime type of the heap cell is updated to the more precise type \( \text{Int} \). Therefore, the subsequent assignment of \( \text{true} \) to \( x \) triggers a runtime error at line 3 because \( \text{Bool} \) is not consistent with \( \text{Int} \). Note that under the guarded semantics this program runs without errors. The difference is that accesses to \( y \) in the monotonic semantics ensures that the value on the heap is of type \( \text{Int} \), while under the guarded semantics a coercion is necessary. For instance, consider changing example 5 at line 3, with a dereference \( !y \). While both semantics yield the same result, operationally there is an important difference: in the guarded semantics, because the content type of the reference is \( ? \) but \( y \) has type \( \text{Ref} \, \text{Int} \), an additional coercion of the underlying value to \( \text{Int} \) occurs. With monotonic references this coercion is not needed, because the semantics enforce that the stored value is of type \( \text{Int} \) as soon as the alias \( y \) is created.

**Permissive references.** The monotonic discipline favors efficiency over flexibility. Siek et al. [39] also develop a flexible notion of **permissive references** on top of the language with monotonic references. In essence, permissive references consist in treating the type of all heap cells as \( ? \). A source-level translation then adds the necessary ascriptions on dereferences and assignments. Note that the transformation would work equivalently using the guarded semantics as target (but not with the invariant semantics).

The following examples present a variation of example 3 using permissive references (left),

\[^1\] As in [39], we use \( \text{ref}^* \) \( t \) to denote the permissive reference constructor, and \( \text{Ref}^* \) \( T \) to denote a permissive reference type.
With the permissive semantics, the program does not produce any runtime error. The first line of the transformed program creates a new reference of type Ref ?. The third line shows that every assigned value is first ascribed to the ? type. Therefore the runtime type of the heap cell does not change: it stays ?. Finally, in the last line, since the variable y originally had type Ref* Bool, the dereference value is ascribed to Bool.

Note that the permissive semantics is even more flexible than the guarded semantics: guarded semantics allows programmers to fix the type of heap cells at more precise types than ?.

2.4. Gradual references, systematically

These four developments reflect different design decisions with respect to gradual references: is the reference type constructor variant under consistency? Can the programmer specify a precise bound on the static type of a reference, and hence on the corresponding heap cell type? Can the heap cell type evolve its precision at runtime, and if yes, how? Although there is no correct answer to any of these questions, in this article we try to answer them for any gradual language derived systematically with AGT. In particular, we answer the following questions: Which semantics for gradually-type references follows by systematically applying AGT? Does AGT justify one of the existing approaches, or does it suggest yet another design? Can we recover other semantics for gradual references, if yes, how?

In the next sections we proceed as follows. First we present \( \lambda_{\text{REF}} \), a standard statically-typed language with references (§3). Second, we systematically apply AGT to \( \lambda_{\text{REF}} \) (§4) and observe the resulting semantics, which we called \( \lambda_{\text{AGS}} \) (§2). We observe that \( \lambda_{\text{AGS}} \) manifests the guarded references semantics of HCC. Third, we formalize this observation by relating \( \lambda_{\text{AGS}} \) with HCC (§5). We present an extension of HCC, called HCC\(^+\), and a type-driven translation from \( \lambda_{\text{AGS}} \) to HCC\(^+\). We prove that a \( \lambda_{\text{AGS}} \) term and its translation to HCC\(^+\) are bisimilar. Fourth, we show that, contrary to HCC\(^+\), \( \lambda_{\text{AGS}} \) is not space-efficient. We then present the changes needed in the dynamic semantics to recover space efficiency (§5). Finally, we present \( \lambda_{\text{pm}} \) an extension of \( \lambda_{\text{AGS}} \) to support other semantics both permissive and monotonic references (§6).

3. Preliminary: the static language \( \lambda_{\text{REF}} \)

We now apply AGT to a simple language with references, called \( \lambda_{\text{REF}} \), whose static and dynamic semantics are defined in Figs. 1 and 2, respectively.

Static semantics. The definition of \( \lambda_{\text{REF}} \) is standard. We use the metavariable \( l \) to range over a countably infinite set Loc of locations. A store typing \( \Sigma \) is a partial function from locations to types. A term \( t \) can be a lambda abstraction, a constant \( b \), a variable, an application, a binary operation on constants \( \odot \), a conditional expression, a type ascription, a reference, a dereference, an assignment, or a location. Types may be base types (we use \( B \) to abstract over all base types), functions, and references. Ref \( T \) represents a reference to a value of type \( T \).

To prepare for the application of AGT, the presentation of the type system follows the convention of Garcia et al. [20], in which the type of each sub-expression is kept opaque, the type relations are made explicit as side conditions, and (partial) type functions are used explicitly instead of relying on matching metavariables. In particular, the dom (resp. cod) partial function is used to obtain the domain (resp. codomain) of a function type; it is undefined otherwise. We similarly introduce the ref partial function to extract the underlying type of a reference type. Save for the use of ref, rules (Tref), (Tderef), (Tasgn) and (TI) are all standard [33]. Type ascriptions are also standard [33], though one could argue they are not essential to a static type system; their essential role will become clearer when turning to the gradual language, as ascriptions allow programmers to control (im)precision and play a key role in the dynamic semantics [20]. We use the \( \theta \) metafunction to determine the type of constants (e.g. \( \theta(\text{true}) = \text{Bool}, \theta(1) = \text{Int} \)).

Dynamic semantics. The dynamic semantics of \( \lambda_{\text{REF}} \) is presented in Fig. 2, and is standard as well. The semantics is straightforward using evaluation contexts to reduce terms. A store \( \mu \) maps locations \( o \) to values \( v \). Here \( \mu[o \mapsto v] \) stands for a new store in which the location \( o \) is mapped to the value \( v \). The domain of a store \( \mu \), written \( \text{dom}(\mu) \), is the set of locations for which the finite map is defined. The expression \( \text{ref } t \) evaluates by reducing the term \( t \) to a value \( v \), obtaining a fresh location in memory and storing the value at that location. The result of \( \text{ref } v \) is the newly created location. A dereference expression \( !t \) first evaluates the term \( t \) to a location \( o \), then returns the value stored in memory at location \( o \). An assignment \( t_1 := t_2 \) evaluates term \( t_1 \) to a location \( o \) and evaluates term \( t_2 \) to a value \( v \). The expression \( o := v \) updates the store at location \( o \) with the new value \( v \), and returns unit.

Properties. Type safety of \( \lambda_{\text{REF}} \) is established as usual: a well-typed closed term is either a value or it can take a step (along a well typed store) to a term of the same type (and a well-typed store that extends the original one).
Proposition 1 (Type safety). Let \( \sigma; \Sigma \vdash t : T \). Then one of the following is true:

1. \( t \) is a value \( \nu \);
2. if \( \Sigma \vdash \mu \) then \( t | \mu \la t' | \mu' \), where \( \sigma; \Sigma' \vdash t' : T \) and \( \Sigma' \vdash \mu' \) some \( \Sigma' \supseteq \Sigma \).

Proof. The proof is standard and follows from progress and preservation [33].  

4. Gradualizing \( \lambda_{\text{ref}} \)

Once we have defined the static language \( \lambda_{\text{ref}} \), the AGT methodology drives the derivation of its gradual counterpart, \( \lambda_{\text{ref}} \), following three steps:

1. Define the syntax of gradual types and give them meaning by concretization to sets of static types; consequently obtain the most precise abstraction, establishing a Galois connection.
2. Derive the gradual type system by using lifted type predicates and type functions in the typing rules.
3. Derive the runtime semantics of the gradual language by proof normalization of gradual typing derivations.

4.1. Syntax and meaning of gradual types

We start by defining the syntax of gradual types. We decide to allow references to gradual types:

\[
G \in \text{GTerm} \\
G ::= B \mid G \rightarrow G \mid \text{Ref}
\]

Terms \( t \) are lifted to gradual terms \( \Gamma \rightarrow \text{GTerm} \), i.e. terms with gradual type annotations.

We then give meaning to gradual types via a concretization function \( \gamma \) from gradual types to non-empty sets of static types. We write \( \gamma^0(\text{Type}) \) to denote the non-empty power set of types. We start from the concretization function for GTFL
\[ v ::= c \mid \lambda x.t \mid o \] (values)
\[ E ::= \square \mid E \oplus t \mid v \oplus E \mid E.t \mid v.E \mid E E \mid \text{if } E \text{ then } t \text{ else } t \mid \text{ref } E \mid !E \mid E := t \mid v := E \mid E :: T \] (Contexts)
\[ \mu ::= \mu, o \mapsto v \] (store)

**Notions of Reduction**

\[ b_1 \oplus b_2 \mid \mu \longrightarrow_s b_2 \mid \mu \text{ where } c_3 = b_1 \oplus b_2 \]
\[ (\lambda x.t) v \mid \mu \longrightarrow_s (v/x)t \mid \mu \]
\[ \text{if } b \text{ then } t_1 \text{ else } t_2 \mid \mu \longrightarrow_s t_1 \mid \mu \text{ if } b = \text{true} \]
\[ t_2 \mid \mu \text{ if } b = \text{false} \]
\[ !o \mid \mu \longrightarrow_s v \mid \mu \text{ where } v = \mu(o) \]
\[ o := v \mid \mu \longrightarrow_s \text{unit } | \mu[o \mapsto v] \quad v : T \mid \mu \longrightarrow_s v \mid \mu \]

**Fig. 2.** \( \lambda_{\text{acc}} \): Dynamic semantics.

given by Garcia et al. [20], adding an extra case to deal with reference types. This is the natural lifting of concretization to the reference type constructor: \( \text{Ref } G \) denotes the set of reference types \( \text{Ref } T \) for each \( T \) in the concretization of \( G \):

**Definition 1 (Concretization).** Let \( \gamma : \text{GType} \rightarrow \mathcal{P}^*(\text{TYPE}) \) be defined as follows:

\[ \gamma(B) = \{ B \} \]
\[ \gamma(G_1 \rightarrow G_2) = \{ T_1 \rightarrow T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2) \} \]
\[ \gamma(\text{Ref } G) = \{ \text{Ref } T \mid T \in \gamma(G) \} \]
\[ \gamma(?) = \text{Type} \]

The notion of type precision between gradual types coincides with set inclusion of their concretizations:

**Definition 2 (Type Precision).** \( G_1 \sqsubseteq G_2 \) if and only if \( \gamma(G_1) \subseteq \gamma(G_2) \).

**Proposition 2 (Precision, inductively).** The following inductive definition of type precision is equivalent to Definition 2.

\[
\begin{align*}
B \sqsubseteq B & \quad \frac{G_1 \sqsubseteq G_2 \quad G_1 \rightarrow G_2 \sqsubseteq G_2'}{G_1 \rightarrow G_2' \sqsubseteq G_2'} \\
& \quad \frac{G_1 \sqsubseteq G_2}{\text{Ref } G_1 \sqsubseteq \text{Ref } G_2} \\
& \quad \frac{G_1 \sqsubseteq G_2}{G \sqsubseteq ?}
\end{align*}
\]

Once \( \gamma \) is defined, we proceed to define its corresponding abstraction function:

**Definition 3 (Abstraction).** Let the abstraction function \( \alpha : \mathcal{P}^*(\text{TYPE}) \rightarrow \text{GType} \) be defined as follows:

\[ \alpha([B]) = B \]
\[ \alpha([T_1 \rightarrow T_2]) = \alpha([T_1]) \rightarrow \alpha([T_2]) \]
\[ \alpha([\text{Ref } T]) = \text{Ref } \alpha([T]) \]
\[ \alpha(T) = ? \text{ otherwise} \]

The abstraction function preserves type constructors and falls back on the unknown type whenever a heterogeneous set is abstracted. As expected, abstraction preserves the \( \text{Ref } \) type constructor when all static types in the set are reference types. This abstraction function is both sound and optimal: it produces the most precise gradual type that over-approximates a given set of static types.

**Proposition 3 (Galois connection).** \( (\gamma, \alpha) \) is a Galois connection, i.e.:

\( a \) (Soundness) for any non-empty set of static types \( S = \{ T \} \), we have \( S \sqsubseteq \gamma(\alpha(S)) \)
\( b \) (Optimality) for any gradual type \( G \), we have \( \alpha(\gamma(G)) \sqsubseteq G \).
Soundness (a) means that $\alpha$ always produces a gradual type whose concretization over-approximates the original set. Optimality (b) means that $\alpha$ always yields the best (i.e. least) sound approximation that gradual types can represent.

4.2. Lifting the type system

In order to obtain the static semantics of $\lambda_{\text{REF}}$, we lift type relations (here, equality) and type functions (dom, cod, tref, equate). Following AGT [20], this lifting is obtained by exploiting the Galois connection we have just established through existential lifting.

**Definition 4** (Consistency). $G_1 \sim G_2$ if and only if $T_1 = T_2$ for some $(T_1, T_2) \in \gamma(G_1) \times \gamma(G_2)$. Inductively:

\[
\begin{align*}
G \sim ? & \quad \\quad \quad ? \sim G & \quad \quad G \sim G \\
G_{21} \sim G_{11} & \quad \quad G_{12} \sim G_{22} & \quad \quad G_1 \sim G_2
\end{align*}
\]

As a first result, the concretization function justifies consistency variance for reference types—as adopted by all gradual reference systems, except the invariant semantics of Siek and Taha [35].

Lifting type functions follows abstract interpretation as well. For example, consider a partial function $F : \text{TYPE} \times \text{TYPE} \rightarrow \text{TYPE}$. The lifting of $F$, called $\tilde{F}$, is defined as $\tilde{F}(G_1, G_2) = \alpha(\tilde{F}(\gamma(G_1), \gamma(G_2)))$. Note that as $F$ is partial, the collecting application of $F$ may be the empty set, which is not part of the domain of $\alpha$; this situation captures the notion of type errors [20].

For instance, the lifting of the equate operator presented in Fig. 1 is defined as follows:

**Definition 5.** $\text{equate}(G_1, G_2) = \alpha(\{\text{equate}(T_1, T_2) \mid (T_1, T_2) \in \gamma(G_1) \times \gamma(G_2)\})$ and it comes as no surprise that this definition coincides with the meet operator in the precision order [20]:

**Proposition 4.** $\text{equate}(G_1, G_2) = G_1 \cap G_2$.

The meet operator is defined as $G_1 \cap G_2 = \alpha(\gamma(G_1) \cap \gamma(G_2))$, and inductively as:

\[
\begin{align*}
B \cap B & = B \\
G_1 \cap G_2 & = G_1 \cap G_2 \\
G \cap ? & = ? \cap G = G \\
(G_{11} \rightarrow G_{12}) \cap (G_{21} \rightarrow G_{22}) & = (G_{11} \cap G_{21}) \rightarrow (G_{12} \cap G_{22})
\end{align*}
\]

$\text{Ref } G_1 \cap \text{Ref } G_2 = \text{Ref } G_1 \cap \text{Ref } G_2$

$G_1 \cap G_2$ is undefined otherwise

Compositional lifting. As previously noted by Garcia et al. [20] we cannot always apply compositionally lifting to predicates that use both type relations and type functions. However, we justify that we can do it for application and assignment rules.

**Proposition 5.** Let $P_1(T_1, T_2) \triangleq T_1 = \text{dom}(T_2)$. Then $\tilde{P}_1(G_1, G_2) \iff G_1 \sim \text{dom}(G_2)$.

**Proposition 6.** Let $P_2(T_1, T_2) \triangleq T_1 = \text{tref}(T_2)$. Then $\tilde{P}_2(G_1, G_2) \iff G_1 \sim \text{tref}(G_2)$.

Consistent reference type function. The algorithmic consistent lifting of the tref type function, $\text{tref}$, is provided in Fig. 3. As expected, it justifies the fact that a term of the unknown type $? \quad ?$ can be dealt with as if it were a reference type Ref $\quad ?$, since $\text{tref}(?) = ?$.

4.3. Static semantics

The type system of $\lambda_{\text{REF}}$ is presented in Fig. 3, along algorithmic definitions of consistent functions; the type rules are obtained by replacing type predicates and functions with their corresponding liftings. For simplicity, we use notation $t : G$ if $\vdash t : G$. Assertions play an important role in the gradual language [20]; they conveniently allow programmers to introduce (im)precision as desired. For instance, the following program typechecks due to the convenient use of assertions: $((\lambda b : \text{bool}. \text{if } b \text{ then } ? \quad ? \text{ else } 1 :: ? \quad ?) \quad \text{false}) + 2$. Note that the type equality premise of ascription in $\lambda_{\text{REF}}$ is lifted to a type consistency premise in $\lambda_{\text{REF}}$. As we will see in the next section, ascriptions are also helpful in the dynamic semantics to ensure that type precision is stable under substitution, hence ensuring typing preservation.

4.4. Dynamic semantics

Traditionally, the dynamic semantics of a gradual language is given by translation to an intermediate cast calculus [35]. One of the salient features of the AGT methodology is that it provides a direct dynamic semantics of gradual programs, defined over gradual typing derivations [20]. The key idea is to apply proof reduction on gradual typing derivations [25].

---

2 $\tilde{F}$ is notation for the collecting function of $F$. 
augmented with evidence for consistent judgments. By the Curry-Howard correspondence, this induces a notion of reduction for gradual terms.

More specifically, the reduction of gradual typing derivations mirrors reasoning steps used in the type safety proof of the static language. The static type safety proof relies on transitivity of type relations, but in a gradual setting, transitivity does not always hold. For instance, equality is a transitive type relation, but type consistency—which only captures plausibility—is not transitive in general: \text{Int} \sim \text{?} and \text{?} \sim \text{Bool}, but \text{Int} \not\sim \text{Bool}. In AGT, gradual typing derivations are augmented with type-based justifications of why a consistent judgment holds, called evidence. Evidence can generally be represented by a pair of gradual types, \( e = (G_1, G_2) \), which capture the implied information about types related by a consistent judgment; these types are at least as precise as the types involved in the judgment [20]. We use notation \( e \vdash G_1 \sim G_2 \) to denote that evidence \( e \) justifies the consistency judgment \( G_1 \sim G_2 \). During proof reduction (which corresponds to a reduction step), when a transitivity claim between two consistent judgments needs to be justified, the corresponding evidences of these judgments are combined via consistent transitivity. If the combination is defined, then the resulting evidence justifies the new consistent judgment and the reduction step can be taken, otherwise the program halts with an error.

Evidence is initially computed by a partial function called an initial evidence operator \( \gamma_{\text{in}} \) [20]. An initial evidence operator computes the most precise evidence that can be deduced from a given judgment. For instance the initial evidence of consistent judgment \( G_1 \sim G_2 \) is \( e = \gamma_{\text{in}}(G_1, G_2) \), i.e. \( \gamma_{\text{in}}(G_1, G_2) \vdash G_1 \sim G_2 \). Formally the initial evidence operator is defined as:

Definition 6 (Initial evidence operator).

\[ \gamma_{\text{in}}(G_1, G_2) = \alpha^2(\{(T_1, T_2) \mid T_1 \in \gamma(G_1), T_2 \in \gamma(G_2), T_1 = T_2\}) \]

Given two sets of static types that belong to the concretization of both gradual types, this function abstracts the sets of pairs of static types such that both types are equal.\(^3\) In this setting with only consistency (and not consistent subtyping), the initial evidence operator coincides with the pair of meets between the two types.

Proposition 7. If \( G_1 \sim G_2 \), then \( \gamma_{\text{in}}(G_1, G_2) = \{G_1 \cap G_2, G_1 \cap G_2\} \).

\(^3\) \( \alpha^2(\{(T_1, T_2)\}) = \{(\alpha(T_1)), \alpha(T_2)\} \).
Illustration. Consider the gradual typing derivation of \((\lambda x : \mathbb{R} . x + 1) \mathbb{R} \) false, called \(\mathcal{D}\) (for simplicity we omit the store typing):

\[
\begin{align*}
\mathcal{D} = & \quad \text{\(\theta((\lambda x : \mathbb{R} . x + 1) \mathbb{R} \) false) = Bool\)} \\
\text{\(\varnothing \vdash \text{false} : \text{Bool}\)} & \quad (\text{Bool} , \text{Bool}) \vdash \text{\(\sim\)} ? \quad \text{\(\theta(1) = \text{Int}\)} \\
\text{\(x : \mathbb{R} \vdash x : \mathbb{R}\)} & \quad \text{\(\varnothing \vdash (\lambda x : \mathbb{R} . x + 1) \mathbb{R} \) false : Int}\end{align*}
\]

where \(\mathcal{D}' = \text{\(\varnothing \vdash \text{false} : \text{Bool}\)} \quad (\text{Bool} , \text{Bool}) \vdash \text{\(\sim\)} ? \quad \text{\(x : \mathbb{R} \vdash x + 1\)} \quad \text{\(\varnothing \vdash (\lambda x : \mathbb{R} . x + 1) \mathbb{R} \) false : Int}\)

In the typing derivation of the function \(\mathcal{D}'\), the consistent judgments \(\sim \text{Int}\) and \(\text{Int} \sim \text{Int}\) support the addition expression, and at the top-level, the judgment \(\text{Bool} \sim ?\) supports the application of the function to false. By knowing that \(\sim \text{Int}\) holds, we learn that the first type can only possibly be \(\text{Int}\), while we do not learn anything new about the right-hand side, which is already fully static. Therefore the evidence of that judgment is \(\varepsilon_1 = \mathbb{G}_G(? , \text{Int}) = (\text{Int} , \text{Int})\), i.e. \((\text{Int} , \text{Int}) \vdash ? \sim \text{Int}\). For the \(\text{Int} \sim \text{Int}\) consistent judgment we cannot learn anything new, therefore its evidence is \(\varepsilon_2 = \mathbb{G}_G(\text{Int}, \text{Int}) = (\text{Int}, \text{Int})\). Similarly, the evidence for the third judgment is \(\varepsilon_3 = \mathbb{G}_G(\text{Bool}, ?) = (\text{Bool}, \text{Bool})\).

At runtime, reduction rules need to combine evidence in order to either justify or refute a use of transitivity in the type preservation argument. The combination operation, called consistent transitivity \(\circ^p\), determines whether two evidences support the transitivity of their corresponding judgments. The definition of consistent transitivity for a type predicate \(P\), \(\circ^p\), is given by the abstract interpretation framework [20]; in particular, for type equality it is defined as follows:

**Definition 7 (Consistent transitivity).** Suppose \(\varepsilon_{ab} \vdash G_a \sim G_b\) and \(\varepsilon_{bc} \vdash G_b \sim G_c\). Evidence for consistent transitivity is deduced as \(\varepsilon_{ab} \circ^p \varepsilon_{bc} \vdash G_a \sim G_c\), where:

\[
(G_1, G_{21}) \circ^p (G_{22}, G_3) = \alpha^2(\{(T_1, T_3) \in \gamma(G_1) \times \gamma(G_3) | \exists T_2 \in \gamma(G_{21}) \cap \gamma(G_{22}), T_1 = T_2 \land T_2 = T_3\})
\]

As \(G_1 = G_{21}\) and \(G_{22} = G_3\), the definition of consistent transitivity corresponds to the meet of gradual types \(\sqcap\):

**Lemma 8.** \((G_1) \circ^p (G_2) = (G_1 \sqcap G_2, G_1 \sqcap G_2)\).

The only operators that create new evidence are the initial evidence and consistent transitivity operators. These two operators always return evidence where both components are the same, therefore for simplicity we use notation \((G)\) instead of \((G, G)\).

**Illustration.** Let us go back to the example above. The gradual typing derivation \(\mathcal{D}\) is reduced by using preservation arguments as follows:

\[
\begin{align*}
\mathcal{D} \quad \text{\(\varnothing \vdash \text{false} : \text{Bool}\)} & \quad (\text{Bool} \vdash \text{\(\sim\)} ?) \\
\text{\(x : \mathbb{R} \vdash x + 1\)} & \quad \text{\(\varnothing \vdash (\lambda x : \mathbb{R} . x + 1) \mathbb{R} \) false : Int}\end{align*}
\]

Note that the use of ascriptions is crucial to represent each step of evaluation as a legal source typing, and most importantly to preserve evidence of different subterms. In this case, instead of replacing \(x\) with false, we replace \(x\) with false \(\sim ?\), otherwise \(1\) the consistent judgment \((\text{Bool} \vdash \text{\(\sim\)} ?)\) would be lost, and \(2\) the resulting gradual typing derivation would be ill-typed. To simplify the new ascription to \(\sim ?\), we need to combine \(\varepsilon_1\) and \(\varepsilon_3\) in order to (try to) obtain a justification for the transitive judgment, namely that \(\text{Bool} \sim \text{Int}\), but \(\varepsilon_3 \circ^p \varepsilon_1 = (\text{Bool} \circ^p \text{Int}) = (\text{Bool} \sqcap \text{Int})\), which is undefined, so a runtime error is raised.

To formalize this approach to the runtime semantics of gradual programs while avoiding writing reduction rules on actual (bi-dimensional) derivation trees, Garcia et al. adopt intrinsic terms [9], which are a flat notation that is isomorphic to typing derivations. In this paper, we use the same technique, and introduce the semantics via a language of intrinsic terms, called \(\lambda^G_{\text{Ref}}\).

4.4.1. Static semantics of \(\lambda^G_{\text{Ref}}\).

The syntax and static semantics of \(\lambda^G_{\text{Ref}}\) is presented in Fig. 4. Intrinsically-typed terms \(\text{t}^G\) comprise a family \(T[G]\) of type-indexed sets, such that ill-typed terms do not exist. Intrinsic terms are built up from disjoint families \(x^G \in V[G]\) and

\[\footnote{The consistent transitivity operator is parametrized by the type predicate being lifted. For instance, in other settings such as for subtyping, we write \(\circ^p\) instead, which is defined analogously.}\]
\[ o^G \in L[G] \] of intrinsically-typed variables and locations respectively. Note that intrinsic terms do not need explicit type environment \( \Gamma \) or store environments \( \Sigma \). Essentially, an intrinsic term \( t^G \in T[G] \) represents the typing derivation for the \( \lambda^G_{\text{ref}} \) judgment \( \Gamma; \Sigma \vdash t : G \), where \( \Gamma \) and \( \Sigma \) correspond to the free (intrinsically-typed) variables and locations in \( t^G \). We omit the type exponent on intrinsic terms when not needed, writing for instance \( t \in T[G] \).

The syntax and type rules for intrinsic terms is presented in Fig. 4. We use notation \( et \) to refer to an evidence term, which are terms augmented with evidence. This evidence justifies why the type of the term is consistent with the corresponding statically determined type.\(^5\) For instance, in term \( \text{Int} 1 :: ?, \) evidence \( \text{Int} \) is the companion of the raw value 1 and justifies that \( \text{Int} \sim ? \). Intrinsic values \( v \) can either be simple values \( u \) or ascribed values \( \text{Ref} :: t \). A simple value \( u \) can be a variable \( x \), a constant \( b \), a lambda abstraction \( \lambda x . t \), or a location \( o^G \). Some terms carry extra type annotations purely to help prove type safety, such as \( G \) in \( et := o^G \) \( et \), and to ensure unicity of typing during reduction such as \( G \) in \( et \circledast o^G \) \( et \). The rules mirror the type rules of \( \lambda^G_{\text{ref}} \) where each consistent judgment is justified by some evidence. The presentation may differ sometimes: for instance in Rule (IGasgn), its extrinsic counterpart has premise \( \text{Ref} \sim G \sim G2 \) which is equivalent to both \( G1 \sim \text{Ref} G3 \) and \( G2 \sim G3 \). We choose the later representation because it allows us to track evidence for each of the subterms. Something similar occurs in rules (IGderef) and (IGref): extrinsic rules (Gderef) and (Gref) has no consistent judgment whatsoever. This judgment is justified as subterms may evolve during reduction into something of a different (but consistent) type. For instance, in rule (IGderef), evidence \( \text{Ref} G1 \) justifies that the type of subterm \( t^G2 \) is consistent with \( G \), the type of the subterm during type checking. Alternatively, rule (IGderef) may also be seen as the intrinsic counterpart of the following \( \lambda^G_{\text{ref}} \) rule:

\[
\frac{\Gamma; \Sigma \vdash t : G1 \quad G1 \sim \text{Ref} G2}{\Gamma; \Sigma \vdash t^G2 : G2}
\]

where statically \( G1 = \text{Ref} G2 \). The elaboration rules for intrinsic terms, i.e. from \( \lambda^G_{\text{ref}} \) to \( \lambda^G_{\text{ref}} \), are explained later in \$4.5.\

---

\(^5\) As illustrated previously, evidence lives in a derivation tree, to justify a consistent judgment. When moving to the flat representation of intrinsic terms, the question arises of where to put the evidence. If a consistent judgment naturally corresponds to a subterm, then we annotate that subterm with evidence. Note however that in some gradual languages, such as security-typed languages, some consistent judgments may not correspond to one subterm, and in that case the practice is to decorate the term constructor itself [41].
4.4.2. Dynamic semantics of $\lambda^e_{\text{Ref}}$

Now we turn to the reduction rules of intrinsic terms, possibly failing with an **error** when combining evidences using consistent transitivity defined above. The reduction rules are presented in Fig. 5. They are defined over configurations $\text{Config_G}$ which consist of a pair of a term and a store. Contrary to [20], instead of using evaluation frames, we define the reduction semantics by using an equivalent representation using evaluation contexts [17], which make it easier to recover space efficiency (§5.3). We explain how to derive Rules (r4), (r5), and (r6), which deal with references, in §4.4.3.

Rules (r1), (r2), and (r3), present no novelty with respect to the presentation of Garcia et al. [20].

Rule (r4) reduces a reference to a new location $o^G$ not already present in the domain of store $\mu$. The store is extended mapping $o^G$ to the evidence value ascribed to $G_2$, the type of the reference determined statically. This use of ascriptions to anchor new evidence and preserve typing upon reduction is used in almost all other reduction rules.

Rule (r5) reduces a dereference to the underlying value $v$ of location $o^G$, ascribed to the statically determined type $G$, where evidence $\langle G_1 \rangle$ justifies that $G_2$ (the type of $v$), is consistent with $G$.

Rule (r6) updates the corresponding value on the heap of location $o^{G_1}$, with raw value $u$ ascribed to $G$. As $\langle G_2 \rangle$ justifies that the type of $u$ is consistent with $G_3$, and by inversion lemmas, $\langle G_1 \rangle$ justifies that $G_3 \sim G$, then evidence $\langle G_2 \rangle o^{G_1} (G_1) = \langle G_2 \cap G_1 \rangle$ (if defined) justifies that the type of $u$ is consistent with $G$. If $G_2 \cap G_1$ is not defined then a runtime error is signaled.

4.4.3. Deriving the reduction rules of $\lambda^e_{\text{Ref}}$

We now intuitively describe how we derive all reference related rules: (r4), (r5) and (r6).

**Rule (r4).** We start with the last intrinsic term before elimination

$$
\begin{align*}
\text{(IGref)} & : u \in T[G_1] & \langle G' \rangle G_1 \sim G_2 \\
\text{ref}^{G_2} & : \langle G' \rangle u \mid \mu \longrightarrow o^{G_2} \mid \mu [o^{G_2} \mapsto u]
\end{align*}
$$

By following the reduction rule for allocating a reference for $\lambda_{\text{Ref}}$ (Fig. 2), we would have to reduce allocations as follows:

$$
\begin{align*}
\text{ref}^{G_2} & : \langle G' \rangle u \mid \mu \longrightarrow o^{G_2} \mid \mu [o^{G_2} \mapsto \langle G_2 \rangle u :: G_2]
\end{align*}
$$

**Rule (r5).** Similarly to (r4), we start from the last intrinsic term before elimination:

$$
\begin{align*}
\text{(IGderef)} & : o^{G_2} \in T[\text{Ref} G_2] & \langle \text{Ref} G_1 \rangle \vdash \text{Ref} G_2 \sim \text{Ref} G \\
\text{Gref} & : \langle \text{Ref} G_1 \rangle o^{G_2} \in T[G]
\end{align*}
$$

Following Fig. 2 we would have to reduce dereferences as follows:

$$
\begin{align*}
\text{Gref} & : \langle \text{Ref} G_1 \rangle o^{G_2} \mid \mu \longrightarrow v \mid \mu
\end{align*}
$$

where $\mu (o^{G_2}) = v$. But as $v \in T[G_2]$ and the expected type of the dereference is $G$, we need to ascribe the dereferenced value to $G$. Now there is the question about what evidence to use. Of course, we cannot make up new evidence from thin air, we have to use and combine evidence already present in the redex. We know from the premise of the redex that $\langle \text{Ref} G_1 \rangle \vdash \text{Ref} G_2 \sim \text{Ref} G$, and by an inversion lemma we also know that $\langle G_1 \rangle \vdash G_2 \sim G$, which is exactly what we need. The final reduction rule for dereferences is therefore:

$$
\begin{align*}
\text{Gref} & : \langle \text{Ref} G_1 \rangle o^{G_2} \mid \mu \longrightarrow \langle G_1 \rangle v :: G \mid \mu
\end{align*}
$$

**Rule (r6).** Let us start from an assignment intrinsic term before elimination:

$$
\begin{align*}
\text{(IGasgn)} & : o^{G_1} \in T[\text{Ref} G_1] & u \in T[G_2] & \langle G_1 \rangle \vdash \text{Ref} G_1 \sim \text{Ref} G_3 \\
\text{assign} & : \langle G_1 \rangle o^{G_1} := G_1 \langle G_2 \rangle u \in T[\text{Unit}]
\end{align*}
$$

If we follow the reduction rule for assignment of $\lambda_{\text{Ref}}$ (Fig. 2), then we would be tempted to reduce assignments as:

$$
\begin{align*}
\langle G_1 \rangle o^{G_1} := G_3 \langle G_2 \rangle u \mid \mu \longrightarrow \text{unit} \mid \mu [o^{G_1} \mapsto u]
\end{align*}
$$
The problem here is that \( u \in T[G_2] \), but \( o^{G_1} \) should map to some value of type \( G_1 \). We can extend the store as \( \mu[o^{G_1} \mapsto cu :: G_1] \), for some \( c \) such that \( G_2 \sim G_1 \). Again, we combine evidences already present in the redex to construct new evidence. Notice that \((Ref \ G_1) \vdash Ref \ G_1 \sim Ref \ G_2 \), then by an inversion lemma \((Ref \ G_1) \vdash G_1 \sim G_3 \). When considering consistency and not subtyping, evidence is also symmetric, then \((G_1) \vdash G_3 \sim G_1 \). Also as \((G_2) \vdash G_2 \sim G_3 \), by consistent transitivity \((G_2) \circ \sim (G_1) \vdash G_2 \sim G_1 \) (if defined). As \((G_2) \circ \sim (G_1) = (G_2 \cap G_1) \), then the final reduction rule is:

\[
(Ref) \ 
\begin{align*}
&\text{if } \mu [G_2] \rightarrow_u G_2 \\
&\text{then } \mu [G_2] \rightarrow_{\sim} G_2 \\
&\text{error}
\end{align*}
\]

if the meet is defined, and \((G_2) \circ \sim (G_1) \vdash G_2 \sim G_1 \) otherwise.

4.5. Elaboration of \( \lambda_{\text{HS}}^e \) terms

So far we have presented intrinsic terms without formally explaining how to derive them. Fig. 6 present the type-driven elaboration rules from \( \lambda_{\text{HS}}^0 \) to \( \lambda_{\text{HS}}^e \). Judgment \( \Gamma ; \Sigma \vdash t : \sim_n t^C : G^G \) denotes the elaboration of term \( t^C \) from term \( t \), where \( t \) is typed \( G \) under environments \( \Gamma \) and \( \Sigma \). For simplicity, we write \( t : \sim_n t : G \) if \( ; \vdash t : \sim_n t : G \). Basically each consistent type judgment is replaced by the initial evidence operator \( \delta_\sim \).

Rule (TR ::) recursively translates the subterm \( t \) and the consistent subtyping judgment \( G^G \sim G \) from \( \Gamma :: \) is replaced with \( \delta_\sim (G^G, G) \), which computes evidence \( e \) for consistent subtyping. This evidence is eventually placed next to the translated term \( t^C \). Most of the elaboration rules follow this same recipe. Rule (TRapp), uses metafunctions \( \delta_\sim \text{ dom} \) and \( \delta_\sim \text{ cod} \) to avoid writing three different elaboration rules, e.g. when \( t_1 \) is typed \( ? \) then \( e_1 = \delta_\sim (?, ? : ?) \). The same is applied in rules (TRderef) and (TRasgn) where we use \( \delta_\sim \text{ ref} \) instead.

---

\(^{6}\) We use the \( e \) subindex to differentiate different translations presented in this chapter.
Note that the elaboration rules only enrich derivations with evidence (by using the initial evidence operator), and such resulting derivations are represented as intrinsic terms. Then by construction, the elaboration rules trivially preserve typing:

**Proposition 9 (Elaboration preserves typing).** If $\Gamma; \Sigma \vdash t : G$ and $\Gamma; \Sigma \vdash t \rightsquigarrow n t^G : G$, then $t^G \in T[G]$.

### 4.6. Properties

In order to establish type safety we first have to define well-typedness of the store $\mu$. Well-typedness of the store is usually defined with respect to a store environment, i.e. $\Sigma \vdash \mu$. Here, as we can see in Fig. 4, intrinsically-typed locations $o^G \in T[Ref G]$ obviate the need for store environment $\Sigma$: the store environment of a term $t$ is simply the set of intrinsically-typed free locations of the term, $\text{freeLocs}(t)$. Therefore, contrary to standard reference type systems, well-typedness of the store is defined with respect to an intrinsic term:

**Definition 8 (\(\mu\) is well typed).** A store $\mu$ is said to be well typed with respect to an intrinsic term $t^G$, written $t^G \vdash \mu$, if

1. $\text{freeLocs}(t^G) \subseteq \text{dom}(\mu)$, and
2. $\forall o^G \in \text{dom}(\mu), \ mu(o^G) \in T[G]$.

A store $\mu$ is well typed if all the free locations of a term are part of the domain of the store. Also for each of the intrinsic locations $o^G \in T[G]$ that are part of the domain of the store, then all the underlying values $v \in T[G]$.

Now we can establish type safety: closed terms do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

**Proposition 10 (Type safety).** Let $t^G$ a closed intrinsic term. If $t^G \in T[G]$ then one of the following is true:

1. $t^G$ is a value $v$;
2. if $t^G \vdash \mu$ then $t^G \mid \mu \mapsto t'^G \mid \mu'$ for some term $t'^G \in T[G]$ and some $\mu'$ such that $t'^G \vdash \mu'$ and $\text{dom}(\mu) \subseteq \text{dom}(\mu')$;
3. $t^G \mid \mu \mapsto \text{error}$.
Also, the gradual type system is a conservative extension of the static type system; i.e. both systems coincide on fully-annotated terms (where every subterm has only static type annotations). We first present the conservative extension of the static semantics of the static language.

**Proposition 11 (Equivalence for fully-annotated terms (statics)).** For any \( t \in \text{TERM}, \models_s t : T \) if and only if \( t : T \).

