Graduality and Parametricity:
Together Again for the First Time

MAX S. NEW, Northeastern University, USA
DUSTIN JAMNER, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

Parametric polymorphism and gradual typing have proven to be a difficult combination, with no language yet produced that satisfies the fundamental theorems of each: parametricity and graduality. Notably, Toro, Labrada, and Tanter (POPL 2019) conjecture that for any gradual extension of System F that uses dynamic type generation, graduality and parametricity are "simply incompatible". However, we argue that it is not graduality and parametricity that are incompatible per se, but instead that combining the syntax of System F with dynamic type generation as in previous work necessitates type-directed computation, which we show has been a common source of graduality and parametricity violations in previous work.

We then show that by modifying the syntax of universal and existential types to make the type name generation explicit, we remove the need for type-directed computation, and get a language that satisfies both graduality and parametricity theorems. The language has a simple runtime semantics, which can be explained by translation to a statically typed language where the dynamic type is interpreted as a dynamically extensible sum type. Far from being in conflict, we show that the parametricity theorem follows as a direct corollary of a relational interpretation of the graduality property.

Additional Key Words and Phrases: gradual typing, graduality, polymorphism, parametricity, logical relation

1 INTRODUCTION

Gradually typed languages support freely mixing statically typed and dynamically code within a single language and enable a transition from dynamic to static typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006, 2008]. They allow for stable, typed libraries to be used by ephemeral dynamically typed scripts with no manual programming overhead, streamlining a commonplace pattern in systems software. Furthermore, when some of these dynamically typed scripts inevitably become feature-rich software, static types can be gradually added to help with optimization, refactoring, type-based IDEs and documentation.

Gradually typed languages in the tradition of Siek and Taha [2006] are based on the presence of a dynamic type, written $\exists$, which is the type of dynamically typed code and is treated specially by the type checker. For instance, if $f$ is a statically typed function with type $\mathbb{I} \rightarrow \mathbb{B}$—where $\mathbb{I}$ and $\mathbb{B}$ represent integer and boolean types, respectively—and $x$ is a dynamically typed input, then the application $f x$ is allowed by the static type checker because it is “plausible” that $x$ will actually satisfy the type $\mathbb{I}$ at runtime. But note here that since $f$ has type $\mathbb{I} \rightarrow \mathbb{B}$, it was written with the expectation that it should only be applied to integers and may, for instance, use arithmetic operations on its argument. In a sound gradually typed language, this type information should be
reliable: the programmer and compiler should be able to refactor or optimize the function $f$ based on its type, which says it will only be used on values of type $I$. In order to ensure that this expectation is met at runtime, the application $fx$ is elaborated to a core language called a cast calculus where a cast is inserted and the application becomes $f(\langle I \leftarrow ? \rangle x)$. If at runtime $x$ is a value that is incompatible with the type $I$, such as a function value, then the cast will error and signal that the input failed to meet the function’s type. While this means that the gradual language admits runtime errors, it ensures the soundness of the type for programmer reasoning and compiler optimization.

When designing the semantics of a gradual language, we must consider not just how programs run, but how their behavior changes throughout the development process. Specifically, a gradual language should ensure a smooth transition from dynamic to static typing, which is formalized in two properties called the static and dynamic gradual guarantee [Siek et al. 2015]. The static gradual guarantee states that making types more precise in a program makes it less likely that the program type-checks. Our focus in this paper is on the dynamic gradual guarantee, also called graduality [New and Ahmed 2018]. The graduality theorem provides a formalization for the intuition that making types more precise should not impact the partial correctness of the program itself. Specifically, it says that if the types in a program are made more precise, then either the more precise program errors, or exhibits the same behavior as before. This means that a programmer can add types to a portion of their program and know that the program as a whole still operates the same way, unless a new dynamic error is raised, in which case there is a flaw either in the code or in the new annotation that was introduced.

Languages can fail to satisfy the graduality theorem for a variety of reasons but a common culprit is type-directed computation. Whenever a form in a gradual language has behavior that is defined by inspection of the type of an argument, rather than by its behavior, there is a potential for a graduality violation, because the computation must be ensured to be monotone in the type. For instance, the Grace language supports a construct the designers call “structural type tests”. That is, it includes a form $M \mathsf{is} A$ that checks if $M$ has type $A$ at runtime. Boyland [2014] show that care must be taken in designing the semantics of this construct if $A$ is allowed to be an arbitrary type. For instance, it might seem reasonable to say that $(\lambda x : ? . x)$ $\mathsf{is}$ $\top \rightarrow \top$ should run to false because the function has type $? \rightarrow ?$. However, if we increase the precision of the types by changing the annotation, we get $(\lambda x : I . x)$ $\mathsf{is}$ $I \rightarrow I$ which should clearly evaluate to true, violating the graduality principle. In such a system, we can’t think of types as just properties whose precision can be tuned up or down: we also need to understand how changing the type might influence our use of type tests at runtime.

Gradual typing researchers have designed languages that support reasoning principles enabled by a variety of advanced static features—such as objects [Siek and Taha 2007; Takikawa et al. 2012], refinement types [Lehmann and Tanter 2017], union and intersection types [Castagna and Lanvin 2017], typestates [Wolff et al. 2011], effect tracking [Bañados Schwerter et al. 2014], subtyping [Garcia et al. 2016], ownership [Sergey and Clarke 2012], session types [Igarashi et al. 2017b], and secure information flow [Disney and Flanagan 2011; Fennell and Thiemann 2013; Toro et al. 2018]. As these typing features become more complicated, the behavior of casts can become sophisticated as well, and the graduality principle is a way of ensuring that these sophisticated mechanisms stay within programmer expectations.

### 1.1 Polymorphism and Runtime Sealing

Parametric polymorphism, in the form of universal and existential types, allows for abstraction over types within a program. Universal types, written $\forall X.A$, allow for the definition of functions that can be used at many different types. Dually, existential types provide a simple model of a module system. A value of type $\exists X.A$ can be thought of as a module that exports a newly defined
type $X$ and then a value $A$ that may include $X$ that gives the interface to the type. Languages with parametric polymorphism provide very strong reasoning principles regarding data abstraction, formalized by the relational parametricity theorem [Reynolds 1983].

The relational parametricity theorem captures the idea that an abstract type is truly opaque to its users: for instance, a consumer of a value of existential type $\exists X.A$ can only interact with $X$ values using the capabilities provided by the interface type $A$. This allows programmers to use existential types to model abstract data types [Mitchell and Plotkin 1985]. For instance, the existential type $\exists X. X \times (X \rightarrow X) \times (X \rightarrow I)$ represents the type of an abstract functional counter. The $X$ represents the state, the first component of the tuple is the initial state, the second component is an increment function, and the final component reads out an observable integer value from the state. One obvious example implementation would use $I$ for $X$, 0 as the initial state, addition by 1 as the increment, and the identity function as the read-out. In a language with proper data abstraction, we should be able to guarantee that with this implementation, the read-out function should only ever produce positive numbers, because even though the type $I$ allows for negative numbers, the interface only enables the construction of positive numbers. This pattern of reasoning naturally generalizes to sophisticated data structure invariants such as balanced trees, sorted lists, etc.

Polymorphic languages can fail to satisfy the parametricity theorem for a variety of reasons but one common culprit is type-directed computation on abstract types. For instance in Java, values of a generic type $T$ can be cast to an arbitrary object type. If the type $T$ happens to be instantiated with the same type as the cast, then all information about the value will be revealed, and data abstraction is entirely lost. The problem is that the behavior of this runtime type-cast is directed by the type of the input: at runtime the input must carry some information indicating its type so that this cast can be performed. A similar problem arises when naively combining gradual typing with polymorphism, as we will see in §2.

While parametric polymorphism ensures data abstraction by means of a static type discipline, dynamic sealing provides a means of ensuring data abstraction even in a dynamically typed language. To protect abstract data from exposure, a fresh "key" is generated and implementation code must "seal" any abstract values before sending them to untrusted parties, "unsealing" them when they are passed into the exposed interface. For instance, we can ensure data abstraction for an untyped abstract functional counter by generating a fresh key $\sigma$, and producing a tuple where the first component is a 0 sealed with $\sigma$, and the increment and read-out function unseal their inputs and the increment function seals its output appropriately. If this is the only way the seal $\sigma$ is used in the program, then the abstraction is ensured. While the programmer receives less support from the static type checker, this runtime sealing mechanism gives much of the same abstraction benefits.

One ongoing research area has been to satisfactorily combine the static typing discipline of parametric polymorphism with the runtime mechanism of dynamic sealing in a gradually typed language [Ahmed et al. 2011, 2017; Igarashi et al. 2017a; Ina and Igarashi 2011; Toro et al. 2019; Xie et al. 2018]. However, no such language design so far proposed has satisfied both of the desired fundamental theorems: graduality for gradual typing and relational parametricity for parametric polymorphism. Recent work by Toro et al. [2019] claims to prove that graduality and parametricity are inherently incompatible, which backed by analogous difficulties for secure information flow [Toro et al. 2018] has led to the impression that the graduality property is incompatible with parametric reasoning. This would be the wrong conclusion to draw, for the following two reasons. First, the claimed proof has a narrow applicability. It is based on the definition of their logical relation, which we show in §2.3 does not capture a standard notion of parametricity. Second, and more significantly, we should be careful not to conclude that graduality and parametricity are incompatible properties, and that language designs must choose one. In this paper, we reframe the problem: both are desirable, and should be demanded of any gradual or parametric language. The
failure of graduality and parametricity in previous work can be interpreted not as an indictment of these properties, but rather points us to reconsider the combination of System F’s syntax with runtime semantics based on dynamic sealing. In this paper, we will show that graduality and parametricity are not in conflict per se, by showing that by modifying System F’s syntax to make the sealing visible, both properties are achieved. Far from being in opposition to each other, both graduality and parametricity can be proven using a single logical relation theorem (§6).

1.2 Overview
We summarize the contributions of this work as follows

• We identify type-directed computation as the common cause of graduality and parametricity violations in previous work on gradual polymorphism.
• We show that certain polymorphic programs in Toro et al. [2019]’s language GSF exhibit non-parametric behavior.
• We present a new surface language PolyG’ that supports a novel form of universal and existential types where the creation of fresh types is exposed in a controlled way. The semantics of PolyG’ is similar to previous gradual parametric languages, but the explicit type creation and sealing eliminates the need for type-directed computation.
• We elaborate PolyG’ into an explicit cast calculus PolyC’. We then give a translation from PolyC’ into a typed target language, CBPV$_{OSum}$, essentially call-by-push-value with polymorphism and an extensible sum type.
• We develop a novel logical relation that proves both graduality and parametricity for PolyG’. Thus, we show that parametricity and graduality are compatible, and we strengthen the connection alluded to by New and Ahmed [2018] that graduality and parametricity are analogous properties.

Complete typing rules, definitions, and proofs are in the technical appendix [New et al. 2020].

2 GRADUALITY AND PARAMETRICITY, FRIENDS OR ENEMIES?
Next, we review the issues in constructing a polymorphic gradual language that satisfies parametricity and graduality that have arisen in previous work. We see in each case that the common obstacle to parametricity and graduality is the presence of type-directed computation. This motivates our own language design, which obviates the need for type-directed computation by making dynamic sealing explicit in code.

2.1 “Naïve” Attempt
Before considering any dynamic sealing mechanisms, let’s see why the most obvious combination of polymorphism with gradual typing produces a language that does not maintain data abstraction. Consider a polymorphic function of type ∀X.X → B. In a language satisfying relational parametricity, we know that the function must treat its input as having abstract type X and so this input cannot have any influence on what value is returned. However, in a gradually typed language, any value can be cast using type ascriptions, such as in the function ΛX.λx:X.(x :: ?) :: B. Here :: represents a type ascription. In a gradually typed language, a term M of type A can be ascribed a type B if it is “plausible” that an A is a B. This is typically formalized using a type consistency relation ∼ or more generally consistent subtyping relation ⊑, but in either case, it is always plausible that an A is a ? and vice-versa, so in effect a value of any type can be cast to any other by taking a detour through the dynamic type. These ascriptions would then be elaborated to casts producing the term ΛX.λx:X.(B ⇐ ?)(? ⇐ X)x If this function is applied to any value that is not compatible with
While this achieves the goal of maintaining data abstraction, it unfortunately violates graduality, which involves the dual coercion using unsealing. The root-cause of this parametricity violation is that we allow casts like $\langle ? \Leftarrow X \rangle$ whose behavior depends on how $X$ is instantiated. To construct a gradual language with strong data abstraction we must somehow avoid the dependency of $\langle ? \Leftarrow X \rangle$ on $X$. One option is to ban casts like $\langle ? \Leftarrow X \rangle$ altogether. Syntactically, this means changing the notion of plausibility to say that ascribing a term of type $X$ with the dynamic type $?$ is not allowed. This is possible using the system presented by Igarashi et al. [2017a] if you only allow $\lambda$s that use the "static" label. This is compatible with parametricity and graduality, but is somewhat against the spirit of gradual typing, where typically all programs could be written as dynamically typed programs, and dynamically typed functions can be used on values of any type. An alternative is to use dynamic sealing to allow casts like $\langle ? \Leftarrow X \rangle$, but ensure that their behavior does not depend on how $X$ is instantiated.

2.2 Type-directed Sealing

In sealing-based gradual parametric languages like $\lambda B[\text{Ahmed et al. 2011, 2017}]$, we ensure that casts of abstract type do not depend on their instantiation by adding a layer of indirection. Instead of the usual $\beta$ rule for polymorphic functions

$$(\Lambda X.M)[A] \mapsto M[A/X],$$

in $\lambda B$, we dynamically generate a fresh type $\alpha$ and pass that in for $X$. This first of all means the runtime state must include a store of fresh types, written $\Sigma$. When reducing a type application, we generate a fresh type $\alpha$ and instantiate the function with this new type

$$\Sigma; (\Lambda X.M)[A] \mapsto \Sigma, \alpha := A; M[\alpha/X]$$

In this case, we interpret $\alpha$ as being a new tag on the dynamic type that tags values of type $A$ but is different from all previously used tags. The casts involving $\alpha$ are treated like a new base type, incompatible with all existing types. However, if we look at the resulting term, it is not well-typed: if the polymorphic function has type $\forall X.B$, then $M[\alpha/X]$ has type $B[\alpha/X]$, but the context of this term expects it to be of type $B[A/X]$. To paper over this difference, $\lambda B$ wraps the substitution with a type-directed coercion, distinct from casts, that mediates between the two types:

$$\Sigma; (\Lambda X.M)[A] \mapsto \Sigma, \alpha := A; M[\alpha/X] : B[\alpha/X] \xrightarrow{+\alpha} B[A/X]$$

This type-directed coercion $M[\alpha/X] : B[\alpha/X] \xrightarrow{+\alpha} B[A/X]$ is the part of the system that performs the actual sealing and unsealing, and is defined by recursion on the type $B$. The $+\alpha$ indicates that we are unsealing values in positive positions and sealing at negative positions. For instance if $B = X \times \mathbb{B}$, and $X = \mathbb{B}$, then on a pair $(\text{seal}_\alpha \text{true, false})$ the coercion will unseal the sealed boolean on the left and leave the boolean on the right alone. If $B$ is of function type, the definition will involve the dual coercion using $-\alpha$, which seals at positive positions. So for instance applying the polymorphic identity function will reduce as follows

$$\Sigma; (\Lambda X.\lambda x : X.x)[\mathbb{B}]\text{true} \mapsto \Sigma, \alpha := \mathbb{B}; (\lambda x : \alpha.x : \alpha \rightarrow \alpha \xrightarrow{+\alpha} \mathbb{B} \xrightarrow{\mathbb{B}} \text{true}$$

$$\mapsto \Sigma, \alpha := \mathbb{B}; (\lambda x : \alpha.x)\text{(true : X \xrightarrow{a} a) : a \xrightarrow{+\alpha} X} \mapsto \Sigma, \alpha := \mathbb{B}; (\lambda x : \alpha.x)(\text{seal}_\alpha \text{true}) : \alpha \xrightarrow{+\alpha} X$$

$$\mapsto \Sigma, \alpha := \mathbb{B}; \text{seal}_\alpha \text{true : a} \xrightarrow{+\alpha} X \mapsto \text{true}$$

While this achieves the goal of maintaining data abstraction, it unfortunately violates graduality, as first pointed out by Igarashi et al. [2017a]. The reason is that the coercion is a type-directed computation, this time directed by the type $\forall X.B$ of the polymorphic function, whose behavior
observably differs at type X from its behavior at type \( \alpha \). Specifically, a coercion \( M : X \rightarrow^\alpha \alpha \) results in sealing the result of \( M \), whereas if \( X \) is replaced by dynamic, then \( M : ? \rightarrow^\alpha \alpha \) is an identity function. An explicit counter-example is given by modifying the identity function to include an explicit annotation. The term \( M_1 = (\lambda x : X. x : X)[B] \text{true} \) reduces by generating a seal \( \alpha \), sealing the input true with \( \alpha \), then unsealing it, finally producing true. On the other hand, if the type of the input were dynamic rather than \( X \), we would get a term \( M_2 = (\lambda x : ?.(x :: X))[B] \text{true} \). In this case, the input is not sealed by the implementation, and the ascription of \( X \) results in a failed cast since \( B \) is incompatible with \( \alpha \). The only difference between the two terms is a type annotation, meaning that \( M_1 \subseteq M_2 \) in the term precision ordering (\( M_1 \) is more precise than \( M_2 \)), and so the graduality theorem states that if \( M_1 \) does not error, it should behave the same as \( M_2 \), but in this case \( M_2 \) errors while \( M_1 \) does not. The problem here is that the type of the polymorphic function determines whether to seal or unseal the inputs and outputs, but graduality says that the behavior of the dynamic type must align with both abstract types \( X \) (indicating sealing/unsealing) and concrete types like \( B \) (indicating no sealing/unsealing). These demands are contradictory since dynamic code would have to simultaneously be opaque until unsealing and available to interact with. So we see that the attempt to remove the type-directed casts which break parametricity by using dynamic sealing led to the need for a type-directed coercion which breaks graduality.