We now present the conservative extension of the dynamic semantics of the static language. The equivalence of the dynamic semantics for fully-annotated terms is more subtle. We cannot rely on a syntactic comparison of values because during reduction \( \lambda^\text{E}_{\text{REF}} \) inserts (possibly redundant) ascriptions. For instance, \((\lambda x : \text{Int}.1)\) is syntactically different, but equivalent to \((\lambda^{\text{Int}} x . (\text{Int}.1) :: \text{Int})\). To capture this relation, we formally connect both languages using logical relations between pairs of terms and stores.

We use notation \( \langle t, \mu \rangle \approx \langle t^T, \mu' \rangle : T \) to denote that the pair of term \( t \) and store \( \mu \) is related to the pair of term \( t^T \) and store \( \mu' \) at type \( T \). Two pairs of values and stores are related values at type \( T \), if first, both stores are related. Two stores are related if for all locations that are common to both stores, the stored values are related. Second, if \( T \) is a constant \( B \) or a reference \( \text{Ref} T \), then both values must be equal. Third, if \( T \) is a function, then both functions applied to related arguments yield two related computations. Two configurations (i.e. term-store pairs) are related computations if both configurations reduce to related values and stores. The complete definition and proofs are presented in C.1.

**Proposition 12 (Equivalence for fully-annotated terms (dynamics)).** For any \( t \in \text{TERM}, \models t : T, t \leadsto t^T : T \), then \( t \models \leadsto^* \langle t, \mu \rangle \approx \langle t^T, \mu' \rangle, \) for some \( \mu, \mu' \) such that \( \langle v, \mu \rangle \approx \langle \langle v', \mu' \rangle : T \).

Precision on terms, noted \( t_1 \subset t_2 \), is the natural lifting of type precision to terms. The gradual type system satisfies the static gradual guarantee of Siek et al. [38], i.e. losing precision preserves typeability: if a program is well-typed, then a less precise version of it also type checks, at a less precise type.

**Proposition 13 (Static gradual guarantee).** If \( t_1 : G_1 \) and \( t_1 \subset t_2 \), then \( t_2 : G_2 \), for some \( G_2 \) such that \( G_1 \subset G_2 \).

We also prove that \( \lambda^\text{E}_{\text{REF}} \) satisfies the dynamic component of the gradual guarantee: “any program that runs without error would continue to do so if it were given less precise types”. For this we must also extend the notion of precision over stores: intuitively a store is more precise than another store if its locations and values are more precise than the locations and values of the other.

**Proposition 14 (Dynamic gradual guarantee).** Suppose \( t^G_1 \subset t^G_2 \) and \( \mu_1 \subset \mu_2 \). Then if \( t^G_1 \models \mu_1 \leadsto t^G_2 \models \mu_2 \), then \( t^G_2 \models \mu_2 \) where \( t^G_1 \subset t^G_2 \) and \( \mu_1 \subset \mu_2 \).

### 4.7. \( \lambda^\text{E}_{\text{REF}} \) in action

\( \lambda^\text{E}_{\text{REF}} \) is semantically equivalent to HCC. The resulting language \( \lambda^\text{E}_{\text{REF}} \) behaves exactly as HCC. Recall the examples from §2.3; \( \lambda^\text{E}_{\text{REF}} \), like HCC, rejects examples 2 and 4, and accepts examples 3 and 5.

For instance, consider example 2. The corresponding \( \lambda^\text{E}_{\text{REF}} \) term is \((!(\text{ref } 4 :: ?) :: \text{Ref Bool})\). Its elaboration reduces as follows:

\[
\begin{align*}
&!(\text{Bool})((\text{Ref Bool})((\text{ref } 4 :: ?) :: \text{Ref Bool}) | \\
&\mapsto !(\text{Bool})((\text{Ref Bool})((\text{ref } 4 :: ?) :: \text{Ref Bool}) | \ [o^2 \mapsto (\text{Int}.4 :: ?)] \\
&\mapsto !(\text{Bool})((\text{ref } 4 :: ?) :: \text{Ref Bool} | \ [o^2 \mapsto (\text{Int}.4 :: ?)] \\
&\mapsto \text{error}
\end{align*}
\]

because \((\text{Int}.o ((\text{Bool})::))\) is not defined. Of course, this is just an example reduction; formally establishing the relation between both languages is the subject of §5.

\( \lambda^\text{E}_{\text{REF}} \) is not space efficient. Even though semantically equivalent, contrary to HCC, \( \lambda^\text{E}_{\text{REF}} \) is not space efficient. We can write programs in \( \lambda^\text{E}_{\text{REF}} \), that accumulate an unbounded number of evidences during reduction.

To illustrate, consider \( \lambda^\text{E}_{\text{REF}} \) term \( \Omega = (\lambda x : ?.x)(\lambda x : ?.x) \). Its elaboration to \( \lambda^\text{E}_{\text{REF}} \) is

\[
\Omega^\text{E} = (\ ? \mapsto ?)(\lambda x : ? \mapsto ?)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)(? \mapsto x)
\]

After multiple steps of reduction, the resulting term accumulates ascriptions as follows:

\[
(?)((?)((?)...((?)?)...)::?)::?
\]
This example illustrates how the order of combination of evidences impacts space efficiency and destroys tail recursion. Note that this problem applies to any language derived with AGT.

In contrast, with HCC, the same program reduces as follows (we omit stores for simplicity):

\[
\Omega^T = \{(\lambda x : ?c_1 \mathsf{xx})\ c_2(\lambda x : ?c_1 \mathsf{xx}) \mapsto c_1 (c_2 (\lambda x : ?c_1 \mathsf{xx}))\ c_2(\lambda x : ?c_1 \mathsf{xx}) \mapsto \Omega^T \mapsto \ldots
\]

where \(c_1\) is a coercion from \(?\) to \(?\), and \(c_2\) from \(?\) to \(\cdot\).

Even though \(\lambda_{\text{ref/}}\) and HCC are different regarding space efficiency, they are semantically equivalent: given a term and its compilations to \(\lambda_{\text{ref/}}\) and HCC\(\dagger\) (an adapted version of HCC), either both terms reduce to values, both terms diverge, or both terms reduce to an error. In the following, we formalize the relation between \(\lambda_{\text{ref/}}\) and HCC (§5), along with the changes needed to recover space efficiency in \(\lambda_{\text{ref/}}\) (§5.3).

5. Comparing \(\lambda_{\text{ref/}}\) and HCC

In this section we compare \(\lambda_{\text{ref/}}\) and HCC, the space-efficient coercion calculus of Herman et al. [24]. We start by presenting the static and dynamic semantics of HCC\(\dagger\), an adapted version of HCC extended with conditionals and binary operations. Then we formalize the relation between both semantics as follows: given a \(\lambda_{\text{ref/}}\) term and its corresponding elaboration to \(\lambda_{\text{ref/}}\) and translation to HCC\(\dagger\), we prove that the resulting terms are bisimilar, and that consequently they either both terminate, both fail, or both diverge. Despite this tight relation, the dynamic semantics of \(\lambda_{\text{ref/}}\) are not space-efficient: ascriptions can be repeatedly accumulated during reduction, contrary to HCC. We finalize this section by adjusting the dynamic semantics of \(\lambda_{\text{ref/}}\) to recover space efficiency.

5.1. The coercion calculus

In this section we present HCC\(\dagger\), an adaptation of HCC extended with conditionals and binary operations. This language is designed as a cast calculus for \(\lambda_{\text{ref/}}\). The following presentation of this language is closely related to the coercion calculus presented by Siek et al. [37].

Usually, the operational semantics of gradual languages generate proxies when reducing function applications which involve casts. This approach may result in an unbounded growth in the number of proxies, which impacts space efficiency and destroys tail recursion [24]. HCC was designed to represent and compress sequences of casts, by using coercions instead of casts (and function proxies). HCC recovers space efficiency by combining and normalizing adjacent coercions to limit their space consumption to a constant factor.

Static semantics. Fig. 7 presents the static semantics of HCC\(\dagger\). The syntax includes gradual types \(G\), ground types \(R\), coercions \(c\), and terms \(t\). Ground types \(R\) are the only types allowed to be coerced directly from/to the unknown type \(?\). A ground type can be either a function \(? \rightarrow ?\), a reference \(\mathsf{Ref} ?\), or a base type \(B\). Terms \(t\) can also be coerced terms \(ct\). The judgment \(c \vdash G_1 \Rightarrow G_2\) represents that coercion \(c\) is used to coerced values of type \(G_1\) to type \(G_2\). The identity coercion \(\mathsf{I_G}\) represents a coercion from a type to itself. The failure coercion \(\mathsf{Fail}\) represents an invalid coercion. The tagging coercion \(\mathsf{R} ?\) represents a coercion from a ground type \(R\) to \(?\). The check-and-untag coercion \(\mathsf{R} ?\) represents a coercion from \(?\) to a ground type \(R\). The function coercion \(c_1 \rightarrow c_2\) represents a coercion where \(c_1\) coerces the function argument, and \(c_2\) coerces the result. The reference coercion \(c_1 \mathsf{Ref} c_2\) represents a coercion where \(c_1\) coerces values written in the heap, and \(c_2\) coerces values read from the heap. Finally, a coercion composition \(c_1 ; c_2\) represents the coercion \(c_1\) followed by coercion \(c_2\). We consider coercions equal up to associativity of composition. The type rules are standard for a cast calculus. Each type rule of Fig. 3 is simplified by replacing uses of consistency with equality. We replace the ascription rule \((G::)\) with rule \((\text{HC})\), used to typecheck coerced terms: a coerced term \(ct\) has type \(G\) if the subterm \(t\) has type \(G^\prime\), and \(c\) is a coercion from \(G^\prime\) to \(G\).

Dynamic semantics. Fig. 8 presents the dynamic semantics of HCC\(\dagger\). A value \(v\) can be a raw value \(u\), or a coerced value \(cu\), where coercion \(c\) is in normal form. We say a coercion is in normal form if it is irreducible, denoted \(\mathsf{nm}\); the predicate is defined in Fig. 9. To reduce programs we use three different evaluation contexts: \(H\) to reduce coercions, and \(F\) and \(E\) to reduce terms. Coercions are combined using the coerced term reduction rule \(\mapsto\). Coercions are maintained in normal form throughout evaluation using big step semantics of the coercion reduction rule \(\mapsto\), and the normal form predicate \(\mathsf{nm}\). The coercion reduction rule combines coercions using the notion of coercion reduction rule \(\mapsto\). A failure coercion is produced when a tagging and a check-and-untag coercions are combined, and the ground types involved are different. When the types are the same, then an identity coercion is produced. The combination of an identity coercion with another coercion \(c\) produces the same coercion \(c\). On the contrary, the combination of a failure coercion with another coercion \(c\) propagates the failure coercion. Reductions of both combination of function coercions and combination of reference coercions are defined inductively. Notice the contravariant combination order for the argument of functions, and in the coercions for write values in the heap respectively. Note that in the reduction of coerced terms, a failure coercion does not trigger a runtime error immediately (i.e. \(\mathsf{Fail} \mapsto\), \(\mathsf{error}\), but after the subterm is reduced to a raw value (i.e. \(\mathsf{Fail} u \mapsto\), \(\mathsf{error}\)). The reason for this is that HCC aims to regain space efficiency without changing the behavior of standard cast calculus/coercion semantics, which combines casts when the subterm is a value. The rest of the dynamic semantics is standard to cast calculus. The reduction of coerced dereferences coerces the value on the heap with the second component of the coercion, and dually the reduction of coerced assignments coerces the updated value using the first component of the coercion.
\[ R \in \text{GROUNDTYPE}, \quad c \in \text{COERCION}, \quad t \in \text{CTERM}. \]

\[
G ::= ? | B | \text{Ref } G | \text{Ref } G \quad \text{(Gradual types)}
\]

\[
R ::= \text{?} | \text{Ref } ? | B \quad \text{(Ground types)}
\]

\[
c ::= t_G | \text{Fail } | ! | \text{Ref } ? | c \quad \text{(coercions)}
\]

\[
t ::= b | (\lambda x : G.t) | o | x | t | t \circ t | \text{if } t \text{ then } t \text{ else } t | c \ t | \text{ref } t | ! t | ? t := t \quad \text{(terms)}
\]

**Coercion typing**

\[
\begin{array}{l}
\Gamma; \Sigma \vdash_H x : G \\
\Gamma; \Sigma \vdash_H b : B
\end{array}
\]

**Translation semantics.** Fig. 10 presents the translation rules from \( \lambda_{\text{eff}} \) to \( \text{HCC}^+ \). The translation is a type-driven coercion insertion. The key idea is to insert coercions where consistency is used in the typing derivation. The translation judgment has the form \( \Gamma; \Sigma \vdash t \sim \gamma \ t' : G \) which represent translation from \( \lambda_{\text{eff}} \) term \( t \) of type \( G \), to \( \text{HCC}^+ \) term \( t' \), under environments \( \Gamma \) and \( \Sigma \). We write \( t \sim \gamma : G \) if \( \vdash t \sim \gamma \ t : G \). Note that we assume that variables \( x \) and \( \lambda x \) refer the same variable. Similarly, constants \( b \) and \( B \), and locations \( o \) and \( o \) refer to the same location. We paint them red only to disambiguate terms of different languages. Coercions are introduced using the \( \Gamma_1 \Rightarrow G_2 \) metafunction, which represents the insertion of a coercion from \( G_1 \) to \( G_2 \). This metafunction avoids the insertion of redundant coercions by checking if \( G_1 \) is syntactically equal to \( G_2 \). If both types are the same, then the coercion is not introduced. Otherwise we use the coercion function \( \Gamma_1 \Rightarrow G_2 \) to elaborate the coercion from \( G_1 \) to \( G_2 \). The inductive definition of the coercion insertion function is presented in Fig. 11, and follows closely the rules for coercion typing. For instance, as \( R ? \Rightarrow ? \Rightarrow R \), then \( \{ ? \Rightarrow R \} = R ? \). These are some subtleties worth mentioning, such as the definition of \( \{ ? \Rightarrow \text{Ref } G \} \). This should result in a coercion from \( ? \) to \( \text{Ref } G \), but there is no direct coercion from unknown to any given type \( \text{Ref } G \). Consequently \( \{ ? \Rightarrow \text{Ref } G \} \) is defined as the composition of a coercion from \( ? \) to \( \text{Ref } G \): \( (\text{Ref } ? ) ? \) (to test if the value is actually a reference), with a coercion from \( \text{Ref } ? \) to \( \text{Ref } G \): \( (\text{Ref } ? \Rightarrow \text{Ref } G) \), which follows the inductive definition. Analogously, \( \{ \text{Ref } G \Rightarrow ? \} \) is defined as the composition of a coercion from \( \text{Ref } G \) to \( \text{Ref } ? \), with a coercion from \( \text{Ref } ? \) to \( ? \). Notice that by construction, we do not need definitions for \( \{ G \Rightarrow G \} \) and \( \{ ? \Rightarrow ? \} \) as these cases are avoided thanks to the \(! \Rightarrow ! \) \text{metacolumn.}

**5.2. Relating \( \lambda_{\text{eff}}^e \) and \( \text{HCC}^+ \)**

We now establish the equivalence of the \( \lambda_{\text{eff}}^e \) and \( \text{HCC}^+ \) semantics for elaborated and translated \( \lambda_{\text{eff}} \) terms respectively, by using a bisimulation relation.

Fig. 12 presents function \( \langle \rangle \), which relates evidence augmented consistent judgments with coercions during the definition of the bisimulation relation. A naive relation between evidence and coercion is ambiguous unless one indicates the gradual types involved in the judgment. For instance, evidence \( \langle \text{int} \rangle \) corresponds to both coercion \( \text{int} ? \) or \( \text{int} ! \), unless we expose the judgment associated with the evidence, so \( \langle \text{int} ? \rangle \sim \text{int} ? \) correspond exactly to the coercion from \( \text{int} \) to \( ? \), \( \langle \text{int} ! \rangle \). The definition follows the definition of the coercion insertion function presented in Fig. 10 (e.g., \( \{ ? \Rightarrow R \} = R ? \), so
\[
\begin{align*}
\mathbf{u} & := b \mid (\lambda x : \mathbf{G}. \mathbf{t}) \mid o \\
\mathbf{v} & := \mathbf{u} \mid \mathbf{cu} \text{ where } \mathbf{nm} \mathbf{c} \\
\mathbf{H} & := \square \mid \square \mathbf{c} \mid c \mid \square \mathbf{c} \mid \square \mathbf{c} \rightarrow \mathbf{c} \mid \square \mathbf{c} \mid \square \mathbf{c} \rightarrow \mathbf{c} \mid \square \mathbf{Ref} \mathbf{c} \mid \square \mathbf{Ref} \mathbf{c} \mathbf{square} \\
\mathbf{F} & := \square \mid \mathbf{E} + \mathbf{v} \mid \mathbf{E} \mathbf{v} \mid \mathbf{E} \mathbf{v} \mid \mathbf{E} = \mathbf{G} \\
\text{if } \mathbf{E} \text{ then } \mathbf{t} \text{ else } \mathbf{t} \mid \mathbf{ref} \mathbf{E} \mid \mathbf{E} = \mathbf{t} \mid \mathbf{v} := \mathbf{E} \\
\mathbf{E} & := \mathbf{F} \mid \mathbf{cF} \\
\end{align*}
\]

(raw values)

(values)

(Coercion contexts)

(Cast-free contexts)

(Evaluation contexts)

\[
\mathbf{c}_1 \rightarrow \mathbf{c}_2 \\
\mathbf{H}[(\mathbf{c}_1) ] \rightarrow \mathbf{H}[\mathbf{c}_2] \\
\mathbf{\Gamma} : \mathbf{\Delta} \rightarrow \mathbf{\Gamma'} : \mathbf{\Delta'} \\
\]

Coercion reduction

(Initial evidence)

(CTerm × (CTerm ∪ \{error\}))

Coerced term reduction

\[
\begin{align*}
\mathbf{\mu} \mathbf{u} & \mid \mathbf{\mu} \rightarrow \mathbf{\epsilon U} \mid \mathbf{\mu} \\
\mathbf{c}_1 (\mathbf{c}_2 \mathbf{t}) & \mid \mathbf{\mu} \rightarrow \mathbf{\epsilon C} \mathbf{t} \mid \mathbf{\mu} \\
\text{if } \mathbf{c}_2 ; \mathbf{c}_1 \rightarrow * \mathbf{c} \wedge \mathbf{nm} \mathbf{c} \\
\text{Fail} \mathbf{u} & \rightarrow \mathbf{\epsilon error} \\
\end{align*}
\]

Fig. 8. HCC⁺⁺: Dynamic semantics.

\[
\begin{align*}
\mathbf{nm} \mathbf{c} \\
\mathbf{nm} \mathbf{i}_G \quad \mathbf{nm} \text{ Fail} \quad \mathbf{nm} \mathbf{R} \quad \mathbf{nm} \mathbf{R'} \quad \mathbf{nm} \mathbf{R'} \quad \mathbf{nm} \mathbf{R} \quad \mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \\
\mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \quad \mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \\
\mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \quad \mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \\
\mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \quad \mathbf{nm} \mathbf{c}_1 \quad \mathbf{nm} \mathbf{c}_2 \\
\mathbf{nm} \mathbf{(Ref) t} ; \mathbf{Ref} \mathbf{c}_1 \mathbf{c}_2 \\
\mathbf{nm} \mathbf{Ref} \mathbf{c}_1 \mathbf{c}_2 ; (\mathbf{Ref}) ? ? \\
\mathbf{nm} \mathbf{Ref} \mathbf{c}_1 \mathbf{c}_2 ; (\mathbf{Ref}) ? ? \\
\mathbf{nm} \mathbf{Ref} \mathbf{c}_1 \mathbf{c}_2 ; (\mathbf{Ref}) ? ? \\
\mathbf{nm} \mathbf{Ref} \mathbf{c}_1 \mathbf{c}_2 ; (\mathbf{Ref}) ? ? \\
\end{align*}
\]

Fig. 9. HCC⁺⁺: Coercion normal forms.

\((\mathbf{R} \rightarrow ? \sim \mathbf{R}) = \mathbf{R})\), save for a few extra cases described next. Reflexive judgments on ground types and the unknown type, where evidence corresponds to the initial evidence, are mapped to identity coercions for ground types and the unknown type respectively. The definition also takes into consideration judgments where both types are unknown, and the evidence
Translation rules

\[
\begin{align*}
\text{(HRx)} & \quad \Gamma ; \Sigma \vdash t \sim_c x : G \quad \text{\(x : G \in \Gamma\)} \\
\text{(HRc)} & \quad \Gamma ; \Sigma \vdash \theta(b) = B \quad \text{\(\theta(b) = B\)}
\end{align*}
\]

\[
\begin{align*}
\text{(HRapp)} & \quad \Gamma ; \Sigma \vdash t_1 \sim_c t_1' : G_1 \quad \Gamma ; \Sigma \vdash t_2 \sim_c t_2' : G_2 \\
\text{(HRop)} & \quad \Gamma ; \Sigma \vdash t_1 \sim_c t_1' : G_1 \quad \Gamma ; \Sigma \vdash t_2 \sim_c t_2' : G_2
\end{align*}
\]

\[
\begin{align*}
\text{(HRref)} & \quad \Gamma ; \Sigma \vdash \text{ref } t \sim_c \text{ref } G \quad \text{\(G \Rightarrow G'\)} \\
\text{(HRderef)} & \quad \Gamma ; \Sigma \vdash \text{ref } t \sim_c \text{ref } G \quad \text{\(G \Rightarrow G'\)}
\end{align*}
\]

\[
\begin{align*}
\text{(HRasgn)} & \quad \Gamma ; \Sigma \vdash t_1 \sim_c t_1' : G_1 \quad \Gamma ; \Sigma \vdash t_2 \sim_c t_2' : G_2 \quad G_3 = \text{\text{ref}(G_1)} \\
\text{(HR)} & \quad \Gamma ; \Sigma \vdash o \sim_c o : \text{Ref } G
\end{align*}
\]

where \(\langle G_1 \Rightarrow G_2 \rangle t = \begin{cases} t & \text{if } G_1 = G_2 \\ \{G_1 \Rightarrow G_2\} t & \text{otherwise} \end{cases}\)

\[\text{Fig. 10. } \lambda_{\text{ref}} \text{ to HCC}^+ \text{ translation rules.}\]

The bimulation relation is formally presented in Fig. 13. This relation syntactically relates a \(\lambda_{\text{ref}}^+\) term and an HCC\(^+\) term. Rules (b\const\) and (b\lambda\) are straightforward. Rules (b\x\) and (b\b\) relate two variables and two constants respectively, where for simplicity we assume that \(x^G\) and \(x\) refer to the same source variable \(x\) and \(b\) and \(b\) refer to the same constant \(b\). Similarly, Rule (b\ref\) relates two locations, assuming that the creation of locations is deterministic, i.e., new references in related executions are always allocated at the same address: \(o^G\) and \(o\) refer to the same location \(o\). Rule (b\app\) relates two application terms inductively, but notice that because evidence terms do not correspond to anything in HCC\(^+\), we build an ascribed term instead, e.g. to relate \(e_{11}\) with \(t_{21}\) we notice that the type of \(t_{21}\) has to be \(G_1 \Rightarrow G_2\), therefore as \(e : G \vdash G_1 \Rightarrow G_2\) where \(t_1 \in \text{\text{T}[G]}\), we can inductively relate \(e_{11} : G \Rightarrow G_2\) with \(t_{21}\) instead. Similarly, we relate \(e_{212} : G_1\) as \(e_2 : G \sim G_1\), where \(t_2 \in \text{\text{T}[G']}\) with \(t_2\). We use the same reasoning for rules (b\ref\), (b\l\), and (b\ic\). Rules (b\eq\), (b\id\), and (b\ic\) are the most important rules. Rule (b\eq\) is the most intuitive rule; it relates an ascribed term with a coerced term, only if the underlying evidence of the ascription is mapped to the coercion. Rule (b\id\) relates a redundant ascription with a term without a coercion. The reason is that a term like \(t_u\) (which is related to \(G)u^G\) by \(b\eq\) if \(u\) is related to \(u^G\) reduces to \(u\), whereas in \(\lambda_{\text{ref}}^+\) this redundant cast is not eliminated. Rule (b\ic\) relates \(\lambda_{\text{ref}}^+\) terms with HCC\(^+\) terms that have eagerly combined coercions starting from the outermost pair of coercions. For instance,

\footnote{Note that if \(G_1 \Rightarrow G_2\) \(\vdash ? \sim ?\) then \(G_1 \Rightarrow G_2\) \(\vdash ? \sim ? \sim ?\).}
Conversely, related

\[ \langle (R) \rangle \leftrightarrow R \sim R \Rightarrow i_R \]

\[ \langle (\exists) \rangle \leftrightarrow \exists \sim \exists \Rightarrow i \]

\[ \langle (R) \rangle \leftrightarrow R \sim R \Rightarrow R \]
Fig. 14. $\lambda^{\epsilon}_{\text{ref}}$: Modifications for a space-efficient dynamic semantics.

$\lambda^{\epsilon}_{\text{ref}}$ term. This is because for cases like application and assignment, $\lambda^{\epsilon}_{\text{ref}}$ does in one step what HCC$^+$ may do in three. For instance, take the following two related terms: $(\varepsilon_1 (\lambda x. t) @ G_1 \Rightarrow G_2 \varepsilon_2 u) \approx ((c_1 \rightarrow c_2) (\lambda x : G.t)c_3 u)$. After a step of reduction the two terms are not related; the HCC$^+$ term has to take two more steps to relate the body of the lambdas (we ignore stores for simplicity):

\[
\begin{align*}
(\varepsilon_1 (\lambda x. t) @ G_1 \Rightarrow G_2 \varepsilon_2 u) & \mapsto i\text{code}(\varepsilon_1) (\{(\varepsilon_2 \circ \text{idom}(\varepsilon_1)u) :: G/\lambda x. t\} :: G_2) \\
((c_1 \rightarrow c_2) (\lambda x : G.t)c_3 u) & \mapsto c_1 ((\lambda x : G.t) (c_2 c_3 u)) \mapsto c_1 ((\lambda x : G.t) (c_3 u)) \mapsto c_1 t [c_3 u/x]
\end{align*}
\]

The key result is that given a $\lambda^{\epsilon}_{\text{ref}}$ term, its elaboration to $\lambda^{\epsilon}_{\text{eff}}$ and its translation to HCC$^+$ are bisimilar.

**Proposition 16** (Translations are bisimilar). Given $t : G$, if $t \rightsquigarrow_n t_1 : G$ and $t \rightsquigarrow \epsilon t_2 : G$, then $t_1 \approx t_2$.

A direct consequence of bisimilarity is that elaborating to $\lambda^{\epsilon}_{\text{eff}}$ and translating to HCC$^+$ yield programs that co-terminate, co-fail, or co-diverge. We write $t \downarrow$ (resp. $t \downarrow \epsilon$), if $t \mapsto^* \mu$ (resp. $t \mapsto^* \epsilon$) for some resulting store $\mu$, and similarly, we write $t \downarrow$ (resp. $t \downarrow \epsilon$), if $t \mapsto^* \mu$ (resp. $t \mapsto^* \epsilon$) for some resulting store $\mu$.

**Corollary 17.** Given $t : G$, if $t \rightsquigarrow_n t_1 : G$ and $t \rightsquigarrow \epsilon t_2 : G$, then $t_1 \downarrow \iff t_2 \downarrow$ and $t_1 \downarrow \epsilon \iff t_2 \downarrow \epsilon$. (Co-divergence follows trivially.)

5.3. Recovering space efficiency in $\lambda^{\epsilon}_{\text{ref}}$

Although we have established that given a $\lambda^{\text{ref}}$ term, its elaboration to $\lambda^{\epsilon}_{\text{eff}}$ and its translation to HCC$^+$ are bisimilar, the dynamic semantics of $\lambda^{\epsilon}_{\text{ref}}$ are not space-efficient, i.e. ascriptions can be repeatedly accumulated during reduction as illustrated in §4.7. We now present the changes needed in the runtime semantics of $\lambda^{\epsilon}_{\text{ref}}$ in order to enjoy a space-efficient operational semantics.

The main space efficiency problem with $\lambda^{\epsilon}_{\text{ref}}$ is that the definition of evaluation contexts allows ascriptions and evidences to accumulate until the corresponding subterm is reduced to a value. Fig. 14 presents a space-efficient dynamic semantics variant with respect to the original dynamic semantics of Fig. 5 (changes are highlighted in gray). To achieve space efficiency, we eliminate the $E :: G$ evaluation context, so as to forbid reduction inside nested ascriptions. Instead of combining evidences starting from the innermost pair of evidences, rule (r7) now combines evidence eagerly starting from the outermost pair of evidences by using the new $\sqcap :: G$ evaluation context, before subterm $t$ is reduced to a value.

**Preserving the failure behavior.** To preserve the failure behavior of the original dynamic semantics (and fail at an point of execution), (r7) cannot simply reduce to an error when consistent transitivity is not defined. For instance, consider $t = \langle \text{Bool} \rangle (\langle \text{Int} \rangle + \langle \text{Int} \rangle)$ evaluation context, so as to forbid reduction inside nested ascriptions. Instead of combining evidences starting from the innermost pair of evidences, rule (r7) now combines evidence eagerly starting from the outermost pair of evidences by using the new $\sqcap :: G$ evaluation context, before subterm $t$ is reduced to a value.

If we combine evidences starting from the outermost pair of evidence, $t$ would reduce to an error immediately, i.e. $t \mapsto \epsilon$, because $\langle \text{Int} \rangle \circ \langle \text{Bool} \rangle$ is not defined.

If combination of evidences is not defined then instead of reducing directly to an error, we reduce to an evidence term using the pending error evidence $\varepsilon_{\text{err}}$. The pending error evidence $\varepsilon_{\text{err}}$ is defined such that $\varepsilon_{\text{err}} \vdash G_1 \rightsquigarrow G_2 \varepsilon_2 u$ for any $G_1$ and $G_2$. We also update the definition of intrinsic values $v$ to raw values $u$, or ascribed simple values $v :: G$ where $\varepsilon \neq \varepsilon_{\text{err}}$. Rule (r8) just propagates evidence $\varepsilon_{\text{err}}$ until rule (r9) finally reduces a pending error evidence combined with a simple value to an error. Using the space-efficient semantics, $t$ now reduces as follows:

\[
t \mapsto \varepsilon_{\text{err}} (\langle \text{Int} \rangle + \langle \text{Int} \rangle) :: ? \mapsto \varepsilon_{\text{err}}2 :: ? \mapsto \epsilon
\]
Space efficiency. We now demonstrate that the total cost of maintaining evidences during reduction in $\lambda_{\text{err}}^e$ is bounded. We proceed analogously to Herman et al. [24], and refer to their work for more details.

First, to reason about the space required by evidences we introduce the notion of size and height of evidences and types:

\[
\begin{align*}
\text{size}(G) & = \text{size}(G) \\
\text{size}(\epsilon_{\text{err}}) & = 1 \\
\text{size}(?) & = 1 \\
\text{size}(B) & = 1 \\
\text{size}(G_1 \rightarrow G_2) & = 1 + \text{size}(G_1) + \text{size}(G_2) \\
\text{size}(\text{Ref } G_1) & = 1 + \text{size}(G_1)
\end{align*}
\]

\[
\begin{align*}
\text{height}(G) & = \text{height}(G) \\
\text{height}(\epsilon_{\text{err}}) & = 1 \\
\text{height}(?) & = 1 \\
\text{height}(B) & = 1 \\
\text{height}(G_1 \rightarrow G_2) & = 1 + \max(\text{height}(G_1), \text{height}(G_2)) \\
\text{height}(\text{Ref } G_1) & = 1 + \text{height}(G_1)
\end{align*}
\]

size and height respectively compute the size and depth of the abstract syntax of evidence and types. Note that the maximum number of children for a given node is two (functions), therefore we can bound the size of evidences in terms of its height as a binary tree:

**Lemma 18.** $\forall \epsilon \vdash G_1 \sim G_2$, $\text{size}(\epsilon) \leq 2^{\text{height}(\epsilon)} - 1$

Note that we get a tighter bound than in HCC because evidence composition in $\lambda_{\text{err}}^e$ is not part of the syntax of evidence. The height of any type computed by the meet operator is bounded by the maximum height of both types.

**Lemma 19.** If $G_1 \sqcap G_2 = G_3$, then $\text{height}(G_3) \leq \max(\text{height}(G_1), \text{height}(G_2))$

This lemma allows us to establish two similar lemmas to bound the maximum height of evidences:

**Lemma 20.** If $\sqcap_{\epsilon}(G_1, G_2) = \epsilon$, then $\text{height}(\epsilon) \leq \max(\text{height}(G_1), \text{height}(G_2))$

**Lemma 21.** If $\epsilon_1 \triangleleft \epsilon_2 = \epsilon_3$, then $\text{height}(\epsilon_3) \leq \max(\text{height}(\epsilon_1), \text{height}(\epsilon_2))$

Given a $\lambda_{\text{err}}^e$ term and its elaboration to $\lambda_{\text{err}}^e$, the size and height of every evidence found at any step of reduction is bounded by some types found during the elaboration.

**Proposition 22.** If $t \sim_\rightarrow t : G$ and $t \mid \cdot \rightarrow^* t' : (\mu')'$, then there exists $G'$ in the derivation of $t \sim_\rightarrow t : G$ such that $\text{height}(\epsilon) \leq \text{height}(G')$ and $\text{size}(\epsilon) \leq 2^{\text{height}(G')} - 1$.

Finally, we bound the total cost of maintaining evidences. To do this we define the size of a program configuration $p = (t, \mu)$ as the sum of the sizes of its term and store subcomponents. Following [24], for the store, we only count the locations that an idealized garbage collector would consider live, by using an auxiliary reachable metafunction:

\[
\begin{align*}
\text{size}(t, \mu) & = \text{size}(t) + \text{size}(\mu |_{\text{reachable}}(t)) \\
\text{size}(\mu) & = \sum_{\alpha \in \text{dom}(\mu)} (\text{size}(\alpha^G) + \text{size}(\mu(\alpha^G))) \\
\text{size}(b) & = = 1 \\
\text{size}(\alpha^G) & = \text{size}(x^G) = 1 + \text{size}(G)
\end{align*}
\]

\[\text{8} \] In HCC the bound is $5(2^{\text{height}(\epsilon)} - 1)$, where 5 represents the maximum width of a normalized coercion.
\[
\text{size}(\lambda x^G.t) = 1 + \text{size}(x^G) + \text{size}(t)
\]
\[
\text{size}(\text{ref}^{G_2} \text{et}^{G_1}) = \text{size}(\text{et}^{G_2} \text{G}^1) = \text{size}(\text{et}^{G_1} : G_2) = 1 + \text{size}(\varepsilon) + \text{size}(t^{G_1}) + \text{size}(G_2)
\]
\[
\text{size}(\text{e}^{G_1} := G_3 \text{e}_2^{G_2}) = \text{size}(\text{e}_1^{G_1} @ G_3 \text{e}_2^{G_2}) = 1 + \text{size}(\varepsilon_1) + \text{size}(t^{G_1}) + \text{size}(\varepsilon_2) + \text{size}(t^{G_2}) + \text{size}(G_3)
\]

We now compare the size of a program configuration with the size of a program configuration reduced in an "oracle" semantics, where evidences require no space. The oracular measure \(\text{size}_\text{OR}\) is defined analogous to size, but where \(\text{size}_\text{OR}(\varepsilon) = 0\).

**Proposition 23.** If \(t \leadsto_n t : G\) and \(t \mid \cdot \leadsto\star t' \mid \mu'\), then there exists \(G'\) in the derivation of \(t \leadsto_n t : G\) such that \(\text{size}(\langle t', \mu' \rangle) \in O(2^{\text{height}(G')} \cdot \text{size}_\text{OR}(\langle t', \mu' \rangle))\).

As explained by Herman et al. [24], this result shows that when reducing a term \(t\), coercions occupy bounded space, which depends on the height of some type used in the type derivation of \(t\).

Relating the space-efficient semantics and \(\text{HCC}^+\). Regarding the result of §5.2, the new space-efficient semantics are now more closely related to \(\text{HCC}^+\). In particular rule (b::leq) of Fig. 13 is not needed anymore as evidences and coercions are reduced in lock-step. The only difference between both semantics is that identity coercions are eliminated during reduction, whereas redundant evidences are not (and this is why we have to keep the (b::id) rule).

**Eager space-efficient dynamic semantics.** Alternatively, we could have defined the space-efficient dynamic semantics without rules (r8) and (r9), and where (r7) would be defined as follows:

\[
\langle G_2 \rangle (\langle G_1 \rangle) \text{E} : G \leadsto \varepsilon \begin{cases} 
\langle G_1 \cap G_2 \rangle t & \text{if } G_1 \cap G_2 \text{ is not defined} \\
\text{error} & \text{otherwise}
\end{cases}
\]

This variant would yield a more eager semantics. Going back to the previous example where \(t = \langle \text{Bool} \rangle (\langle \text{Int} \rangle [1 + \langle \text{Int} \rangle 1] :: ? :: ?) :: ? \), \(t\) would now reduce immediately to an error after trying to combine the \(\langle \text{Int} \rangle\) and \(\langle \text{Bool} \rangle\) outer evidences: \(t \mid \cdot \leadsto\star \text{error}\).

The main difference with the previous approach is that a program that may diverge using the original dynamic semantics may now also fail with an error (therefore the bisimulation would be weaker). To illustrate this, consider the following program, where \(\Omega\) is a non-terminating term:

\[
\langle \text{Int} \rangle (\langle ? \rangle (\langle \text{Bool} \rangle (\langle ? \rangle \Omega :: ?) :: \text{Bool} :: ?) :: ?) :: \text{Int}
\]

Using the original dynamic semantics, this program diverges because \(\Omega\) is evaluated first before combining the outer evidences. Using the first variant of the space-efficient semantics, this program also diverges because the outer evidence \(\varepsilon\) never triggers an error because \(\Omega\) never reduces to a value. But using the eager variant of the space-efficient semantics, the program reduces to an error just after combining the \(\langle \text{Bool} \rangle\) and \(\langle \text{Int} \rangle\) evidences.

### 6. Encoding permissive and monotonic references in \(\lambda^\text{pm}_\text{REF}\)

In this section we present \(\lambda^\text{pm}_\text{REF}\), an extension of \(\lambda^\text{REF}\) with support for both permissive and monotonic references [39]. We codify permissive and monotonic references by introducing new term constructors for each form of reference in \(\lambda^\text{REF}_\text{pm}\). Encoding monotonic references is more difficult than encoding permissive references, as it involves extending the dynamic semantics of \(\lambda^\text{REF}_\text{pm}\).