### 2.3 To Seal, or not to Seal

The language GSF was introduced by Toro et al. [2019] to address several criticisms of the type system and semantics of \( \lambda B \). We agree with the criticisms of the type system and so we will focus on the semantic differences. GSF by design has the same violation of graduality as \( \lambda B \), but has different behavior when using casts.

One motivating example for GSF is what happens when casting the polymorphic identity function to have a dynamically typed output: \(((\lambda x : X. x :: X) :: \forall X.X \rightarrow ?) [I] 1) + 2 \). In \( \lambda B \), the input 1 is sealed as dictated by the type, but the dynamically typed output is not unsealed when it is returned from the function, resulting in an error when we try to add it. Ahmed et al. [2011] argue that it should be a free theorem that the behavior of a function of type \( \forall X.X \rightarrow ? \) should be independent of its argument: it always errors, diverges or it always returns the same dynamic value, based on the intuition that the dynamic type \( ? \) does not syntactically contain the free variable \( X \), and that this free theorem holds in System F. This reasoning is suspect since at runtime, the dynamic type does include a case for the freshly allocated type \( X \), so intuitively we should consider \( ? \) to include \( X \) (and any other abstract types in scope).

Toro et al. [2019] argue on the other hand that intuitively the identity function was written with the intention of having a sealed input that is returned and then unsealed, and so casting the program to be more dynamic should result in the same behavior and so the program should succeed. The function application runs to the equivalent of \( (? \leftrightarrow I) 1 \) which is then cast to \( I \) and added to 2, resulting in the number 3. The mechanism for achieving this semantics is a system of runtime evidence, based on the Abstracting Gradual Typing (AGT) framework [Garcia et al. 2016]. An intuition for the behavior is that the sealing is still type-directed, but rather than being directed by the static type of the function being instantiated, it is based on the most precise type that the function has had. So here because the function was originally of type \( \forall X.X \rightarrow X \), the sealing behavior is given by that type.

However, while we agree that the analysis in Ahmed et al. [2011] is incomplete, the behavior in GSF is inherently non-parametric, because the polymorphic program produces values with different dynamic type tags based on what the input type is. As a user of this function, we should be able to replace the instantiating type \( I \) with \( B \) and give any boolean input and get related behavior at
the type $\exists$, but in the program $((\Lambda X.\lambda x : X.x) :: \forall X.X \to ?)[B]\text{true} + 2$ the function application reduces to $(\exists \leftarrow B)\text{true}$ which errors when cast to $I$. Intuitively, this behavior is not parametric because the first program places an $I$ tag on its input, and the second places a $B$ tag on its input.

The non-parametricity is clearer if we look at a program of type $\forall X.? \to B$ and consider the following function, a constant function with abstract input type cast to have dynamic input:

$$\text{const} = (\Lambda X.\lambda x : X.\text{true}) :: \forall X.? \to B$$

$X$ now has no effect on static typing, so both $\text{const}[I]3$ and $\text{const}[B]$ are well-typed. However, since the sealing behavior is actually determined by the type $\forall X.X \to B$, the program will try to seal its input after downcasting it to whatever type $X$ is instantiated at. So the first program casts $\langle I \leftarrow ?\rangle(\exists \leftarrow I)3$, which succeeds and returns $\text{true}$, while the second program performs the cast $\langle B \leftarrow ?\rangle(\exists \leftarrow I)3$ which fails. In effect, we have implemented a polymorphic function that for any type $X$, is a recognizer of dynamically typed values for that type, returning $\text{true}$ if the input matches $X$ and erring otherwise. Any implementation of this behavior would clearly require passing of some syntactic representation of types at runtime.

Formally, the GSF language does not satisfy the following *defining principle of relational parametricity*, as found in standard axiomatizations of parametricity such as Dunphy [2002]; Ma and Reynolds [1991]; Plotkin and Abadi [1993]. In a parametric language, the user of a term $M$ of a polymorphic function type $\forall X.A \to B$ should be guaranteed that $M$ will behave uniformly when instantiated multiple times. Specifically, a programmer should be able to instantiate $M$ with two different types $B_1, B_2$ and choose any relation $R \in \text{Rel}[B_1, B_2]$ (where the notion of relation depends on the type of effects present), and be ensured that if they supply related inputs to the functions, they will get related outputs. Formally, for a Kripke-style relation, the following principle should hold:

$$\frac{M : \forall X.A \to B \quad R \in \text{Rel}[B_1, B_2] \quad (w, V_1, V_2) \in \mathcal{V}[A] \rho [X \mapsto R]}{(w, M[B_1]V_1, M[B_2]V_2) \in \mathcal{E}[B] \rho [X \mapsto R]}$$

Here $w$ is a “world” that gives the invariants in the store and $\rho$ is the relational interpretation of free variables. $\mathcal{V}[\cdot]$ and $\mathcal{E}[\cdot]$ are value and expression relations formalizing an approximation ordering on values and expressions respectively, and $X \mapsto R$ means that the relational interpretation of $X$ is given by $R$.

Toro et al. [2019] use an unusual logical relation for their language based on a similar relation in Ahmed et al. [2017], so there is no direct analogue of the relational mapping $X \mapsto R$. Instead, the application extends the world with the association of $a$ to $R$ and the interpretation sends $X$ to $\alpha$. However, we can show that this parametricity principle is violated by any $\rho$ we pick for the term $\text{const}$ above, using the definition of $\mathcal{E}[\cdot]$ given in [Toro et al. 2019]. Instantiating the lemma would give us that $(w, \text{const}[I]3, \text{const}[B]3) \in \mathcal{E}[B] \rho [\exists \leftarrow ?] \rho$ since $(w, 3, 3) \in \mathcal{V}[?] \rho$ for any $\rho$. The definition of $\mathcal{E}[B] \rho$ then says (again for any $\rho$) that it should be the case that since $\text{const}[I]3$ runs to a value, it should also be the case that $\text{const}[B]3$ runs to a value as well, but in actuality it errors, and so this parametricity principle must be false.

How can the above parametricity principle be false when Toro et al. [2019] prove a parametricity theorem for GSF? We have not found a flaw in their *proof*, but rather a mismatch between their *theorem* statement and the expected meaning of parametricity. The definition of $\mathcal{V}[\forall X.A]$ in Toro et al. [2019] is not the usual interpretation, but rather is an adaptation of a non-standard definition used in Ahmed et al. [2017]. Neither of their definitions imply the above principle, so we argue that neither paper provides a satisfying proof of parametricity. With GSF, we see that the above behavior violates some expected parametric reasoning, using the definition of $\mathcal{V}[?]$ given in Toro

---

1They use slightly different notation, but we use notation that matches the logical relation we present later
et al. [2019]. With $\lambda B$, we know of no counterexample to the above principle, and we conjecture that it would satisfy a more standard formulation of parametricity.

It is worth noting that the presence of effects—such as nontermination, mutable state, control effects—requires different formulations of the logical relation that defines parametricity. However, those logical relations capture parametricity in that they always formalize uniformity of behavior across different type instantiations. For instance, for a language that supports nontermination, the logical relation for parametricity ensures that two different instantiations have the same termination behavior: either both diverge, or they both terminate with related values. Because of this, the presence of effects usually leads to weaker free theorems—in pure System F all inhabitants of $\forall X. X \to X$ are equivalent to the identity function, but in System F with non-termination, every inhabitant is either the identity or always errors. Though the free theorems are weaker, parametricity still ensures uniformity of behavior. As our counterexample above (const[I]3 vs. const[B]3) illustrates, GSF is non-parametric since it does not ensure uniform behavior. However, since the difference in behavior was between error and termination, it is possible that GSF satisfies a property that could be called “partial parametricity” (or parametricity modulo errors) that weakens the notion of uniformity of behavior: either one side errors or we get related behaviors. However, it is not clear to us how to formulate the logical relation for the dynamic type to prove this. We show how this weakened reasoning in the presence of $?$ compares to reasoning in our language PolyG$^\nu$ in §6.4.

Our counter-example crucially uses the dynamic type, and we conjecture that when the dynamic type does not appear under a quantifier, that the usual parametric reasoning should hold in GSF. This would mean that in GSF once polymorphic functions become “fully static”, they support parametric reasoning, but we argue that it should be the goal of gradual typing to support type-based reasoning even in the presence of dynamic typing, since migration from dynamic to static is a gradual process, possibly taking a long time or never being fully completed.

### 2.4 Resolution: Explicit Sealing

Summarizing the above examples, we see that

1. The naïve semantics leads to type-directed casts at abstract types, violating parametricity.
2. $\lambda B$’s type-directed sealing violates graduality because of the ambiguity of whether or not the dynamic type indicates sealing/unsealing or not.
3. GSF’s variant of type-directed sealing based on the most precise type violates graduality and parametricity because the polymorphic function gets to determine which dynamically typed values are sealed (i.e. abstract) and which are not.

We see that in each case, the use of a type-directed computational step leads to a violation of graduality or parametricity. The GSF semantics makes the type-directed sealing of $\lambda B$ more flexible by using the runtime evidence attached to the polymorphic function rather than the type at the instantiation point, but unfortunately this makes it impossible for the continuation to reason about which dynamically typed values it passes will be treated as abstract or concrete. This analysis motivates our own language design PolyG$^\nu$, where

1. We depart from the syntax of System F.
2. Sealing/unsealing of values is explicit and programmable, rather than implicit and type-directed.
3. The party that instantiates an abstract type is the party that determines which values are sealed and unsealed. For existential types, this is the package (i.e., the module) and dually for universal types it is the continuation of the instantiation.
The dynamic semantics of PolyG$^ν$ are similar to $\lambda B$ without the type-directed coercions, removing the obstacle to proving the graduality theorem. By allowing user-programmable sealing and unsealing, more complicated forms of sealing and unsealing are possible: for instance, we can seal every prime number element of a list, which would require a very rich type system to express using type-directed sealing! We conjecture that the language is strictly more expressive than $\lambda B$ in the sense of Felleisen [1990]: $\lambda B$ should be translatable into PolyG$^ν$ in a way that simulates its operational semantics. Because the sealing is performed by the instantiating party rather than the abstracting party, the expressivity of PolyG$^ν$ is incomparable to GSF.

3 PolyG$^ν$: A GRADUAL LANGUAGE WITH POLYMORPHISM AND SEALING

Next, we present our gradual language, PolyG$^ν$, that supports a variant of existential and universal quantification while satisfying parametricity and graduality. The language has some unusual features, so we start with an extended example to illustrate what programs look like, and then in § 3.2 introduce the formal syntax and typing rules.

3.1 PolyG$^ν$ Informally

Let’s consider an example of existential types, since they are simpler than universal types in PolyG$^ν$. In a typed, non-gradual language, we can define an abstract “flipper” type, FLIP = $\exists X. X \times (X \rightarrow X) \times (X \rightarrow B)$. The first element is the initial state, the second is a “toggle” function and the last element reads out the value as a concrete boolean.

Then we could create an instance of this abstract flipper using booleans as the carrier type $X$ and negation as the toggle function pack($B$, $(\text{true}, (\text{NOT}, \text{ID}))$) as FLIP. Note that we must explicitly mark the existential package with a type annotation, because otherwise we wouldn’t be able to tell which occurrences of $B$ should be hidden and which should be exposed. With different type annotations, the same package could be given types $\exists X. B \times ([B \rightarrow B] \times [B \rightarrow B])$ or $\exists X. X \times (X \rightarrow X) \times (X \rightarrow X)$.

The PolyG$^ν$ language existential type works differently in a few ways. We write $\exists^ν$ rather than $\exists$ to emphasize that we are only quantifying over fresh types, and not arbitrary types. The equivalent of the above existential package would be written as

$$\text{pack}^ν(X \equiv B, (\text{seal}_X \text{true}, ((\lambda x : X. \text{seal}_X \text{NOT}(\text{unseal}_X x)), (\lambda x : X. \text{unseal}_X x))): \text{FLIP}$$

The first thing to notice is that rather than just providing a type $B$ to instantiate the existential, we write a declaration $X \equiv B$. The $X$ here is a binding position and the body of the package is typed under the assumption that $X \equiv B$. Then, rather than substituting $B$ for $X$ when typing the body of the package, the type checker checks that the body has type $X \times ((X \rightarrow X) \times (X \rightarrow B))$ under the assumption that $X \equiv B$:

$$X \equiv B \vdash (\text{seal}_X \text{true}, ((\lambda x : X. \text{seal}_X \text{NOT}(\text{unseal}_X x)), (\lambda x : X. \text{unseal}_X x))): X \times ((X \rightarrow X) \times (X \rightarrow B))$$

Crucially, $X \equiv B$ is a weaker assumption than $X = B$. In particular, there are no implicit casts from $X$ to $B$ or vice-versa, but the programmer can explicitly “seal” $B$ values to be $X$ using the form seal$_X M$, which is only well-typed under the assumption that $X = A$ for some $A$ consistent with $B$. We also get a corresponding unseal form unseal$_X M$, and the runtime semantics in § 4.4 defines these to be a bijection. At runtime, $X$ will be a freshly generated type with its own tag on the dynamic type. An interesting side-effect of making the difference between $X$ and $B$ explicit in the term is that existential packages do not require type annotations to resolve any ambiguities. For instance, unlike in the typed case, the gradual package above could not be ascribed the type $\exists^ν X. B \times ((B \rightarrow B) \times (B \rightarrow B))$ because the functions explicitly take $X$ values, and not $B$ values.

The corresponding elimination form for $\exists^ν$ is a standard unpack: unpack $(X, x) = M; N$, where the continuation for the unpack is typed with just $X$ and $x$ added to the context, it doesn’t know...
that $X \equiv A$ for any particular $A$. We call this ordinary type variable assumption an *abstract type variable*, whereas the new assumption $X \equiv A$ is a *known type variable* which acts more like a *type definition* than an abstract type. At runtime, when an existential is unpacked, a fresh type $X$ is created that is isomorphic to $A$ but whose behavior with respect to casts is different.

While explicit sealing and unsealing might seem burdensome to the programmer, note that this is directly analogous to a common pattern in Haskell, where modules are used in combination with newtype to create a datatype that at runtime is represented in the same way as an existing type, but for type-checking purposes is considered distinct. We give an analogous Haskell module as follows:

```haskell
module Flipper(State, start, toggle, toBool) where
newtype State = Seal { unseal :: Bool }
start :: State
start = Seal True
inc :: State -> State
inc s = Seal (not (unseal s))
toBool :: State -> Bool
toBool = unseal
```

Then a different module that imports Flipper is analogous to an unpack, as its only interface to the `State` type is through the functions provided.

We also add universal quantification to the language, using the duality between universals and existentials as a guide. Again we write the type differently, as $\forall \nu. X. A$. In an ordinary polymorphic language, we would write the type of the identity function as $\forall X. X \rightarrow X$ and implement it using a $\Lambda$ form: $\Lambda X. \lambda x : X.x$. The elimination form passes in a type for $X$. For instance applying the identity function to a boolean would be written as $\text{ID} \{ X \rightarrow B \}(\text{seal} X \text{true}) : B$. which introduces a known type variable $X \equiv B$ into the context. Rather than the resulting type being $B \rightarrow B$, it is $X \rightarrow X$ with the assumption $X \equiv B$. Then the argument to the function must be explicitly sealed as an $X$ to be passed to the function. The output of the function is also of type $X$ and so must be explicitly unsealed to get a boolean out. However, there is something quite unusual about this term: the $X \equiv B$ binding site is not binding $X$ in a subterm of the application, but rather into the context: the argument is sealed, and the continuation is performing an unseal! These bindings in $\forall \nu$ instantiations follow this “inside-out” structure and complicate the typing rules: every term in the language “exports” known type variable bindings that go outwards in addition to the other typing assumptions coming inwards from the context. While unusual, they are intuitively justified by the duality with existentials: we can think of the continuation for an instantiation of a $\forall \nu$ as being analogous to the body of the existential package.