\(\lambda^\text{pm}_\text{REF}\) supports two constructors to create references: \(\text{ref}\) for guarded reference, and \(\text{mref}\) for monotonic references. For instance, to emulate the behavior of monotonic references in examples 3 and 5, we use \(\text{mref}\) as illustrated below:

\begin{verbatim}
1 let x = mref (4 :: ?) 1 let x = mref (4 :: ?)
2 let y: Ref Bool = x ← runtime error 2 let y: Ref Int = x
3 y := true 3 x := true ← runtime error
4 !y

Example 3

Example 5
\end{verbatim}

#### 6.1. Static semantics

We start by extending the syntax of \(\lambda^\text{REF}\) as follows:

\[
\begin{align*}
\text{z} & ::= g \mid p \mid m & \text{(reference mode)} \\
\nu & ::= \ldots \alpha_z & \text{(values)} \\
\text{t} & ::= \ldots \text{ref}_z^m \text{t} & \text{(terms)}
\end{align*}
\]
A reference mode $z$ may be a guarded reference $g$, a permissive reference $p$, or a monotonic reference $m$ (notice that
the $m_{\text{ref}}$ and ref constructors correspond to $m_{\text{ref}}$ and $\text{ref}_m$ respectively). Locations are now indexed by
a reference mode $o_z$, e.g. $o_m$ represent a monotonic reference. Reference terms are also indexed with a tag $z$ to
which kind of reference to create during reduction.

Fig. 15 highlights the changes to the typing rules of Fig. 3. Rule (Gref) is split into (Grefz) and (Grefp), the former
to type check guarded and monotonic references, and the latter for permissive references. Note that in $\lambda_{\text{ref}}^\text{pm}$,
references created at type $? \cdot \text{Ref}$ already behave like permissive references. This is because stored values of locations
typed Ref never change its type $?$ allowing for any arbitrary update. Therefore, to support permissive references in $\lambda_{\text{ref}}^\text{pm}$ we
simply add the (Grefp) type rule, which assigns type Ref $?$ to any permissive reference, as it may be used freely with any value of
any type.

6.2. Dynamic semantics

Analogous to $\lambda_{\text{ref}}$, the dynamic semantics of $\lambda_{\text{ref}}^\text{pm}$ are defined via elaboration to their intrinsic
representation. The extended language of intrinsic terms is called $\lambda_{\text{ref}}^\text{pm}$. The elaboration rules are identical to
Fig. 6, save for terms and types corresponding to references, which are now indexed by a reference mode.

Fig. 16 presents selected rules of the dynamic semantics of $\lambda_{\text{ref}}^\text{pm}$. We highlight in gray the key changes with respect
to Fig. 5. Following Siek et al. [39], we use in some rules evolving stores to reduce programs. An evolving store $ν$ is a
mapping between locations and terms, and intuitively it represents a store with pending evidence combinations. Compared
to Siek et al. [39], the fact that $λ_{\text{ref}}^\text{pm}$ does not have pairs allows us to simplify the definition of an evolving store as a store
with a pending location combination. Evolving stores are used to propagate ascriptions on monotonic locations
recursively. Configurations are pairs of an intrinsic term and an evolving store. Rule (r2) is factorized to perform the
ascription of the argument separately, because the argument can be a monotonic reference, and thus the ascription
needs to be propagated into the store. Rules (r4), (r5) and (r6) are adapted by adding a reference mode $z$ to the corresponding
rules and types constructors. Rule (r4), instead of reducing to a location, now reduces to an ascribed location. Although this
ascription may seem redundant, it is used later by other rules to push more precise evidence information in the store when working
with monotonic locations as shown in rule (r7). Rule (r6) is adapted for monotonic locations: instead of updating the location
cell to a new value, the cell is updated to a new ascribed value with evidence information of what was before in that
cell. By doing this, we make sure that the evidence of a cell can only gain precision. Rule (r7) reduces an evidence term
and a store, as the store may change during combination of evidence. There is also a new special case when the raw value $u$
is a monotonic location. In that case, the underlying value in the store is ascribed with information of evidence (G3) as
it may gain precision (or fail). Rule (r7) uses function $\text{ev}$ which returns the outermost evidence of a term, and is defined as
$\text{ev}(\text{et} :: ?G) = e$. As there may be cycles in the store, this new ascription can trigger the same reduction again in the
future. Following Siek et al. [39], to avoid infinite loops, this special case is considered only if we are gaining precision. But
instead of demanding that $G' \not\equiv \text{tref}(G_3)$, to prove the dynamic gradual guarantee we have to impose a stronger condition:

$$G' \not\equiv \text{tref}(G_3)$$

To illustrate rule (r7), consider the following step of reduction:

$$\langle \text{Ref}(? \rightarrow ?) \rangle \langle \text{Ref} (? \rightarrow ?) \rangle o_{\text{m}}^{2 \rightarrow ?} : \text{Ref} \langle ? \rightarrow ? \rangle \langle \lambda x : ?.x :: ? \rightarrow ? \rangle$$

$$\text{ev} \langle \text{Ref}(? \rightarrow ?) \rangle o_{\text{m}}^{2 \rightarrow ?} : \text{Ref} \langle ? \rightarrow ? \rangle \langle \lambda x : ?.x :: ? \rightarrow ? \rangle$$

The corresponding value in the store of the monotonic location is updated to a new term: its evidence will gain precision
when the evolving store is reduced to a store. Following Siek et al. [39], rules (Rv) and (RvErr) are added to reduce evolving
stores. Rule (RvErr) steps to an error when one of the terms in the store reduces to an error. Notice that, differently
from Siek et al. [39], propagation of ascriptions stops when a non-monotonic location is encountered. In the example,
the evolving store is reduced as follows:

$$\langle \text{Ref}(? \rightarrow ?) \rangle o_{\text{m}}^{2 \rightarrow ?} : \text{Ref} \langle ? \rightarrow ? \rangle \langle \lambda x : ?.x :: ? \rightarrow ? \rangle$$

$$\text{ev} \langle \text{Ref}(? \rightarrow ?) \rangle o_{\text{m}}^{2 \rightarrow ?} : \text{Ref} \langle ? \rightarrow ? \rangle \langle \lambda x : ?.x :: ? \rightarrow ? \rangle$$

Finally, contexts (RF) and (RFErr) are also adapted to include the store when combining evidences.
\[ ... \\
\mu ::= \cdot | \mu, \sigma^G \rightarrow v \\
v ::= \cdot | \mu, \sigma^G \rightarrow et :: G, \mu \\
\]

Notions of Reduction

\[ \text{CONFIG}_C = T[G] \times \text{EVOLVINGSTORE} \]

\[ ... \]

(2) \((G_1 \rightarrow G_2)(\lambda x^{G_1}.t)@G_1 \rightarrow G_2 ((G_2)'u) | \mu \longrightarrow \{(G_1')(\langle G_2' | \emptyset \rangle) :: G_2 | v \}
\]

where \((G_1')(\langle G_2' | \emptyset \rangle) :: G_1 \) and \(\mu \rightarrow v | v \)

\[ ... \]

(4) \[ \text{ref}^G_\sigma (G_1) u | \mu \longrightarrow (\text{Ref} G_2) \sigma^G_\mu :: \text{Ref} G_2 | \mu[\sigma^G_\mu] \rightarrow (G_1) u :: G_2 \]

where \(\sigma^G_\mu \notin \text{dom}(\mu)\)

(5) \[ \text{ref}^G_\sigma (G_1) u | \mu \longrightarrow (G_1) v :: G | \mu \text{ where } v = \mu[\sigma^G_\mu] \]

(6) \[ (\text{Ref} G_1) \sigma^G_\mu ::= G_1 (G_2) u | \mu \longrightarrow \begin{cases} \text{unit} | \mu[\sigma^G_\mu] \rightarrow t \} \\
\text{error} & \text{if } G_2 \cap G_3 \text{ is not defined} \\
\end{cases} \]

where \(\mu[\sigma^G_\mu] = (G') u' :: G, t = (G_1)(G_2) u :: G_3 \) and \(G_3\) is not defined

(7) \[ (G_2)((G_1) u :: G) | v \longrightarrow c \begin{cases} (G_3) u | v \text{ if } u \neq \sigma_m^G \}}} \\
(G_3) u | v[u \rightarrow (G_4) u(u) :: G_5] & \text{if } u = \sigma_m^{G_3}, G' \not\subseteq \text{ref}(G_3) \\
\text{error} & \text{if } G_3 \text{ or } G_4 \text{ are not defined} \\
\end{cases} \]

where \(\text{ev} \cdot (v(u)) = (G'), G_3 \subseteq G_1 \cap G_2, \) and \(G_4 \subseteq \text{ref}(G_3) \cap G' \)

Reduction

\[ \text{REF} \]

\[ ... \]

(Ref) \[ \text{et} | \mu \longrightarrow \text{et}' | v \]

(RFerr) \[ \text{et} | \mu \longrightarrow \text{error} \]

(Rv) \[ \text{v}(\sigma^G_\mu) = et :: G, \text{et} | v \longrightarrow \text{et}' | v' \]

(RvErr) \[ \text{v}(\sigma^G_\mu) = et :: G, \text{et} | v \longrightarrow \text{error} \]

Fig. 16. \(\lambda_{\text{pm}}^G\) : Dynamic semantics (selected rules).

6.3. Properties

\(\lambda_{\text{pm}}^G\) satisfies all the properties described in §4.6. As \text{ref} is not part of the source language \(\lambda_{\text{pm}}^G\), to make sense of the conservative extension of the static discipline properties (Propositions 11 and 58), any \text{ref} term must be converted to either a \text{ref} or a \text{ref} term (when considering fully precise programs, a guarded reference and a monotonic reference behave identically). Notice that converting a \text{ref} term to a \text{ref} term breaks Propositions 11, e.g. \(\vdash \text{ref} \rightarrow \text{Ref Int} \) but \(\vdash \text{ref} \rightarrow \text{Ref ?} \).

For monotonic references we state two properties that best describe their behavior.

Proposition 24 (Monotonicity of the evolving heap). If \(t^G | v \rightarrow t^G | v'\), then \(\forall \sigma_m^G \in \text{dom}(\mu), \text{ev}(v'(\sigma_m^G)) \subseteq \text{ev}(v(\sigma_m^G))\).

Monotonicity of the evolving heap means that, by taking a step, for every monotonic reference the outermost evidence of the corresponding term in the heap may only gain precision.

Proposition 25 (Monotonicity of the heap). If \(t^G | \mu \rightarrow \ast t^G | \mu'\), then \(\forall \sigma_m^G \in \text{dom}(\mu), \sigma_m^G \in \text{ev}(\mu', G') \)

\(\mu' = \varepsilon u :: G'\)

and \(\varepsilon' \subseteq \varepsilon\).
Monotonicity of the heap states that by reducing a term, starting from a heap and ending from a heap, the evidence of the underlying value of a monotonic evidence may only gain precision.

To the best of our knowledge, the dynamic gradual guarantee has never been proven for monotonic references. We prove this result here, which turns out to be challenging, and requires subtle considerations. We start by defining some operations on intrinsic terms: flat is a partial function that combines every evidence of nested ascriptions (which we call flattened evidence), and uval extracts a simple value from a nested ascription or a value:

\[
\begin{align*}
\text{flat}(\epsilon t^G :: G) &= \epsilon \circ^m \text{flat}(t^G) \\
\text{uval}(\epsilon t^G :: G) &= \text{uval}(t)
\end{align*}
\]

With these definitions we can now establish the precision relation between evolving stores.

\[
\forall o^G_1 \in \text{dom}(v_1), \exists o^G_2 \in \text{dom}(v_2) \text{ s.t.} \\
\nu \vdash o^G_1 \subseteq o^G_2 \quad G_1 \subseteq G_2 \\
\text{uval}(v_1(o^G_1)) \subseteq \text{uval}(v_2(o^G_2)) \\
\text{flat}(v_1(o^G_1)) \text{ is defined } \Rightarrow \text{flat}(v_1(o^G_1)) \subseteq \text{flat}(v_2(o^G_2))
\]

The difference with respect to precision of regular stores is that the values in the evolving stores of the same location must satisfy that the simple values and the combination of nested evidences are related by precision. Note that if the simple values are related, but consistent transitivity is not defined for \(\text{flat}(v_1(o^G_1))\), then the condition holds for that location. The intuition is that when defining the dynamic gradual guarantee, we need to relate evolving stores that may potentially fail in future steps. As we are only interested in the cases where the more precise term reduces, then the \(\text{flat}(v_1(o^G_1)) \subseteq \text{flat}(v_2(o^G_2))\) requirement makes sense only when \(\text{flat}(v_1(o^G_1))\) is defined. Note also that if both evolving stores are regular stores, then this definition coincides with the precision relation of stores defined for \(\lambda^m_{\text{REF}}\).

We introduce the notion of monotonic well-formedness, written \(\vdash_m (t, v)\), which states that, for every monotonic location \(\nu o^G_m\) found in either the term \(t\) or the evolving store \(v\), if the flattened evidence of the underlying value in the evolving heap \(\text{flat}(v(o^G_m))\) is defined, then \(\text{flat}(v(o^G_m)) \subseteq \text{uval}(\epsilon)\). Intuitively, if a monotonic reference gains precision, then its underlying value must also gain precision (sometimes in future steps). Crucially, reduction preserves monotonic well-formedness:

**Lemma 26 (Monotonic well-formedness preservation).** If \(\vdash_m (t, v)\) and \(t | v \rightarrow t' | v'\), then \(\vdash_m (t', v')\).

With the definitions of precision for evolving stores and monotonic well-formedness, we state the dynamic gradual guarantee as follows:

**Proposition 27 (Dynamic guarantee).** Suppose \(\vdash_m (t_1, v_1)\), \(t_1 \subseteq t_2\) and \(v_1 \subseteq v_2\). If \(t_1 | v_1 \rightarrow t'_1 | v'_1\) then \(t_2 | v_2 \rightarrow^* t'_2 | v'_2\), such that \(t'_1 \subseteq t'_2\) and \(v'_1 \subseteq v'_2\).

Note that the dynamic gradual guarantee is stated in an unusual way. First it requires that monotonic well-formedness holds for each related term. Second, as rule (r7) may reduce differently (subject to the type precision test, which may potentially endanger the dynamic gradual guarantee), both terms are not necessarily related after each step of reduction: one can reduce to an evolving store whereas the other does not. For this reason, we state the relation after taking zero or more steps for the less precise term and evolving store. The proof of the dynamic gradual guarantee differs from the proof of that property in \(\lambda^m_{\text{REF}}\) in the parts involving reduction of evolving stores. For these, the proof depends on the following two lemmas.

First, reducing the more precise evolving store preserves the precision relation:

**Lemma 28.** Let \(t_1 | v_1 \subseteq t_2 | v_2\). If \(t_1 | v_1 \rightarrow t_1 | v'_1\), then \(v'_1 \subseteq v_2\).

Second, we can safely reduce a less precise evolving store into a regular store:

**Lemma 29.** If \(\vdash_m (t_1, \mu_1)\) and \(t_1 | \mu_1 \subseteq t_2 | v_2\) then \(t_2 | v_2 \rightarrow^* t_2 | \mu_2\), such that \(\mu_1 \subseteq \mu_2\).

This last property requires that no infinite cycles occur when reducing an evolving store. This fact depends on the monotonic well-formedness of terms and stores:

**Lemma 30.** If \(\vdash_m (t_1 | v_1(o^G_m) \rightarrow \epsilon_1(\epsilon_2 u :: G) :: G)\), \(t | \mu_1 | v(o^G_m) \rightarrow \epsilon_1(\epsilon_2 u :: G) :: G\) \(\rightarrow^* t | v(o^G_m) \rightarrow \epsilon_3 u :: G\).

Then \(t | v(o^G_m) \rightarrow \epsilon_3 u :: G\) \(\rightarrow^* t | v(o^G_m) \rightarrow \epsilon_4(\epsilon_3 u :: G) :: G\).
7. Related work

We have already extensively discussed the four main approaches to gradual references found in the literature [35,24,39]. Vitousek et al. [46] present Reticulated Python, a tool for experimenting with gradual typing in Python 3 with support for references. They give two different dynamic semantics for casts: guarded semantics (with support for guarded references), and transient semantics. Instead of performing proxying of function or wrapping of runtime values, the transient semantics translate source programs by inserting type checks at all elimination forms, and at the entry and output of function definitions. These checks only test if values shallowly conform to a given type: only immediately-checkable information is considered. Greenman and Felleisen [22] compare guarded and transient semantics, and show that soundness for the transient semantics is a weaker notion that only guarantees preservation of the top-level constructor of the static type of an expression. For instance, consider \( h = (\lambda x : ? \rightarrow \text{Int}). f :: \text{Int} \rightarrow \text{Int}, \ g = (\lambda x : \text{Bool}, x), \) and term \( h \ g. \) Using guarded semantics and either space efficient coercions, threesome, or evidences based semantics, this program reduces to an error after reducing the body of \( h, \) because \( \text{Bool} \to \text{Bool} \) (the type of the returned value, \( g \)) is not consistent with the expected return type, \( \text{Int} \to \text{Int}. \) On the contrary, using the transient semantics, this program reduces successfully to \( g \) as its conforms with the expected top-level type constructor (a function). Of course, if we evaluate the program \( (h \ g) \) 1 then we will get an error right after applying \( g \) with 1 (thanks to the check at the entry of the body of \( g \)). Similarly, checks involving pair types only test if a given value is a pair. In Reticulated Python, for references (objects) the story is slightly different. Checks involving reference types (object types) recursively inspect a given value. For instance, consider the previous application of \( h \ g \) where now \( h = (\lambda x : \text{Ref} ? x :: \text{Ref} \text{Int}), \) and \( g = \text{ref true}. \) Using transient semantics this program reduces to an error as expected. After reducing the body of \( h, \) the resulting location content does not conform with the expected type \( \text{Ref} \text{Int}. \) But when we combine references and functions, the same issue as before manifests: if \( h = (\lambda x : \text{Ref} (?) x :: \text{Ref} \text{Int} \rightarrow \text{Int}), \) and \( g = \text{ref } (\lambda x : \text{Bool}, x), \) then \( h \ g \) reduces successfully to a location whose content conforms with the expected resulting type: a reference to a function. We believe that the discussion about transient semantics is orthogonal to references, and is extensively analyzed by Greenman and Felleisen [22], therefore we did not include it in the main body of this work.

Much prior work on gradual security typing also supports references [14,18,19,41], although imprecision is introduced exclusively via security labels, e.g. types like \( \text{Ref} \text{Int}? \) are supported but not types like \( \text{Ref} ?. \) Toro et al. [41] derive their gradual security language using AGT. The semantics of references corresponds to guarded references in which imprecision is limited to security labels.

Cimini and Siek [10] present the Gradualizer, a methodology and algorithm to systematically derive the static and cast insertion semantics of a gradual language from a static type system. They illustrate the application of this methodology to a language with references. We conjecture that the resulting gradual language treats gradual references as guarded or monotonic references, but it is hard to know precisely as the dynamic semantics were left as future work. They later extend the Gradualizer to also be able to derive the dynamic semantics of a gradual language [11]. The handling of auxiliary structures such as the heap is however not supported. The authors mention informally how the algorithm could be adapted to references, and conjecture that the resulting dynamic semantics correspond to the guarded semantics of Herman et al. [24]; however the precise formal treatment of this extension is left as future work.

There are many languages that integrate static and dynamic typing in some way, and support references: TypeScript [31], Flow [16], Hack [15], Dart [13], and Typed Clojure [6]. These languages adopt another approach called optional typing [7], which allows programmers to partially introduce type annotations to capture some errors statically, but do not perform any additional runtime checks at runtime beyond those that are performed for fully-untyped programs. This means that the runtime semantics are not sound with respect to its static type system.

Finally, there are many efforts related to gradually advancing typing disciplines, such as typestates [47,21], ownership types [34], annotated type systems [40], effects [4,5,43], refinement types [30,29], parametric polymorphism [2,27,42,32], and the security type systems discussed above, among others. Since the formulation of the refined criteria for gradually-typed languages [38], however, only refinement types [30], data type refinements [29] and a non-standard polymorphic language with explicit sealing [32] have been shown to fully respect such guarantees. Toro et al. [41,42] reveal some tensions between semantic type-based properties (noninterference, parametricity) and the dynamic gradual guarantee when following a type-driven approach to gradual typing, as that induced by AGT among others. The present work contributes to the gradualization of advanced typing disciplines by deriving a gradual language with references that satisfies the refined criteria. Additionally, this work presents the first formal statement and proof of the conservative extension of the dynamic semantics of the static language for a gradual language derived using AGT. Finally, the proof of the dynamic gradual guarantee for monotonic references is also novel.

8. Conclusion

We present \( \lambda_{\text{REF}}^\text{ext} \), a gradual language with support for mutable references. This language is derived step-by-step using the AGT methodology [20]. We compare the resulting language with other gradual languages with mutable references: monotonic references, permissive references, and guarded references. We find that \( \lambda_{\text{REF}}^\text{ext} \) treats references as guarded references, similar to how references are treated in the coercion calculus of Herman et al. [24] (HCC).

We formalize this relation by introducing HCC\(^*\), an adapted version of HCC with conditionals and binary operations. We prove semantic correspondence between \( \lambda_{\text{REF}}^\text{ext} \) (the intrinsic semantics of \( \lambda_{\text{REF}}^\text{ext} \)) and HCC\(^*\) for elaborated and translated
The main difference between both semantics has nothing to do with references, but with the order in which evidences/casts are combined. Under certain conditions (and contrary to HCC), a gradual language derived with AGT may accumulate an unbounded number of runtime checks. We describe the changes needed in the dynamic semantics of $\lambda_{\text{ref}}$ to recover space efficiency. We also present $\lambda_{\text{pm}}$, an extension of $\lambda_{\text{ref}}$ that supports both permissive and monotonic references. Finally, we formally prove that monotonic references satisfy the dynamic gradual guarantee, a non-trivial novel result that requires a careful consideration of updates to the store.

An interesting perspective for future work is to extend $\lambda_{\text{ref}}$ with nominal subtyping. We anticipate that a refined interpretation of evidences (such as pair of type intervals) might be needed to precisely capture type bounds at runtime, similarly to the security label intervals used in evidence by Toró et al. [41]. It would also be interesting to port the space-efficiency technique developed here to other AGT-derived gradual languages.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Gradualizing $\lambda_{\text{ref}}$, elaborating $\lambda_{\text{e}}$

In this section we present some proofs used in the gradualization of $\lambda_{\text{ref}}$ and elaboration of $\lambda_{\text{e}}$.

Proposition 31 (Precision, inductively). The following inductive definition of type precision is equivalent to Definition 2.

$$
\frac{B \subseteq B}{G_1 \subseteq G_1'} \quad \frac{G_1 \subseteq G_2 \quad G_2 \subseteq G_2'}{G_1 \rightarrow G_2 \subseteq G_1' \rightarrow G_2'} \quad \frac{G_1 \subseteq G_2}{\text{Ref } G_1 \subseteq \text{Ref } G_2} \quad \frac{G \subseteq ?}{
}
$$

Proof. We have to prove that $\gamma(G_1) \subseteq \gamma(G_2) \iff G_1 \subseteq G_2$, where $G_1 \subseteq G_2$ correspond to the inductive definition of type precision. We prove $\Rightarrow$ (the other direction is analogous). We proceed by induction on $\gamma(G_1) \subseteq \gamma(G_2)$.

Case ($\gamma(B) \subseteq \gamma(G_2)$). If $G_2 = B$ then we have to prove that $\gamma(B) \subseteq \gamma(B) \Rightarrow B \subseteq B$, which is trivial. If $G_2 = ?$ then we have to prove that $\gamma(B) \subseteq \text{Type} \Rightarrow B \subseteq ?$, which is also trivial.

Case ($\gamma(G_1) \rightarrow G_2) \subseteq \gamma(G_2)$). If $\gamma(G_1 \rightarrow G_2) = \{T_1 \rightarrow T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2)\}$, then $G_2$ is either $?$ and $\gamma(G_2) = \text{Type}$, but $G_1 \subseteq G_2$ and the result holds, or $G_2$ is $G_1 \rightarrow G_2$ such that $\gamma(G_2 \rightarrow G_2) = \{T_1 \rightarrow T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2)\}$ and $\{T_1 \rightarrow T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2)\} \subseteq \{T_1 \rightarrow T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2)\}$. For this to be true then $\gamma(G_1) \subseteq \gamma(G_1') \subseteq \gamma(G_2)$, and $\gamma(G_2) = \{T_1 \in \gamma(G_2)\}$ and $\gamma(G_2) = \{T_2 \in \gamma(G_2)\}$ we know that $G_1 \subseteq G_2$ and $G_2 \subseteq G_2$. Therefore $G_1 \rightarrow G_2 \subseteq G_2$ and the result holds.

Case ($\gamma(\text{Ref } G_1) \subseteq \gamma(G_2)$). We proceed similar to case function. □

Proposition 32 (Galois connection). $(\gamma, \alpha)$ is a Galois connection, i.e.:

a) (Soundness) for any non-empty set of static types $S = \{T\}$, we have $S \subseteq \gamma(\alpha(S))$

b) (Optimality) for any gradual type $G$, we have $\alpha(\gamma(G)) \subseteq G$.

Proof. We first proceed to prove a) by induction on the structure of the non-empty set $S$.

Case ($\{B\}$). Then $\alpha(\{B\}) = B$. But $\gamma(B) = \{B\}$ and the result holds.

Case ($\{\{T_1 \rightarrow T_2\}\}$). Then $\alpha(\{\{T_1 \rightarrow T_2\}\}) = \alpha(\{\{T_1\}\}) \rightarrow \alpha(\{\{T_2\}\})$. By definition of $\gamma$, $\gamma(\alpha(\{\{T_1\}\}) \rightarrow \alpha(\{\{T_2\}\})) = \{T_1 \rightarrow T_2 \mid T_1 \in \gamma(\alpha(\{\{T_1\}\})) \land T_2 \in \gamma(\alpha(\{\{T_2\}\}))\}$. By induction hypotheses, $\{\{T_1\}\} \subseteq \gamma(\alpha(\{\{T_1\}\}))$ and $\{\{T_2\}\} \subseteq \gamma(\alpha(\{\{T_2\}\}))$, therefore $\{\{T_1 \rightarrow T_2\}\} \subseteq \{T_1 \rightarrow T_2 \mid T_1 \in \gamma(\alpha(\{\{T_1\}\})) \land T_2 \in \gamma(\alpha(\{\{T_2\}\}))\}$ and the result holds.

Case ($\{\text{Ref } T\}$). Then $\alpha(\{\text{Ref } T\}) = \text{Ref } \alpha(\{T\})$. But by definition of $\gamma$, $\gamma(\text{Ref } \alpha(\{T\})) = \{\text{Ref } T \mid T \in \gamma(\alpha(\{T\}))\}$. By induction hypothesis, $\{\{T\}\} \subseteq \gamma(\alpha(\{T\}))$, therefore $\{\text{Ref } T\} = \{\text{Ref } T \mid T \in \gamma(\alpha(\{T\}))\}$ and the result holds.

Case ($\{T\}$ homogeneous). Then $\alpha(\{T\}) = ?$ and therefore $\gamma(\alpha(\{T\})) = \text{Type}$, but $\{T\} \subseteq \text{Type}$ and the result holds.
Now let us proceed to prove b) by induction on gradual type \( G \).

**Case \((B)\).** Trivial because \( \gamma(B) = \{B\} \), and \( \alpha(\{B\}) = B \).

**Case \((G_1 \rightarrow G_2)\).** We have to prove that \( \alpha(\gamma(G_1 \rightarrow G_2)) \subseteq G_1 \rightarrow G_2 \), which is equivalent to prove that \( \gamma(\alpha(\overline{T})) \subseteq \overline{T} \), where \( \overline{T} = \gamma(G_1 \rightarrow G_2) = \{T_1 \rightarrow T_2 | T_1 \in \gamma(G_1), T_2 \in \gamma(G_2)\} \). Then \( \overline{T} \) has the form \( \{T_{11} \rightarrow T_{12}\} \), such that \( \forall i, T_{ii} \in \gamma(G_i) \) and \( T_{12} \in \gamma(G_2) \). Also note that \( \{T_{11}\} = \gamma(G_1) \) and \( \{T_{12}\} = \gamma(G_2) \). But by definition of \( \alpha \), \( \alpha(\{T_{11}\}) \rightarrow \alpha(\{T_{12}\}) \) and therefore \( \gamma(\alpha(\{T_{11}\})) \rightarrow \gamma(\alpha(\{T_{12}\})) \) = \( \{T_1 \rightarrow T_2 | T_1 \in \gamma(\alpha(\{T_{11}\})), T_2 \in \gamma(\alpha(\{T_{12}\}))\} \). But by induction hypotheses \( \gamma(\alpha(\{T_{11}\})) \subseteq \gamma(G_1) \) and \( \gamma(\alpha(\{T_{12}\})) \subseteq \gamma(G_2) \) and the result holds.

**Case \((\text{Ref} G)\).** We have to prove that \( \alpha(\gamma(\text{Ref} G)) \subseteq \text{Ref} G \), which is equivalent to prove that \( \gamma(\alpha(\overline{T})) \subseteq \overline{T} \), where \( \overline{T} = \gamma(\text{Ref} G) = \{\text{Ref} T | T \in \gamma(G)\} \). Then \( \overline{T} \) has the form \( \{\text{Ref} T_i\} \), such that \( \forall i, T_i \in \gamma(G) \). Also note that \( \{T_i\} = \gamma(G) \). But by definition of \( \alpha \), \( \alpha(\{\text{Ref} T_i\}) = \text{Ref} \alpha(\{T_i\}) \) and therefore \( \gamma(\text{Ref} \alpha(\{T_i\})) = \{\text{Ref} T | T \in \gamma(\alpha(\{T_i\}))\} \). But by induction hypothesis \( \gamma(\alpha(\{T_i\})) \subseteq \gamma(G) \) and the result holds.

**Case \((?)\).** Then we have to prove that \( \gamma(\alpha(?)) \subseteq \gamma(?) \) is True, but this is always true and the result holds immediately. \(\square\)

**Proposition 33.** \(\text{equate}(G_1, G_2) = G_1 \cap G_2 \).

The meet operator is defined as \( G_1 \cap G_2 = \alpha(\gamma(G_1) \cap \gamma(G_2)) \), and inductively as:

\[
B \cap B = B \quad G_1 \cap G_2 = G_2 \cap G_1 \quad G \cap ? = ? \cap G = G \quad (G_1 \rightarrow G_2) \cap (G_{21} \rightarrow G_{22}) = (G_{11} \rightarrow G_{12}) \rightarrow (G_{12} \cap G_{22})
\]

\[
\text{Ref} G_1 \cap \text{Ref} G_2 = \text{Ref} G_1 \cap G_2 \quad G_1 \cap G_2 \text{ is undefined otherwise}
\]

**Proof.** By induction on \( G_1 \) and \( G_2 \). \(\square\)

**Proposition 34.** Let \( P_1(T_1, T_2) \triangleq T_1 = \text{dom}(T_2) \). Then \( P_1(G_1, G_2) \iff G_1 \sim \text{dom}(G_2) \).

**Proof.** The \( \Rightarrow \) direction by induction on \( P_1(G_1, G_2) \), and the \( \Leftarrow \) direction by induction on \( G_1 \sim \text{dom}(G_2) \). \(\square\)

**Proposition 35.** Let \( P_2(T_1, T_2) \triangleq T_2 = \text{tref}(T_2) \). Then \( P_2(G_1, G_2) \iff G_1 \sim \text{tref}(G_2) \).

**Proof.** The \( \Rightarrow \) direction by induction on \( P_2(G_1, G_2) \), and the \( \Leftarrow \) direction by induction on \( G_1 \sim \text{tref}(G_2) \). \(\square\)

**Proposition 36.** If \( G_1 \sim G_2 \), then \( \gamma_G(G_1, G_2) = (G_1 \cap G_2, G_1 \cap G_2) \).

**Proof.** Notice that in this setting \( \gamma_G(G_1, G_2) = \alpha((\{T\} | T \in \gamma(G_1), T \in \gamma(G_2))) = \alpha((\{T\} | T \in \gamma(G_1) \cap \gamma(G_2))) = (\alpha(\{T\} | T \in \gamma(G_1) \cap \gamma(G_2))) = (G_1 \cap G_2) \). \(\square\)

**Lemma 37.** \( (G_1) \circ \gamma \ (G_2) = (G_1 \cap G_2, G_1 \cap G_2) \).

**Proof.** Similar to Proposition 7. \(\square\)

**Proposition 38 (Elaboration preserves typing).** If \( \Gamma; \Sigma \vdash t : G \) and \( \Gamma; \Sigma \vdash t \sim_n t^G : G \), then \( t^G \in T[G] \).

**Proof.** Straightforward induction on \( \Gamma; \Sigma \vdash t : G \). We only present one case as the other are analogous.

**Case \((\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit})\).** We know by \((\text{Gasgn})\) that

\[
\begin{align*}
(\text{Gasgn}) & \quad \Gamma; \Sigma \vdash t_1 : G_1 \quad \Gamma; \Sigma \vdash t_2 : G_2 \quad G_2 \sim \text{tref}(G_1) \\
\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit}
\end{align*}
\]

Then by \((\text{TRasgn})\):

\[
\begin{align*}
\text{(TRasgn)} & \quad \Gamma; \Sigma \vdash t_1 \sim_n t^{G_1} : G_1 \quad \Gamma; \Sigma \vdash t_2 \sim_n t^{G_2} : G_2 \\
& \quad G_3 = \text{tref}(G_1) \quad \epsilon_1 = \gamma_G(G_1, \text{Ref} G_3) \quad \epsilon_2 = \gamma_G(G_2, G_3) \\
& \quad \Gamma; \Sigma \vdash t_1 := t_2 \sim_n \epsilon_1 t^{G_1} := \epsilon_2 t^{G_2} : \text{Unit}
\end{align*}
\]
By induction hypothesis on $\Gamma; \Sigma \vdash t_1 : G_1$, if $\Gamma; \Sigma \vdash t_1 \sim_n t^{G_1} : G_1$ then $t^{G_1} \in T[G_1]$. Similarly by induction hypothesis on $\Gamma; \Sigma \vdash t_2 : G_2$, if $\Gamma; \Sigma \vdash t_2 \sim_n t^{G_2} : G_2$ then $t^{G_2} \in T[G_2]$. Also by definition of the interior function, $\varepsilon_1 \vdash G_1 \sim \text{Ref } G_3$ and $\varepsilon_2 \vdash G_2 \sim G_3$. Then by (Lg asgn):

$$\frac{t^{G_1} \in T[G_1]}{\varepsilon_1 \vdash G_1 \sim \text{Ref } G_3}$$

and the result holds. □

Appendix B. Type safety

In this section we present the proof of type safety for $\lambda_{\text{IV}}$.

**Lemma 39** (Canonical forms). Consider a value $v \in T[G]$. Then either $v = u$, or $v = \varepsilon u :: G$ with $u \in T[G']$ and $\varepsilon \vdash G' \sim G$. Furthermore:

1. If $G = \text{Bool}$ then either $v = b$ or $v = \varepsilon b :: \text{Bool}$ with $b \in T[\text{Bool}]$.
2. If $G = \text{Int}$ then either $v = n$ or $v = \varepsilon n :: \text{Int}$ with $n \in T[\text{Int}]$.
3. If $G = G_1 \rightarrow G_2$ then either $v = (\lambda x^{G_1}. t^{G_2})$ with $t^{G_2} \in T[G_2]$ or $v = \varepsilon (\lambda x^{G_1}. t^{G_2}) :: G_1 \rightarrow G_2$ with $t^{G_2} \in T[G_2]$ and $\varepsilon \vdash G_1 \rightarrow G_2' \sim G_1 \rightarrow G_2$.
4. If $G = \text{Ref } G'$ then either $v = o^{G'}$ or $v = \varepsilon o^{G'}$ with $o^{G'} \in T[\text{Ref } G']$ and $\varepsilon \vdash \text{Ref } G' \rightarrow \text{Ref } G$.

**Proof.** By direct inspection of the formation rules of gradual intrinsic terms (Fig. 4). □

**Lemma 40** (Substitution). If $t^G \in T[G]$ and $v \in T[G_1]$, then $[v/x^{G_1}]t^G \in T[G]$.

**Proof.** By induction on the derivation of $t^G$. □

**Proposition 41** ($\rightarrow$ is well defined). If $\varepsilon \vdash \mu'$ and $t^G \rightarrow r$, then $r \in \text{CONFIG}_G \cup \{ \text{error} \}$, and if $r = t^G \mid \mu'$, then also $t^G \vdash \mu'$ and $\text{dom}(\mu) \subseteq \text{dom}(\mu')$.

**Proof.** By induction on the structure of a derivation of $t^G \rightarrow r$, considering the last rule used in the derivation.

**Case** ($r_1$). Then $t^G = t^B_3 = \varepsilon_1 b_1 \uplus \varepsilon_2 b_2$. Then

$$\frac{b_1 \in T[B_1]}{\varepsilon_1 \vdash B_1 \sim B_1} \frac{b_2 \in T[B_2]}{\varepsilon_2 \vdash B_2 \sim B_2} \frac{\text{ty}(\oplus) = B_1 x B_2 \rightarrow B_3}{\varepsilon_1 b_1 \uplus \varepsilon_2 b_2 \in T[B_3]}$$

Therefore

$$\varepsilon_1 b_1 \uplus \varepsilon_2 b_2 \mid \mu \rightarrow b_3 \mid \mu \text{ where } b_3 = b_1 \llbracket \oplus \rrbracket b_2$$

But $b_3 \in T[B_3]$ and the result holds.

**Case** ($r_2$). Then $t^G = \varepsilon_1 (\lambda x^{G_11}. t^{G_12}_1) \circ^{G_1 \rightarrow G_2} (\varepsilon_2 u)$ and $G = G_2$. Then

$$\frac{D_1}{t^{G_12}_1 \in T[G_{12}]} \frac{D_2}{u \in T[G_2]} \frac{\text{ty}(\circ) = G_{12} \rightarrow G_2}{\varepsilon_1 \vdash G_{11} \sim G_{12} \rightarrow G_1 \sim G_2}$$

If $\varepsilon' = (\varepsilon_2 \circ \text{idom}(\varepsilon_1))$ is not defined, then $t^G \rightarrow \text{error}$, and then the result hold immediately. Suppose that consistent transitivity does hold, then

$$\varepsilon_1 (\lambda x^{G_11}. t^{G_12}_1) \circ^{G_1 \rightarrow G_2} (\varepsilon_2 u \mid \mu \rightarrow \text{icod}(\varepsilon_1)((\varepsilon' u :: G_{11})/x^{G_11}[t]) :: G_2 \mid \mu$$

As $\varepsilon_2 \vdash G_2 \sim G_1$ and by inversion lemma $\text{idom}(\varepsilon_1) \vdash G_{11} \sim G_{11}$, then $\varepsilon' \vdash G_{11} \sim G_{11}$. Therefore $\varepsilon' u :: G_{11} \in T[G_{11}]$, and by Lemma 40, $t^{G_12} = [(\varepsilon' u :: G_{11})/x^{G_11}[t^{G_12}] \in T[G_{12}]$. 


and the result holds.

**Case** \((r3 \text{ -- true})\). Then \(t^G = \text{if } ε \text{ then } \epsilon_1 \text{ else } \epsilon_2\) and \(G = G_2 \cap G_3\) and

\[
\begin{align*}
G &\in T[G_2] \\
\epsilon_1 &\vdash G_1 \sim \text{Bool} \quad G = (G_2 \cap G_3) \\
\epsilon_2 &\vdash G_2 \sim G \\
\epsilon_3 &\vdash G_3 \sim G
\end{align*}
\]

\[
(\text{IRef}) \quad \begin{array}{c}
t^G \in T[G_2] \\
\epsilon \vdash G_2 \sim G_2 \cap G_3 \\
\epsilon_2 \vdash G_2 \cap G_3 \in T[G_2 \cap G_3]
\end{array}
\]

Therefore

\[
\begin{array}{c}
\text{if } ε \text{ then } \epsilon_2 \text{ else } \epsilon_3 \in T[G]
\end{array}
\]

But

\[
\begin{array}{c}
\text{if } ε \text{ then } \epsilon_2 \text{ else } \epsilon_3 \in T[G]
\end{array}
\]

and the result holds.

**Case** \((r3 \text{ -- false})\). Analogous to case (if-true).

**Case** \((r4)\). Then \(t^G = \text{ref}^G \epsilon u\). Then

\[
\begin{array}{c}
\epsilon \vdash G_1 \sim G_2 \\
\epsilon_2 \vdash G_2 \sim G_2 \cap G_3
\end{array}
\]

Then

\[
\begin{array}{c}
\text{ref}^G \epsilon u \mid \mu \rightarrow o^G \mid \mu [o^G \mapsto \epsilon u :: G_2]
\end{array}
\]

where \(o \notin \text{dom(μ)}\). But as \(\epsilon u :: G_2 \in T[G_2]\), then \(o^G \vdash \mu [o^G \mapsto \epsilon u :: G_2]\). Also \(o^G \in T[\text{Ref} G_2]\) and the result holds.

**Case** \((r5)\). Then \(t^G = \epsilon^G (\epsilon o^G_1)\). Then

\[
\begin{array}{c}
\epsilon \vdash \text{Ref} G_1 \sim \text{Ref} G_2 \\
\epsilon^G (\epsilon o^G_1) \in T[G_2]
\end{array}
\]

Then

\[
\begin{array}{c}
\epsilon^G (\epsilon o^G_1) \mid \mu \rightarrow \text{iref}(\epsilon) v :: G_2 \mid \mu
\end{array}
\]

where \(v = \mu (o^G_1)\). As \(\mu\) is well formed, then \(v \in T[G_1]\). Then by inversion lemma \(\text{iref}(\epsilon) \vdash G_1 \sim G_2\), therefore \(\text{iref}(\epsilon) v :: G_2 \in T[G_2]\) and the result holds.

**Case** \((r6)\). Then \(t^G = \epsilon_1 o^G_1 := \epsilon_2 u\). Then

\[
\begin{array}{c}
o^G_1 \in T[\text{Ref} G_1] \\
u \in T[G_2] \\
\epsilon_1 \vdash \text{Ref} G_1 \sim \text{Ref} G_3 \\
\epsilon_2 \vdash G_2 \sim G_3
\end{array}
\]

\[
(\text{IAsgn}) \quad \begin{array}{c}
o^G_1 \in T[\text{Ref} G_1] \\
\epsilon_1 \vdash \text{Ref} G_1 \sim \text{Ref} G_3 \\
\epsilon_2 \vdash G_2 \sim G_3 \\
\epsilon_1 o^G_1 := \epsilon_3 \epsilon_2 u \in T[\text{Unit}]
\end{array}
\]

If \(\epsilon' = (\epsilon_2 \cap \text{iref}(\epsilon_1))\) is not defined, then \(t^G \rightarrow \text{error}\), and then the result hold immediately. Suppose that consistent transitivity does hold, then

\[
\begin{array}{c}
\epsilon_1 o^G_1 := \epsilon_3 \epsilon_2 u \mid \mu \rightarrow \text{unit} \mid \mu [o^G_1 \mapsto \epsilon' u :: G_1]
\end{array}
\]

As \(\epsilon_2 \vdash G_2 \sim G_3\) and by inversion lemma \(\text{iref}(\epsilon_1) \vdash G_1 \sim G_3\), and as evidence is symmetrical \(\text{iref}(\epsilon_1) \vdash G_3 \sim G_1\), then \(\epsilon' \vdash G_2 \sim G_1\). Therefore \(\epsilon' u :: G_1 \in T[G_1]\), and therefore \(\text{unit} \vdash \mu [o^G_1 \mapsto \epsilon' u :: G_1]\). Also

\[
\begin{array}{c}
\theta(\text{unit}) = \text{Unit} \\
\text{unit} \in T[\text{Unit}]
\end{array}
\]

and the result holds. □
Proposition 42 (** is well defined). If \( t^G \vdash \mu \) and \( t^G \mid \mu \longrightarrow r \), then \( r \in \text{CONFIG}_G \cup \{\text{error}\} \), and if \( r = t^G \mid \mu' \), then also \( t^G \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

**Proof.** By induction on the structure of a derivation of \( t^G \longrightarrow r \).

Case (\( \longrightarrow \)). \( t^G \mid \mu \longrightarrow r \). By well-definedness of \( \longrightarrow \) (Proposition 41), \( r \in \text{CONFIG}_G \cup \{\text{error}\} \), and if \( r = t^G \mid \mu' \), then also \( t^G \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

Case (\( R.E \)). \( t^G = E[t^G_1] \), \( E[t^G_1] \in T[G] \), \( t^G_1 \mid \mu \longrightarrow t^G_2 \mid \mu' \), \( t^G_2 \in T[G'] \), and \( E : T[G'] \rightarrow T[G] \). By induction hypothesis, \( t^G_2 \in T[G'] \), so \( E[t^G_2] \in T[G] \).

By induction hypothesis we also know that \( t^G \vdash \mu' \).

If \( \text{freeLocs}(t^G_1) \subseteq \mu', \text{freeLocs}(f(t^G_1)) \subseteq \mu \), and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \), then it is easy to see that \( \text{freeLocs}(f(t^G_2)) \subseteq \mu' \), and therefore conclude that \( f(t^G_2) \vdash \mu' \).

Case (\( R.Err, R.Ferr \)). \( r = \text{error} \).

Case (\( R.F \)). Let \( \text{EvTERM}_{G_2} \) be notation for the family of evidence terms \( e_1 t^G_1 \) such that \( e \vdash G_1 \sim G_2 \). Then \( t^G = F[et] \), \( F[et] \in T[G] \), and \( F : \text{EvTERM}_{G_2} \rightarrow T[G] \), and \( et \longrightarrow e' \). Then there exists \( G_v, G_x \) such that \( et = e_1 t^G_1 \) and \( e_2 \vdash G_v \sim G_x \). Also, \( t_e = e_2 u : G_e \), with \( u \in T[G_e] \) and \( e_y \vdash G \sim G_e \).

We know that \( e_{\epsilon_2} = e_y o^w e_x \) is defined, and \( et = e_1 t_e \longrightarrow e' \epsilon_2 u = e' \). By definition of \( o^w \) we have \( e_{\epsilon_2} \vdash G_v \sim G_x \), so \( F[et'] \in T[G] \).

As \( \text{freeLocs}(et) = \text{freeLocs}(et') \) and \( \mu' = \mu \), then it is easy to conclude that \( F[et'] \vdash \mu \). □

Now we can establish type safety: programs do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

Proposition 43 (Type Safety). If \( t^G \in T[G] \) then one of the following is true:

1. \( t^G \) is a value \( v \);
2. \( t^G \vdash \mu \) then \( t^G \mid \mu \longrightarrow t^G \mid \mu' \) for some term \( t^G \in T[G] \) and some \( \mu' \) such that \( t^G \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \);
3. \( t^G \mid \mu \longrightarrow \text{error} \).

**Proof.** By induction on the structure of \( t^G \). We only present some cases as all proceed the same way.

Case (\( IGc, IGx, IG\lambda, IG0 \)). \( t^G \) is a value.

Case (\( IG :: \)). \( t^G = e_1 t^G_1 :: G_2 \), and

\[
\begin{array}{c}
(l::) \quad t^G_1 \in T[G_1] \\
\quad e_1 \vdash G_1 \sim G_2 \\
\end{array}
\]

By induction hypothesis on \( t^G_1 \), one of the following holds:

1. \( t^G_1 \) is a simple value \( u \), in which case \( t^G \) is also a value.
2. \( t^G_1 \) is an ascribed value \( v \), then the result holds by Proposition 42 and either (RF), or (RFerr).
3. \( t^G \mid \mu \longrightarrow r_1 \) for some \( r_1 \in \text{CONFIG}_G \cup \{\text{error}\} \). Hence \( t^G \mid \mu \longrightarrow r \) for some \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Proposition 42 and either (RF), or (RFerr).

Case (\( Igf \)). \( t^G = if \ e_1 t^G_1 \ then \ e_2 t^G_2 \ else \ e_3 t^G_1 \) and

\[
\begin{array}{c}
(lGf) \quad t^G_1 \in T[G_1] \\
\quad e_1 \vdash G_1 \sim \text{Bool} \\
\quad G = (G_2 \cap G_3) \\
\quad t^G_2 \in T[G_2] \\
\quad e_2 \vdash G_2 \sim G \\
\quad t^G_3 \in T[G_3] \\
\quad e_3 \vdash G_3 \sim G \\
\end{array}
\]

By induction hypothesis on \( t^G_1 \), one of the following holds:

1. \( t^G_1 \) is a simple value \( u \), then by (\( \longrightarrow \)), \( t^G \mid \mu \longrightarrow r \) and \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Proposition 42.
2. $t^{G_1}$ is an ascribed value $v$, then, $e_1 t^{G_1} \rightarrow_{e} r'$ for some $r' \in \text{EvTerm}_{B\text{odl}} \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).

3. $t^{G_1} | \mu \mapsto r_1$ for some $r_1 \in T[G_1] \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).

**Case (IGapp).** $t^{G} = (e_1 t^{G_1}) \odot^{G_1 \rightarrow G_1} (e_2 t^{G_2})$

By induction hypothesis on $t^{G_1}$, one of the following holds:

1. $t^{G_1}$ is a value $(\lambda x^{G_1}, t^{G_1}_{12})$ (by canonical forms Lemma 39), posing $G_1 = G^{'}_{11} \rightarrow G^{'}_{12}$.

   Then by induction hypothesis on $t^{G_2}$, one of the following holds:
   (a) $t^{G_2}$ is a simple value $u$, then by (R$\rightarrow_{e}$), $t^{G} | \mu \mapsto r$ and $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42.
   (b) $t^{G_2}$ is an ascribed value $v$, then, $e_2 t^{G_2} \rightarrow_{e} r'$ for some $r' \in \text{EvTerm}_{G_{11}} \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).
   (c) $t^{G_2} | \mu \mapsto r_2$ for some $r_2 \in \text{Config}_{G_{2}} \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).

2. $t^{G_1}$ is an ascribed value $v$, then, $e_1 t^{G_1} \rightarrow_{e} r'$ for some $r' \in \text{EvTerm}_{G_{11}} \rightarrow G_{12} \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).

3. $t^{G_1} | \mu \mapsto r_1$ for some $r_1 \in \text{Config}_{G_{1}} \cup \{\text{error}\}$. Hence $t^{G} | \mu \mapsto r$ for some $r \in \text{Config}_{G} \cup \{\text{error}\}$ by Proposition 42 and either (RF), or (RFerr).

**Case.** Other cases are similar to the app case. □

**Appendix C. Gradual guarantee.**

In this section we present the proof of the conservative extensions of the static discipline and the static and the dynamic gradual guarantee.

**C.1. Conservative extensions of the static discipline**

**Proposition 44 (Equivalence for fully-annotated terms (statics)).** For any $t \in \text{Term}$, $t \vdash_{S} t : T$ if and only if $t : T$

**Proof.** By induction over the typing derivations. The proof is trivial because static types are given singleton meanings via concretization. □

The equivalence for the dynamics of fully-annotated terms is defined in terms of a logical relation between terms of the static language $\lambda_{\text{REF}}$, and $\lambda_{\text{REF}}^{c}$. The logical relation is presented in Fig. C.17.

**Definition 9 (Related substitutions).** We say that tuples $(\sigma_1, \mu_1)$ and $(\sigma_2, \mu_2)$ are related under $\Gamma$ and $\Sigma$, notation $\Gamma ; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$ if $\sigma_1 = \Gamma$, $\sigma_2 = \Gamma$, $\Sigma \vdash \mu_1$, $\Sigma \vdash \mu_2$, $\mu_1 \approx \mu_2$, and
\[ \forall x \in \text{dom}(\Gamma), \langle \sigma_1(x), \mu_1 \rangle \approx \langle \sigma_2(x), \mu_2 \rangle : \Gamma(x) \]

**Definition 10 (Semantic equivalence).**

\[ \Gamma; \Sigma \vdash t \approx t^T : T \iff \forall \sigma_1, \sigma_2, \mu_1, \mu_1, \Sigma \vdash \mu_1, \Sigma \vdash \mu_2, \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle , \]

we have \( \langle \sigma_1(t), \mu_1 \rangle \approx \langle \sigma_2(t^T), \mu_2 \rangle : T \)

**Proposition 45 (Fundamental property).** For any \( t \in \text{TERM}, \Gamma; \Sigma \vdash t : T, \Gamma; \Sigma \vdash t \sim_n t^T : T, \text{then } \Gamma; \Sigma \vdash t \approx t^T : T \).

**Proof.** By induction on the type derivation of \( t \).

**Case (Tx).** Then \( t = x \) and therefore

\[ (Tx) \quad x : T \in \Gamma \]

\[ \Gamma; \Sigma \vdash x : T \]

Then we have to prove that \( x \approx x^T : T \). But the result follows directly by Proposition 48.

**Case (Tb).** Then \( t = b \) and therefore

\[ (Tc) \quad \theta(b) = B \quad \Gamma; \Sigma \vdash b : B \]

and \( t^T = b \). The result follows by Proposition 47.

**Case (Tapp).** Then \( t = t_1 \oplus t_2 \) and \( T = T_2 \) where

\[ (Tapp) \quad \Gamma; \Sigma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma; \Sigma \vdash t_2 : T_1 \quad T_2 = \text{dom}(T_1) \]

and

\[ \Gamma; \Sigma \vdash t_1 \sim_n t^{G_1} : T_1 \rightarrow T_2 \quad \Gamma; \Sigma \vdash t_2 \sim_n t^{G_2} : T_1 \]

\[ \theta_1 = (T_1 \rightarrow T_2) = \theta_2 \quad \theta_1 \oplus \theta_2 \rightarrow \text{dom}(T_1 \rightarrow T_2) \rightarrow \text{cod}(T_1 \rightarrow T_2) \]

\[ (Tapp) \quad \Gamma; \Sigma \vdash t_1 \oplus t_2 : T_1 \rightarrow T_2 \]

We have to prove that \( \Gamma; \Sigma \vdash t_1, t_2 \approx \epsilon_1 t_1^{T_1} \oplus t_2 \rightarrow T_1 \oplus t_1^{T_1} \oplus t_2 \rightarrow T_2 \). By induction hypotheses we know that \( \Gamma; \Sigma \vdash t_1 \approx t_1^{T_1} : T_1 \rightarrow T_2 \) and that \( \Gamma; \Sigma \vdash t_2 \approx t_1^{T_1} : T_1 \). The result follows directly by Proposition 49.

**Case (Top).** Then \( t = t_1 @ t_2 \) and \( T = B_3 \), where

\[ (Top) \quad \Gamma; \Sigma \vdash t_1 : B_1 \quad \Gamma; \Sigma \vdash t_2 : B_2 \]

\[ \text{ty}(@) = B_1 \times B_2 

\]

\[ \Gamma; \Sigma \vdash t_1 @ t_2 : B_3 \]

and

\[ \Gamma; \Sigma \vdash t_1 \sim_n t^{B_1} : B_1 \quad \Gamma; \Sigma \vdash t_2 \sim_n t^{B_2} : B_2 \quad \text{ty}(@) = B_1 \times B_2 

\]

\[ \epsilon_1 = (B_1) = \theta_2(B_1, B_1) \quad \epsilon_2 = (B_2) = \theta_2(B_2, B_2) \]

\[ (Top) \quad \Gamma; \Sigma \vdash t_1 @ t_2 : t^{B_1} \oplus t^{B_2} : B_3 \]

We have to prove that \( \Gamma; \Sigma \vdash t_1, t_2 \approx \epsilon_1 t_1^{B_1} \oplus \epsilon_2 t_1^{B_2} : T_2 \). By induction hypotheses we know that \( \Gamma; \Sigma \vdash t_1 \approx t_1^{B_1} : B_1 \) and that \( \Gamma; \Sigma \vdash t_2 \approx t_1^{B_2} : B_2 \). The result follows directly by Proposition 50.

**Case (Tif).** Then \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \), where

\[ (Tif) \quad \Gamma; \Sigma \vdash t_1 : \text{Bool} \quad \Gamma; \Sigma \vdash t_2 : T \quad \Gamma; \Sigma \vdash t_3 : T \]

and

\[ \Gamma; \Sigma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \]

\[ \theta_1 = (\text{Bool}) = \theta_2 \quad \theta_1 \oplus \theta_2 \rightarrow \text{dom}(T, T) \rightarrow \text{dom}(T, T) \]

\[ \Gamma; \Sigma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : t^{\theta_1^{B_0}} \oplus \epsilon_1 t^{T_1} \oplus \epsilon_2 t^{T_2} : T \]


We have to prove that $\Gamma; \Sigma \vdash t_1$ then $t_2$ else $t_3 \approx t^\text{ref} \downarrow e_1 t^\text{ref} \downarrow e_2 t^\text{ref} : T$. By induction hypotheses we know that $\Gamma; \Sigma \vdash t_1 \approx t^\text{ref} \downarrow e_1 t^\text{ref} \downarrow e_2 t^\text{ref} : T$ and that $\Gamma; \Sigma \vdash t_2 \approx t^\text{ref} \downarrow e_1 t^\text{ref} \downarrow e_2 t^\text{ref} : T$, and $\Gamma; \Sigma \vdash t_3 \approx t^\text{ref} \downarrow e_1 t^\text{ref} \downarrow e_2 t^\text{ref} : T$. The result follows directly by Proposition 51.

**Case** ($T\lambda$). Then $t = (\lambda x : T_1.t')$ and $T = T_1 \rightarrow T_2$, and therefore

$$\frac{\Gamma, x : T_1; \Sigma \vdash s \vdash t' : T_2}{\Gamma; \Sigma \vdash (\lambda x : T_1.t') : T_1 \rightarrow T_2}$$

and

$$\frac{\Gamma, x : T_1 \vdash t' \sim^n t : T_2}{\Gamma; \Sigma \vdash (\lambda x : T_1.t') : T_1 \rightarrow T_2}$$

Then we have to prove that $\Gamma; \Sigma \vdash (\lambda x : T_1.t') \approx (\lambda^\text{ref} T_1.t^\text{ref} : T_1 \rightarrow T_2$. By induction hypothesis we already know that $\Gamma, x : T_1; \Sigma \vdash t' \approx t^\text{ref} : T_2$. But the result follows directly by Proposition 52.

**Case** ($T::$). Then $t = t' :: T$

$$\frac{\Gamma; \Sigma \vdash s \vdash t' : T \quad T = T}{\Gamma; \Sigma \vdash (t' :: T) : T}$$

and

$$\frac{\Gamma, \Sigma \vdash t' \sim^n t^\text{ref} : T \quad \epsilon = (T) = \emptyset}{\Gamma; \Sigma \vdash (t' :: T) : T}$$

We have to prove that $\Gamma; \Sigma \vdash t' :: T \approx t^\text{ref} :: T : T$. By induction hypothesis we know that $\Gamma; \Sigma \vdash t' \approx t^\text{ref} : T$. The result follows directly by Proposition 53.

**Case** ($T\text{ref}$). Then $t = \text{ref} t'$ and $T = \text{Ref} T$, where

$$\frac{\Gamma; \Sigma \vdash s \vdash t' : T}{\Gamma; \Sigma \vdash \text{ref} t' : \text{Ref} T}$$

and

$$\frac{\Gamma, \Sigma \vdash t' \sim^n t^\text{ref} : T \quad \epsilon = (T) = \emptyset}{\Gamma; \Sigma \vdash \text{ref} t' \sim^\text{ref} t^\text{ref} : \text{Ref} T}$$

We have to prove that $\Gamma; \Sigma \vdash \text{ref} t' \approx \text{ref}^\text{ref} t^\text{ref} : \text{Ref} T$. By induction hypothesis we know that $\Gamma; \Sigma \vdash t' \approx t^\text{ref} : T$. The result follows directly by Proposition 54.

**Case** ($T\text{deref}$). Then $t = \text{deref} t'$, where

$$\frac{\Gamma; \Sigma \vdash s \vdash t' : \text{Ref} T}{\Gamma; \Sigma \vdash \text{deref} t' : T}$$

and

$$\frac{\Gamma, \Sigma \vdash t' \sim^\text{ref} t^\text{ref} : \text{Ref} T \quad \epsilon = (\text{Ref} T) = \emptyset}{\Gamma; \Sigma \vdash \text{deref} t' \sim^\text{ref} t^\text{ref} : \text{Ref} T}$$

We have to prove that $\Gamma; \Sigma \vdash \text{deref} t' \approx \text{deref} \text{ref} t^\text{ref} : T$. By induction hypothesis we know that $\Gamma; \Sigma \vdash t' \approx t^\text{ref} : T$. The result follows directly by Proposition 55.

**Case** ($T\text{asgn}$). Then $t = t_1 := t_2$ and $T = \text{Unit}$, where

$$\frac{\Gamma; \Sigma \vdash s \vdash t_1 : \text{Ref} T \quad \Gamma; \Sigma \vdash t_2 : T}{\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit}}$$

and

$$\frac{\Gamma; \Sigma \vdash t_1 \sim^\text{ref} t^\text{ref} : \text{Ref} T \quad \Gamma; \Sigma \vdash t_2 \sim^\text{ref} t^\text{ref} : T \quad \epsilon_1 = (\text{Ref} T) = \emptyset}{\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit}}$$

We have to prove that $\Gamma; \Sigma \vdash t_1 := t_2 \approx \text{ref} \text{ref} t^\text{ref} : \text{Ref} T$ and that $\Gamma; \Sigma \vdash t_2 \approx t^\text{ref} : T$. The result follows directly by Proposition 56.
Case (To). Then \( t = o \) and \( T = \text{Ref} T \), where

\[
\frac{o : T \in \Sigma}{\Gamma; \Sigma \vdash o : \text{Ref} T}
\]

and

\[
\frac{G \in \Sigma}{\Gamma; \Sigma \vdash o \sim_\rho o^G : \text{Ref} G}
\]

Then we have to prove that \( o \approx o^G : \text{Ref} T \). But the result follows directly by Proposition 57. □

**Lemma 46.** Consider \( \langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle \) and \( \mu'_1 \approx \mu'_2 \); such that \( \mu_1 \subseteq \mu'_1 \) and \( \mu_2 \subseteq \mu'_2 \); then \( \langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle \).

**Proof.** Direct as evolution of the store to related store does not alter the relation between values that not depend on new locations. □

**Proposition 47 (Compatibility Tb).** If \( b \in B \), then \( \Gamma; \Sigma \vdash b \approx b : B \).

**Proof.** Trivial as \( b = b \). □

**Proposition 48 (Compatibility Tx).** If \( x : T \in \Gamma \), then \( \Gamma; \Sigma \vdash x \approx x^T : T \).

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2 \), such that \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). We are required to show that:

\[
\langle \sigma_1(x), \mu_1 \rangle \approx \langle \sigma_2(x^T), \mu_2 \rangle : T
\]

which is immediately by the definition of \( \Gamma; \Sigma \vdash \); \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). □

**Proposition 49 (Compatibility Tapp).** If \( \Gamma; \Sigma \vdash t_1 \approx t_1^{T_1 \rightarrow T_2} : T_1 \rightarrow T_2 \), \( \Gamma; \Sigma \vdash t_2 \approx t_2^{T_1 \rightarrow T_2} : T_1 \rightarrow T_2 \), \( T_1 \rightarrow T_2 \), \( \varepsilon_1 \vdash T_1 \approx T_2 \), \( \varepsilon_2 \vdash T_1 \approx T_2 \), \( \varepsilon_1 \vdash T_1 \rightarrow T_2 \).

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2 \), such that \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). We are required to show that:

\[
\langle \sigma_1(t_1 t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t_1^{T_1 \rightarrow T_2} @ t_2^{T_1 \rightarrow T_2} \varepsilon_2 t_1^{T}), \mu_2 \rangle : T_1 \rightarrow T_2
\]

which, by definition of substitution, is equivalent to prove that

\[
\langle \sigma_1(t_1) \sigma_1(t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t_1^{T_1 \rightarrow T_2} @ t_2^{T_1 \rightarrow T_2} \varepsilon_2 t_1^{T}), \mu_2 \rangle : T_1 \rightarrow T_2
\]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t_1^{T_1 \rightarrow T_2} : T_1 \rightarrow T_2 \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2 \). We know then that \( \langle t_1, \mu_1 \rangle \approx \langle t_1^{T_1 \rightarrow T_2}, \mu_2 \rangle : T_1 \rightarrow T_2 \). Then suppose \( \sigma_1(t_1) \mid \mu_1 \mapsto v_1 \mid \mu'_1 \) and \( \sigma_2(t_1^{T_1 \rightarrow T_2}) \mid \mu_2 \mapsto v_2 \mid \mu'_2 \) (otherwise the result holds immediately). We know that \( \langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T_1 \rightarrow T_2 \). Similarly we instantiate \( \Gamma; \Sigma \vdash t_2 \approx t_1^{T_1 \rightarrow T_2} : T_1 \rightarrow T_1 \) with \( \sigma_1, \sigma_2, \mu'_1 \) and \( \mu'_2 \). Notice \( \mu_1 \subseteq \mu'_1 \) (\( \mu_2 \subseteq \mu'_2 \) resp.), therefore \( \Sigma \vdash \mu'_1 \) (\( \Sigma \vdash \mu'_2 \) resp.). Then we know that \( \langle t_2, \mu'_1 \rangle \approx \langle t_1^{T_1}, \mu'_2 \rangle : T_1 \). Then suppose \( \sigma_1(t_2) \mid \mu'_1 \mapsto v_1 \mid \mu'_2 \) and \( \sigma_2(t_1^{T_1}) \mid \mu'_2 \mapsto v_2 \mid \mu'_2 \) (otherwise the result holds immediately). We know that \( \langle v_2, \mu'_2 \rangle \approx \langle v_2, \mu'_2 \rangle : T_1 \). Let us assume \( v_1 = v_2 \) (the other case is analogous modulo one trivial step of reduction). Then the result holds by definition of related lambdas instantiating with \( \varepsilon_1, \varepsilon_2, \mu'_1, \mu'_2, v_1 \) and \( v_2 \). □

**Proposition 50 (Compatibility Top).** If \( \Gamma; \Sigma \vdash t_1 \approx t_1^{B_1} : B_1, \Gamma; \Sigma \vdash t_2 \approx t_2^{B_2} : B_2, \varepsilon_1 \vdash t_1 \approx t_1^{B_1}, \varepsilon_2 \vdash t_2 \approx t_2^{B_2} : B_2, t y(\Box) = B_1 x B_2 \rightarrow B_3, \) then \( \Gamma; \Sigma \vdash t_1 \oplus t_2 \approx \varepsilon_1 t_1^{B_1} \oplus \varepsilon_2 t_2^{B_2} : B_3 \).

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2 \), such that \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). We are required to show that:

\[
\langle \sigma_1(t_1 \oplus t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t_1^{B_1} \oplus \varepsilon_2 t_2^{B_2}), \mu_2 \rangle : B_3
\]

which, by definition of substitution, is equivalent to prove that

\[
\langle \sigma_1(t_1) \oplus \sigma_1(t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t_1^{B_1} \oplus \varepsilon_2 t_2^{B_2}), \mu_2 \rangle : B_3
\]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t_1^{B_1} : B_1 \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2 \). We know then that \( \langle t_1, \mu_1 \rangle \approx \langle t_1^{B_1}, \mu_2 \rangle : B_1 \). Then suppose \( \sigma_1(t_1) \mid \mu_1 \mapsto v_1 \mid \mu'_1 \) and \( \sigma_2(t_1^{B_1}) \mid \mu_2 \mapsto v_2 \mid \mu'_2 \) (otherwise the result holds immediately). We know that \( \langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : B_1 \). Similarly we instantiate \( \Gamma; \Sigma \vdash t_2 \approx t_1^{B_2} : B_2 \) with \( \sigma_1, \sigma_2, \mu'_1 \) and \( \mu'_2 \). Notice \( \mu_1 \subseteq \mu'_1 \) (\( \mu_2 \subseteq \mu'_2 \) resp.), therefore \( \Sigma \vdash \mu'_1 \) (\( \Sigma \vdash \mu'_2 \) resp.). Then we know that \( \langle t_2, \mu'_1 \rangle \approx \langle t_2, \mu'_2 \rangle : B_2 \). Then
suppose $\sigma_1(t_2) \mid \mu_1 \mapsto v_{12} \mid \mu'_1$ and $\sigma_2(t_2) \mid \mu_2 \mapsto v_{22} \mid \mu'_2$ (otherwise the result holds immediately). We know that $(v_{21}, \mu'_1) \approx (v_{22}, \mu'_2) : B_2$. Let us assume $v_{21} = u_{21}$ and $v_{22} = u_{22}$ (the other cases are analogous modulo one or two trivial steps of reduction).

Then $\sigma_1(t_1) \oplus \sigma_1(t_2) \mid \mu_1 \mapsto v_{11} \oplus v_{12} \mid \mu'_1$ and $\sigma_1(\epsilon_{B_1}) \oplus \sigma_2(B_2) \mid \mu_2 \mapsto \epsilon_{11} u_{21} \oplus \epsilon_{22} u_{22} \mid \mu'_2 \mapsto u_{21} \oplus u_{22}$. But as $v_{11} = u_{21}$ and $v_{12} = u_{22}$, then $v_{11} \oplus v_{12} \approx u_{21} \oplus u_{22}$ and the result holds.

**Proposition 51 (Compatibility T if)**. If $\Gamma; \Sigma \vdash t_1 \approx t_1^\text{Bool}; \text{Boo}, \Gamma; \Sigma \vdash t_2 \approx t_2^T : T, \Gamma; \Sigma \vdash t_3 \approx t^T : T, \epsilon_1 \vdash \text{Bool} \sim \text{Bool}, \epsilon \vdash t \sim T, \Gamma; \Sigma \vdash t_1$ then $t_2$ else $t_3 \approx \epsilon t_2^T \text{ else } \epsilon t_3^T : T$.

**Proof**. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1(\text{if } t_1 \text{ then } t_2 \text{ else } t_3), \mu_1 \rangle \approx \langle \sigma_2(\text{if } \epsilon t_1^\text{Bool} \text{ then } \epsilon t_2^T \text{ else } \epsilon t_3^T), \mu_2 \rangle : T$$

which, by definition of substitution, is equivalent to prove that:

$$\langle \sigma_1(\ell_1) \text{ then } \sigma_2(\ell_2), \mu_1 \rangle \approx \langle \sigma_2(\ell_1^\text{Bool} \text{ then } \ell_2^T \text{ else } \ell_3^T), \mu_2 \rangle : T$$

We instantiate $\Gamma; \Sigma \vdash t_1 \approx t_1^\text{Bool} \text{ Bool}$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1, \mu_2$ such that $\Sigma \vdash \sigma_1 \mu_1 \approx \sigma_2 \mu_2$. We know then that $(t_1, \mu_1) \approx (t_1^\text{Bool}, \mu_2) : \text{Bool}$. Then suppose $\sigma_1(t_1) \mid \mu_1 \mapsto v_{11} \mid \mu'_1$ and $\sigma_2(\text{Bool}) \mid \mu_2 \mapsto v_{21} \mid \mu'_2$ (otherwise the result holds immediately). We know that $(v_{11}, \mu'_1) \approx (v_{21}, \mu'_2) : \text{Bool}$. Let us assume $v_{21} = u_{21}$ (the other case is analogous modulo one trivial step of reduction). Also let us assume $v_{11} = \text{true}$. The case $(v_{11}, \mu'_1) \approx (\text{true}, \mu'_2)$ is analogous, therefore as $v_{11} = \text{true}$, we know:

$$\text{true} \equiv \text{true} \quad \text{true} \equiv \text{true} \quad \text{true} \equiv \text{true} \quad \text{true} \equiv \text{true} \quad \text{true} \equiv \text{true} \quad \text{true} \equiv \text{true}$$

But by instantiating $\Gamma; \Sigma \vdash t_2 \approx t_2^T : T$ with $\sigma_1, \sigma_2, \mu_1', \mu_2', \mu_2'$. Then $(t_2, \mu_1') \approx (t_2^T, \mu_2')$. As then, $t_2 \mid \mu_1' \mapsto v_{12} \mid \mu'_2$ and $t_2^T \mid \mu_2' \mapsto v_{22} \mid \mu'_2$, and $(v_{12}, \mu'_2) \approx (v_{22}, \mu'_2)$. Let us assume $v_{22} = u_{22}$ (the other case is analogous), then as $(v_{12}, \mu'_2) \approx (v_{22}, \mu'_2)$, the result holds.

**Proposition 52 (Compatibility T \lambda)**. If $\Gamma; \Sigma \vdash t \approx t^T : T_2, \text{then } \Gamma; \Sigma \vdash (\lambda x : T_1. t') \approx (\lambda x^T. t^T_2) : T_1 \rightarrow T_2$.

**Proof**. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1((\lambda x : T_1. t'), \mu_1) \rangle \approx \langle \sigma_2((\lambda x^T. t^T_2), \mu_2) : T_1 \rightarrow T_2$$

which, by definition of substitution, is equivalent to prove that:

$$\langle \lambda x : T_1. \sigma_1(t'), \mu_1' \rangle \approx \langle \lambda x^T. \sigma_2(t^T_2), \mu_2' \rangle : T_1 \rightarrow T_2$$

Consider $\nu_1', \nu_2', \epsilon_1, \epsilon_2, \mu_1', \mu_2'$ such that $\mu_1 \subseteq \mu_1', \mu_2 \subseteq \mu_2'$, $\langle \nu_1', \mu_1' \rangle \approx \langle \nu_2', \mu_2' \rangle : T_1, \epsilon_1 = (T_1 \rightarrow T_2) \vdash T_1 \rightarrow T_2 \rightarrow T_1 \rightarrow T_2$, and $\epsilon_2 = (T_1) \vdash T_1$. We have to prove that:

$$\langle \lambda x : T_1. \sigma_1(t') \rangle \nu_1' \mu_1' \approx \langle \epsilon_1 (\lambda x^T. \sigma_2(t^T_2)) \nu_1' \mu_1' : (T_1)u_2 :: T_1 \rightarrow x^T \rangle \nu_2' \mu_2' : T_2$$

Consider $v_2 = u_2$ (the other case is trivial because if $v_2 = \epsilon_2 u_2 :: T_1$, then as everything is static the only possibility is that $\epsilon_2 = (T_1)$ and as $\epsilon_2 = (T_1)$, consistent transitivity does not fail). By taking one step of evaluation (where consistent transitivity trivially does not fail):

$$\langle \lambda x : T_1. \sigma_1(t') \rangle \nu_1' \mu_1' \approx \sigma_1(t') [\nu_1'/x]$$

and

$$\langle \epsilon_1 (\lambda x^T. \sigma_2(t^T_2)) \nu_1' \mu_1' : (T_1)u_2 :: T_1 \rightarrow x^T \rangle \nu_2' \mu_2' : T_2$$

Notice that $\sigma_1(t') [\nu_1'/x] = \sigma_1(t')$, where $\sigma_1' = \sigma_1[x \mapsto \nu_1']$, and analogously $\sigma_2(t^T_2)[(T_1)u_2 :: T_1/x^T] = \sigma_2'(t^T_2)$, where $\sigma_2' = \sigma_2[x^T \mapsto (T_1)u_2 :: T_1]$. Also, as

$$\langle \nu_1', \mu_1' \rangle \approx \langle (T_1)u_2 :: T_1, \mu_2' \rangle : T_1$$

then $\Gamma'; x : T_1; \Sigma \vdash (\sigma_1', \mu_1) \approx (\sigma_2', \mu_2)$. Then by instantiating premise $\Gamma', x : T_1; \Sigma \vdash t \approx t^T : T_2$ with $\sigma_1', \mu_1, \sigma_2', \mu_2$, we know that $\sigma_1'(t') \mid \mu_1' \mapsto v_{12}' \mid \mu'_1 \approx \sigma_2'(t^T) \mid \mu_2' \mapsto v_{22}' \mid \mu_2'$, where $\langle v_{12}', \mu_1' \rangle \approx \langle v_{22}', \mu_2' \rangle : T_2$. If $v_{22}' = (T_2)u_2 :: T_2$ (the other case is similar), $v_{12}' \\ u_2 :: T_2 \rightarrow (T_2)u_2 :: T_2 \rightarrow T_2$, but $\langle v_{12}', \mu_1' \rangle \approx (T_2)u_2 :: T_2, \mu_2' : T_2$, and the result holds.

**Proposition 53 (Compatibility T : :)**. If $\Gamma; \Sigma \vdash t \approx t^T : T, \epsilon \vdash t \sim T, \text{then } \Gamma; \Sigma \vdash t : T \leftrightarrow T : T$. 