To get an understanding of how PolyG$^\nu$ compares to $\lambda B$ and GSF and why it avoids their violation of graduality, let’s consider how we might write the examples from the previous section. In PolyG$^\nu$, if we apply a function of type $\forall X. X \rightarrow X$, we have to mark explicitly that the input is sealed, and furthermore if we want to use the output as a boolean, we must unseal the output:

$\text{unseal}_X((\lambda X. \lambda x : X.x :: X)(X \equiv B)(\text{seal}_X \text{true})): \star \rightarrow \text{true}$

Then if we change the type of the input from $X$ to $\nu$ the explicit sealing and unsealing remain, so even though the input is dynamically typed it will still be a sealed boolean, and the program exhibits the same behavior:
we include sealing
seal
which is well formed when the types are consistent
with the ordinary typing assumptions \( x \) \( \sim \) \( X \) explicit and not implicit, because the behavior of casts involving \( X \) explicit bijection between the types \( X \) and \( X \) explicit is standard in as being the least congruence relation including equality and rules making \( M \) connectives, but lax when the dynamic type is involved. The first in the style of \[ \text{Siek and Taha 2006} \]: type checking is strict when checking compatibility of different necessary if this type was removed. We follow the usual formulation of gradual surface languages the rules, which only concern the inside-out scoping needed for the \( \Gamma \) type variable assumptions \( \exists \) \( X.\alpha \) ascription \( M \) language is a typical gradual functional language, using \( ? \) \( X \) does not occur in \( \alpha \) \( \sim \) \( B \) \( \sim \) \( X \) \( ? \) \( \Gamma \) \( X \) \( \Gamma \) \( X \sim A \) \( A \) \( X \) \( B \) \( \Gamma \) \( x : A \) \( \Gamma \), \( X \) \( \Gamma \), \( X \sim A \) \( \Gamma \), \( x : A \) \( \Gamma \), \( X \) \( \Gamma \), \( X \sim A \)

\[
\text{unseal}_X((\lambda X . \lambda x : ? . x :: X)(X \equiv B)(\text{seal}_X \text{true})) \mapsto^* \text{true}
\]

If we remove the seal of the input, then the cast to \( X \) in the function will fail, giving us the behavior of \( \lambda B / \text{GSF} \):

\[
\text{unseal}_X((\lambda X . \lambda x : ? . x :: X)(X \equiv B)(\text{true})) \mapsto^* U
\]

but crucially this involved changing the term, not just the type, so the graduality theorem does not tell us that the programs should have related behavior.

Next, let’s consider the parametricity violation from GSF. When we instantiate the constant \( \text{const} \) \( X \equiv B \) \( \text{seal}_X \) then the program is not well typed because \( X \equiv B \) and \( 3 \) has type \( I \) which is not compatible. We can replicate the outcome of the GSF program by not sealing the \( 3 \):

\[
\text{const}(X \equiv B) \mapsto^* U
\]

But this is not a parametricity violation because the \( 3 \) here will be embedded at the dynamic type with the \( I \) tag, whereas above the \( 3 \) was tagged with the \( X \) tag, which is not related.

\subsection{PolyG$^\nu$ Formal Syntax and Semantics}

Figure 1 presents the syntax of the surface language types, terms and environments. Most of the language is a typical gradual functional language, using \( ? \) as the dynamic type, and including type ascription \( M :: A \). The unusual aspects of the language are the \( \text{seal}_X \) and \( \text{unseal}_X \) forms and the “fresh” existential \( \exists X . \alpha \) and universal \( \forall X . \alpha \). Note also the non-standard environments \( \Gamma \), which include ordinary typing assumptions \( \alpha : A \), abstract type variable assumptions \( X \) and known type variable assumptions \( X \equiv A \). For simplicity, we assume freshness of all type variable bindings, i.e. when we write \( \Gamma, X \) or \( \Gamma, X \equiv A \) that \( X \) does not occur in \( \Gamma \).

The typing rules are presented in Figure 2. On a first pass, we suggest ignoring all shaded parts of the rules, which only concern the inside-out scoping needed for the \( \forall X . \alpha \) forms and would not be necessary if this type was removed. We follow the usual formulation of gradual surface languages in the style of \[ \text{Siek and Taha 2006} \]: type checking is strict when checking compatibility of different connectives, but lax when the dynamic type is involved. The first \( M :: B \) form is type-ascription, which is well formed when the types are consistent with each other, written \( A \sim B \). We define this in the standard way in as being the least congruence relation including equality and rules making \( ? \) consistent with every type.

We include variable and let-binding rules, which are standard other than the shaded parts. Next, we include sealing \( \text{seal}_X \) and unsealing \( \text{unseal}_X \) forms. The sealing and unsealing forms are valid when the assumption \( X \equiv A \) is in the environment and give the programmer access to an explicit bijection between the types \( X \) and \( A \). It is crucial for graduality to hold that this bijection is explicit and not implicit, because the behavior of casts involving \( X \) and \( A \) are very different. To
show that it has no adverse effect on the calculus, we also include a form \( \text{is}(G)\)? \( M \) that checks at runtime whether \( M \) returns a value that is compatible with the ground type \( G \). \( M \) can have any type in this case because it is always a safe operation, but the result is either trivially true or false unless \( M \) has type \( ? \).
Next, we have booleans, whose values are true and false, and whose elimination form is an if-statement. The if-statement checks that the scrutinee has a type compatible with \( \mathbb{B} \), and as in previous work uses gradual meet \( B_t \sqcap B_f \) for the output type [Garcia and Cimini 2015]. Gradual meet is only partially defined, since this ensures that if the two sides have different (non-?) head connectives then type checking errors, in keeping with the philosophy of strict checking when precise types are used.

Next, we have pairs and functions, which are fairly standard. We use pattern-matching as the elimination form for pairs. To reduce the number of rules, we present the elimination forms in the style of Garcia and Cimini [2015], using partial functions \( \pi_i \), \( \text{dom} \), \( \text{cod} \) and later \( \text{un}\nu^\pi \), \( \text{un}\exists \nu^\pi \) to extract the subformula from a type “up to \( ? \)”. For the correct type this extracts the actual subformula, but for \( ? \) is defined to be \( ? \) and for other connectives is undefined. We define these at the bottom of the figure, where uncovered cases are undefined. Next, we have existentials, which are as described in §3.1.

Finally, we consider the shaded components of the judgment. The full form of the judgment is \( \Gamma \vdash M : A; \Gamma_o \) where \( \Gamma_o \) is the list of bindings that are generated by \( M \) and exported outward. Note that the type \( A \) of \( M \) can use variables in \( \Gamma_o \) as well as variables in \( \Gamma \). Also, while we write these as \( \Gamma_o, \Gamma_M \), etc., they only contain sequences of known type variables, and never any abstract type variables or typing assumptions \( x : A \). These bindings are generated in the \( V^\nu \) elimination rule, where the instantiation \( M\{X \equiv B\} \) adds \( X \equiv B \) to the output context. Rules that produce delayed thunks—the function, existential and universal introduction rules—have bodies that generate bindings, but these are not exported because these bindings will only be generated at the point where the thunk is forced to evaluate. The rest of the rules work similarly to an effect system: for instance in the function application rule \( M \ N \) the bindings generated in \( M \) are bound in \( N \), and the application produces all of the bindings they generate, and similarly for product introduction. In the unpack form, care must be taken to make sure that the \( X \) from the unpack is not leaked in the output \( \Gamma_N \), in addition to making sure the output type \( B \) does not mention \( X \). Any known type variables that mention \( X \) are removed from the output context, using the restriction form \( \Gamma^{-} \) defined at the bottom of Figure 2. Finally, in the if form, each branch might export different known type variables, so the if statement as a whole only exports the intersection of the two branches, since these are the only ones guaranteed to be generated.

4 PolyC\(^{\nu}\): CAST CALCULUS

As is standard in gradual languages, rather than giving the surface language an operational semantics directly, we define a cast calculus that makes explicit the casts that perform the dynamic type checking in gradual programs. We present the cast calculus syntax in Figure 3. The cast calculus syntax is almost the same as the surface syntax, though the typing is quite different. First, the type ascription form is removed, and several forms are added to replace it. Based on the analysis in [New and Ahmed 2018], we add two cast forms: an upcast \( \langle A^c \rangle \uparrow M \) and a downcast \( \langle A^c \rangle \downarrow M \), whereas most prior work includes a single cast form \( \langle A \leftarrow B \rangle \). The \( A^c \) used in the upcast and downcast forms here is a proof that \( A_t \sqsubseteq A_r \) for some types \( A_t, A_r \), i.e., that \( A_t \) is a more precise (less dynamic) type than \( A_r \). This type precision definition is key to formalizing the graduality property, but previous work has shown that it is useful for formalizing the semantics of casts as well. We emphasize the structure of these proofs because the central semantic constructions of this work: the operational semantics of casts, the translation of casts into functions and finally our graduality logical relation are all naturally defined by recursion on these derivations.