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1(t :: T), \mu_1 \rangle \approx \langle \sigma_2(eT T :: T), \mu_2 \rangle : T$$

which, by definition of substitution, is equivalent to prove that

$$\langle \sigma_1(t :: T), \mu_1 \rangle \approx \langle \epsilon \sigma_2(t^T :: T), \mu_2 \rangle : T$$

We instantiate $\Gamma; \Sigma \vdash t \approx t^T : T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that

$$\langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T.$$  

Then suppose $\sigma_1(t) \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$ and $\sigma_2(t^T) \mid \mu_2 \mapsto^* v_2 \mid \mu'_2$ (otherwise the result holds immediately). We know that $\langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T$. Let us assume $v_2 = u_2$ (the other case is analogous modulo one trivial step of reduction). Then $v_1 : T \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$, so we have to prove that $\langle v_1, \mu'_1 \rangle \approx \langle (T)u_2 :: T, \mu'_2 \rangle : T$, which holds because $\langle v_1, \mu'_1 \rangle \approx \langle u_2, \mu'_2 \rangle : T$. □

Proposition 54 (Compatibility Tref). If $\Gamma; \Sigma \vdash t \approx t^T : T$, $\epsilon \vdash T \sim T$, then $\Gamma; \Sigma \vdash t \approx \epsilon T^T : T$:

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1(t), \mu_1 \rangle \approx \langle \sigma_2(eT^T), \mu_2 \rangle : T$$

which, by definition of substitution, is equivalent to prove that

$$\langle \sigma_1(t), \mu_1 \rangle \approx \langle \epsilon \sigma_2(t^T), \mu_2 \rangle : T$$

We instantiate $\Gamma; \Sigma \vdash t \approx t^T : T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that

$$\langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T.$$  

Then suppose $\sigma_1(t) \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$ and $\sigma_2(t^T) \mid \mu_2 \mapsto^* v_2 \mid \mu'_2$ (otherwise the result holds immediately). We know that $\langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T$. Let us assume $v_2 = u_2$ (the other case is analogous modulo one trivial step of reduction). Then $v_1 : T \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$, and by Lemma 46, $\langle v_1, \mu'_1 \rangle \approx \langle u_2 :: T, \mu'_2 \rangle : T$, by (S$\mu$), $\langle v_1, \mu'_1 \rangle \approx \langle oT, \mu'_2 \rangle : T$ and the result holds. □

Proposition 55 (Compatibility Tdef). If $\Gamma; \Sigma \vdash t \approx t^T : T$, $\epsilon \vdash \Gamma \sim T$, then $\Gamma; \Sigma \vdash t \approx \epsilon T^T : T$:

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1(t), \mu_1 \rangle \approx \langle \sigma_2(eT T), \mu_2 \rangle : T$$

which, by definition of substitution, is equivalent to prove that

$$\langle \sigma_1(t), \mu_1 \rangle \approx \langle \epsilon \sigma_2(t^T T), \mu_2 \rangle : T$$

We instantiate $\Gamma; \Sigma \vdash t \approx t^T : T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that

$$\langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T.$$  

Then suppose $\sigma_1(t) \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$ and $\sigma_2(t^T) \mid \mu_2 \mapsto^* v_2 \mid \mu'_2$ (otherwise the result holds immediately). We know that $\langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T$. Let us assume $v_1 = o$ and $v_2 = o^T$ (the other case is analogous modulo one trivial step of reduction). Then $v_1 : T \mid \mu'_1 \mapsto^* \mu'_1 \mid \mu'_1$, and $t^T \epsilon \mu_2 \mid \mu'_2 \mapsto^* (T)\mu'_2(o^T) :: T \mid \mu_2$. By definition of $\mu'_1 \approx \mu'_2$. $\langle \mu'_1(o), \mu'_2 \rangle \approx \langle \mu'_2(o^T), \mu'_2 \rangle : T$, therefore the result follows by (S$\epsilon$). □

Proposition 56 (Compatibility Tasgn). If $\Gamma; \Sigma \vdash t_1 \approx t^T : T$, $\alpha \vdash \Gamma \sim T$, $\epsilon \vdash \Gamma \sim T$, $\epsilon \vdash T \sim T$, then $\Gamma; \Sigma \vdash t_1 := t_2 \approx T^T : T$:

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash (\sigma_1, \mu_1) \approx (\sigma_2, \mu_2)$. We are required to show that:

$$\langle \sigma_1(t_1 := t_2), \mu_1 \rangle \approx \langle \sigma_2(eT T), \mu_2 \rangle : Unit$$

which, by definition of substitution, is equivalent to prove that

$$\langle \sigma_1(t_1 := t_2), \mu_1 \rangle \approx \langle \epsilon \sigma_2(t^T T), \mu_2 \rangle : Unit$$

We instantiate $\Gamma; \Sigma \vdash t_1 \approx t^T : T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that

$$\langle t_1, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T.$$  

Then suppose $\sigma_1(t_1) \mid \mu_1 \mapsto^* v_1 \mid \mu'_1$ and $\sigma_2(t^T) \mid \mu_2 \mapsto^* v_2 \mid \mu'_2$ (otherwise the result holds immediately). We know that $\langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T$. Similarly we instantiate $\Gamma; \Sigma \vdash t_2 \approx t^T : T$ with $\sigma_1, \sigma_2, \mu'_1$ and $\mu'_2$. Notice $\mu_1 \subseteq \mu'_1 \triangleq \mu_2 \subseteq \mu'_2$ resp., therefore $\Sigma \vdash \mu'_1 \subseteq \mu'_2$ resp.). Then we know that

$$\langle t_2, \mu'_1 \rangle \approx \langle t_2^T, \mu'_2 \rangle : T.$$  

Then suppose $\sigma_1(t_2) \mid \mu'_1 \mapsto^* v_1 \mid \mu'_1$ and $\sigma_2(t_2^T) \mid \mu'_2 \mapsto^* v_2 \mid \mu'_2$ (otherwise the result holds immediately). We know that $\langle v_21, \mu'_1 \rangle \approx \langle v_22, \mu'_2 \rangle : T$. Let us assume $v_21 = u_21 = o^T$ and $v_22 = u_22$ (the other cases are
analogue modulo one or two trivial steps of reduction. Then \( \sigma_1(t_1) = \sigma_1(t_2) \mid \mu_1 \rightarrow^* \circ := \nu_{12} \mid \mu_1^\circ \rightarrow \text{unit} \mid \mu_1^\circ | o \rightarrow \nu_{12} \) and 
\( \epsilon_1 \sigma_2(t^T) := \epsilon_1 \sigma_2(t^T) \mid \mu_2 \rightarrow^* (\text{Ref } T) \sigma_2(o^T) := \epsilon_1 \sigma_2(t^T) | o \rightarrow^* (T)u_{22} \mid \mu_2^o \rightarrow \text{unit} \mid \mu_2^o | o \rightarrow^* (T)u_{22} \rightarrow T \).

Let \( \mu_1'' = [o \rightarrow \nu_{12}] \) and \( \mu_2'' = [o \rightarrow (T)u_{22} \rightarrow T] \). By (S\( \vdash \)), \( \langle \nu_{12}, \mu_1'' \rangle \approx (\langle T \rangle u_{22} \rightarrow T, \mu_2'') : T \), and by Lemma 46, \( \langle \nu_1, \mu_1'' \rangle \approx \langle \langle T \rangle u_{22} \rightarrow T, \mu_2'' \rangle : T \) and the result holds.  

**Proposition 57 (Compatibility To).** If \( o : T \in \Sigma \), then \( \Gamma; \Sigma \vdash o \approx o^T : T \).

**Proof.** Direct by definition of related stores.  

**Proposition 58 (Equivalence for fully-annotated terms (dynamics)).** For any \( t \in \text{TERM} \), \( o \vdash t : T \), \( t \rightarrow_n t^T : T \), then \( t \mid \cdot \rightarrow_n^* \nu \mid \mu \leftrightarrow t^T \mid \cdot \rightarrow_n^* \nu' \mid \mu' \), for some \( \nu, \mu, \nu', \mu' \) such that \( \langle \nu, \mu \rangle \approx \langle \nu', \mu' \rangle : T \).

**Proof.** As a special case of the fundamental Property 45 and the unfolding of related computations.  

**C.2. Static gradual guarantee**

**Definition 11 (Term precision).**

\[
\begin{align*}
\text{(Px)} & \quad x \subseteq x \quad \text{(Pc)} & \quad c \subseteq c \quad \text{(Pd)} & \quad t \subseteq t' \quad G_1 \subseteq G_1' \quad t_1 \subseteq t_1' \quad t_2 \subseteq t_2' \quad \rightarrow \quad (\lambda x : G_1, t) \subseteq (\lambda x : G_1', t') \quad (\lambda x : G_1, t_1 + t_2) \subseteq (\lambda x : G_1', t_1 + t_2') \quad (\lambda x : G_1, t_1 \cdot t_2) \subseteq (\lambda x : G_1', t_1' \cdot t_2') \\
\text{(Papp)} & \quad t_1 \subseteq t_1' \quad t_2 \subseteq t_2' \quad t_1 \cdot t_2 \subseteq t_1' \cdot t_2' \quad (\text{Pf}) & \quad t \subseteq t' \quad t_1 \subseteq t_1' \quad t_2 \subseteq t_2' \quad t \cdot t_1 \subseteq t' \cdot t_1' \quad \text{if } t \text{ then } t_1 \text{ else } t_2 \subseteq t' \cdot t_1' \text{ else } t_2' \\
\text{(Pref)} & \quad t \subseteq t' \quad G \subseteq G' \quad \text{ref}G \cdot t \subseteq \text{ref}G' \cdot t' \quad (\text{Pder}) & \quad t \subseteq t' \quad t_1 \subseteq t_1' \quad t_2 \subseteq t_2' \quad \text{ref}G \cdot t_1 \cdot t_2 \subseteq \text{ref}G' \cdot t_1' \cdot t_2' \\
\text{(Pass) } & \quad t \subseteq t' \quad \text{ref}G \cdot t \subseteq \text{ref}G' \cdot t' \quad (\text{Po}) & \quad o \subseteq o
\end{align*}
\]

**Definition 12 (Type environment precision).**

\[
\begin{align*}
\text{. } \subseteq . \\
\Gamma \subseteq \Gamma' \quad G \subseteq G' \quad \Gamma, x : G \subseteq \Gamma', x : G' \\
\Sigma \subseteq \Sigma' \quad G \subseteq G' \quad \Sigma, o : G \subseteq \Sigma', o : G'
\end{align*}
\]

**Lemma 59.** If \( \Gamma; \Sigma \vdash t : G \), \( \Gamma \subseteq \Gamma' \) and \( \Sigma \subseteq \Sigma' \), then \( \Gamma'; \Sigma' \vdash t : G' \) for some \( G \subseteq G' \).

**Proof.** Simple induction on typing derivations.

**Lemma 60.** If \( G_1 \sim G_2 \) and \( G_1 \subseteq G_1' \) and \( G_2 \subseteq G_2' \) then \( G_1' \sim G_2' \).

**Proof.** By definition of \( \sim \), there exists \( (T_1, T_2) \in (\tilde{T}_1, \tilde{T}_2) \in \gamma^2(G_1, G_2) \) such that \( T_1 = T_2 \), \( G_1 \subseteq G_1' \) and \( G_2 \subseteq G_2' \) mean that \( \gamma(G_1) \subseteq \gamma(G_1') \) and \( \gamma(G_2) \subseteq \gamma(G_2') \), therefore \( (T_1, T_2) \in (\tilde{T}_1, \tilde{T}_2) \) \( \in \gamma^2(G_1, G_2') \).

**Proposition 61 (Static gradual guarantee).** If \( t_1 : G_1 \) and \( t_1 \subseteq t_2 \), then \( t_2 : G_2 \) for some \( G_2 \) such that \( G_1 \subseteq G_2 \).

**Proof.** We prove the property on opens terms instead of closed terms: If \( \Gamma; \Sigma \vdash t_1 : G_1 \) and \( t_1 \subseteq t_2 \) then \( \Gamma; \Sigma \vdash t_2 : G_2 \) and \( G_1 \subseteq G_2 \).

The proof proceeds by induction on the typing derivation.

**Case (Gx, Gb).** Trivial by definition of \( \subseteq \) using \( (Px), (Pb) \) respectively.

**Case (Gλ).** Then \( t_1 = (\lambda x : G_1, t) \) and \( G_1 = G_1' \rightarrow G_2' \). By \( (G\lambda) \) we know that:

\[
\text{(G}\lambda) \quad \Gamma, x : G_1'; \Sigma \vdash t : G_2' \quad \Gamma \vdash (\lambda x : G_1', t) : G_1' \rightarrow G_2'
\]
Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = (\lambda x : G'_1, t')' \) and therefore

\[
(G\lambda) \quad \frac{t \subseteq t'}{G'_1 \subseteq G''_1} \quad \frac{(\lambda x : G'_1.t) \subseteq (\lambda x : G''_1.t')}{(\lambda x : G'_1.t')\subseteq G''_1}
\] (C.2)

Using induction hypotheses on the premise of C.1, \( \Gamma, x : G'_1; \Sigma' \vdash t' : G''_2 \) with \( G'_2 \subseteq G''_2 \). By Lemma 59, \( \Gamma, x : G'_1 \vdash t' : G''_2 \). Then we can use rule \((G\lambda)\) to derive:

\[
(G\lambda) \quad \frac{\Gamma, x : G'_1; \Sigma' \vdash t' : G''_2}{\Gamma; (\lambda x : G''_1.t') : G_1 \rightarrow G''_2}
\]

Where \( G_2 \subseteq G''_2 \). Using the premise of C.2 and the definition of type precision we can infer that

\[
G'_1 \rightarrow G'_2 \subseteq G''_2 \rightarrow G''_2
\]

and the result holds.

**Case** \((G\oplus)\). Then \( t_1 = t'_1 \oplus t'_2 \) and \( G_1 = \text{Int} \). By \((G\oplus)\) we know that:

\[
(T\oplus) \quad \frac{\Gamma; \Sigma \vdash t_1 : G_1 \quad \Gamma; \Sigma \vdash t_2 : G_2 \quad \Gamma_1 \sim B_1 \quad \Gamma_2 \sim B_2}{\Gamma; \Sigma \vdash t_1 \oplus t_2 : B_3}
\] (C.3)

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 \oplus t''_2 \) and therefore

\[
(P\oplus) \quad \frac{t'_1 \subseteq t''_1 \quad t'_2 \subseteq t''_2}{t'_1 \oplus t'_2 \subseteq t''_1 \oplus t''_2}
\] (C.4)

Using induction hypotheses on the premises of C.3, \( \Gamma; \Sigma \vdash t'_1 : G'_1 \) and \( \Gamma; \Sigma \vdash t'_2 : G'_2 \), where \( G_1 \subseteq G'_1 \) and \( G_2 \subseteq G'_2 \). By Lemma 60, \( G'_1 \sim B_1 \) and \( G'_2 \sim B_2 \). Therefore we can use rule \((G\oplus)\) to derive:

\[
(T\oplus) \quad \frac{\Gamma; \Sigma \vdash t'_1 : G'_1 \quad \Gamma; \Sigma \vdash t'_2 : G'_2 \quad \Gamma'_1 \sim B_1 \quad \Gamma'_2 \sim B_2}{\Gamma; \Sigma \vdash t'_1 \oplus t'_2 : B_3}
\]

and the result holds.

**Case** \((G\text{app})\). Then \( t_1 = t'_1 \cdot t'_2 \) and \( G_1 = G_{12} \). By \((G\text{app})\) we know that:

\[
(G\text{app}) \quad \frac{\Gamma; \Sigma \vdash t'_1 : G'_1 \quad \Gamma; \Sigma \vdash t'_2 : G'_2 \quad \Gamma \vdash \text{dom}(G'_1)}{\Gamma; \Sigma \vdash t'_1 \cdot t'_2 : \text{cod}(G'_1)}
\] (C.5)

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 \cdot t''_2 \) and therefore

\[
(P\text{app}) \quad \frac{t'_1 \subseteq t''_1 \quad t'_2 \subseteq t''_2}{t'_1 \cdot t'_2 \subseteq t''_1 \cdot t''_2}
\] (C.6)

Using induction hypotheses on the premises of C.5, \( \Gamma; \Sigma \vdash t''_1 : G''_1 \) and \( \Gamma; \Sigma \vdash t''_2 : G''_2 \), where \( G'_1 \subseteq G''_1 \) and \( G'_2 \subseteq G''_2 \). By definition precision (Definition 2) and the definition of \( \text{dom} \), \( \text{dom}(G'_1) \subseteq \text{dom}(G''_1) \) and, therefore by Lemma 60, \( G''_2 \sim \text{dom}(G''_1) \). Also, by the previous argument \( \text{cod}(G'_1) \subseteq \text{cod}(G''_1) \). Then we can use rule \((G\text{app})\) to derive:

\[
(G\text{app}) \quad \frac{\Gamma' \vdash t''_1 : G''_1 \quad \Gamma' \vdash t''_2 : G''_2 \quad \Gamma' \vdash \text{dom}(G''_1)}{\Gamma' \vdash t''_1 \cdot t''_2 : \text{cod}(G''_1)}
\]

and the result holds.

**Case** \((G\text{if})\). Then \( t_1 = \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 \) and \( G_1 = (G_2 \cap G_3) \). By \((G\text{if})\) we know that:

\[
(G\text{if}) \quad \frac{\Gamma; \Sigma \vdash t'_1 : G'_1 \quad \Gamma; \Sigma \vdash t'_2 : G'_2 \quad \Gamma; \Sigma \vdash t'_3 : G'_3}{\Gamma; \Sigma \vdash \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 : (G_2 \cap G_3)}
\] (C.7)
Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{if } t'_2 \text{ then } t''_2 \text{ else } t'''_2 \) and therefore

\[
\begin{align*}
(Pf) & \quad t'_1 \sqsubseteq t'_2 \quad t'_1 \sqsubseteq t''_2 \quad t'_2 \sqsubseteq t''_2 \\
\text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 \sqsubseteq t'_2 \text{ else } t'_3 \sqsubseteq t''_2 \\
\end{align*}
\]

(C.8)

Then we can use induction hypotheses on the premises of C.7 and derive:

\[
\begin{align*}
(Gdf) & \quad \Gamma' : \Sigma \vdash t'_1 : G'_1 \
& \quad \Gamma' : \Sigma \vdash t'_2 : G'_2 \
& \quad \Gamma' : \Sigma \vdash t'_3 : G'_3 \\
\Gamma' : \Sigma' \vdash \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 : (G'_2 \cap G'_3) \\
\end{align*}
\]

Where \( G'_1 \subseteq G'_2 \) and \( G'_2 \subseteq G'_3 \). Using the definition of type precision (Definition 2) we can infer that

\[
(G'_1 \cap G'_2) \sqsubseteq (G'_1 \cap G'_2)
\]

and the result holds.

**Case (G::).** Then \( t_1 = t :: G_1 \). By (G::) we know that:

\[
\begin{align*}
(G::) & \quad \Gamma \vdash t : G'_1 \
& \quad G'_1 \sim G_1 \\
\Gamma \vdash t :: G'_1 : G_1 \\
\end{align*}
\]

(C.9)

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t' :: G_2 \) and therefore

\[
\begin{align*}
(P::) & \quad t \sqsubseteq t' \
& \quad G_1 \sqsubseteq G_2 \\
& \quad t :: G_1 \sqsubseteq t' :: G_2 \\
\end{align*}
\]

(C.10)

Using induction hypotheses on the premises of C.9, \( \Gamma \vdash t' : G'_2 \) where \( G'_1 \subseteq G'_2 \). We can use rule (G::) and Lemma 60 to derive:

\[
\begin{align*}
(G::) & \quad \Gamma \vdash t' : G'_2 \\
& \quad G'_2 \sim G_2 \\
\Gamma \vdash t' :: G'_2 : G_2 \\
\end{align*}
\]

(C.11)

Where \( G_1 \subseteq G_2 \) and the result holds.

**Case (Gref).** Then \( t_1 = \text{ref}^G t'_1 \). By (Gref) we know that:

\[
\begin{align*}
(Gref) & \quad \Gamma ; \Sigma \vdash t'_1 : G'_1 \\
& \quad G'_1 \sim G \\
\Gamma ; \Sigma \vdash \text{ref } t'_1 : \text{Ref } G \\
\end{align*}
\]

(C.11)

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{ref}^G t'_2 \) and therefore

\[
\begin{align*}
(Pref) & \quad t'_1 \sqsubseteq t'_2 \
& \quad G \sqsubseteq G' \\
& \quad \text{ref}^G t'_1 \sqsubseteq \text{ref}^G t'_2 \\
\end{align*}
\]

(C.12)

Using induction hypotheses on the premise of C.11, \( \Gamma ; \Sigma \vdash t'_1 : G'_1 \), where \( G'_1 \subseteq G'_2 \). By definition of precision on types \( \text{Ref } G \sqsubseteq \text{Ref } G' \). Then we can use rule (Gref) to derive:

\[
\begin{align*}
(Gref) & \quad \Gamma' ; \Sigma' \vdash t'_2 : G'_2 \\
& \quad G'_2 \sim G' \\
\Gamma' ; \Sigma' \vdash \text{ref } t'_2 : \text{Ref } G' \\
\end{align*}
\]

and the result holds.

**Case (Gderef).** Then \( t_1 = \text{lref}_1 \). By (Gderef) we know that:

\[
\begin{align*}
(Gderef) & \quad \Gamma ; \Sigma \vdash t'_1 : G \\
\end{align*}
\]

(C.13)

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{lref}_2 \) and therefore

\[
\begin{align*}
(Pref) & \quad t'_1 \sqsubseteq t'_2 \\
& \quad \text{lref}_1 \sqsubseteq \text{lref}_2 \\
\end{align*}
\]

(C.14)
Using induction hypotheses on the premise of C.13, \( \Gamma; \Sigma \vdash t'_1 : G'_1 \). By definition of \( \text{tref}, \text{tref} G \subseteq \text{tref} G' \). Then we can use rule \((\text{Gderef})\) to derive:

\[
\frac{\Gamma'; \Sigma' \vdash t'_2 : G'}{\Gamma'; \Sigma' \vdash \text{tref}(G')} \tag{C.15}
\]

and the result holds.

**Case** \((\text{Gassign})\). Then \( t_1 = t'_1 := t'_2 \) and \( G_1 = G_{12} \). By \((\text{Gassign})\) we know that:

\[
\frac{\Gamma; \Sigma \vdash t'_1 : G'_1 \quad \Gamma; \Sigma \vdash t'_2 : G'_2 \quad G'_2 \sim \text{tref}(G'_1)}{\Gamma; \Sigma \vdash t'_1 := t'_2 : \text{cod}(G'_1)} \tag{C.15}
\]

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 := t''_2 \) and therefore

\[
\frac{t'_1 \sqsubseteq t''_1 \quad t'_2 \sqsubseteq t''_2}{t'_1 := t_2 \sqsubseteq t''_1 := t''_2} \tag{C.16}
\]

Using induction hypotheses on the premises of C.15, \( \Gamma; \Sigma \vdash t''_1 : G''_1 \) and \( \Gamma; \Sigma \vdash t''_2 : G''_2 \), where \( G'_1 \subseteq G''_1 \) and \( G'_2 \subseteq G''_2 \). By definition precision (Definition 2) and the definition of \( \text{tref}, \text{tref}(G'') \subseteq \text{tref}(G'_1) \) and, therefore by Lemma 60, \( G''_2 \sim \text{tref}(G''_1) \). Then we can use rule \((\text{Gassign})\) to derive:

\[
\frac{\Gamma'; \Sigma' \vdash t''_1 : G''_1 \quad \Gamma'; \Sigma' \vdash t''_2 : G''_2 \quad G''_2 \sim \text{tref}(G''_1)}{\Gamma'; \Sigma' \vdash t''_1 := t''_2 : \text{Unit}} \tag{C.15}
\]

and the result holds. \( \square \)

**C.3. Dynamic gradual guarantee**

In this section we present the proof the Dynamic Gradual Guarantee for \( \lambda^{\mathcal{E}}_{\text{ker}} \).

**Definition 14** \((\text{Intrinsic term precision})\). Let \( \Omega \in \mathcal{P}(\mathcal{V}[*] \times \mathcal{V}[*]) \cup \mathcal{P}(\text{Loc}_c \times \text{Loc}_c) \) be defined as \( \Omega := \{ x^{G_1} \subseteq x^{G_2}, \theta^{G_1} \subseteq \theta^{G_2} \} \)

We define an ordering relation \((\vdash \subseteq)\) in \( (\mathcal{P}(\mathcal{V}[*] \times \mathcal{V}[*]) \cup \mathcal{P}(\text{Loc}_c \times \text{Loc}_c)) \times \mathcal{T}[*] \times \mathcal{T}[*] \) shown in Fig. C.18.

**Definition 15** \((\text{Well Formedness of } \Omega)\). We say that \( \Omega \) is well formed iff \( \forall \{ x^{G_1} \subseteq x^{G_2} \} \in \Omega.G_{11} \subseteq G_{12} \)
Before proving the gradual guarantee, we first establish some auxiliary properties of precision. For the following propositions, we assume Well Formedness of \( \Omega \) (Definition 15).

**Proposition 62.** If \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \) for some \( \Omega \in \mathcal{P}(\forall[n] \times \forall[n]) \cup \mathcal{P}(\text{Loc}_n \times \text{Loc}_n) \), then \( G_1 \sqsubseteq G_2 \).

**Proof.** Straightforward induction on \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \), since the corresponding precision on types is systematically a premise (either directly or transitively). \( \square \)

**Proposition 63 (Substitution preserves precision).** If \( \Omega \cup \{x^{G_1} \sqsubseteq x^{G_2}\} \vdash t^{G_1} \sqsubseteq t^{G_2} \) and \( \Omega \vdash t^{G_3} \sqsubseteq t^{G_4} \), then \( \Omega \vdash \{t^{G_1}/x^{G_1}\}t^{G_3} \sqsubseteq \{t^{G_4}/x^{G_4}\}t^{G_2} \).

**Proof.** By induction on the derivation of \( t^{G_1} \sqsubseteq t^{G_2} \), and case analysis of the last rule used in the derivation. All cases follow either trivially (no premises) or by the induction hypotheses. \( \square \)

**Proposition 64 (Monotonicity of evidence).** If \( \epsilon_1 \sqsubseteq \epsilon_2 \), \( \epsilon_3 \sqsubseteq \epsilon_4 \), and \( \epsilon_1 \circ^m \epsilon_3 \) is defined, then \( \epsilon_1 \circ^m \epsilon_3 \sqsubseteq \epsilon_2 \circ^m \epsilon_4 \).

**Proof.** By definition of consistent transitivity for \( = \) and the definition of precision. \( \square \)

**Proposition 65.** If \( G_{11} \sqsubseteq G_{12} \) and \( G_{21} \sqsubseteq G_{22} \) then \( G_{11} \cap G_{21} \sqsubseteq G_{12} \cap G_{22} \).

**Proof.** By induction on the type derivation of the types and meet. \( \square \)

**Proposition 66 (Dynamic guarantee for \( \rightarrow \)).** Suppose \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \) and \( \mu_1 \sqsubseteq \mu_2 \). If \( t^{G_1} \mid \mu_1 \rightarrow t^{G_1} \mid \mu_1' \) then \( t^{G_2} \mid \mu_2 \rightarrow t^{G_2} \mid \mu_2' \), where \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \), \( \mu_1' \sqsubseteq \mu_2' \) for some \( \Omega' \supseteq \Omega \).

**Proof.** By induction on reduction \( t^{G_1} \mid \mu_1 \rightarrow t^{G_1} \mid \mu_1' \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case (r1).** We know that \( t^{G_1}_1 = (\epsilon_{11}(c_1) \oplus \epsilon_{12}(c_2)) \) then by \( (\sqsubseteq \mu) \) \( t^{G_2}_1 = (\epsilon_{21}(c_1) \oplus \epsilon_{22}(c_2)) \) for some \( \epsilon_{21}, \epsilon_{22} \) such that \( \epsilon_{11} \sqsubseteq \epsilon_{21} \) and \( \epsilon_{12} \sqsubseteq \epsilon_{22} \). If \( t^{G_1}_1 \mid \mu_1 \rightarrow c_3 \mid \mu_1 \) where \( c_3 = (c_1 \oplus c_2) \), then \( t^{G_2}_1 \mid \mu_2 \rightarrow c_3' \mid \mu_2 \) where \( c_3' = (c_1 \oplus c_2) \). But \( c_3 = c_3' \) and therefore \( t^{G_2}_1 \sqsubseteq t^{G_2}_1 \) and the result holds.

**Case (r2).** We know that \( t^{G_1}_1 = \epsilon_{11}(\lambda x^{G_{11}}.t^{G_{12}}) \sqsubseteq \epsilon_{21}u \) then by \( (\sqsubseteq \mu) \) \( t^{G_2}_1 \) must have the form \( t^{G_2}_1 = \epsilon_{21}(\lambda x^{G_{21}}.t^{G_{22}}) \) \( \sqsubseteq \epsilon_{22}u \). Let us pose \( \epsilon_1 = \epsilon_{12} \circ^m \text{dom}(\epsilon_{11}) \). Then

\[
\begin{align*}
t^{G_1}_1 \mid \mu_1 \rightarrow \text{icod}(\epsilon_{11})t'_2 : \text{G}_2 \mid \mu_1 \text{ with } t'_2 &= \{((\epsilon_1u_1 : \text{G}_{11})/x^{G_{11}})t^{G_{12}}\}.\end{align*}
\]

Also, by \( 64, \epsilon_2 = \epsilon_{22} \circ^m \text{dom}(\epsilon_{21}) \) is defined. Then

\[
\begin{align*}
t^{G_2}_1 \mid \mu_2 \rightarrow \text{icod}(\epsilon_{21})t'_2 : \text{G}_4 \mid \mu_2 \text{ with } t'_2 &= \{(\epsilon_2u_2 : \text{G}_{21})/x^{G_{21}}t^{G_{22}}\}.\end{align*}
\]

As \( \Omega \vdash t^{G_1}_1 \sqsubseteq t^{G_2}_1 \), then \( u_1 \sqsubseteq u_2 \), \( \epsilon_{12} \sqsubseteq \epsilon_{22} \) and \( \text{dom}(\epsilon_{11}) \sqsubseteq \text{dom}(\epsilon_{21}) \) as well, then by Proposition 64 \( \epsilon_1 \sqsubseteq \epsilon_2 \). Then \( \epsilon_1u_1 : \text{G}_{11} \sqsubseteq \epsilon_2u_2 : \text{G}_{21} \) by \( (\sqsubseteq \mu) \).

We also know by \( (\sqsubseteq \mu) \) and \( (\sqsubseteq \nu) \) that \( \Omega \cup \{x^{G_{12}} \sqsubseteq x^{G_{11}}\} \vdash t^{G_{12}} \sqsubseteq t^{G_{22}} \). By Substitution preserves precision (Proposition 63) \( t^{G_1}_1 \sqsubseteq t^{G_2}_1 \), therefore \( \text{icod}(\epsilon_{11})t'_2 : \text{G}_2 \sqsubseteq \text{icod}(\epsilon_{21})t'_2 : \text{G}_4 \) by \( (\sqsubseteq \mu) \). Then \( t^{G_2}_1 \sqsubseteq t^{G_2}_1 \).

Case (r3 - true). \( t^{G_1}_1 = \epsilon_1u_1 \) then \( \epsilon_{12} \circ^m \text{dom}(\epsilon_{11}) \) is defined. Then by \( (\sqsubseteq \mu) \) \( t^{G_2}_1 \) has the form \( t^{G_2}_1 = \epsilon_{21}u \) then \( \epsilon_{21} \circ^m \text{dom}(\epsilon_{11}) \) is defined. Then \( \epsilon_1u_1 : \text{G}_{11} \sqsubseteq \epsilon_2u_2 : \text{G}_{21} \) by \( (\sqsubseteq \mu) \).

Using the fact that \( t^{G_1}_1 \sqsubseteq t^{G_2}_1 \) we know that \( \epsilon_{12} \sqsubseteq \epsilon_{22} \) and \( \epsilon_{12} \circ^m \text{dom}(\epsilon_{11}) \) is defined. Then by Proposition 62 \( \epsilon_1u_1 \sqsubseteq \epsilon_2u_2 \) and \( \epsilon_{12} \circ^m \text{dom}(\epsilon_{11}) \) is defined.

Case (r3 - false). Same as case \( \rightarrow \text{if-true} \), using the fact that \( \epsilon_{13} \sqsubseteq \epsilon_{23} \) and \( \epsilon_{13} \circ^m \text{dom}(\epsilon_{11}) \) is defined.

Case (r4). We know that \( t^{G_1}_1 = \text{ref}^{G_1} \epsilon_1u_1 \) where \( G_1 = \text{Ref} G' \), then by \( (\sqsubseteq \mu) \) \( t^{G_2}_1 \) must have the form \( t^{G_2}_1 = \text{ref}^{G_1} \epsilon_2u_2 \) for some \( \epsilon_2, u_2, G_2 \) such that \( \epsilon_1 \sqsubseteq \epsilon_2, u_1 \sqsubseteq u_2, \) and \( G_1 \sqsubseteq G_2 \).
Then
\[ t_1^{G_1} \mid \mu_1 \rightarrow o^{G_1} \mid \mu_1 [o^{G_1} \mapsto \varepsilon_1 u_1 :: G'_1]. \]

Also, \( t_1^{G_2} \mid \mu_2 \rightarrow o^{G_2} \mid \mu_2 [o^{G_2} \mapsto \varepsilon_2 u_2 :: G'_2]. \)

Then by \((\varepsilon,.)\), \( \varepsilon_1 u_1 :: G'_1 \subseteq \varepsilon_2 u_2 :: G'_2 \), and then \( \mu_1 [o^{G_1} \mapsto \varepsilon_1 u_1 :: G'_1] \subseteq \mu_2 [o^{G_2} \mapsto \varepsilon_2 u_2 :: G'_2] \). Also by \((\varepsilon,.)\), as \( G'_1 \subseteq G'_2 \), \( o^{G_1} \subseteq o^{G_2} \) and the result holds.

**Case (r5).** We know that \( t_1^{G_1} = \varepsilon_1 t_1^{G_1} = \varepsilon_1 t_1^{G_1} \). Then by \((\varepsilon,.)\) \( t_1^{G_2} \) must have the form \( t_1^{G_2} = \varepsilon_2 t_2^{G_2} \) for some \( \varepsilon_2, o^{G_2}, G'_2 \) such that \( \varepsilon_1 \subseteq \varepsilon_2, o^{G_1} \subseteq o^{G_2}, \) and \( G'_1 \subseteq G'_2 \).

Then \( t_1^{G_1} \mid \mu_1 \rightarrow \varepsilon_1 \mu_1 (G_1') :: G_1 \mid \mu_1 \).

Also, \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 t_2^{G_2} (G'_2) :: G_2 \).

As \( \mu_1 \subseteq \mu_2 \), then \( \varepsilon_1 \mu_1 (G_1') \subseteq \mu_2 (G'_2) \). Then by \((\varepsilon,.)\), \( \varepsilon_1 \mu_1 (G_1') :: G_1' \subseteq \varepsilon_2 \mu_2 (G'_2) :: G_2' \), and the result holds.

**Case (r6).** We know that \( t_1^{G_1} = \varepsilon_1 t_1^{G_1} = \varepsilon_1 t_1^{G_1} \). Then by \((\varepsilon,.)\) \( t_1^{G_2} \) must have the form \( t_1^{G_2} = \varepsilon_2 t_2^{G_2} \) for some \( \varepsilon_2, G_2, G_2' \) such that \( \varepsilon_1 \subseteq \varepsilon_2, \varepsilon_2 \subseteq \varepsilon_2, u_1 \subseteq u_2, G_1 \subseteq G_2, G_2' \subseteq G_2 \).

Let us pose \( \varepsilon_1 = \varepsilon_2 o^{G_1} \mapsto \varepsilon_2 \). Then \( t_1^{G_1} \mid \mu_1 \rightarrow \varepsilon_1 \mu_1 (G_1') :: G_1 \mid \mu_1 \).

By inspection of evidence and inversion lemma, as \( \varepsilon_2 \subseteq \varepsilon_2 \) then \( \varepsilon_2 \). Also, by \( \varepsilon_2 = \varepsilon_2 o^{G_1} \mapsto \varepsilon_2 \). Then, \( t_1^{G_1} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_1') :: G_2 \).

Then by \((\varepsilon,.)\), \( \varepsilon_1 u_1 :: G_1 \subseteq \varepsilon_2 u_2 :: G_2 \), and then \( \mu_1 [o^{G_1} \mapsto \varepsilon_1 u_1 :: G_1] \subseteq \mu_2 [o^{G_2} \mapsto \varepsilon_2 u_2 :: G_2] \) and the result holds.

**Proposition 67 (Dynamic gradual guarantee).** Suppose \( t_1^{G_1} \subseteq t_1^{G_1} \) and \( \mu_1 \subseteq \mu_2 \). Then if \( t_1^{G_1} \mid \mu_1 \rightarrow \varepsilon_1 \mu_1 (G_1') :: G_1' \) then \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \).

**Proof.** We prove the following property instead: Suppose \( \Omega \vdash t_1^{G_1} \subseteq t_1^{G_2} \) and \( \mu_1 \subseteq \mu_2 \). If \( t_1^{G_1} \mid \mu_1 \rightarrow \varepsilon_1 \mu_1 (G_1') :: G_1' \), then \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \

where \( \Omega \vdash \varepsilon_1 \mu_1 (G_1') \subseteq \varepsilon_2 \mu_2 (G_2') \), and \( \mu_1 \subseteq \mu_2 \) for some \( \Omega \subseteq \Omega \).

By induction on reduction \( t_1^{G_1} \mid \mu_1 \rightarrow \varepsilon_1 \mu_1 (G_1') :: G_1' \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case (r5).** By dynamic guarantee of \( \vdash \) (Proposition 66), \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \) where \( \Omega \vdash t_2^{G_2} \subseteq G_2' \), \( \mu_1 \subseteq \mu_2 \) for some \( \Omega \supseteq \Omega \). And the result holds immediately.

**Case (r6).** By dynamic guarantee of \( \vdash \) (Proposition 66), \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \) where \( \Omega \vdash t_2^{G_2} \subseteq G_2' \), \( \mu_1 \subseteq \mu_2 \) for some \( \Omega \supseteq \Omega \). And the result holds immediately.

**Case (r5).** By dynamic guarantee of \( \vdash \) (Proposition 66), \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \) where \( \Omega \vdash t_2^{G_2} \subseteq G_2' \), \( \mu_1 \subseteq \mu_2 \) for some \( \Omega \supseteq \Omega \). And the result holds immediately.

**Case (r6).** By dynamic guarantee of \( \vdash \) (Proposition 66), \( t_1^{G_2} \mid \mu_2 \rightarrow \varepsilon_2 \mu_2 (G_2') :: G_2' \) where \( \Omega \vdash t_2^{G_2} \subseteq G_2' \), \( \mu_1 \subseteq \mu_2 \) for some \( \Omega \supseteq \Omega \). And the result holds immediately.
Appendix D. Space efficiency

Lemma 68. ∀ε \vdash G_1 \sim G_2, \text{size}(ε) \leq 2^{\text{height}(ε)} - 1

Proof. By induction on ε. □

Lemma 69. If G_1 \cap G_2 = G_3, then \text{height}(G_3) \leq \max(\text{height}(G_1), \text{height}(G_2))

Proof. By induction on G_1 \cap G_2 = G_3.

Case \(G \cap \emptyset = \emptyset \cap G = G\). We know that \text{height}(G) \geq 1, and that \text{height}(\emptyset) = 1, therefore \max(\text{height}(G), \text{height}(\emptyset)) = \text{height}(G), and the result holds immediately.

Case \((G_1 \rightarrow G_2) \cap (G_2 \rightarrow G_3) = (G_1 \cap G_2 \rightarrow G_3) \cap (G_3 \cap G_2)\). We know by induction hypothesis that \text{height}(G_1 \cap G_2) \leq \max(\text{height}(G_1), \text{height}(G_2)), and \text{height}(G_2 \cap G_3) \leq \max(\text{height}(G_2), \text{height}(G_3)). We also know that \text{height}(G_1 \rightarrow G_2) = 1 + \max(\text{height}(G_1), \text{height}(G_2)), and \text{height}(G_2 \rightarrow G_3) = 1 + \max(\text{height}(G_2), \text{height}(G_3)).

Then we have to prove that
\[
\text{height}((G_1 \cap G_2) \rightarrow (G_1 \cap G_3) \rightarrow (G_1 \cap G_3)) \leq \max(\text{height}(G_1 \cap G_2), \text{height}(G_2 \cap G_3))
\]
But we know that
\[
\text{height}((G_1 \cap G_2) \rightarrow (G_1 \cap G_3)) = 1 + \max(\text{height}(G_1 \cap G_2), \text{height}(G_2 \cap G_3))
\]
and that
\[
\text{max}(\text{height}(G_1 \rightarrow G_2), \text{height}(G_2 \rightarrow G_3)) = \max(1 + \max(\text{height}(G_1), \text{height}(G_2)), 1 + \max(\text{height}(G_1), \text{height}(G_2)))
\]
therefore the result follows.
Case (Ref \( G_1 \sqcap \text{Ref} \ G_2 = \text{Ref} \ G_1 \sqcap \text{Ref} \ G_2 \)). We know by induction hypothesis that \( \text{height}(G_1 \sqcap G_2) \leq \max(\text{height}(G_1), \text{height}(G_2)) \). We also know that \( \text{height}(\text{Ref} \ G_1) = 1 + \max(\text{height}(G_1), \text{height}(G_{12})) \). Then we have to prove that

\[
\text{height}(\text{Ref} \ G_1 \sqcap G_2) \leq \max(\text{height}(\text{Ref} \ G_1), \text{height}(\text{Ref} \ G_2))
\]

But we know that

\[
\text{height}(\text{Ref} \ G_1 \sqcap G_2) = 1 + \text{height}(G_1 \sqcap G_2) \\
\leq 1 + \max(\text{height}(G_1), \text{height}(G_2)) \\
= \max(1 + \text{height}(G_1), 1 + \text{height}(G_2)) \\
= \max(\text{height}(\text{Ref} \ G_1), \text{height}(\text{Ref} \ G_2))
\]

and the result holds. □

Lemma 70. If \( \gamma = (G_1, G_2) = \epsilon \), then \( \text{height}(\epsilon) \leq \max(\text{height}(G_1), \text{height}(G_2)) \)

Proof. Direct by 19 as \( (G_1) \circ^= (G_2) = (G_1 \sqcap G_2) \) if defined. □

Lemma 71. If \( \epsilon_1 \circ^= \epsilon_2 = \epsilon_3 \), then \( \text{height}(\epsilon_3) \leq \max(\text{height}(\epsilon_1), \text{height}(\epsilon_2)) \)

Proof. Direct by 19 as \( (G_1) \circ^= (G_2) = (G_1 \sqcap G_2) \) if defined. □

Lemma 72. If \( \epsilon; \sigma \vdash t \leadsto_n t : G \), then if \( \epsilon \) occurs in \( t \), then \( \exists G' \) in the derivation \( \sigma \vdash t \leadsto_n t : G \), such that \( \text{height}(\epsilon) \leq \text{height}(G') \) and \( \text{size}(\epsilon) \leq 2^{\text{height}(G')} - 1 \).

Proof. By induction on \( \sigma; \sigma \vdash t \leadsto_n t : G \) using Lemmas 20 and 18. □

Lemma 73. Let \( \epsilon \vdash G \leadsto_1 G_2 \), then

1. \( \text{height}(\text{idom}(\epsilon)) < \text{height}(\epsilon) \) if \( \text{idom}(\epsilon) \) is defined.
2. \( \text{height}(\text{icon}(\epsilon)) < \text{height}(\epsilon) \) if \( \text{icon}(\epsilon) \) is defined.
3. \( \text{height}(\text{idref}(\epsilon)) < \text{height}(\epsilon) \) if \( \text{idref}(\epsilon) \) is defined.

Proposition 74. If \( t \leadsto_n t : G \) and \( t \vdash \cdot \mapsto^{*} t' \mid \mu' \) such that \( \epsilon \) occurs in \( (t', \mu') \), then there exists \( G' \) in the derivation of \( t \leadsto_n t : G \) such that \( \text{height}(\epsilon) \leq \text{height}(G') \) and \( \text{size}(\epsilon) \leq 2^{\text{height}(G')} - 1 \).

Proof. By induction on the length of reduction \( t \vdash \cdot \mapsto^{*} t' \mid \mu \).

Case (\( t \vdash \cdot \mapsto^{0} t \mid \cdot \)). Direct by Lemma 72.

Case (\( t \vdash \cdot \mapsto^{k} t'' \mid \mu'' \), and \( t' \mid \mu'' \mapsto \cdot \mid \mu'' \)). We only show representative cases. By induction hypothesis we know that, \( \forall \epsilon' \) such that \( \epsilon' \) occurs in \( (t'', \mu'') \), then \( \exists G' \) in the derivation \( \sigma; \sigma \vdash t \leadsto_n t : G \), such that \( \text{height}(\epsilon') \leq \text{height}(G') \) and \( \text{size}(\epsilon') \leq 2^{\text{height}(G')} - 1 \) (1). One set of cases is when \( \epsilon \) occurs in \( (t'', \mu'') \), then the result follows immediately. Otherwise, the only cases that produces new evidences are the following:

- Case (RE) and (r2). Let \( G_1 = G_{11} \rightarrow G_{12} \).

  \[
  \epsilon_1 \circ^= (\lambda \cdot G_{11} \cdot t'). \epsilon_2 \leadsto_2 G_2 \mid \mu'' \mapsto \text{icon}(\epsilon_1)((\epsilon_2 \circ^= \text{idom}(\epsilon_1))u \cdot G_{11} / x^{G_{11}})[\cdot] \circ^= G_2 \mid \mu''
  \]

  By induction hypotheses, \( \exists G'_1, G'_2, \text{height}(\epsilon_1) \leq \text{height}(G'_1) \), and \( \text{height}(\epsilon_2) \leq \text{height}(G'_2) \). Let \( G' \) the tallest of types \( G'_1 \) and \( G'_2 \). Note that by Lemma 73, \( \text{height}(\text{idom}(\epsilon_1)) \leq \text{height}(\epsilon_1) \), and \( \text{height}(\text{icon}(\epsilon_1)) \leq \text{height}(\epsilon_1) \). Let \( \epsilon' = \epsilon_2 \circ^= \text{idom}(\epsilon_1) \). By Lemma 21, \( \text{height}(\epsilon') \leq \max(\text{height}(\epsilon_2), \text{height}(\text{idom}(\epsilon_1))) \leq \max(\text{height}(\epsilon_2), \text{height}(\epsilon_1)) \leq \text{height}(G') \). Then by Lemma 18, \( \text{size}(\epsilon') \leq 2^{\text{height}(\epsilon')} - 1 \leq 2^{\text{height}(G')} - 1 \). Also by Lemma 18, and as \( \text{height}(\text{icon}(\epsilon_1)) \leq \text{height}(\epsilon_1) \leq \text{height}(G') \), then \( \text{size}(\text{idom}(\epsilon_1)) \leq 2^{\text{height}(\text{idom}(\epsilon_1))} - 1 \leq 2^{\text{height}(G')} - 1 \), and the result holds.

- Case (RE) and (r5). Analogously to (r2), noticing that by Lemma 73, \( \text{height}(\epsilon') \leq \text{height}(\epsilon) \).

- Case (RE) and (r6).

  \[
  \epsilon_{10} G^1 := G_2 \mid \mu'' \mapsto \text{unit} \mid \mu''[\sigma^G \mapsto (\epsilon_2 \circ^= \text{idref}(\epsilon_1))u \cdot G_1]
  \]
By induction hypotheses, \( \exists G', G_2. \text{height}(e_1) \leq \text{height}(G') \), and \( \text{height}(e_2) \leq \text{height}(G_2) \). Let \( G' \) the tallest of types \( G'_1 \) and \( G'_2 \). Note that by Lemma 73, \( \text{height}(\text{iref}(e_1)) \prec \text{height}(e_1) \). Let \( e' = e_2 \circ^\mu \text{iref}(e_1) \). By Lemma 21, \( \text{height}(e') \leq \max(\text{height}(e_2), \text{height}(\text{iref}(e_1))) \leq \max(\text{height}(e_2), \text{height}(e_1)) \leq \text{height}(G') \). Then by Lemma 18, \( \text{size}(e') \leq 2^{\text{height}(e') - 1} \leq 2^{\text{height}(G') - 1} \), and the result holds.

- Case \((R') \) and \((r7)\). Then \( e_1 t_1 | \mu'' \mapsto e'_1 t'_1 | \mu' \). Suppose \( e_1 t_1 = e_1 (e'_1 t'_1 : G_1) t_1 \). Then \( e'_1 = e_1 \circ^\mu \varepsilon'' \). By Lemma 21, \( \text{height}(e'_1) \leq \max(\text{height}(e_1), \text{height}(e'_1)) \), but by \((1)\) we know that \( \exists G' \) in the elaboration, such that \( \text{height}(e_1) \leq \text{height}(G') \) and \( \text{height}(e'_1) \leq \text{height}(G') \). Therefore \( \text{height}(e'_1) \leq \text{height}(G') \). By Lemma 18, \( \text{size}(e'_1) \leq 2^{\text{height}(e'_1) - 1} \) but as \( \text{height}(e'_1) \leq \text{height}(G') \), then \( \text{size}(e'_1) \leq 2^{\text{height}(G') - 1} \) and the result follows. \( \Box \)

Proposition 75. If \( t \leadsto_n t : G \) and \( t | \cdot \leadsto^* t' | \mu' \), then there exists \( G' \) in the derivation of \( t \leadsto_n t : G \) such that \( \text{size}((t', \mu')) \in O(2^{\text{height}(G')}) \). 

Proof. Rule \((r7)\) prevents nesting of adjacent ascriptions in any term in the evaluation context, redex or store. Therefore the number of evidences of a program is proportional to the size of the program state (in the worst case, although assignments and applications introduce two evidences, each correspond to each of its subterms). Therefore by Proposition 22, the size of each evidence is in \( O(2^{\text{height}(G')}) \) for some \( G' \) in the elaboration of \( t \). \( \Box \)

Appendix E. Relation to the coercion calculus

Lemma 76. \( G_1 \cap G_2 = G \iff G_2 \cap G_1 = G \).

Proof. We prove both sides of the proposition by induction on the premise, i.e. by induction on \( G_1 \cap G_2 \) for the \( \Rightarrow \) case, and induction on \( G_2 \cap G_1 \) for the \( \Leftarrow \) (both cases as identical). \( \Box \)

Lemma 77. \( e_1 \circ^\mu e_2 = e \iff e_2 \circ^\mu e_1 = e \).

Proof. Direct by Proposition 76. \( \Box \)

Lemma 78. \( \varepsilon = \{ \varepsilon \vdash G_1 \sim G_2 \} \), then \( \text{nm} \varepsilon \).

Proof. Straightforward induction on judgment \( \varepsilon \vdash G_1 \sim G_2 \). \( \Box \)

Proposition 79. Let \( c_1 = \{ \varepsilon_1 \vdash G_1 \sim G_2 \} \) and \( c_2 = \{ \varepsilon_2 \vdash G_2 \sim G_3 \} \). Then

1. \( c_1 ; c_2 \leadsto^* \text{Fail} \iff \varepsilon_1 \circ^\mu e_2 \text{ is undefined} \)
2. \( c_1 ; c_2 \leadsto^* c \land \text{nm} c \iff \varepsilon_1 \circ^\mu e_2 \text{ is defined. Furthermore } c = \{ (\varepsilon_1 \circ^\mu e_2) \vdash G_1 \sim G_3 \} \).

Proof. Direct by induction on types \( G_1, G_2 \) and \( G_3 \), inspection on the coercion reduction rules and transitivity of evidence. We only present interesting cases.

Case \((G_1 = R_1, G_2 = ?, G_3 = R_2)\). Then \( \varepsilon_1 = \{ (R_1) \}, \varepsilon_2 = \{ (R_2) \}, c_1 = R_1, \) and \( c_2 = R_2 \). If \( R_1 \neq R_2 \) then \( R_1 \cap R_2 \) is not defined and then \( e_1 \circ^\mu e_2 \) is not defined. Also \( R_1 = R_2 \Rightarrow \text{Fail} \) and the result holds.

If \( R_1 = R_2 = R \) then \( R_1 \cap R_2 = R \) and then \( e_1 \circ^\mu e_2 \mapsto (R) \). Also \( R = \{ (R) \vdash R \sim R \} \) and the result holds.

Case \((G_1 = ?, G_2 = R, G_3 = ?)\). Then \( \varepsilon_1 = \{ (R) \}, \varepsilon_2 = \{ (R) \}, c_1 = R_1, \) and \( c_2 = R_2 \). As \( R \cap R = R \) then \( e_1 \circ^\mu e_2 = (R) \). Also \( R = R \) is in normal form, but \( R = \{ (R) \vdash ? \sim ? \} \) and the result holds.

Case \((G_1 = ?, G_2 = ?, G_3 = ?)\). Then we proceed on cases \( \varepsilon_1 \) and \( e_2 \).

1. \( (\varepsilon_1 = (?) \land \varepsilon_2 = (?) \). Then \( c_1 = i_\gamma \) and \( c_2 = i_\gamma \). But \( i_\gamma ; i_\gamma \mapsto i_\gamma, \text{nm} i_\gamma, \varepsilon_1 \circ^\mu e_2 = (?) \), and \( i_\gamma = \{ (?) \vdash ? \sim ? \} \) and the result holds.
2. \( (\varepsilon_1 = (R), \varepsilon_2 = (?) \). Then \( c_1 = R; R! \) and \( c_2 = i_\gamma \). But \( R; R! \leadsto R; R!, \text{nm} R; R!, \varepsilon_1 \circ^\mu e_2 = (R) \), and \( R; R! = \{ (R) \vdash ? \sim ? \} \) and the result holds.
3. \( (\varepsilon_1 = (?), \varepsilon_2 = (R)) \). Analogous to previous sub-case.
4. \( (\varepsilon_1 = (R_1), \varepsilon_2 = (R_2)) \). Then \( c_1 = R_1; R_1! \) and \( c_2 = R_2; R_2! \). If \( R_1 \neq R_2 \), then \( R_1; R_1! \leadsto R_2; R_2! \mapsto \text{Fail} \), but also \( \varepsilon_1 \circ^\mu e_2 = (R_1) \circ^\mu (R_2) \) is undefined as \( R_1 \cap R_2 \) is not defined.
If $R_1 = R_2 = R$, then

$$R_1?; R_1!; R_2?; R_2!$$

$$= R?; R!; R?; R!$$

$$\rightarrow R?; i_R; R!$$

$$\rightarrow R?; R!$$

$$\rightarrow i_R$$

where nm $i_R$. But also $e_1 \circ \epsilon e_2 = (R_1) \circ \epsilon (R_2) = (R) \circ \epsilon (R) = (R)$, and $i_R = \langle (R) \vdash \sim \rangle$ and the result holds.

**Case** $(G_1 = R_1, G_2 = ?, G_3 = ?)$. Then we proceed on cases for $e_2$.

1. $(e_1 = (R_1), e_2 = (?))$. Then $c_1 = R_1?; R_1!$ and $c_2 = i_R$. But $R_1?; R_1!; i_R \rightarrow R_1?; R_1!$, nm $R_1?; R_1!$, $e_1 \circ \epsilon e_2 = (R_1)$, and $R_1?; R_1! = \langle (R_1) \vdash \sim \rangle$ and the result holds.

2. $(e_1 = (R_1), e_2 = (R_2))$. Then $c_1 = R_1?; R_1!$ and $c_2 = R_2?; R_2!$. If $R_1 \neq R_2$, then $R_1?; R_1!; R_2?; R_2! \rightarrow 3$ Fail, but also $e_1 \circ \epsilon e_2 = (R_1) \circ \epsilon (R_2)$ is undefined as $R_1 \cap R_2$ is not defined.

If $R_1 = R_2 = R$, then

$$R_1?; R_1!; R_2?; R_2!$$

$$= R?; R!; R?; R!$$

$$\rightarrow R?; i_R; R!$$

$$\rightarrow R?; R!$$

$$\rightarrow i_R$$

where nm $i_R$. But also $e_1 \circ \epsilon e_2 = (R_1) \circ \epsilon (R_2) = (R) \circ \epsilon (R) = (R)$, and $i_R = \langle (R) \vdash \sim \rangle$ and the result holds.

**Case** $(G_1 = ?, G_2 = ?, G_3 = R_2)$. Analogous to previous case.

**Case** $(G_1 = \text{Ref } G_1' \neq R_1, G_2 = \text{Ref } G_2' \neq R_2, G_3 = \text{Ref } G_3' \neq R_3)$. Then $e_1 = \langle \text{Ref } G_1' \rangle, e_2 = \langle \text{Ref } G_2' \rangle, c_1 = \text{Ref } c_{21}, c_{12}$, where $c_{21} = \langle (G_{12}) \vdash G_2' \sim G_1' \rangle$ and $c_{12} = \langle (G_{12}) \vdash G_1' \sim G_2' \rangle$, and $c_2 = \text{Ref } c_{32}, c_{23}$, where $c_{32} = \langle (G_{23}) \vdash G_3' \sim G_2' \rangle$ and $c_{23} = \langle (G_{23}) \vdash G_2' \sim G_3' \rangle$.

But

$$c_1; c_2 = \langle \text{Ref } c_{21}, c_{12}; \text{Ref } c_{32}, c_{23} \rangle$$

and $e_1 \circ \epsilon e_2 = (\text{Ref } G_1') \circ \epsilon (\text{Ref } G_2') = (\text{Ref } G_2') \circ \epsilon (\text{Ref } G_1')$ (Proposition 77).

By induction hypothesis on $c_{32} = \langle (G_{23}) \vdash G_3' \sim G_2' \rangle$ and $c_{21} = \langle (G_{12}) \vdash G_2' \sim G_1' \rangle$, if $c_{32}; c_{21} \rightarrow^{*} \text{Fail}$, and $G_2' \cap G_1'$ is not defined, therefore $\text{Ref } (c_{32}; c_{21}) (c_{12}; c_{23}) \rightarrow^{*} \text{Fail}$, and $\text{Ref } (G_2') \circ \epsilon (\text{Ref } G_1')$ is not defined and the result holds.

Similarly by induction hypothesis on $c_{12} = \langle (G_{12}) \vdash G_1' \sim G_2' \rangle$ and $c_{23} = \langle (G_{23}) \vdash G_2' \sim G_3' \rangle$, if $c_{32}; c_{21} \rightarrow^{*} \text{Fail}$ and $G_1' \cap G_2'$ is not defined, therefore the result holds.

The only case left is that if we apply both induction hypotheses and we know that $c_{32}; c_{21} \rightarrow^{*} c_{31}$, nm $c_{31}, c_{31} = \langle (G_{23} \cap G_{12}) \vdash G_3' \sim G_1' \rangle = \langle (G_{12} \cap G_{23}) \vdash G_1' \sim G_2' \rangle$ (Proposition 77), $c_{12}; c_{23} \rightarrow^{*} c_{13}$, nm $c_{13}$, and $c_{13} = \langle (G_{12} \cap G_{23}) \vdash G_1' \sim G_2' \rangle$. Then Ref $(c_{32}; c_{21}) (c_{12}; c_{23}) \rightarrow^{*} \text{Ref } c_{31}, c_{13}$, nm $c_{31}, c_{13}$, and $e_1 \circ \epsilon e_2 = (\text{Ref } G_1') \circ \epsilon (\text{Ref } G_2') = (\text{Ref } G_2') \circ \epsilon (\text{Ref } G_1')$.

But $(\text{Ref } G_{12}' \cap G_{23}) \vdash \text{Ref } G_1' \sim \text{Ref } G_2' = \text{Ref } (G_{12}' \cap G_{23}) \vdash G_1' \sim G_2' \Rightarrow (G_{12}' \cap G_{23}) \vdash G_1' \sim G_2'$, and the result holds.

**Case** $(G_1 = \text{Ref } G_1' \neq R_1, G_2 = \text{Ref } G_2' \neq R_2, G_3 = ?)$. Then $e_1 = \langle \text{Ref } G_1' \rangle, e_2 = \langle \text{Ref } G_2' \rangle, c_1 = \text{Ref } c_{21}, c_{12}$, where $c_{21} = \langle (G_{12}) \vdash G_2' \sim G_1' \rangle$ and $c_{12} = \langle (G_{12}) \vdash G_1' \sim G_2' \rangle$, and $c_2 = \text{Ref } c_{32}, c_{23}$, where $c_{32} = \langle (G_{23}) \vdash G_3' \sim G_2' \rangle$ and $c_{23} = \langle (G_{23}) \vdash G_2' \sim G_3' \rangle$, and we proceed analogous to previous case.

**Case** $(G_1 = \text{Ref } G_1' \neq R_1, G_2 = ?, G_3 = ?)$. Then $e_1 = \langle \text{Ref } G_1' \rangle, c_1 = \langle (\text{Ref } G_1') \vdash \text{Ref } G_1' \sim \text{Ref } G_2' \rangle$, and we proceed by cases for $e_2$.

1. $(e_2 = (?))$. Then $c_2 = i_R$, and $c_1; c_2 \rightarrow c_1$. We also know nm $c_1$ (by Lemma 78), and $e_1 \circ \epsilon e_2 = (\text{Ref } G_1') \circ \epsilon (\text{Ref } G_2') = (\text{Ref } G_1')$, but $c_1 = \langle (\text{Ref } G_{12}) \vdash G_1' \sim ? \rangle$, and the result holds.
2. \((\varepsilon_2 = \text{Ref} \ ?)\). Then \(c_2 = (\text{Ref} \ ?); (\text{Ref} \ ?)!\), and

\[
\begin{align*}
& c_1; c_2 = \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim \text{Ref} ? \rangle; (\text{Ref} ?); (\text{Ref} ?); (\text{Ref} ?)! \\
& \quad \rightarrow \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim \text{Ref} ? \rangle; (\text{Ref} ?)! \\
& \quad \rightarrow \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim \text{Ref} ? \rangle; \text{Ref} ?; (\text{Ref} ?)! \\
& = c_1
\end{align*}
\]

i.e. \(c_1; c_2 \rightarrow^* c_1\). We also know \(c_1\) (by Lemma 78), and \(\varepsilon_1 \circ^\varepsilon_2 = \langle (\text{Ref} G'_{12}) \cap (\text{Ref} ?) \rangle = (\text{Ref} G'_{12})\), but \(c_1 = \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim ? \rangle\), and the result holds.

3. \((\varepsilon_2 = \langle R \rangle); R \neq \text{Ref} \ ?\). Then \(c_2 = R?; R!\), and

\[
\begin{align*}
& c_1; c_2 = \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim \text{Ref} ? \rangle; (\text{Ref} ?); R?; R! \\
& \quad \rightarrow \langle (\text{Ref} G'_{12}) \vdash \text{Ref} G'_1 \sim \text{Ref} ? \rangle; \text{Fail} \\
& \quad \rightarrow^* \text{Fail}
\end{align*}
\]

We also know that \(\varepsilon_1 \circ^\varepsilon_2\) is not defined as \(\text{Ref} G'_{12}) \cap (\langle R \rangle)\) is undefined, and the result holds immediately. \(\Box\)

Lemma 80. If \(\varepsilon_1 \circ^\varepsilon_2\) is not defined then \(\forall \varepsilon'_2 \subseteq \varepsilon_2, \varepsilon_1 \circ^\varepsilon_2\) is not defined.

Proof. By induction on \(\varepsilon_1\) and \(\varepsilon_2\) subject to consistent transitivity being not defined. \(\Box\)

Lemma 81. If \(\varepsilon_1 \circ^\varepsilon_2\) is not defined then \(\forall \varepsilon'_1 \subseteq \varepsilon_1, \varepsilon_1 \circ^\varepsilon_2\) is not defined.

Proof. Direct by Lemma 80 and Lemma 77. \(\Box\)

Proposition 82. If \(G = G_1 \cap G_2\) is defined, then \(G \subseteq G_1\) and \(G \subseteq G_2\).

Proof. Straightforward induction on \(G_1 \cap G_2\). \(\Box\)

Proposition 83. If \((\text{Ref} G) \vdash \text{Ref} G_1 \sim \text{Ref} G_2\) then \(G \vdash G_1 \sim G_2\).

Proof. Straightforward induction on \((\text{Ref} G) \vdash \text{Ref} G_1 \sim \text{Ref} G_2\). \(\Box\)

Proposition 84. If \((G_1 \rightarrow G_2) \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}\) then \(G_1 \vdash G_{21} \sim G_{11}\).

Proof. Straightforward induction on \((G_1 \rightarrow G_2) \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}\). \(\Box\)

Proposition 85. If \((G_1 \rightarrow G_2) \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}\) then \(G_2 \vdash G_{12} \sim G_{22}\).

Proof. Straightforward induction on \((\text{Ref} G) \vdash \text{Ref} G_1 \sim \text{Ref} G_2\). \(\Box\)

Proposition 86 (Optimality). If \(\varepsilon = \varepsilon_1 \circ^\varepsilon_2\) is defined, then \(\pi_1(\varepsilon) \subseteq \pi_1(\varepsilon_1)\) and \(\pi_2(\varepsilon) \subseteq \pi_2(\varepsilon_2)\).

Proof. Direct by Lemma 82, as evidences can be represented as singletons. \(\Box\)

Lemma 87. \((c_1; c_2) \rightarrow^* c \land c; c_3 \rightarrow^* c' \) \iff \( (c_1; c_2); c_3 \rightarrow^* c' \)

Proof. Straightforward induction on \((c_1; c_2)\) and then induction on \(c_3\). \(\Box\)

Lemma 88. \((c_2; c_3) \rightarrow^* c \land c_1; c \rightarrow^* c' \) \iff \( (c_1; c_2); c_3 \rightarrow^* c' \)

Proof. For proving \((\Rightarrow)\), we use straightforward induction on \((c_2; c_3)\) and then induction on \(c_1\). For proving the other direction \((\Leftarrow)\), we use induction on \((c_1; c_2); c_3\). \(\Box\)

Proposition 89 (Associativity). Let \(\varepsilon_1 \vdash G_1 \sim G_2, \varepsilon_2 \vdash G_2 \sim G_3\) and \(\varepsilon_3 \vdash G_3 \sim G_4\). Then \((\varepsilon_1 \circ^\varepsilon_2) \circ^\varepsilon \varepsilon_3 \equiv \varepsilon_1 \circ^\varepsilon \varepsilon_2 \circ^\varepsilon \varepsilon_3\) or both are undefined.
Proof. By straightforward induction on evidences \(e_1, e_2,\) and \(e_3,\) noticing that if \(e_1 = (G_{12}), e_2 = (G_{23}), e_3 = (G_{34}),\) then it is equivalent to prove that \((G_{12} \cap G_{23}) \cap G_{34} = G_{12} \cap (G_{23} \cap G_{34})\) or both are undefined. We only present interesting cases.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{2})).\) Then the result is trivial as \((G) \circ^m (G) = (G).\)

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = ((G_{1}) \circ^m (?)) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3})).\) As \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3}), G_1 \cap G_2 \text{ not defined}).\) We know that \((e_1 \circ^m e_2) \circ^m e_3 = ((G_{1}) \circ^m (G_{2})) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) if \(G_1 \cap G_2\) is defined, and \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m e_2 = (G_{1}) \circ^m (G_{2}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{2}) = (G_{1}) \circ^m (G_{2}),\) if \(G_1 \cap G_2\) is defined, the result holds immediately.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3}), G_2 \cap G_3 \text{ not defined}).\) We know that \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m ((? \circ^m e_2) + G_{3}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{3}),\) if \(G_1 \cap G_2\) is not defined then the result holds immediately by next argument.

By Proposition 86, \((G_2 \cap G_3) \subseteq (G_2).\) Then by Proposition 80 \((G_1) \circ^m (G_2 \cap G_3)\) is not defined, and the result holds.

Case \((e_1 = (G_{1}), e_1 = (G_{2}), e_1 = (G_{3}), G_2 \cap G_3 \text{ not defined}).\) We know that \((e_1 \circ^m e_2) \circ^m e_3 = (G_{1}) \circ^m ((? \circ^m e_2) + G_{3}) = (G_{1}) \circ^m ((? \circ^m e_2) + G_{3}),\) if \(G_1 \cap G_2\) or \(G_2 \cap G_3\) are not defined then the result holds immediately by next argument.

Then by Proposition 86, \((G_1 \cap G_2) 
subseteq \) (G_1) and \(G_2 \cap G_3 \subseteq (G_3),\) therefore by Proposition 81 \((G_1 \cap G_2) \circ^m (G_3)\) is not defined, and by Proposition 80, \((G_1) \circ^m (G_2 \cap G_3)\) is not defined, and the result holds as both combinations of evidence fail, regardless if \(G_1 \cap G_2\) or \(G_2 \cap G_3\) are defined or not.

Case \((e_1 = (\text{Ref } G_1), e_1 = (\text{Ref } G_2), e_1 = (\text{Ref } G_3)).\) Notice that \((\text{Ref } G_1) \circ^m (\text{Ref } G_2) \circ^m (\text{Ref } G_3) = \text{Ref } G_1) \circ^m (\text{Ref } G_2) \circ^m (\text{Ref } G_3)\) is defined if and only if \((G_1) \circ^m ((? \circ^m e_2) + G_{2}) = (G_1) \circ^m ((? \circ^m e_2) + G_{2}),\) if \(G_1 \cap G_2\) or \(G_2 \cap G_3\) are not defined then the result holds immediately by next argument.

Then the result holds immediately as by induction hypothesis \((G_1) \circ^m (G_2) = (G_1) \circ^m (G_2)\) or both are undefined.

Case \((e_1 = (G_{11} \rightarrow G_{12}), e_1 = (G_{21} \rightarrow G_{22}), e_1 = (G_{31} \rightarrow G_{32})).\) Analogous to the previous case but using inversion Lemmas 85 and 84. □

Proposition 90. Let \(c_1 \vdash G_1 \Rightarrow G_2, c_2 \vdash G_2 \Rightarrow G_3\) and \(c_3 \vdash G_3 \Rightarrow G_4.\) Then either

- \((c_1; c_2): c_1 \rightarrow^* \text{ c and } c_1; (c_2; c_3) \rightarrow^* \text{ c, or}
- (c_1; c_2): (c_1; c_3) \rightarrow^* \text{ Fail and } (c_2; c_3) \rightarrow^* \text{ Fail}

Proof. By induction on \(c_1, c_2,\) and \(c_3.\) Alternatively, by Proposition 79 and Proposition 89, noticing that \(c_1 \vdash (G_1 \sim G_{i+1})\) for some \(e_i.\) □

Proposition 91 (Confluence). Let \(c_1 \vdash G_1 \Rightarrow G_2, \text{ nm } c_1, c_2 \vdash G_2 \Rightarrow G_3,\) and \(\text{ nm } c_2.\) If \(c_1; c_2 \rightarrow^* c_1'\) and \(\text{ nm } c_1',\) and \(c_1; c_2 \rightarrow^* c_2'\) and \(\text{ nm } c_1,\) then \(c_1' = c_2'.\)
**Proof.** By a lengthy induction on $c_1 \vdash G_1 \Rightarrow G_2 \land nm$ $c_1$ and then induction on $c_2 \vdash G_2 \Rightarrow G_3 \land nm$ $c_2$. We show a representative case:

**Case** $(c_1 = (c_1 →)?$; $c_11 \rightarrow c_12; (c_1 →)!)$. where $nm$ $c_11$, and $G_2 = ?$. We now proceed by induction on $c_2 \vdash ? \Rightarrow G_3 \land nm$ $c_2$:

- **(Case** $c_2 = $ **Fail**). Then

  \[(c_1 →)?; c_11 → c_12; (c_1 →)!!; \text{ Fail} \mapsto (c_1 →)??; c_11 → c_12; \text{ Fail} (\text{as } nm (c_1 →)??; c_11 → c_12) \]

  \[\mapsto (c_1 →)??; \text{ Fail (as } nm (c_1 →)??) \]

  \[\mapsto \text{ Fail (as } nm (c_1 →)??) \]

  and as $nm$ **Fail**, the result holds.

- **(Case** $c_2 = $ **Ref** ?). Then

  \[(c_1 →)?; c_11 → c_12; (c_1 →)!!; \text{ Ref } ?? \mapsto (c_1 →)??; c_11 → c_12; \text{ Fail (as } nm (c_1 →)??; c_11 → c_12) \]

  \[\mapsto (c_1 →)??; \text{ Fail (as } nm (c_1 →)??) \]

  \[\mapsto \text{ Fail (as } nm (c_1 →)??) \]

  and as $nm (c_1 →)??; c_11 → c_12$, the result holds.

- **(Case** $c_2 = $ **B** ??). Similar to $c_2 = $ **Ref** ?? case.

- **(Case** $c_2 = (c_1 →)?$; $c_12 \rightarrow c_22; (c_1 →)!!)$. Then

  \[(c_1 →)?; c_11 → c_12; (c_1 →)!!; (c_1 →)??; c_21 → c_22; (c_1 →)!! \]

  \[\mapsto (c_1 →)??; c_11 → c_12; ?_? → ?; c_21 → c_22; (c_1 →)!! \]

  as $nm (c_1 →)??; c_11 → c_12$ and $nm c_21 → c_22; (c_1 →)!!$. Then

  \[(c_1 →)?; c_11 → c_12; i_? → ?; c_21 → c_22; (c_1 →)!! \]

  \[\mapsto (c_1 →)??; c_11 → c_12; c_21 → c_22; (c_1 →)!! \]

  Notice that we get to the same result either if we reduce the sub coercion $c_11 → c_12; ?_? → ?$, or $i_? → ?; c_21 → c_22$. Then,

  \[(c_1 →)?; c_11 → c_12; c_21 → c_22; (c_1 →)!! \]

  \[\mapsto (c_1 →)??; (c_21; c_11) → (c_12; c_22); (c_1 →)!! \]

  as $nm (c_1 →)??$ and $nm (c_1 →)!!$. Now we apply induction hypotheses:

  (1) if $c_21; c_11 \mapsto c_12$ and $c_21; c_11 \mapsto c_12'$, $nm c_11$, and $nm c_12$, then $c_11 = c_12'$.

  (2) if $c_12; c_22 \mapsto c_12$ and $c_12; c_22 \mapsto c_12'$, $nm c_21$, and $nm c_22$, then $c_21 = c_22$. Then

  \[(c_1 →)?; (c_21; c_11) → (c_12; c_22); (c_1 →)!! \]

  \[\mapsto (c_1 →)?; c_11 → c_12'; (c_1 →)!! \]

  But as $nm c_11$ and $nm c_12$, then $nm (c_1 →)?; c_11 → c_12'; (c_1 →)!!$ and the result follows.

- **(Case** $c_2 = (c_1 →)?$; $c_21 → c_22$). Then

  \[(c_1 →)?; c_11 → c_12; (c_1 →)!!; (c_1 →)??; c_21 → c_22 \]

  \[\mapsto (c_1 →)?; c_11 → c_12; i_? → ?; c_21 → c_22 \]

  as $nm (c_1 →)??; c_11 → c_12$ and $nm c_21 → c_22$. Then
\((\rightarrow \rightarrow?); c_{11} \rightarrow c_{12}; \iota \rightarrow?; c_{21} \rightarrow c_{22}\)

\(\llbracket (\rightarrow \rightarrow?); c_{11} \rightarrow c_{12}; c_{21} \rightarrow c_{22}\)

Notice that we get to the same result either if we reduce the sub coercion \(c_{11} \rightarrow c_{12}; \iota \rightarrow?; c_{21} \rightarrow c_{22}\). Then,

\[(\rightarrow \rightarrow?); c_{11} \rightarrow c_{12}; c_{21} \rightarrow c_{22}\]

\(\llbracket (\rightarrow \rightarrow?); (c_{21}; c_{11}) \rightarrow (c_{12}; c_{22})\)

as \(nm \rightarrow ??\). Now we apply induction hypotheses:

1. If \(c_{21}; c_{11} \llbracket \rightarrow c_{11} \circ nm \rightarrow c_{12}, \) and \(c_{21} \llbracket \rightarrow c_{12}\), then \(c_{11} = c_{12}\).
2. If \(c_{12}; c_{22} \llbracket \rightarrow c_{21} \circ c_{12} \llbracket \rightarrow c_{22}, \) and \(c_{21} \llbracket \rightarrow c_{22}\), then \(c_{21} = c_{22}\).

Then

\[(\rightarrow \rightarrow?); (c_{21}; c_{11}) \rightarrow (c_{12}; c_{22})\]

\(\llbracket (\rightarrow \rightarrow?); c_{11} \rightarrow c_{12}\)

But as \(nm c_{11}\) and \(nm c_{12}\), then \(nm (\rightarrow \rightarrow?); c_{11} \rightarrow c_{12}\) and the result follows.

- (Case \(c_{2} = \text{Ref} ??; \text{Ref} c_{21} c_{22}; \text{Ref} ??\)). Analogous to the function case.
- (Case \(c_{2} = \text{Ref} ??; \text{Ref} c_{21} c_{22}\)). Analogous to the function case. □

Lemma 92. \(e_{1} t_{1} @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e'_{1} t'_{1} @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu' \) if and only if \(e_{1} t_{1} :: G_{1} \rightarrow G_{2} | \mu \llbracket \rightarrow e'_{1} t'_{1} :: G_{1} \rightarrow G_{2} | \mu'\).