4.1 PolyC\(^{\nu}\) Type Precision

We present the definition of type precision in Figure 4. The judgment \( \Gamma \vdash A^c : A_t \sqsubseteq A_r \) is read as “using the variables in \( \Gamma^c \), \( A^c \) proves that \( A_t \) is more precise/less dynamic than \( A_r \). If you ignore the
while we give a syntax for derivations, there is at most one

The static type system for the cast calculus is given Figure 5. The cast calculus type system differs from the surface language in that all type checking is strict and precise. This manifests in two ways. First, the dynamic type is not considered implicitly compatible with other types. Instead, in the translation from PolyG to PolyC, we insert casts wherever consistency is used in the judgment. Second, in the if rule, the branches must have the same type, and an upcast is inserted in the translation to make the two align. Finally, the outward scoping of known type variables is handled more explicitly. We add a new form hide $X \equiv A; M$ that delimits the scope of $X \equiv A$ from going further outward, enforced by the side condition that $\Gamma, \Gamma_M \vdash \Gamma'$. Then in rules that include

| type names | $\alpha$ ::= $\sigma \mid X$ |
| types | $A, B$ ::= $\sigma$ |
| ground types | $G$ ::= $\alpha \mid B \mid ? \times ? \mid \? \rightarrow ? \mid \exists X. ? \mid \forall X. ?$ |
| precision derivations | $\Lambda{A}, B{\Lambda}$ ::= $? \mid \text{tag}_G(A{\Lambda}) \mid \alpha \mid B \mid A{\Lambda} \times A{\Lambda} \mid A{\Lambda} \rightarrow B{\Lambda}$ |
| $\exists X. A{\Lambda} \mid \forall X. A{\Lambda}$ |
| values | $V$ ::= $\text{seal}_\alpha V \mid \text{true} \mid \text{false} \mid x \mid (V, V) \mid \lambda(x : A). M$ |
| $| A^X. M \mid \text{inj}_G V \mid (A{\Lambda} \rightarrow A{\Lambda})\uparrow M \mid (A{\Lambda} \rightarrow A{\Lambda})\downarrow M$ |
| $\mid \forall (\forall X. A{\Lambda})\uparrow M \mid (\forall (\forall X. A{\Lambda})\downarrow M \mid \text{pack}(X \equiv A',[A{\Lambda} \downarrow])$, $M)$ |
| expressions | $M, N$ ::= $(M : A)$ |
| ::= $U \mid \langle A{\Lambda}\rangle\uparrow M \mid A{\Lambda}\downarrow M \mid \text{hide}(X \equiv A; M) \mid \text{inj}_G M$ |
| $| \text{pack}(X \equiv A',[A{\Lambda} \downarrow],...,M) \mid \text{seal}_\alpha M \mid \text{unseal}_\alpha M$ |
| Evaluation Context | $E$ ::= $[\_] \mid (E, M) \mid (V, E) \mid E[A] \mid E M \mid V E \mid \text{inj}_G E$ |
| $| \text{if } E \text{ then } M \text{ else } M \mid \text{let } (x, y) = E, M \mid \langle A{\Lambda}\rangle\uparrow E$ |
| $| \text{unpack}(X, x) = E, M \mid \text{seal}_\alpha E \mid \text{unseal}_\alpha E \mid \langle A{\Lambda}\rangle\downarrow E$ |

Fig. 3. PolyC$^\forall$ Syntax

Fig. 4. PolyC$^\forall$ Type Precision

precision derivations, our definition of type precision is a simple extension of the usual notion: type variables are only related to the dynamic type and themselves, and similarly for $\forall$ and $\exists$. Since we have quantifiers and type variables, we include a context $\Gamma$ of known and abstract type variables. Crucially, even under the assumption that $X \equiv A, X$ and $A$ are unrelated precision-wise unless $A$ is ?. As before, $X \in \Gamma$ ranges over both known and abstract type variables. It is easy to see that precision reflexive and transitive, and that ? is the greatest element. Finally, ? is the least precise type, meaning for any type $A$ there is a derivation that $A \subseteq ?$. The precision notation is a natural extension of the syntax of types: with base types $?, B$ serving as the proof of reflexivity at the type and constructors $\times, \rightarrow$, etc. serving as syntax for congruence proofs. It is important to note that while we give a syntax for derivations, there is at most one derivation $\Lambda{A}$ that proves any given $A_l \subseteq A_r$. We prove these and several more lemmas about type precision in the appendix [New et al. 2020].

4.2 PolyC$^\forall$ Type System

The static type system for the cast calculus is given Figure 5. The cast calculus type system differs from the surface language in that all type checking is strict and precise. This manifests in two ways. First, the dynamic type is not considered implicitly compatible with other types. Instead, in the translation from PolyG$^\forall$ to PolyC$^\forall$, we insert casts wherever consistency is used in the judgment. Second, in the if rule, the branches must have the same type, and an upcast is inserted in the translation to make the two align. Finally, the outward scoping of known type variables is handled more explicitly. We add a new form hide $X \equiv A; M$ that delimits the scope of $X \equiv A$ from going further outward, enforced by the side condition that $\Gamma, \Gamma_M \vdash \Gamma'$.
We define the elaboration of PolyG to PolyC as well. The dynamic semantics of PolyC is formalized using the metafunction \( \Gamma \vdash \exists \cdot A \subseteq A_r \) for delayed computations, i.e., values of function, existential and universal type, whereas in the surface language the delayed term could produce any names, now in PolyC, they must all be manually hidden. Similarly in the branches of an if statement, the two sides must have the same generated names, and hides must be used in the elaboration to make them align.

### 4.3 Elaboration from PolyG to PolyC

We define the elaboration of PolyG into the cast calculus PolyC in Figure 6. Following [New and Ahmed 2018], an ascription is interpreted as a cast up to ? followed by a cast down to the ascribed type. Most of the elaboration is standard, with elimination forms being directly translated to the corresponding PolyC form if the head connective is correct, and inserting a downcast if the elimination position has type ?. We formalize this using the metafunction \( G \vdash ? \) which defined towards the bottom of the figure. For the if case, in PolyC the two branches of the if have to have the same output type and export the same names, so we downcast each branch, but also we hide any names not generated by both sides, using the metafunction hide \( \Gamma \subseteq \Gamma' \) defined at the bottom of the figure, which hides all names present in \( \Gamma' \) that are not in \( \Gamma \). Finally, in the values that are thunks (pack, \( \lambda \) and \( \lambda \)), the bodies of the thunks must not generate names in PolyC, so we hide names there as well.

### 4.4 PolyC Dynamic Semantics

The dynamic semantics of PolyC, presented in Figure 7, extends traditional cast semantics with appropriate rules for our name-generating universals and existentials. The runtime state is a pair of a term \( M \) and a case store \( \Sigma \). A case store \( \Sigma \) represents the set of cases allocated so far in the
\[(M : B)^+ = (B^\Sigma)^\downarrow \{A^\Sigma\}^\uparrow M^+ \quad \text{(where } M : A, A^\Sigma : A \subseteq ?, B^\Sigma : B \subseteq ?)\]
\[x^+ = x\]
\[\text{(let } x = M; N)^+ = \text{let } x = M^+; N^+\]
\[\text{(seal}_X\text{)} M^+ = \text{seal}_X(M : A)^+ \quad \text{(where } X \cong A)\]
\[\text{(unseal}_X\text{)} M^+ = \text{unseal}_X(X^M)\]
\[\text{(is}(G) \cdot M)^+ = \text{is}(G) \cdot (\{A^\Sigma\}^\uparrow M) \quad \text{(where } M : A, A^\Sigma : A \subseteq ?)\]
\[b^+ = b \quad (b \in \{\text{true, false}\})\]
\[(\text{if } M \text{ then } N_f \text{ else } N_f)^+ = \text{if } B \ni \xi M \text{ then } ((B^\Sigma)^\downarrow \text{hide } \Gamma_f \subseteq \Gamma_b \cap \Gamma_f; N_f^+) \text{ else } ((B^\Sigma)^\downarrow \text{hide } \Gamma_f \subseteq \Gamma_b \cap \Gamma_f; N_f^+)\]
\[(\text{and } B^\Sigma : B_f \cap B_f; \Gamma_M, \Gamma_f \cap \Gamma_f)\]
\[\text{(M}_1, M_2)^+ = (M_1^+, M_2^+)\]
\[\text{(let } (x, y) = M; N)^+ = \text{let } (x, y) = ? x ? \xi M; N^+\]
\[(\lambda x : A.M)^+ = \lambda x : \text{hide } \Gamma_0 \subseteq \Gamma_0; M^+ \quad \text{(where } \Gamma, x : A + M : B; \Gamma_0)\]
\[(N \ni \lambda x : A.M)^+ = (\lambda x : \text{hidden } \Gamma_0 \subseteq \Gamma_0; M^+)\]
\[\text{(pack}_\nu(X \ni A, M)^+ = \text{pack}_\nu(X \ni A, \text{hide } \Gamma_0 \subseteq \Gamma_0; M^+)\]
\[\text{(unpack} (x, y) = M; N)^+ = \text{unpack} (x, y) = ? x ? X.\xi M; \text{hide } \Gamma_N|X \subseteq \Gamma_N; N^+\]
\[(A^\nu X.M)^+ = A^\nu X.\text{hide } \Gamma_0 \subseteq \Gamma_0; M^+ \quad \text{(where } M : A; \Gamma_0)\]
\[M[X \ni B]^+ = (\nu^\nu X.\xi M)[X \ni B] \quad \text{(when } M : ?)\]
\[G \xi M = (\text{tag}_G(G))^\uparrow M^+ \quad \text{(otherwise)}\]
\[\text{hide } \Gamma_f \subseteq (\Gamma_b, X \cong A); M = \text{hide } \Gamma_f \subseteq \Gamma_b; \text{hide } X \cong A; M \quad (X \notin \Gamma_f)\]
\[\text{hide } (\Gamma_b, X \cong A) \subseteq (\Gamma_b, X \cong A); M = \text{hide } \Gamma_b \subseteq \Gamma_b; M\]
\[\text{hide } \cdot \subseteq \cdot; M = M\]