**Proof.** We start by proving \(\Rightarrow\) by case analysis on \(t_{1}\) (the other direction is analogous).

- If \(t_{1} = e_{3} t_{3} :: G_{3}\) where \(e_{1}(e_{3} t_{3} :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e_{1}(e'_{3} t'_{3} :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu', \) then it is easy to see that \(e_{1}(e_{3} t_{3} :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e_{1}(e'_{3} t'_{3} :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu', \) and the result holds.
- If \(t_{1} = e_{3} u :: G_{3}\) where \(e_{1}(e_{3} u :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e'_{1} u :: G_{1} \rightarrow G_{2} | \mu \llbracket \rightarrow e'_{1} u :: G_{1} \rightarrow G_{2} | \mu, \) and \(e'_{1} = e_{1} \circ e_{1},\) then also \(e_{1}(e_{3} u :: G_{3}) @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e'_{1} u :: G_{1} \rightarrow G_{2} | \mu \llbracket \rightarrow e'_{1} u :: G_{1} \rightarrow G_{2} | \mu, \) and the result holds. □

Lemma 93. \(e_{1} t_{1} @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu \llbracket \rightarrow e_{1} t_{1} @^{G_{1} \rightarrow G_{2}} e_{2} t_{2} | \mu' \) if and only if \(e_{2} t_{2} :: G_{1} | \mu \llbracket \rightarrow e'_{2} t'_{2} :: G_{1} | \mu'.\)

**Proof.** Analogous to Lemma 92. □

Lemma 94. \(e_{1} t_{1} :: G_{1} e_{2} t_{2} | \mu \llbracket \rightarrow e_{1} t'_{1} :: G_{1} e_{2} t_{2} | \mu' \) if and only if \(e_{1} t_{1} :: \text{Ref} G_{3} | \mu \llbracket \rightarrow e_{1} t'_{1} :: \text{Ref} G_{3} | \mu'.\)

**Proof.** Similar to Lemma 92. □

Lemma 95. \(e_{1} t_{1} :: G_{1} e_{2} t_{2} | \mu \llbracket \rightarrow e_{1} t_{1} :: G_{1} e'_{2} t'_{2} | \mu' \) if and only if \(e_{2} t_{2} :: G_{3} | \mu \llbracket \rightarrow e_{2} t'_{2} :: G_{3} | \mu'.\)

**Proof.** Analogous to Lemma 94. □

Lemma 96. \(\text{ref}^{G} e t | \mu \llbracket \rightarrow \text{ref}^{G} e' t' | \mu' \) if and only if \(e t :: G | \mu \llbracket \rightarrow e' t' :: G | \mu'.\)

**Proof.** Similar to Lemma 92. □

Lemma 97. \(\text{ref}^{G} e t | \mu \llbracket \rightarrow ! e' t' | \mu' \) if and only if \(e t :: \text{Ref} G | \mu \llbracket \rightarrow e' t' :: \text{Ref} G | \mu'.\)

**Proof.** Similar to Lemma 92. □

Lemma 98. \(e_{1} t_{1} \text{then} e_{2} t_{2} \text{else} e_{3} t_{3} | \mu \llbracket \rightarrow e'_{1} t'_{1} \text{then} e_{2} t_{2} \text{else} e_{3} t_{3} | \mu' \) if and only if \(e_{1} t_{1} :: \text{Bool} | \mu \llbracket \rightarrow e'_{1} t'_{1} :: \text{Bool} | \mu'.\)

**Proof.** Similar to Lemma 92. □

Lemma 99. \(\text{if} \ c_{1} \rightarrow c_{2} = (\llbracket e \mapsto G_{1} \rightarrow G_{2} \sim G_{1} \rightarrow G_{2} \rrbracket,\) then \(c_{1} = \llbracket \text{idom}(e) \mapsto G_{1} \sim G'_{1} \rrbracket\) and \(c_{2} = \llbracket \text{icod}(e) \mapsto G'_{2} \sim G_{2} \rrbracket\) which is equal to \(c_{1} \rightarrow c_{2},\) and the result holds immediately. □
Lemma 100. If $\text{Ref } c_1 \ c_2 = \{e \vdash \text{Ref } G' \leadsto \text{Ref } G\}$, then $c_1 = \{\text{iref } e\} \vdash \text{Ref } G \leadsto \text{Ref } G'$ and $c_2 = \{\text{iref } e\} \vdash \text{Ref } G' \leadsto \text{Ref } G$.

Proof. By definition of the map function between evidence augmented consistent judgments and coercions we know that $(e \vdash \text{Ref } G' \leadsto \text{Ref } G) = \text{Ref } \{\text{iref } e\} \vdash \text{Ref } G' \leadsto \text{G} \text{Ref } e \vdash \text{Ref } G'$ which is equal to $\text{Ref } c_1 \ c_2$, and the result holds immediately. □

Lemma 101. If $t_1 \approx t_2, v_1 \in T[G]$, and $v_1 \approx v_2$, then $t_1[v_1/x^G] \approx t_2[v_2/x]$.

Proof. By induction on the derivation $t_1 \approx t_2$. □

Lemma 102. Consider $(G) \vdash G \leadsto G$, then

1. $\forall \epsilon \vdash G' \leadsto G, (G) \circ \epsilon = \epsilon$, and
2. $\forall \epsilon \vdash G' \rightarrow G, \epsilon \circ (G) = \epsilon$.

Proof. By induction on evidence augmented consistent judgment $(G) \vdash G \leadsto G$. □

Lemma 103 (Weak bisimulation between $\lambda^+\text{ref}$ and HCC$^+$). If $t_1 \in T[G], \Sigma \vdash t_2 : G, \mu_1 \vdash \Sigma, \mu_1 \approx \mu_2$, and $t_1 \approx t_2$, then

1. If $t_1 \mid \mu_1 \mapsto t'_1 \mid \mu'_1$, then $t_2 \mid \mu_2 \mapsto t'_2 \mid \mu'_2$ such that $t'_1 \approx t'_2$ and $\mu'_1 \approx \mu'_2$.
2. If $t_2 \mid \mu_2 \mapsto t'_2 \mid \mu'_2$, then $\exists \mu, 0 \leq j \leq 2, t'_2 \mid \mu'_2 \mapsto t'_2 \mid \mu'_2$ and $t_1 \mid \mu_1 \mapsto t'_1 \mid \mu'_1$ such that $t'_1 \approx t'_2$ and $\mu'_1 \approx \mu'_2$.

Proof. 1. We proceed by induction on $t_1 \mid \mu_1 \mapsto t'_1 \mid \mu'_1$.

Case $(\epsilon_1(\lambda x^G. t'_1) \circ G \circ t_1, \epsilon_2 u_1) \mid \mu_1 \mapsto (\epsilon_1' t'_1[\epsilon_2 u_1 :: G_{11}/X^{G_{11}}] : G_{12} :: \mu_1)$. By Lemma 99, $c_1 = \{\text{idom}(\epsilon_1) \vdash G_{12} \sim G_{11}\}$, and $c_2 = \{\text{idom}(\epsilon_2) \vdash \text{idom}(\epsilon_1) \sim G_{12}\}$.

If $u_{22} = c_2 u_{22}$, where $c_2 = \{\text{id } t_2 \vdash G_{12} \sim G_{11}\}$ and $u_{11} \approx u_{22}$, then by Proposition 79, $c'_2 = c_2$ and $c_1 = \{\text{id } t_2 \vdash G_{12} \sim G_{11}\}$. Then if we assume $c'_1 \neq c_1$ (the other case is analogous)

$$((c_1 \rightarrow c_2) (\lambda x^G t_2)) c_2 u_{22} | \mu_2 \mapsto c_2 (((\lambda x^G t_1) c_1 (c_2 u_{22})) | \mu_2)$$

But we know that $t'_1 \approx t'_2$, and that by (b::eq) $\epsilon_2 u_1 :: G_{11} \approx c'_1 u_{22}$, by Lemma 101 and (b::eq), $\text{idom}(\epsilon_1) t'_1[\epsilon_2 u_1 :: G_{11}/X^{G_{11}}] : G_{12} \approx c_1 (t_2 c_2 u_{22}/x)$ and the result holds.

If $u_{22} = u_{22}$ where $u_1 \approx u_{22}$, then by (b::id) $\epsilon_2 = (G_1)$ and $u_1 \in T[G_1]$, therefore $(G) \vdash G \sim G_1$. Therefore by Lemma 102, $\epsilon_2 = \text{idom}(\epsilon_1)$. Then

$$(((c_1 \rightarrow c_2) (\lambda x^G t_1)) u_{22} | \mu_2 \mapsto c_1 (((\lambda x^G t_2) c_1 (u_{22})) | \mu_2)$$

But we know that $t'_1 \approx t'_2$, and that by (b::eq) $\epsilon_2 u_1 :: G_{11} \approx c_1 u_{22}$, by Lemma 101 and (b::eq), $\text{idom}(\epsilon_1) t'_1[\epsilon_2 u_1 :: G_{11}/X^{G_{11}}] : G_{12} \approx c_1 (c_2 u_{22}/x)$ and the result holds.

If $t_1 \approx t_2 = (\lambda x^G t_2) v_{22}$. Then $G_1 \vdash G_1, t_1 \in T[G_2], \epsilon_1 = (G_1 \rightarrow G_2)$, and $\epsilon_1 \vdash (G_1 \rightarrow G_2) \vdash G_1 \rightarrow G_2 \rightarrow G_1 \rightarrow G_2$. By the inversion lemmas $\text{idom}(\epsilon_2) = (G_1) \vdash G_1 \rightarrow G_1$ and $\text{idom}(\epsilon_2) = (G_1) \vdash G_1 \rightarrow G_2$. But we know that $t'_1 \approx t'_2$, therefore by Lemma 102, $\epsilon'_2 = \epsilon_2$. Finally by (b::id) and as $\epsilon_2 u_1 :: G_1 \approx v_{22}$, by Lemma 101, $\text{idom}(\epsilon_1) t'_1[\epsilon_2 u_1 :: G_{11}/X^{G_{11}}] : G_{12} \approx (t'_2 v_{22}/x)$ and the result holds.

Case $(\text{ref } G) u_1 | \mu_1 \mapsto o^{G_1} | \mu_1 [o^{G_1} \vdash u_1 :: G_{11}]$. We know by (b::ref) that $t_2 = \text{ref } v_2$, for some $v_2$ such that $e u_1 :: G_1 \approx v_2$. But ref $v_2 | \mu_2 \mapsto o | \mu_2 [o \vdash v_2]$. As $o^{G_1} \approx o$, and $\mu_1 \approx \mu_2$, we only have to prove that $e u_1 :: G_1 \approx v_2$, but we already know that by (b::ref), and the result holds immediately.

Case $(\text{ref } e v_1) | \mu_1 \mapsto \text{iref } e v_1 :: G | \mu_1$. Where $\mu_1 (x^{G_2}) = v_1$. By inspection of (b!), we know that $t_2 = !v_2$, for some $v_2$ such that $e v_1 :: G_2 \approx v_2$. We proceed by case analysis on $v_2$. 
• If \( v_2 = (\text{Ref } c_1 c_2) o \). Where \( \text{Ref } c_1 c_2 = \{ e \vdash \text{Ref } G_2 \sim \text{Ref } G \}, \) and \( o^{G_2} \approx o \). By Lemma 100, \( c_1 = \{ \text{iref}(e) \vdash G \sim G_2 \}, \) and \( c_2 = (\text{iref}(e) \vdash G_2 \sim G) \). Then

\[
\vdash (\text{Ref } c_1 c_2) o \mid \mu_2 \quad \rightarrow \quad c_2 v_2 \mid \mu_2
\]

where \( \mu_2(o) = v_2 \), and \( v_2 \approx v_1 \). The result follows by applying (b::eq).

• If \( v_2 = 0 \). Then \( G_2 = G \) and \( e = (\text{Ref } G) \). By the inversion lemma on evidence \( \text{iref}(e) = (G) \sim G \). Then \( ! 0 \mid \mu_2 \quad \rightarrow \quad v_2 \mid \mu_2 \), where \( \mu_2(o) = v_2 \), and \( v_2 \approx v_1 \). By (b::id) we know that \( (G) v_1 \sim G \approx v_2 \) and the result holds.

**Case** \( (e_1 G_1^{c_1} := G_2 u_{12} \mid \mu_1 \quad \rightarrow \quad \text{unit} \mid \mu_1' \) **Case** \( (e_1 G_1^{c_1} := G_2 u_{12} \mid \mu_1 \quad \rightarrow \quad \text{unit} \mid \mu_1' \.

where \( \mu_1' = \mu_1[0^{G_1} \mapsto e_2' u_{12} :: G_1] \), and \( e_2' = 0 \circ \text{iref}(e_1) \). By inspection of (b::eq), we know that \( t_2 = t_{21} := t_{22} \), for some \( t_{21}, t_{22} \) such that \( e_1 G_1^{c_1} \vdash \text{Ref } G_3 \sim t_{21} \) and \( e_2 u_{12} :: G_3 \approx t_{22} \). We proceed by case analysis on \( t_{21} := t_{22} \).

• If \( t_{21} := t_{22} = ((\text{Ref } c_1 c_2) o) v_{22} \). Where \( \text{Ref } c_1 c_2 = \{ e \vdash \text{Ref } G_1 \sim \text{Ref } G_3 \} \), and \( c_2 = (\text{iref}(e_1) \vdash G_1 \sim G_3) \).

If \( v_{22} = c_4 u_{22} \), where \( c_4 = \{ (e \vdash G_2 \sim G_3) \) and \( u_1 \approx u_{22} \), then by Proposition 79, \( c_4 = c_4 \); \( c_1 = \{ (e \vdash G_2 \sim G_3) \). Then if we assume \( c_1 \neq c_0 \) (the other case is analogous)

\[
((\text{Ref } c_1 c_2) o) := c_4 u_{22} \mid \mu_2 \quad \rightarrow \quad 0 := c_1 (c_4 u_{22}) \mid \mu_2
\]

but we know that \( \text{unit} \approx \text{unit} \), and that by (b::eq) \( e_2' u_1 :: G_1 \approx c_1 u_{22} \), and so \( \mu_1[0^{G_1} \mapsto e_2' u_{12} :: G_1] \approx \mu_2[0 \mapsto c_1 u_{22}] \), and the result holds.

If \( v_{22} = u_{22} \) where \( u_1 \approx u_{22} \), then by (b::id) \( e_2 = (G_3) \) and \( u_1 \in \mathbb{T}[G_3] \), therefore \( (G_3) \sim G_3 \sim G_3 \). Therefore by Lemma 102, \( e_2' = \text{iref}(e_1) \).

Then

\[
((\text{Ref } c_1 c_2) o) := u_{22} \mid \mu_2 \quad \rightarrow \quad 0 := c_1 (u_{22}) \mid \mu_2
\]

but we know that \( \text{unit} \approx \text{unit} \), and that by (b::eq) \( e_2' u_1 :: G_1 \approx c_1 u_{22} \), and so \( \mu_1[0^{G_1} \mapsto e_2' u_{12} :: G_1] \approx \mu_2[0 \mapsto c_1 u_{22}] \), and the result holds.

**Case** \( (e_1 b \vdash e_2 t_{12} \quad \rightarrow \quad e_3 t_{13} \mid \mu_1 \rightarrow e_2 t_{12} :: G \mid \mu_1) \). Where \( b = \text{true} \) and \( e_1 = (\text{Bool}) \).

By inspection of (b), we know that \( t_2 = t_2 \) and \( t_2 \) else \( t_2 \), for some \( t_{21}, t_{22}, t_{32} \), such that \( (\text{Bool})b := \text{Bool} \sim t_{21}, e_2 t_{12} :: G \sim t_{22} \), and \( e_3 t_{13} :: G \sim t_{23} \).

By (b::eq) or (b::id) and (bb), we know that either \( t_{21} = \text{true} \) or \( t_{21} = (\text{true}) \).

Let us assume \( t_{21} = \text{true} \) (the other case is analogous modulo one extra step of evaluation).

Then \( t_2 \mid \mu_2 \quad \rightarrow \quad t_{23} \mid \mu_2 \), but \( e_2 t_{12} :: G \sim t_{22} \) and the result holds immediately.

**Case** \( (e_1 b \vdash e_2 t_{12} \quad \rightarrow \quad e_3 t_{13} \mid \mu_1 \rightarrow e_2 t_{13} :: G \mid \mu_1) \). Where \( b = \text{false} \) and \( e_1 = (\text{Bool}) \).

By inspection of (b), we know that \( t_2 = t_{21} \) and \( t_2 \) else \( t_2 \), for some \( t_{21}, t_{22}, t_{23} \), such that \( (\text{ Bool })b :: \text{Bool} \sim t_{21}, e_2 t_{12} :: G \sim t_{22} \), and \( e_3 t_{13} :: G \sim t_{23} \).

By (b::eq) or (b::id) and (bb), we know that either \( t_{21} = \text{false} \) or \( t_{21} = (\text{false}) \).

Let us assume \( t_{21} = \text{false} \) (the other case is analogous modulo one extra step of evaluation).

Then \( t_2 \mid \mu_2 \quad \rightarrow \quad t_{23} \mid \mu_2 \), but \( e_2 t_{13} :: G \sim t_{23} \) and the result holds immediately.

**Case** \( (B_1 b_1) \oplus (B_2 b_2) \mid \mu_1 \rightarrow b_1 \mid \mu_1 \) \( \oplus \) \( b_2 \). Where \( b_1 = b_1 \oplus b_2 \). Then either \( t_2 = t_{21} \oplus t_{22} \).

By (b::eq) or (b::id) and (bb), \( t_{21} = c_1[1] \) or \( t_{21} = \text{false} \), and \( t_{22} = \text{false} \).

Let us assume \( t_{21} = \text{false} \) and \( t_{22} = \text{false} \) (the other cases is analogous modulo one or two extra steps of evaluation).

Then \( b_1 \oplus b_2 = b_3 \), where \( b_3 = b_1 \oplus b_2 \), and the result holds immediately by (b::b).

**Case** \( (e_1 t_{11} \oplus G_1 \sim G_2 \mid \mu_1 \rightarrow e_1' t_{11} \oplus G_1 \sim G_2 \mid \mu_1' \).

Then by (b::app) \( t_2 = t_{21} \oplus t_{22} \).

By Lemma 92 we know that \( e_1 t_{11} :: G_1 \rightarrow G_2 \mid \mu_1 \rightarrow e_1' t_{11} :: G_1 \rightarrow G_2 \mid \mu_1' \).

Also by (b::app) we know that \( e_1 t_{11} :: G_1 \rightarrow G_2 \sim t_{21} \).

Then by induction hypothesis we know that \( t_{21} \mid \mu_2 \rightarrow^{*} t_{21} \mid \mu_2 \), and that \( e_1' t_{11} :: G_1 \rightarrow G_2 \sim t_{12} \) and \( \mu_1' \sim \mu_2' \).

The result follows directly by (b::app).
Case \((ε₁u₁₁ \oplus G₁ \rightarrow G₂ \varepsilon₂t₁₂ \mid μ₁) \rightarrow ε₁u₁₁ \oplus G₁ \rightarrow G₂ \varepsilon₂t₁₂ \mid μ₁').\) Analogous to previous case but using Lemma 93

Case \((ε₁t₁₁ \rightarrow ε₂t₁₂ \mid \varepsilon₁t₁₁ \mid μ₁) \rightarrow ε₁t₁₁ \rightarrow ε₂t₁₂ \mid \varepsilon₁t₁₁ \mid μ₁').\) Then by \((nbf)\) \(t₂ = \varepsilon₁t₁₁ \oplus \varepsilon₂t₂₂\). By Lemma 98 we know that \(ε₁t₁₁ \oplus \text{Bool} \mid μ₁ \rightarrow ε₁t₁₁ \oplus \text{Bool} \mid μ₁'.\) Also by \((nbf)\) we know that \(ε₁t₁₁ \oplus \text{Bool} \approx ε₂t₁₁.\) Then by induction hypothesis we know that \(t₂ \mid μ₂ \rightarrow ε₂t₂₁ \mid μ₂',\) and that \(ε₁t₁₁ \oplus \text{Bool} \approx ε₂t₁₁ \mid μ₁ \approx μ₂.\) The result follows directly by \((nbf)\).

Case \((ε₁t₁₁ \oplus G₁ \rightarrow t₂ \mid μ₁) \rightarrow ε₁t₁₁ \oplus G₁ \rightarrow t₂ \mid μ₁').\) Analogous to previous case but using Lemma 95

Case \((\text{ref}^G \mid ε₁u₁ \mid μ₁) \rightarrow \text{ref}^G \mid ε₁u₁ \mid μ₁').\) Then by \((nbf)\) \(t₂ = \text{ref} \varepsilon₂t₂₁.\) By Lemma 96 we know that \(ε₁u₁ \oplus G' \mid μ₁ \rightarrow ε₁u₁ \oplus G' \mid μ₁'.\) Also by \((nbf)\) we know that \(ε₁u₁ \oplus G' \approx \varepsilon₂u₂₁.\) Then by induction hypothesis we know that \(t₂ \mid μ₂ \rightarrow ε₂t₂₁ \mid μ₂',\) and that \(ε₁u₁ \oplus G' \approx ε₂u₂₁ \mid μ₁ \approx μ₂.\) The result follows directly by \((nbf)\).

Case \((\varepsilon^G \mid ε₁t₁₁ \mid μ₁) \rightarrow \varepsilon^G \mid ε₁t₁₁ \mid μ₁').\) Then by \((nbf)\) \(t₂ = \varepsilon₂t₂₁.\) By Lemma 97 we know that \(ε₁t₁₁ \oplus \text{Ref} G' \mid μ₁ \rightarrow ε₁t₁₁ \oplus \text{Ref} G' \mid μ₁'.\) Also by \((nbf)\) we know that \(ε₁t₁₁ \oplus \text{Ref} G' \approx ε₂t₂₁.\) Then by induction hypothesis we know that \(t₂ \mid μ₂ \rightarrow ε₂t₂₁ \mid μ₂',\) and that \(ε₁t₁₁ \oplus \text{Ref} G' \approx ε₂t₂₁ \mid μ₁ \approx μ₂.\) The result follows directly by \((nbf)\).

Case \((ε₁t₁₁ \oplus G \mid μ₁) \rightarrow ε₁t₁₁ \oplus G \mid μ₁').\) Without losing generality let us assume that \(ε₁t₁₁ \oplus G = ε₁(...(εn₁₁ \oplus Gn)...) \oplus G,\) where \(t₁n \in T \oplus \text{Gn} - 1\) is not an assigned term.

Then by \((nbf)\), \((nbf)\) and \((nbf)\), either \(t₂ = c₁(...(cₘtₘ)...)(m ≤ n)\) or \(t₂ = t₂₁,\) where \(t₂m\) and \(t₂₁\) are not coerced terms, such that \(t₁n \approx t₂m\) or \(t₁n \approx t₂₁\) respectively.

Let us assume that \(t₂ = c₁(...(cₘtₘ)...)\) (the other case is analogous to the second subcase below). Then we know that \(c₁(...(cₘtₘ)...) \rightarrow c₁t₂m,\) where \(cₘ,cₙ,...,c₁ \rightarrow \text{c₁} \) and \(n ≠ m.\) Also by repeatedly applying \((nbf)\) and \((nbf)\), \(ε₁(...(εn₁₁ \oplus Gn)...) \oplus G \approx c₁t₂m.\) Additionally, by inspection of \((nbf)\) and \((nbf)\), \(∃c₁',c₂,...,c₁n,\) such that \(c₁n = \{c₁ \mid G₁n+1 \sim G₁\} \), and that \(c₁n;...;c₁₁ \rightarrow ε₁^G.\)

We now proceed by case analysis on \(t₁n.\)

- If \(t₁n = u₁₁.\) Then \(ε₁(...(εn₁₁ \oplus Gn)...) \oplus G = ε₁(...(εn₁₁ \oplus Gn)...) \oplus G,\) where \(εn₁₁ = (εn \oplus εn) - 1.\) Then \(c₁n;\) \(c₁n \rightarrow c₁n - 1,\) for some \(c₁n - 1,\) therefore by Lemma 79, \(c₁n - 1 = \{c₁n - 1 \mid G₁n+1 \sim G₁n - 1\}.\) Then by \((nbf)\), \(c₁n - 1 \rightarrow c₁n - 1-u₁₁.\) Then the result holds by using \((nbf)\) and \((nbf)\) repeatedly and using \(c₁,1,...,c₁n - 1; c₁n - 1\) and Lemma 90.

- If \(t₁n\) is not a simple value, and therefore \(t₁n \mid μ₁ \rightarrow t₁n \mid μ₁'.\) By induction hypothesis \(t₂m \mid μ₂ \rightarrow t₂m \mid μ₂',\) such that \(t₁n \approx t₂m\) and \(μ₁ \approx μ₂.\) Therefore by \((nbf)\) and \((nbf)\), \(ε₁(...(εn₁₁ \oplus Gn)...) \oplus G \approx c₁t₂m \text{ and the result holds.}\)

The proof of \((nbf)\) is similar but choosing sometimes \(j = 1\) or \(j = 2\) in cases for application, dereference or assignment. □

**Lemma 104.** If \(c = \{b \rightarrow G' \sim G\},\) then \(nm c.\)

**Proof.** Direct by induction on \(ε \rightarrow G' \sim G\) and definition of \(\{\}(\text{Fig. 12).}\)

**Lemma 105.** If \(t₁ \in T \mid G; ε; μ \vdash t₂ \mid G, \mu₂ \vdash ε; \vDash t₁ \mid μ₁ \vdash t₂,\) then \(t₁ \mid μ₁ \approx t₂ \mid μ₂.\)

**Proof.**

Case \((⇒).\) We proceed by induction on \(t₁ \mid μ₁ \rightarrow t₂ \mid μ₂.\)

Case \((t₁ = v₁).\) As \(t₁ \approx t₂\) and by Lemma 15, then \(t₂ \mid μ₂ \rightarrow t₂ \mid μ₂ \) and \(v₁ \approx t₂ \mid μ₂ \) and \(v₁ \approx t₂ \mid μ₂ \). If \(v₁ = u,\) then the result holds immediately by inspection of \((bb), (bl),\) and \((bo).\) If \(v₁ = εu \oplus G \) then either by \((nbf)\) \(t₂ = u \) and the result holds, or by \((nbf)\) \(t₂ = c₁u,\) where \(c = \{b \rightarrow G_u \sim G\} \) (and therefore \(c \neq \text{Fail}\) and by Lemma 104 the result holds.

Case \((t₁ \mid μ₁ \rightarrow t₁ \mid μ₁' \) and \(t₁ \mid μ₁' \rightarrow t₂ \mid μ₂ \) and \(t₁ \approx t₂ \mid μ₂ \).) By Lemma 15 then \(t₂ \mid μ₂ \rightarrow t₂ \mid μ₂ \) and \(t₁ \approx t₂ \mid μ₂ \) and \(t₁ \approx t₂ \mid μ₂ \). Then by induction hypothesis, \(t₂ \mid μ₂ \downarrow \) and therefore \(t₂ \mid μ₂ \downarrow \) and the result holds.

Case \((⇒).\) We proceed similarly by induction on \(t₂ \mid μ₂ \rightarrow t₂ \mid μ₂ \).
Case $t_2 = v_2$. Similar to the $(t_1 = v_1)$ case.

Case $t_2 \mid_\mu_2 \mapsto^k t'_2 \mid \mu''_2$ and $t'_2 \mid \mu''_2 \mapsto^{n-k} v_2 \mid \mu'_2$. By Lemma 15 we know that there exists some $j \in \{1, 2, 3\}$, such that $t_1 \mid \mu_1 \mapsto^j t'_1 \mid \mu''_1$ and $t'_1 \mid \mu''_1 \mapsto t_2$ and $\mu' \approx \mu_2$. We choose $k = j$, and by induction hypothesis, $t'_1 \mid \mu''_1 \mapsto$, and therefore $t_1 \mid \mu_1 \mapsto$ and the result holds. \hfill $\square$

Lemma 106. If $t_1 \in T[G], \varepsilon, \Sigma \vdash H : G, \mu_2 \vdash \Sigma, \mu_1 \approx \mu_2$, and $t_1 \approx t_2$, then:

1. If $t_1 \mid \mu_1 \mapsto \text{error}$, then $t_2 \mid \mu_2 \mapsto* \text{error}$.
2. If $t_2 \mid \mu_2 \mapsto r, \text{for some } r = t_2 \mid \mu' \text{ or } r = \text{error}$, then $\exists j, 0 \leq j \leq 2. \text{If } r \mapsto j \text{ error}, \text{ then } t_1 \mid \mu_1 \mapsto* \text{error}$

Proof. 1. We proceed by induction on $t_1 \mid \mu_1 \mapsto \text{error}$.

Case $(\varepsilon_1 (\lambda x^{G_1} \cdot t'_1) \circ \overline{G_2} \circ u_1) \mid \mu_1 \mapsto \text{error})$. Where $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is not defined. By inspection of (bapp), we know that $t_2 = t_2, t_2$, for some $t_2, t_2$ such that $\varepsilon_1 (\lambda x^{G_1} \cdot t'_1) :: G_1 \rightarrow G_2 \approx t_2, t_2$. We proceed by case analysis on $t_2, t_2$.

- If $t_2 : u_2$. This case cannot happen: as $u_1 \approx u_2$. Then by (b::id) $\varepsilon_2 = (G_1)$ and $u_1 \in T[G_1]$, therefore $(G_1) \vdash G_1 \rightarrow G_1$. Therefore by Lemma 102, $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is defined, which is a contradiction.

- If $t_2 = \text{error}$. This case cannot happen as $\varepsilon_1 = (G_1 \rightarrow G_2)$ and as $\varepsilon_2 \vdash G_2 \sim G_1$, $\varepsilon_2 \circ \text{dom}(\varepsilon_1) = \varepsilon_2 \circ \text{dom}(G_1)$ by Lemma 102 it would never fail.

Case $(\varepsilon_1 (\lambda x^{G_1} \cdot t'_1) \circ \overline{G_2} \circ u_1) \mid \mu_1 \mapsto \text{error})$. Where $\varepsilon_2 \circ \text{iref}(\varepsilon_1)$ is not defined. By inspection of (b::=), we know that $t_2 = t_2, t_2$ for some $t_2, t_2$ such that $\varepsilon_1 (\lambda x^{G_1} \cdot \text{Ref} G_3) \approx t_2, t_2$. We proceed by case analysis on $t_2, t_2$.

- If $t_2 := t_2$. This case cannot happen: as $u_1 \approx u_2$. Then by (b::id) $\varepsilon_2 = (G_3)$ and $u_1 \in T[G_3]$, therefore $(G_3) \vdash G_3 \rightarrow G_3$. Therefore by Lemma 102, $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is defined, which is a contradiction.

- If $t_2 := \text{error}$. This case cannot happen as $\varepsilon_1 = (\text{Ref} G_3)$ and as $\varepsilon_2 \vdash G_2 \sim G_3$, $\varepsilon_2 \circ \text{iref}(\varepsilon_1) = \varepsilon_2 \circ \text{dom}(G_3)$ by Lemma 102 it would never fail.

Case $(\varepsilon_1 (\lambda x^{G_1} \cdot t'_1) \circ \overline{G_2} \circ u_1) \mid \mu_1 \mapsto \text{error})$, $t_1 \neq u$. Then by (bapp) $t_2 = t_2, t_2$. By Lemma 92 we know that $\varepsilon_1 t_1 : G_1 \rightarrow G_2 \mid \mu_1 \mapsto \text{error}$. Also by (bapp) we know that $\varepsilon_1 t_1 \circ \text{Boo} \mid \mu_1 \mapsto \text{error}$. Also by (bapp) we know that $\varepsilon_1 t_1 \circ \text{Boo} \approx t_2, t_2$. Then by induction hypothesis we know that $t_2, t_2 \mid \mu_2 \mapsto* \text{error}$, and the result holds.

Case $(\varepsilon_1 (\lambda x^{G_1} \cdot t'_1) \circ \overline{G_2} \circ u_1) \mid \mu_1 \mapsto \text{error})$, $t_1 \neq u$. Analogous to previous case but using Lemma 93.
Case \((\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}, t_1 \neq u)\). Then by \((b\mapsto\equiv)\) \(t_2 = t_21\mapsto t_22\). By Lemma 94 we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Also by \((b\equiv)\) we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Then by induction hypothesis we know that \(t_21 | \mu_2 \mapsto \text{error}\), and the result follows.

Case \((\varepsilon t_1 | \mu_1 \mapsto \text{error}, t_1 \neq u)\). Analogous to previous case but using Lemma 95.

Case \((\text{ref}^G \varepsilon t_1 | \mu_1 \mapsto \text{error}, t_1 \neq u)\). Then by \((b\text{ref}^G)\) \(t_2 = \text{ref}^G t_21\). By Lemma 96 we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Also by \((b\mapsto)\) we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Then by induction hypothesis we know that \(t_21 | \mu_2 \mapsto \text{error}\), and the result follows.

Case \((\text{t}^G \varepsilon t_1 | \mu_1 \mapsto \text{error}, t_1 \neq u)\). Then by \((b!\text{t}^G)\) \(t_2 = \text{t}^G t_21\). By Lemma 97 we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Also by \((b!)\) we know that \(\varepsilon t_1 \varepsilon t_2 | \mu_1 \mapsto \text{error}\). Then by induction hypothesis we know that \(t_21 | \mu_2 \mapsto \text{error}\), and the result follows.

Case \((\varepsilon t_1 | G | \mu_1 \mapsto \varepsilon t_1 | G | \mu_1')\). Without loosing generality let us assume that \(\varepsilon t_1 | G = \varepsilon_1(\varepsilon t_1 | G_1, G_2, G_3)\). where \(t_1 \in T|G\). By Lemma 97 we know that \(\varepsilon t_1 | G \mapsto \varepsilon t_2 | G\). Then by \((b|\equiv)\), \((b|\text{id})\) and \((b|\text{eq})\), either \(t_2 = c_1(\cdots c_m t_2m)\) where \(m \leq n\) or \(t_2 = t_21\). where \(t_2m \mapsto t_2\) and \(t_21 \mapsto t_22\) respectively.

Let us assume that \(t_2 = c_1(\cdots c_m t_2m)\) (the other case is analogous to the second subcase below). Then we know that \(t_2m \mapsto c_1(\cdots c_m t_2m)\) (the other case is analogous to the second subcase below). Then we know that \(c_1(\cdots c_m t_2m) \mapsto c_1(\cdots c_m t_2m)\). Additionally, by inspection of \((b|\equiv)\) and \((b|\text{eq})\), \(c_2(\cdots c_{2n}) \mapsto c_2(\cdots c_{2n})\).

Then by \((b|\equiv)\) and \((b|\text{eq})\), \(c_1(\cdots c_{2n}) \mapsto c_1(\cdots c_{2n})\). Additionally, by inspection of \((b|\equiv)\) and \((b|\text{eq})\), \(c_2(\cdots c_{2n}) \mapsto c_2(\cdots c_{2n})\).

We now proceed by case analysis on \(t_1\).

- If \(t_1 = u_1\). Then \(t_2m = u_2\) for some \(u_2\), also \(t_2 = c_1(\cdots c_{2n})\) (where \(m \leq n\) or \(t_2 = t_21\). where \(t_2m \mapsto t_2\) and \(t_21 \mapsto t_22\) respectively). Then by \((b|\equiv)\), \((b|\text{id})\) and \((b|\text{eq})\), either \(t_2 = c_1(\cdots c_m t_2m)\) where \(m \leq n\) or \(t_2 = t_21\). where \(t_2m \mapsto t_2\) and \(t_21 \mapsto t_22\) respectively. Let us assume that \(t_2 = c_1(\cdots c_m t_2m)\) (the other case is analogous to the second subcase below). Then we know that \(t_2m \mapsto c_1(\cdots c_m t_2m)\) (the other case is analogous to the second subcase below). Then we know that \(c_1(\cdots c_m t_2m) \mapsto c_1(\cdots c_m t_2m)\).

- If \(t_1 = \text{in}\). Then \(t_2m = u_2\) for some \(u_2\), also \(t_2 = c_1(\cdots c_{2n})\) (where \(m \leq n\) or \(t_2 = t_21\). where \(t_2m \mapsto t_2\) and \(t_21 \mapsto t_22\) respectively). Then by \((b|\equiv)\), \((b|\text{id})\) and \((b|\text{eq})\), either \(t_2 = c_1(\cdots c_m t_2m)\) where \(m \leq n\) or \(t_2 = t_21\). where \(t_2m \mapsto t_2\) and \(t_21 \mapsto t_22\) respectively.

The proof of (2) is similar but choosing sometimes \(j = 2\) or \(j = 3\) in cases for application, dereference or assignment. □

Lemma 107. Let \(G_1 \neq G_2\) such that \(G_1 \leadsto G_2\), then \(\{G_1 \Rightarrow G_2\} = G_1 \Rightarrow G_2\)

Proof. Straightforward induction on \(G_1 \leadsto G_2\). □

Lemma 108. If \(t_1 \in T|G\), \(\sigma; \Sigma \vdash t_2 : G, \mu_2 \vdash \Sigma, \mu_1 \equiv \mu_2, \) and \(t_1 \equiv t_2\), then \(t_1 \in T|G\), \(\mu_1 \equiv \mu_2, \) and \(t_1 \equiv t_2\).

Proof. Similar to Lemma 105 □

Proposition 109 (Translations are bisimilar). Given \(t : G, t \equiv \varepsilon t_1 : G, t \equiv \varepsilon t_2 : G, \) then \(t_1 \equiv t_2\).

Proof. We prove the proposition on open terms: If \(\Gamma; \sigma \vdash t : G, \Gamma; \sigma \vdash t \equiv \varepsilon t_1 : G, \) and \(\Gamma; \sigma \vdash t \equiv \varepsilon t_2 : G, \) then \(t_1 \equiv t_2\).

We proceed by induction on \(\Gamma; \sigma \vdash t : G\) (we only show some cases as the others are analogous).

Case \((\Gamma; \sigma \vdash t' : G : G)\).

\[
\frac{\text{TR} : \quad \Gamma; \sigma \vdash t' \equiv \varepsilon t_1 | G' : G' \quad \varepsilon = G_0(G_1, G_2)}{\text{HR} : \quad \Gamma; \sigma \vdash t' \equiv \varepsilon t_2 | G' : G_0(G_1, G_2)}
\]

By \((G ;)\) we know that \(\Gamma; \sigma \vdash t' : G'\), then by induction hypothesis \(t' \equiv t'\). If \(G' \neq G\), then \(\varepsilon = G_0\) and \(G \Rightarrow G\), therefore the result holds immediately by \((b|\text{id})\). If \(G' \neq G\), then by Lemma 107, \(\{G' \Rightarrow G\} = \{G \Rightarrow G\}\), and the result holds immediately by \((b|\text{eq})\).

Case \((\Gamma; \sigma \vdash \lambda x : G_1, t' : G_1 \rightarrow G_2)\).

\[
\frac{\text{TR} : \quad \Gamma, x : G_1 \vdash t' \equiv \varepsilon t_1 | G_2} {\text{HR} : \quad \Gamma, x : G_1 \vdash t' \equiv \varepsilon t_2 | G_2}
\]

By (\(G ;\)) we know that \(\Gamma, x : G_1 \vdash t' : G_1\), then by induction hypothesis \(t' \equiv t'\). If \(G' \neq G\), then \(\varepsilon = G_0\) and \(G \Rightarrow G\), therefore the result holds immediately by \((b|\text{id})\). If \(G' \neq G\), then by Lemma 107, \(\{G' \Rightarrow G\} = \{G \Rightarrow G\}\), and the result holds immediately by \((b|\text{eq})\).
By \((G\lambda)\) we know that \(\Gamma, x : G_1 \vdash t' : G_2\), then by induction hypothesis \(t^{G_2} \simeq t'_2\). Then the result holds immediately by \((b\lambda)\).

**Case** \((\Gamma ; \theta \vdash t_1 := t_2 : \text{Unit})\).

\[
\begin{align*}
\Gamma ; \theta \vdash t_1 \sim_n t^{G_1} : G_1 & \quad \Gamma ; \theta \vdash t_2 \sim_n t^{G_2} : G_2 \\
\text{(Trasn)} & \Rightarrow G_3 = \text{tref}(G_1) \quad \epsilon_1 = \text{g}(G_1, \text{Ref} G_3) \quad \epsilon_2 = \text{g}(G_2, G_3) \\
\Gamma ; \theta \vdash t_1 := t_2 \sim_n \epsilon_1 t^{G_1} := \text{G}_1 \quad \epsilon_2 t^{G_2} : \text{Unit} \\
\text{(HRasn)} & \Rightarrow \Gamma ; \theta \vdash t_1 := t_2 \sim_{\text{c}} t_1^t : G_1 \quad \Gamma ; \theta \vdash t_2 := t_2 \sim_{\text{c}} t_2^t : G_2 \quad G_3 = \text{tref}(G_1) \\
& \Rightarrow \Gamma ; \theta \vdash t_1 := t_2 \sim_{\text{c}} \langle G_1 \Rightarrow \text{Ref} G_3 \rangle t_1^t := \langle G_2 \Rightarrow G_3 \rangle t_2^t : \text{Unit}
\end{align*}
\]

By \((G :=)\) \(\Gamma ; \theta \vdash t_1 : G_1\) and \(\Gamma ; \theta \vdash t_2 : G_2\), therefore by induction hypothesis \(t^{G_1} \simeq t_1^t\) and \(t^{G_2} \simeq t_2^t\). Let us consider \(G_1 \neq \text{Ref} G_3\) and \(G_2 \neq G_3\) (the other cases are similar – see case for ascription). By Lemma 107, \(\langle G_1 \Rightarrow \text{Ref} G_3 \rangle = \langle G_1 \Rightarrow G_1 \Rightarrow \text{Ref} G_3 \rangle = \\{e_1 \vdash G_1 \sim \text{Ref} G_3\}\), therefore by \((b:\text{eq})\), \(\epsilon_1 t^{G_1} \Rightarrow \text{Ref} G_3 \approx \{G_1 \Rightarrow \text{Ref} G_3\} t_1^t\). By using similar argument, \(\epsilon_2 t^{G_2} \Rightarrow \text{Ref} G_3 \approx \{G_2 \Rightarrow G_3\} t_2^t\). Then the result holds by \((\theta:\text{=}\)\).

**Corollary 110.** Given \(t : G\), if \(t \sim_n t_1 : G\) and \(t \sim_{\text{c}} t_2 : G\), then \(t_1 \Downarrow \Longleftrightarrow t_2 \Downarrow\) and \(t_1 \Downarrow \text{error} \Longleftrightarrow t_2 \Downarrow \text{error}\). (Co-divergence follows trivially.)

**Proof.** By Proposition 16, and then combining Lemmas 105 and 108.

**Appendix F.** Encoding permissive and monotonic references in \(\lambda_{\text{Ref}}\).

**Lemma 111.** If \(\text{et} \mid v \longmapsto_{c} \text{et}' \mid v'\), then

1. \(\text{ev}(\text{et}') \subseteq \text{ev}(\text{et})\)
2. \(\forall o_m^G \in \text{dom}(v), \text{ev}(v'(o_m'^G)) \subseteq \text{ev}(v(o_m^G))\)

**Proof.**

**Case** \((\text{et} = (G_2)((G_1) o_m^{G_5} :: G_5)), v(o_m^G) = (G') u :: G_5, G_1 \cap G_2, and G' \neq \text{tref}(G_3)\). Then \(\text{et} \mid v \longmapsto (G_2) o_m^{G_5} \mid v o_m^{G_5} \longmapsto (G' \cap \text{tref}(G_3) v o_m^{G_5} :: G_5)\). We have to prove that \(G_1 \cap G_2 \subseteq G_2\), and that \(G' \cap \text{tref}(G_3) \subseteq G'\) which is immediate from Proposition 82.

**Case** (otherwise). Then \(\langle G_2 ((G_1) u :: G') \mid v \longmapsto (G_3) u \mid v, where G_3 = G_1 \cap G_2\). We only have to prove that \(G_1 \cap G_2 \subseteq G_2\) which is immediate from Proposition 82.

**Proposition 112 (Monotonicity of the evolving heap).** If \(t^G \mid v \longmapsto t'^G \mid v'\), then \(\forall o_m^G \in \text{dom}(\mu), \text{ev}(v'(o_m'^G)) \subseteq \text{ev}(v(o_m^G))\).

**Proof.** We proceed by induction on \(t^G \mid v \longmapsto t'^G \mid v'\). We only illustrate representative cases.

**Case** (RE and r4). Then \(\text{ref}^G (G_1) u \mid v \longmapsto (\text{Ref} G_1) o_m^G :: \text{Ref} G_2 \mid v', where v' = v o_m^G \longmapsto (G_1) u :: G_2\) and \(o_m^G \notin \text{dom}(v)\). Then \(\forall o_m^G \in \text{dom}(v), v(o_m^G) = v'(o_m'^G)\) and the result holds immediately.

**Case** (RE and r6). Then \(t^G = (\text{Ref} G_1) o_m^G :: G_3 (G_2) u\). Then \(t^G \mid v \longmapsto \text{unit} \mid v o_m^G \longmapsto (G'(G_2 \cap G_1) u :: G_4) :: G_4\), where \(v(o_m^G) = (G') u :: G_4\). Then we have to prove that \(\langle G' \rangle \subseteq \langle G' \rangle\), but is trivial.

**Case** (RE and r6). Then \(t^G = (\text{Ref} G_1) o_m^G :: G_3 (G_2) u, z \neq m\). The result is immediate as the updated location is not monotone.

**Case** \((Rv)\). We know that \(t \mid v \longmapsto t \mid v o_m^G \longmapsto \text{et}' :: G'\), where \((v(o_m^G)) = et :: G'\) and \(et \mid v \longmapsto et' :: v' \mid v'\), and \(\text{ev}(v(o_m^G)) = \text{ev}(v'(o_m'^G))\). By Lemma 111, \(et' \subseteq et\) and \(\forall o_m^G \in \text{dom}(v), \text{ev}(v'(o_m'^G)) \subseteq \text{ev}(v(o_m^G))\). We have to prove that \(\forall o_m^G \in \text{dom}(v), \text{ev}(v'(o_m'^G) \longmapsto \text{et}' :: [G'(G_1) o_m^G :: G_4]) \subseteq \text{ev}(v(o_m^G))\), which means that we only have to show that \(et' :: G' \subseteq et :: G'\), but as \(et' \subseteq et\), the result is immediate.

**Proposition 113 (Monotonicity of the heap).** If \(t^G \mid \mu \longmapsto^{*} t'^G \mid \mu'\), then \(\forall o_m^G \in \text{dom}(\mu), \mu(o_m^G) = \epsilon u :: G'\), then \(\mu'(o_m^G) = \epsilon' u' :: G'\) and \(\epsilon' \subseteq \epsilon\).
**Proof.** By induction on \( t^G | \mu \mapsto t'^G | \mu' \) and Proposition 24. \( \square \)

**Proposition 114** (\( \cdashdash \) is well defined). If \( t^G | \mu \vdash t^G | \mu \rightarrow r \), then \( r \in \text{CONFIG}_G \cup \{ \text{error} \} \), and if \( r = t'G | \mu \), then also \( t'G | \nu \) and \( \text{dom}(\mu) \subseteq \text{dom}(\nu) \).

**Proof.** By induction on the structure of a derivation of \( t^G \rightarrow r \), considering the last rule used in the derivation. The proof is analogous to some cases considered in Proposition 24. We only illustrate representative cases.

**Case** (r4). Then \( t^G = \text{ref}G^2 (G_1)u \). Then

\[
(\text{IRef}) \quad \frac{u \in T[G_1]}{(G_1) \vdash G_2} \quad \frac{G_1 \vdash G_2}{\text{ref}G^2 (G_1)u \in T[\text{Ref} G_2]}
\]

Then

\[
\text{ref}G^2 (G_1)u | \mu \rightarrow (\text{Ref} G_2) oG^2 :: \text{Ref} G_2 | \mu [oG^2 \mapsto (G_1)u :: G_2]
\]

where \( oG^2 \notin \text{dom}(\mu) \). But as \( (G_1)u :: G_2 \in T[G_2] \), then \( (\text{Ref} G_2) oG^2 :: \text{Ref} G_2 \vdash \mu [oG^2 \mapsto (G_1)u :: G_2] \). Also as \( (\text{Ref} G_2) \vdash \text{Ref} G_2 \rightarrow \text{Ref} G_2, (\text{Ref} G_2) oG^2 :: \text{Ref} G_2 \in T[\text{Ref} G_2] \) and the result holds.

**Case** (r6). Then \( t^G = \varepsilon_1 oG^1 :: G_3 \varepsilon_2 u \). Then

\[
(\text{IAsgn}) \quad \frac{G_1 \vdash \text{Ref} G_3}{u \in T[G_2]} \quad \frac{\varepsilon_1 \vdash \text{Ref} G_1 \rightarrow \text{Ref} G_3}{\varepsilon_2 \vdash G_2 \rightarrow G_3} \quad \varepsilon_1 oG^1 :: G_3 \varepsilon_2 u \in T[\text{Unit}]
\]

Suppose \( \mu (oG^1) = \varepsilon_3 u' :: G_1 \), and \( z = m \). If \( \varepsilon' = (\varepsilon_2 \cap \varepsilon_3 \cap \text{iref}(\varepsilon_1)) \) is not defined, then \( t^G \rightarrow \text{error} \), and then the result hold immediately. We know that \( v = \varepsilon_2 u :: G_3 \in T[G_3] \). Also \( \text{iref}(\varepsilon_1) \vdash G_3 \rightarrow G_1 \), and \( \varepsilon_3 \vdash G_2 \rightarrow G_1 \), and \( \varepsilon_3 \vdash G_2 \rightarrow G_2' \), therefore \( \varepsilon_3 \vdash G_1 \rightarrow G_1, \text{and } \text{iref}(\varepsilon_1) \circ \varepsilon_3 \vdash G_1 \rightarrow G_1 \), then \( t = \text{iref}(\varepsilon_1) \circ \varepsilon_3 \vdash G_1 \in T[G_1] \). If \( z \neq m \), then by using arguments analogous to the other case we know that \( t = \text{iref}(\varepsilon_1)v :: G_1 \in T[G_1] \). Therefore as \( \text{freeLocs}(\text{unit}) = \emptyset \subseteq \text{dom}(\mu) \), we know from \( t^G \vdash \mu \) that \( \text{VoG} \in \text{dom}(\mu), \mu (oG^1) \in T[G'] \), and as \( t \in T[G_1] \), therefore \( \text{unit} \vdash \mu [oG^1 \mapsto t] \). Also

\[
\frac{\theta(\text{unit}) = \text{Unit}}{\text{unit} \in T[\text{Unit}]}
\]

and the result holds. \( \square \)

**Proposition 115** (\( \cdashdash \varepsilon \) is well defined). If \( \varepsilon t \in \text{ETERM}_G \), \( t \vdash v \) and \( \varepsilon t | v \rightarrow \varepsilon r, \) then \( r \in (\text{ETERM}_G \times \text{EVOLVINGSTORE}) \cup \{ \text{error} \} \), and if \( r = \varepsilon t' | v' \), then also \( t' \vdash v' \) and \( \text{dom}(v) \subseteq \text{dom}(v') \).

**Proof.** As \( \varepsilon t | v \rightarrow \varepsilon r, \) then \( t = \varepsilon g u :: G', \) and as \( \varepsilon (\varepsilon g u :: G') \in \text{ETERM}_G \) then \( \varepsilon g u \vdash G_1 \rightarrow G' \) and \( \varepsilon' \rightarrow G' \rightarrow G, \) for some \( G_u, G' \), with \( u \in T[G_u] \). If \( \varepsilon \cap \varepsilon' \) is not defined then \( r = \text{error} \) and the result holds immediately. Let us assume that \( \varepsilon = \varepsilon \cap \varepsilon' \) is defined. Then \( \varepsilon t | v \rightarrow \varepsilon t \vdash v' \rightarrow v' \). By definition of consistent transitivity \( \varepsilon t | v \vdash \varepsilon u | v' \), therefore \( t' = \varepsilon t \in \text{ETERM}_G \). If \( u \neq g u \), or if \( (u = g u \text{ and } v(u) = \text{iref}(\varepsilon)u) \), then \( v = v' \) and the result holds. Let us assume that \( u = g u \), \( v(u) = v'u, \varepsilon = v'u :: G', \) and \( \varepsilon \neq \text{iref}(\varepsilon) \), then \( v' = v[u \mapsto v'(u) :: G'] \), where \( \varepsilon' = \varepsilon \cap \text{iref}(\varepsilon) \). Also \( G_u = \text{Ref} G_u \) and \( G = \text{Ref} G'' \), for some \( G'' \). No new locations are created then \( \text{dom}(v) = \text{dom}(v') \Rightarrow \text{dom}(v) \subseteq \text{dom}(v') \). We have the obligation to prove that \( u \vdash v' \). As the domains are the same it is easy to see that \( \text{freeLocs}(u) \subseteq \text{dom}(v') \). Then we only have to prove that \( \varepsilon' \vdash v' :: G_u \in T[G_u] \). But as \( \varepsilon \vdash \text{Ref} G_u \rightarrow \text{Ref} G'' \), then \( \text{iref}(\varepsilon) \vdash G_u \rightarrow G'' \), and also \( \text{iref}(\varepsilon) \vdash G'' \rightarrow G'' \), therefore \( \text{iref}(\varepsilon) \vdash G_u \rightarrow G_u \). Similarly \( v'u \rightarrow G_u \), and then \( \varepsilon' \vdash G_u \rightarrow G_u \). Finally then \( \varepsilon' \vdash v(u) :: G_u \in T[G_u] \) and the result holds. \( \square \)

**Proposition 116** (\( \cdashdash \) is well defined). If \( t^G | v \rightarrow t^G | v' \) and \( r \in \text{CONFIG}_G \cup \{ \text{error} \} \), and if \( r = t'G | \nu \), then also \( t'G \vdash v' \) and \( \text{dom}(v) \subseteq \text{dom}(v') \).

**Proof.** By induction on the structure of a derivation of \( t^G \rightarrow r \). We proceed almost identical to 42, therefore we only highlight main differences.

**Case** (RF). Let \( \text{EvTERM}_G \) be notation for the family of evidence terms \( \text{et}G^1 \) such that \( \varepsilon \vdash G_1 \rightarrow G_2 \). Then \( t^G = F[\text{et}], F[\text{et}] \in T[G], \) and \( F : \text{EvTERM}_G \rightarrow T[G], \) and \( \text{et} | v \rightarrow \text{et}' | v' \). By Lemma 115, \( \text{et}' \in \text{EvTERM}_{G^1}, \text{et}' \vdash v' \), and \( \text{dom}(v) \subseteq \text{dom}(v') \). Then \( F[\text{et}'] \in T[G], \) and as \( \text{freeLocs}(\text{et}) = \text{freeLocs}(\text{et}') \) we can conclude that \( F[\text{et}'] \vdash v \).
Case \( \text{(Ref)} \). We know that \( t \vdash v \iff t' \vdash v' [o_2^{G'} = et' :: G'] \), where \( v(o_2^{G'}) = et :: G' \) and \( et \vdash v \iff et' :: v' \), and \( \text{ev}(v(o_2^{G'})) = \text{ev}(v' (o_2^G)) \). By Lemma 115, \( et' \in \text{ETerm}_{G'} \), \( et' \vdash v' \), and \( \text{dom}(v) \subseteq \text{dom}(v') \). As \( \text{dom}(v) \subseteq \text{dom}(v') \) then freeLocs(t) \( \subseteq \text{dom}(v') \). As \( et' \vdash v' \), we know that \( \forall o_2^{G''} \in \text{dom}(v') \), \( v'(o_2^{G''}) \in T[G'] \). Then we only have to prove that \( et' :: G' \), but we know that \( et' \in \text{ETerm}_{G'} \), therefore \( et' :: G' \in T[G'] \) and the result holds.

Case \( \text{(vErr)} \). Trivial as \( t \vdash v \iff \text{error} \). 

To define the dynamic gradual guarantee, first we have to extend the notion of precision to evolving stores. Note that rule \( (r7) \) propagates casts into the store based on a type test, which may seem to jeopardize the dynamic gradual guarantee. Consider two terms and store in the precision relation, one of the two terms can reduce to a new term and evolving store (that needs to be reduced) whereas the other to another term and store (that does not need to be reduced). Therefore to maintain the precision relation in lock step, we define precision between evolving stores as follows:

\[
\begin{align*}
\flat(\xi_2^{G} :: G) &= \xi_1 o^\mu \flat(t^{G}) \\
\uval(\xi_2^{G} :: G) &= \uval(t) \\
\end{align*}
\]

Note that (1) if \( G \subseteq G' \), \( \uval(t^{G}) \subseteq \uval(t^{G'}) \), but consistent transitivity is not defined in \( \flat \) of \( t^{G} \), then the relation hold, and (2) if both evolving stores are stores, then this definition coincides with the precision relation of stores defined for \( \xi_{\text{Ref}}^{G} \).

**Proposition 117 (Dynamic guarantee for \( \text{---} \)).** Suppose \( \Omega \vdash t_1^{G_1} \subseteq t_1^{G_2} \) and \( \mu_1 \subseteq G_2 \). If \( t_1^{G_1} \mid \mu_1 \longrightarrow t_2^{G_1} \mid v_1 \) then \( t_2^{G_2} \mid \mu_2 \longrightarrow t_2^{G_2} \mid v_2 \), where \( \Omega \vdash t_1^{G_1} \subseteq t_2^{G_1} \), \( v_1 \in v_2 \) for some \( \Omega' \supseteq \Omega \).

**Proof.** By induction on reduction \( t_1^{G_1} \mid \mu_1 \longrightarrow t_2^{G_1} \mid v_1 \). We proceed almost identical to 66, therefore we only illustrate main differences. For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case \( (r2) \).** We know that \( t_1^{G_1} = \xi_{\text{ref}}^{G_1} \xi_1 u_1 \) where \( G_1 = \text{Ref} G'_1 \), then by \( (\text{Ref}) \) \( t_1^{G_2} \) must have the form \( t_1^{G_2} = \xi_{\text{ref}}^{G_2} \xi_2 u_2 \) for some \( \xi_{\text{ref}}^{G_2} \in \text{Ref} G'_2 \). Let us pose \( \xi_1 = \xi_{\text{ref}}^{G_2} \Omega o^{\xi_{\text{ref}}^{G_2}} \). Then \( \xi_1 = \xi_{\text{ref}}^{G_2} \Omega o^{\xi_{\text{ref}}^{G_2}} \in \text{Ref} G'_2 \) and \( \xi_1 = \text{Ref} G'_2 \).

Also, by \( 64, \xi_{\text{ref}}^{G_2} \Omega = \Omega_{\text{ref}}(\xi_{\text{ref}}^{G_2}) \). Then \( \xi_1 u_1 \in \text{Ref} G'_2 \). By substitution preserves precision (Proposition 63) \( t_1^{G_2} \in t_2^{G_2} \), therefore \( \text{icod}(\xi_{\text{ref}}^{G_2}) t_1^{G_2} \in \text{icod}(\xi_{\text{ref}}^{G_2}) t_2^{G_2} \).

**Case \( (r4) \).** We know that \( t_1^{G_1} = \text{ref} G_1 \xi_1 u_1 \) where \( G_1 = \text{Ref} G'_1 \), then by \( (\text{Ref}) \) \( t_1^{G_2} \) must have the form \( t_1^{G_2} = \text{ref} G' \xi_2 u_2 \) for some \( \xi_2^{G_2} \in \text{Ref} G'_2 \).

Then \( t_1^{G_2} \vdash \mu_1 \longrightarrow (\text{Ref} G'_{2} o^{\xi_{\text{ref}}^{G_2}} :: \text{Ref} G'_{2} | \mu_1 o^{\xi_{\text{ref}}^{G_2}} \rightarrow \xi_{\text{ref}}^{G_2} :: G'_1 \).

Also, \( t_1^{G_2} \vdash \mu_2 \longrightarrow (\text{Ref} G'_{2} o^{\xi_{\text{ref}}^{G_2}} :: \text{Ref} G'_{2} | \mu_2 o^{\xi_{\text{ref}}^{G_2}} \rightarrow \xi_{\text{ref}}^{G_2} :: G'_2 \).

Then by \( (\xi_{\text{ref}}^{G_2}) \), \( \xi_{\text{ref}}^{G_2} \xi_1 u_1 :: G_1 \subseteq \xi_{\text{ref}}^{G_2} \xi_2 u_2 :: G_2 \). Also by \( (\xi_{\text{ref}}^{G_2}) \), as \( G_1 \subseteq G_2 \) and by \( (\Omega) \), \( \text{Ref} G'_{2} o^{\xi_{\text{ref}}^{G_2}} :: \text{Ref} G'_{2} | \mu_2 o^{\xi_{\text{ref}}^{G_2}} \rightarrow \xi_{\text{ref}}^{G_2} :: G'_2 \). Also by \( (\xi_{\text{ref}}^{G_2}) \), as \( G_1 \subseteq G_2 \) and by \( (\xi_{\text{ref}}^{G_2}) \), \( \text{Ref} G'_{2} o^{\xi_{\text{ref}}^{G_2}} :: \text{Ref} G'_{2} | \mu_2 o^{\xi_{\text{ref}}^{G_2}} \rightarrow \xi_{\text{ref}}^{G_2} :: G'_2 \) and the result holds.

**Case \( (r6) \).** We know that \( t_1^{G_1} = \xi_{\text{ref}}^{G_1} \xi_1 u_1 \) where \( G_1 = \text{Unit} \), then by \( (\xi_{\text{ref}}^{G_1}) \) \( t_1^{G_2} \) must have the form \( t_1^{G_2} = \xi_{\text{ref}}^{G_2} \xi_2 u_2 \) for some \( \xi_{\text{ref}}^{G_2} \subseteq \text{Unit} \).

**Suppose \( \mu_1 o^{\xi_{\text{ref}}^{G_2}} = \xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1 \).**

Then \( t_1^{G_2} \vdash \mu_1 \longrightarrow (\xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1) \).

By inspection of evidence and inversion lemma, as \( \xi_{\text{ref}}^{G_2} \subseteq \xi_{\text{ref}}^{G_2} \), \( \xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1 \). Also, by Lemma 64, \( \xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1 \). Also, by \( (\xi_{\text{ref}}^{G_2}) \), \( \xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1 \). Also by \( (\xi_{\text{ref}}^{G_2}) \), as \( G_1 \subseteq G_2 \) and by \( (\xi_{\text{ref}}^{G_2}) \), \( \xi_{\text{ref}}^{G_2} o^{\xi_{\text{ref}}^{G_2}} \xi_1 u_1 :: G_1 \) and the result holds.
Now we know that $e_2 \circ e_1$ is defined, and as $e_1 \not\sqsubseteq e_2$ and $e_1 \sqsubseteq e_2$, by Lemma 64, $e_2 \circ e_2$ is also defined. Therefore $\text{flat}((e_1(e_2u_2 :: G_1)) :: G_1) \not\subseteq \text{flat}((e_2(e_2u_2 :: G_2)) :: G_2))$ and the result holds. 

**Definition 16.**

$\vdash_m (t, v) \iff e_0^G_m$ appears in $(t, v)$, either $\text{flat}(v(o^G_m))$ is not defined, or $\text{flat}(v(o^G_m)) \not\subseteq \text{iref}(e)$. 

**Lemma 118.** If $e_1 \not\sqsubseteq e_2$ and $e_1 \circ e_2$ is defined, then $e_1 \circ e_2 \not\sqsubseteq e_2$.

**Proof.** We know that $e_2 \circ e_2 = e_2$. Then by Proposition 64, $e_1 \circ e_2 \not\subseteq e_2$, and the result holds.

**Lemma 119.** If $e_1 \not\sqsubseteq e_2$ and $e_1 \circ e_2$ is defined, then $e_1 \circ e_3 \not\subseteq e_2$.

**Proof.** Direct by formal definition of meet and precision using the concretization function.

**Lemma 120.** If $e_1 \not\sqsubseteq e_2, e_1 \not\sqsubseteq e_3$, then $e_2 \circ e_3$ is defined, and $e_1 \not\sqsubseteq e_2 \circ e_3$.

**Proof.** By induction on $e_1$.

**Lemma 121 (Monotonic well-formedness preservation).** If $\vdash_m (t, v)$ and $t \mid v \mapsto t' \mid v'$, then $\vdash_m (t', v')$.

**Proof.** By induction on $t \mid v \mapsto t' \mid v'$. We only consider interesting cases where references are involved.

**Case (r2).** Then

$$F[(\langle G_1' \to G_2' \rangle \circ \lambda x^G_1 \cdot t)] @ G_1 \to G_2 \mid \mu \mapsto F(G_1') \mid t \mid G_1 \mid \mu \mapsto v \mid v.$$

where $\langle G_1' \rangle \circ \lambda x^G_1 \cdot t \mid G_1 \mid t = G_2$. We know that $\vdash_m (t, u)$, then the result follows from induction hypothesis on $\langle G_1 \rangle \circ \lambda x^G_1 \cdot t \mid G_1 \mid t = G_2$.

**Case (r4).** Then

$$F[\text{ref}^G_m (G_1) \mid \mu \mapsto F(\text{Ref} G_2) \mid \mu \mapsto v \mid v]$$

where $\text{ref}^G_m (G_1) \mid G_1 = G_2$. We have to prove that $\langle G_1 \rangle \subseteq \text{iref} (\text{Ref} G_2) = \langle G_2 \rangle$ which follows immediately. The result follows because from $\vdash_m (t, v)$ we know that if $u = o_m^G$, then $\text{flat}(\mu (u)) \not\subseteq \text{iref}(G_1)$.

**Case (r5).** Then

$$F[\text{G} (\text{Ref} G_1) o^G_m \mid \mu \mapsto F(\text{G} G_1) \mid \mu \mapsto v]$$

where $v = \mu (o^G_m)$. The result is immediately as we know that $\vdash_m (v, \mu)$.

**Case (r6).** Then

$$F[\text{G} (\text{Ref} G_1) o^G_m := G_3 (G_2) \mid t \mapsto F[\text{unit}] \mid t \mapsto t']$$

where $\mu (o^G_m) = \langle G' \rangle u = G$, $t = \langle G_1 \cap G' \rangle \mid G_2 \mid G_3 = G$. We have two obligations: (1) if $e^G_m$ appears in $\text{cod}(\mu)$, then $\text{flat}(t') \not\subseteq \text{iref}(e')$, and (2) if $u$ is a monotonic location then $\text{iref}(G_2) \not\subseteq \text{flat}(\mu (u))$.

Let us prove (1). We know by $\vdash_m (t, \mu)$, that if $e^G_m$ appears in $\text{cod}(\mu)$, then $\langle G' \rangle \not\subseteq \text{iref}(e')$. If $\text{flat}(t') = \langle G_2 \rangle o_m^G \mid G \cap G'$ is defined (if not defined the result is trivial) then by Proposition 118, $\text{flat}(t') \subseteq \langle G' \rangle$, and therefore $\text{flat}(t') \subseteq \text{iref}(e')$.

Now to prove (2), we already know that $\vdash_m (G_2) \mid G_1 \mid G_2, \mu$, then it is trivial to see that $\vdash_m (t', \mu)$ and the result holds.

**Case (r7). Then**

$$\langle G_2 \rangle (G_1 o^G_m : G) \mid v \mapsto G_3 o^G_m \mid v[\langle G_3 \rangle o^G_m \mapsto (G_4) o^G_m \mapsto G_5]$$

where $G' \not\sim \text{iref}(G_3)$. Then we have to prove that $\vdash_m (G_3) o^G_m \cdot v[\langle G_3 \rangle o^G_m \mapsto (G_4) o^G_m \mapsto G_5]$.
1. If $e' o_{m}^{G'}$ appears in cod($v$), then $\text{flat}((G_4) v(o_{m}^{G'}) :: G_3) \subseteq \text{iref}(e')$. Notice that $\text{flat}((G_4) v(o_{m}^{G'}) :: G_5) = (G'' \cap G_4)$, for some $G'' \subseteq G'$ (if not defined the result follows). We know from $\vdash_{m} (t, v)$ that $\text{flat}(v(o_{m}^{G_5})) = (G') \subseteq \text{iref}(e')$, then it by Proposition 118, $(G'' \cap G_4) \subseteq (G') \subseteq \text{iref}(e')$ and the result follows.

2. $\vdash_{m} ((G_4) v(o_{m}^{G'}) :: G_3, v')$. We know from $\vdash_{m} (t, v)$ that $e' o_{m}^{G''}$ appears in $v(o_{m}^{G'})$, then $\text{flat}(v(o_{m}^{G''})) \subseteq \text{iref}(e')$. Then as $e' o_{m}^{G'}$ also appears in $(G_4) v(o_{m}^{G'_3}) :: G_5$ the result follows.

Now if $(Rv)$ is used, i.e.

\[
(Rv) \quad \frac{v(o_{m}^{G'}) = et :: G}{t \mid v \mapsto t \mid v'[o_{m}^{G'} \mapsto et' :: G]}
\]

Then we have to prove that

1. If $e' o_{m}^{G'}$ appears in $(t, v)$, then $\text{flat}((G_4) v(o_{m}^{G'}) :: G_3) \subseteq \text{iref}(e')$. Notice that $\text{flat}((G_4) v(o_{m}^{G'}) :: G_5) = (G'' \cap G_4)$, for some $G'' \subseteq G'$ (if not defined the result follows). We know from $\vdash_{m} (t, v)$ that $\text{flat}(v(o_{m}^{G_5})) = (G') \subseteq \text{iref}(e')$, then by Proposition 118, $(G'' \cap G_4) \subseteq (G') \subseteq \text{iref}(e')$ and the result follows.

2. If $e' o_{m}^{G'}$ appears in $(t, v)$, then $\text{flat}(v(G_3 o_{m}^{G'} :: G) = (G_3) \subseteq \text{iref}(e')$. We know from $\vdash_{m} (t, v)$ that $\text{flat}(v(o_{m}^{G'}) = (G_1) \subseteq \text{iref}(e')$, then by Proposition 118, $(G_1, G_2) \subseteq (G_1) \subseteq \text{iref}(e')$ and the result follows.

3. $\text{flat}((G_4) v(o_{m}^{G'}) :: G_3) \subseteq \text{iref}(G_3)$. We know that $\text{flat}(v(o_{m}^{G})) \subseteq \text{iref}((G_1))$. Then by Proposition 118 $\text{flat}((\text{iref}(G_2)) v(o_{m}^{G'}) :: G_3) = \text{flat}(\text{iref}(G_2)) v(o_{m}^{G'}) :: G_3 \subseteq \text{flat}(\text{iref}(G_2)) v(o_{m}^{G'}) :: G_3$.

Therefore from Proposition 118 $\text{flat}((G_4) v(o_{m}^{G'}) :: G_3) \subseteq \text{flat}(v(o_{m}^{G''})) \subseteq \text{iref}(G_3)$ and the result holds.

Case $(RF)$ is analogous to $(Rv)$ (and simpler).

Lemma 122. If $\text{iref}(G) \cap G$ is defined, then $G \subseteq \text{iref}(\text{iref}(G))$.

Proof. We proceed by induction on $\text{iref}(\text{iref}(G))$.

Case $(G = ?)$. Then we have to prove that $? \subseteq ?,$ which is direct.

Case $(G = \text{Ref} ?)$. Then we have to prove that $\text{Ref} ? \subseteq ?,$ which is direct.

Case $(G = \text{Ref} G' = \text{Ref} 2G' )$. We have to prove that $\text{Ref} 2G' \subseteq G'$. We analyze two possible cases. If $G' = ?$, then the result is trivial as $\text{Ref} \emptyset \subseteq ?. If G' = \text{Ref} G''$ for some $G''$, then we have to prove that $\text{Ref} 3G'' \subseteq \text{Ref} G'$, but by inspection of $\subseteq$, it is equivalent to prove that $G'' \subseteq \text{Ref} G'$, which is equivalent to prove that $\text{Ref} 2G'' \subseteq \text{Ref} 2G'$. We know that $\text{Ref} G'' \cap \text{Ref} G''$ is defined, therefore by definition of $\subseteq$, it must be the case that $G'' \subseteq \text{Ref} 2G''$. Then by induction hypothesis $\text{Ref} 2G'' \subseteq \text{Ref} 2G'$, and the result holds.

Lemma 123. If $\vdash_{m} (t, \mu(o_{m}^{G'} \mapsto \epsilon_1(\epsilon_2 u :: G) :: G), t \mid \mu(o_{m}^{G'} \mapsto \epsilon_1(\epsilon_2 u :: G) :: G) \mapsto t \mid v[o_{m}^{G'} \mapsto \epsilon_3 u :: G], then t \mid v'[o_{m}^{G'} \mapsto \epsilon_3 u :: G] \mapsto * t \mid v'[o_{m}^{G'} \mapsto \epsilon_4(\epsilon_5 u :: G) :: G].$

Proof. We proceed by contradiction. Without losing generality, let us suppose that there is the following cycle: $v(o_{m}^{G'}) = \epsilon_1(\epsilon_2 u :: G) :: G_1, v(o_{m}^{G'}) = \epsilon_2 o_{m}^{G'} :: G_2$. Then suppose

\[
t \mid v \mapsto t \mid \mu(o_{m}^{G_1} \mapsto \epsilon_1(\epsilon_2 o_{m}^{G'}) :: G_2, o_{m}^{G_2} \mapsto \epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2)])
\]

\[
\mapsto t \mid \mu(o_{m}^{G_1} \mapsto \epsilon_2(\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2)])
\]

where $\epsilon' = \epsilon_2 o_{m}^{G'}$. From last step of reduction we know that $\epsilon_1(\epsilon_2 o_{m}^{G'}) \subseteq \epsilon_2(\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2))$. From $\vdash_{m} (t, v)$ and Lemma 26, we know that $\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2)$ is defined. Also as $\epsilon_2(\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2)) = (\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2)) \subseteq \epsilon_2(\epsilon_3(\epsilon_4(\epsilon_5 u :: G) :: G_2))$. From Lemma 122, $G_2 \cap G_2 \subseteq \text{iref}(G_2)$, but we know that $G_2 \cap G_2 \subseteq \text{iref}(G_2)$, therefore by Lemma 120, $G_2 \subseteq \text{iref}(G_2)$ which is a contradiction and the result holds.

Lemma 124. If $\vdash_{m} (t, \mu_1), t_1 \mid \mu_1 \equiv t_2 \mid v_2$, then $t_2 \mid v_2 \mapsto t_2 \mid v'_2$, and $\mu_1 \equiv v'_2$. 

Proof. Let us assume \( \nu_2(\nu_2^G) = \varepsilon_2(\varepsilon_1^u \triangleright G) \triangleright G \), then

\[
    \frac{\varepsilon_2(\varepsilon_1^u \triangleright G) \triangleright \nu_2}{t_2 \mid \nu_2 \longmapsto t_2 \mid \nu_2^G (\nu_2^G \rangle \triangleright \varepsilon_2^u \triangleright G)}
\]

We know that if \( \mu_1(\nu_2^G) = \varepsilon_1u \triangleright G \), then \( \mu_1 \subseteq \mu_2 \). If \( \nu_2, \nu_2' \) are not locations then the result holds immediately. If \( \nu_2 = \nu_2^G \) and \( \nu_2' = \nu_2^G \), suppose that \( \mu_1(\nu_2) = \varepsilon_4u_4 \triangleright G_4 \), \( \varepsilon_4(\nu_2) = \varepsilon_4u_4 \triangleright G_4 \). We know that \( \varepsilon_4 \subseteq \varepsilon_3 \), and thus \( \nu_2 \triangleright \varepsilon_3 \). As \( \varepsilon_4 \subseteq \varepsilon_4' \), then by Lemma 120 \( \varepsilon_4 \triangleright \nu_2 \triangleright \varepsilon_3 \triangleright \varepsilon_4' \). Then \( \nu_2' = \nu_2^{G_4} \triangleright \nu_2^{G_4} \triangleright G \). Notice that \( \nu_2 \triangleright \varepsilon_3 \triangleright \varepsilon_4 \triangleright \varepsilon_4' \) and the result holds.

Lemma 125. If \( \Gamma \triangleright \mu_1 \) and \( \Gamma \triangleright \mu_2 \) then \( \Gamma \triangleright \mu_2 \), such that \( \mu_1 \subseteq \mu_2 \).

Proof. By Lemma 124 we make sure that the precision relation holds after every step, and by Lemma 30 we notice that there are no cycles, so the biggest amount of steps before getting to a \( \mu_2 \) is \( \text{size}(\text{dom}(\nu_2)) \) − 1.

Lemma 126. Let \( \Gamma \triangleright \varepsilon_3 \triangleright \Gamma \); then \( \Gamma \triangleright \varepsilon_3 \triangleright \Gamma \), with \( \varepsilon_3 \triangleright \varepsilon_3 \triangleright \varepsilon_3 \), such that \( \varepsilon_3 \triangleright \varepsilon_3 \triangleright \varepsilon_3 \).

Proof. We prove the following property instead: Suppose \( \Omega \triangleright t_1^G \triangleright t_2^G \) and \( \nu_1 \triangleright \nu_2 \). If \( t_1^G \triangleright t_2^G \triangleright \nu_1 \) then \( t_2^G \triangleright \nu_2 \). By induction on reduction \( t_1^G \triangleright t_2^G \triangleright \nu_1 \). We proceed almost identical to 14, therefore we will only illustrate main differences. For simplicity we omit the \( \Omega \triangleright - \) notation on precision relations when it is not relevant for the argument. Note that in all cases we are using Lemma 26, to pose that \( \Gamma \triangleright (t_2^G, \nu_1) \). Also in every rule where the starting store is not an evolving store, then we can always apply Lemma 124, to advance an evolving store of the less precise term into a non-evolving store in the precision relation.

Case (RF) and \( \nu \triangleright \Gamma \triangleright (\nu_1 \triangleright \nu_1) \triangleright \Gamma \triangleright \nu_1 \). Inspect by \( \nu_1 \triangleright \nu_1 \), for some \( \Gamma \triangleright \nu_1 \). If \( \nu_1 \triangleright \nu_1 \) is an evolving store then by Lemma 29, \( \nu_1^G \rangle \nu_2 \triangleright \Gamma \triangleright \nu_2 \), such that \( \nu_1 \triangleright \nu_2 \). Suppose \( \nu_1 \triangleright \nu_1 \). Then \( \nu_1 \triangleright \nu_2 \) is defined, and \( \nu_2 \triangleright \nu_1 \). By Proposition 64 \( \nu_1 \triangleright \nu_1 \triangleright \nu_2 \). Then \( \nu_2 \triangleright \nu_1 \triangleright \nu_1 \triangleright \nu_2 \), and the result holds by Lemma 64.

If \( \nu_1 \triangleright \nu_1 \), then we have to prove that \( \nu_2 \triangleright \nu_2 \), which holds by Lemma 119.

Case (RF) and \( \nu_1 \triangleright \nu_1 \). The result holds by Lemma 28.

Case (RF) and \( \nu_1 \triangleright \nu_1 \). The result holds by Lemma 28, and then Lemma 124. □
References