Fig. 6. Elaborating PolyG\nu to PolyC\nu

program. Formally, a store \(\Sigma\) is just a pair of a number \(\Sigma.n\) and a function \(\Sigma.f : [n] \rightarrow Ty\) where Ty is the set of all types and \([n] = \{m \in \mathbb{N} \mid m < n\}\) is from some prefix of natural numbers to types. All rules take configurations \(\Sigma \Rightarrow M\) to configurations \(\Sigma ' \Rightarrow M '\). When the step does not change the store, we write \(M \mapsto M '\) for brevity.

The first rule states that all non-trivial evaluation contexts propagate errors. Next, unsealing a seal gets out the underlying value, and \text{is}(G) \cdot V \text{ literally checks if the tag of } V \text{ is } G. \text{The hide form generates a fresh seal } \sigma : A \text{ and substitutes it into the continuation. The pack form steps to an intermediate state used for building up a stack of casts that will be used again in the existential cast rule. The unpack rule generates a fresh seal for the } X \cong A \text{ and then applies all of the accumulated casts to the body of the pack. Here we use } \uparrow \text{ to indicate one of } \downarrow \text{ and } \downarrow. \text{For } \nu^\nu \text{ instantiation, we do not need to generate the seal, because it must have already been generated by a hide form further up the term, so the rule is just a substitution. As is typical for a cast calculus, the remaining types have ordinary call-by-value } \beta \text{ reduction and so we elide them.}

The remaining rules give the behavior of casts. Other than the use of type precision derivations, the behavior of our casts is mostly standard: identity casts for \(\mathbb{B}, \sigma \) and \? \text{ are just the identity, and the product cast proceeds structurally. Function casts are values, and when applied to a value, the}
We present the syntax of our typed language CBPV push-value calculus [Levy 2003], which we use as a convenient metalanguage to extend with features of the dynamic type: rather than being a finitary sum of a few statically fixed cases, the dynamic and universal quantification combined with a somewhat well-studied programming feature: a cast to ordinary terms in the typed language that raise errors. The key benefit of the typed translation decomposes these features into the combination of new type connectives and inside-out scoping of the name generation and gradual type casts. However, this ad hoc design addition of behavior of the cast calculus with an operational semantics defining the ideal for proving meta-theoretic properties of the system.

5 TYPED INTERPRETATION OF THE CAST CALCULUS

In the previous section we developed a cast calculus with an operational semantics defining the name generation and gradual type casts. This ad hoc design addition of new type connectives and inside-out scoping of \( \forall \) -instantiations make the cast calculus less than ideal for proving meta-theoretic properties of the system.

Instead of directly proving meta-theoretic properties of the cast calculus, we give a contract translation of the cast calculus into a statically typed core language, translating the gradual type casts to ordinary terms in the typed language that raise errors. The key benefit of the typed language is that it does not have built-in notions of fresh existential and universal quantification. Instead, the type translation decomposes these features into the combination of ordinary existential and universal quantification combined with a somewhat well-studied programming feature: a dynamically extensible “open” sum type we call OSum. Finally, it gives a static type interpretation of the dynamic type: rather than being a finitary sum of a few statically fixed cases, the dynamic type is implemented as the open sum type which includes those types allocated at runtime.

5.1 Typed Metalanguage

We present the syntax of our typed language CBPVOsum in Figure 8, an extension of Levy’s Call-by-push-value calculus [Levy 2003], which we use as a convenient metalanguage to extend with features.
of interest. Call-by-push-value (CBPV) is a typed calculus with highly explicit evaluation order, providing similar benefits to continuation-passing style and A-normal form [Sabry and Felleisen 1992]. The main distinguishing features of CBPV are that values $V$ and effectful computations $M$ are distinct syntactic categories, with distinct types: value types $A$ and computation types $B$. The two "shift" types $U$ and $F$ mediate between the two worlds. A value of type $UB$ is a first-class "thunk" of a computation of type $B$ that can be forced, behaving as a $B$. A computation of type $FA$ is a computation that can perform effects and return a value of type $A$, and whose elimination form is a monad-like $bind$. Notably while sums and (strict) tuples are value types, function types $A \rightarrow B$ are computations since a function interacts with its environment by receiving an argument. We include existentials as value type and universals as computation types, that in each case quantify over value types because we are using it as the target of a translation from a call-by-value language.

We furthermore extend CBPV with two new value types: $OSum$ and $Case$. Since $OSum$ is an open sum type, it is unknown what cases $x \cdot M$ might use, so the pattern-match has two branches: the one $inj \cdot M$ binds the underlying value to $x : A$ and proceeds as $M$ and the other is a catch-all case $N$ in case $V_0$ was not constructed using $V_c$. Finally, there is a form $newcase_A x; M$ that allocates a fresh $Case A$, binds it to $x$ and proceeds as $M$. In addition to the similarity to ML exception types, they are also similar to the dynamically typed sealing mechanism introduced in Sumii and Pierce [2004].

5.2 Static and Dynamic Semantics

We show a fragment of the typing rules for CBPV$_{OSum}$ in Figure 9. There are two judgments corresponding to the two syntactic categories of terms: $\Delta; \Gamma \vdash V : A$ for typing a value and $\Delta; \Gamma \vdash M : A$ for typing a computation. $\Delta$ is the environment of type variables and $\Gamma$ is the environment for term variables. Unlike in PolyG$^v$ and PolyC$^v$, these are completely standard, and there is no concept of a known type variable.

First, an error $U$ is a computation and can be given any type. Variables are standard and the $OSum/Case$ forms are as described above. Existentials are a value form and are standard as in CBPV using ordinary substitution in the pack form. In all of the value type elimination rules, the discriminatee is restricted to be a value. A computation $M : B$ can be thunked to form a value thunk $M : UB$, which can be forced to run as a computation. Like the existentials, the universal quantification

functions, quantification and an open sum type, this gives a simple explanation of the semantics of PolyC thought of as an alternate semantics to the operational semantics for PolyC.

Next, we present the "contract translation" of PolyC.

5.3 Translation

A representative fragment of the operational semantics is given in Figure 10, the full semantics are in the appendix [New et al. 2020]. $S$ represents a stack, the CBPV analogue of an evaluation context, defined in Figure 8. Here $\Sigma$ is like the $\Sigma$ in PolyC$^\nu$, but maps to value types. The semantics is standard, other than the fact that we assign a count to each step of either 0 or 1. The only steps that count for 1 are those that introduce non-termination to the language, which is used later as a technical device in our logical relation in §6.

5.3 Translation

Next, we present the "contract translation" of PolyC$^\nu$ into CBPV$_{OSum}$. This translation can be thought of as an alternate semantics to the operational semantics for PolyC$^\nu$, but with a tight correspondence given in §5.4. Since CBPV$_{OSum}$ is a typed language that uses ordinary features like functions, quantification and an open sum type, this gives a simple explanation of the semantics of PolyC$^\nu$ in terms of fairly standard language features.

In the left side of Figure 11, we present the type translation from PolyC$^\nu$ to CBPV$_{OSum}$. Since PolyC$^\nu$ is a call-by-value language, types are translated to CBPV$_{OSum}$ value types. Booleans and
which inserts a preamble to generate a case of the OSum type for each ground type. This allows $\nu x \Gamma$, $\sigma : A \in \Sigma$ to make the type be instantiated with a freshly generated case. On the other hand, since known type variables $X \equiv A$ are translated to $\nu x \exists X. A$ as with an unknown type variable. Finally, the empty context $\emptyset$ is translated to a pair of empty contexts.

To translate a whole program, written $\nu x : M$, we insert a preamble that generates the cases of the open sum type for each ground type. In Figure 12, we show our whole-program translation which inserts a preamble to generate a case of the OSum type for each ground type. This allows us to assume the existence of these cases in the rest of the translation. These can be conveniently
Graduality and Parametricity: Together Again for the First Time 46:21

\[
[M]_\rho = \text{newcase}[\mathbb{B}] c_{\text{Bool}}; \\
\text{case}(\mathbb{B}) = c_{\text{Bool}} \\
\text{newcase}[? \rightarrow ?] c_{\text{Fun}}; \\
\text{case}(A \rightarrow B) = c_{\text{Fun}} \\
\text{newcase}[? \times ?] c_{\text{Times}}; \\
\text{case}(A \times B) = c_{\text{Times}} \\
\text{newcase}[? \vee ?] c_{\text{All}}; \\
\text{case}(X) = c_X \\
\text{case}(\sigma) = \sigma \\
\Gamma_p = \text{Bool} \equiv \mathcal{B}, \text{Fun} \equiv ? \rightarrow ?, \text{Times} \equiv ? \times ?, \text{Ex} \equiv \exists^v X.?, \text{All} \equiv \forall^v X.?
\]

\[
\text{Fig. 12. Ground type tag management}
\]

\[
[x] = \text{ret } x \\
[\text{let } x = M; N] = x \leftarrow [M]; [N] \\
[\mathcal{U}_A] = \mathcal{U} \\
[\text{seal}_X M] = [M] \\
[\text{unseal}_X M] = [M] \\
[\text{inj}_G M] = r \leftarrow [M]; \text{ret inj}_{\text{case}(G)} r \\
[\text{is}(G)? M] = r \leftarrow [M]; \text{match } r \text{ with } \text{case}(G)\{ \text{inj } y \text{. ret } \text{true } | \text{ ret } \text{false} \} \\
[\text{hide } X \equiv A; M] = \text{newcase}_X c_X; [M] \\
[(A^E)\downarrow M] = [A^E] \llbracket [M] \rrbracket \\
[\lambda(x : A) . M] = \text{ret thunk } \lambda(x : [A]).[M] \\
[M N] = \text{force } x_1 \leftarrow \text{ret } [M]; a \leftarrow \text{ret } [N]; (\text{force } f) \ a \\
[\text{pack}(X \equiv A, M)] = \text{ret pack}(A, \text{thunk } (\lambda c_X : \text{Case } A).[M]) \\
[\text{unpack } (X, x) = M; N] = r \leftarrow [M]; \text{unpack } (X, f) = r; \text{newcase}_X c_X; x \leftarrow \text{(force } f) \ c_X; [N] \\
[A^E \times A^E; M] = \text{ret } (\text{thunk } (\lambda x : A.\lambda a : A.to \text{. ret } [A^E] \llbracket \text{ret } y \rrbracket; [A^E] \llbracket \text{force } x \ a \rrbracket)) \\
\text{where } A^E \uparrow \equiv A_1 \leftrightarrow A_1 \text{, and if } \downarrow \equiv \downarrow \uparrow, \ A' = A_1 \text{, else } A' = A_1 \text{.} \\
[\text{newcase}_X c_X] = \text{ret thunk } (\lambda X.\lambda c_X : \text{Case } X).[A^E] \llbracket \text{force } x [X].c_X \rrbracket) \\
\]

\[
\text{Fig. 13. PolyC}^v \text{ term translation (fragment)}
\]

modeled as a sequence of “global” definitions of some known type variables, which we write as \(\Gamma_p\). We also define a function case(\cdot) from types to their corresponding case value, which is a case variable for all types except those generated at runtime \(\sigma\).

Next, we consider the term translation, which is defined with the below type preservation Theorem 5.1 in mind. First, all PolyC\(^v\) terms of type \(A\) are translated to CBPV\(_{\text{OSum}}\) computations, with type \(F[A]\), which is standard for translating CBV to CBPV. Also, note that the output environment of fresh type names in a term is just translated as an extension to the input environment, the difference is irrelevant in the translated code, because the names themselves are actually generated in the translation of the hide form. Finally, we include the preamble context \(\Gamma_p\) to the front of the terms to account for the fact that all terms can use the cases generated in the preamble.

\[\text{Theorem 5.1. If } \Gamma_1 \vdash M : A, \Gamma_2 \text{ then } \Delta; \Gamma \vdash [M] : F[\Sigma; \Gamma_1, \Gamma_2 \vdash A] \text{ where } [\Gamma_p, \Gamma_1, \Gamma_2] = \Delta; \Gamma.\]

We show most of the term translation in Figure 13. To reduce the context clutter in the translations, we elide the contexts $\Sigma, \Gamma$ in the definition of the semantics. While they are technically needed to translate type annotations, they do not affect the operational semantics and so can be safely ignored. We put the bool, pair, and our pack-cast intermediate form cases in the appendix [New et al. 2020].

Variables translate to a return of the variable, let is translated bind, and errors are translated to errors. Since the type translation maps known type variables to their bound types, the target language seal and unseal disappear in the translation. Injection into the dynamic type translates to injection into the open sum type and ground type checks in PolyC$^\nu$ are implemented using pattern matching on OSum in CBPV$_{OSum}$. Next, the hide form is translated to a newcase form.

Next, we cover the cases involving thunks. As a warmup, the functions follow the usual CBV translation into CBPV: a CBV $\lambda$ is translated to a thunk of a CBPV $\lambda$, and the application translation makes the evaluation order explicit and forces the function with an input. We translate existential packages in the cast calculus to CBPV$_{OSum}$ packages containing functions from a case of the open sum type to the body of the package. In PolyC$^\nu$ we delay execution of pack bodies, so the translation inserts a thunk to make the order of execution explicit. Since pack bodies translate to functions, the translation of an unpack must provide a case of the open sum type to the package it unwraps. Type abstractions ($\Lambda$), like packs, wrap their bodies in functions that, on instantiation, expect a case of the open sum type matching the instantiating type. Since hide generates the requisite type name, it translates to a newcase. A type application then simply plugs its given type and the tag associated with its type variable into the supplied type abstraction.

Next, we define the implementation of casts as “contracts”, i.e., ordinary functions in the CBPV$_{OSum}$. Reflexive casts at atomic types, $?$, $\alpha$, and $\beta$, translate away. Structural casts at composite types, pair types, function types, universals, and existentials, push casts for their sub-parts into terms of each type. Function and product casts are entirely standard, noting that we use $r(A^\nu) = A_r$. Universal casts delay until type application and then cast the output. Existential casts push their subcasts into whatever package they are given.

5.4 Simulation

In §6, we establish graduality and parametricity theorems for PolyG$^\nu$/PolyC$^\nu$ by analysis of the semantics of terms translated into CBPV$_{OSum}$. But since we take the operational semantics of PolyC$^\nu$ as definitional, we need to bridge the gap between the operational semantics in CBPV$_{OSum}$ and PolyC$^\nu$ by proving the following adequacy theorem that says that the final behavior of terms in PolyC$^\nu$ is the same as the behavior of their translations:

**Theorem 5.2 (Adequacy).** If $\cdot \vdash M : A$; $\cdot$, then $M \upharpoonleft$ if and only if $[M]_p \upharpoonleft$ and $M \downharpoonright V$ if and only if $[M]_p \downharpoonright V'$ and $M \downharpoonright U$ if and only if $[M]_p \downharpoonright U$.

The proof of the theorem follows by a forward simulation argument given in the appendix, adapting a similar CBPV simulation given by Levy [2003], and proves that the $V$ and $V'$ in the adequacy proof are in the simulation relation [New et al. 2020].

6 GRADUALITY AND PARAMETRICITY

In this section we prove the central metatheoretic results of the paper: that our surface language satisfies both graduality and parametricity theorems. Each of these is considered a technical challenge to prove: parametricity is typically proven by a logical relation and graduality is proven either by a simulation argument [Siek et al. 2015] or a logical relation [New and Ahmed 2018; New et al. 2019], so in the worst case this would require two highly technical developments. However, we show that this is not necessary: the logical relations proof for graduality is general enough that
We formalize this as a judgment $\Gamma \vdash_M M_l \subseteq M_r : A \Gamma'$. As they bridge the worlds of type and term precision, the analogy between parametricity and graduality originally proposed in [New and Ahmed 2018].

The key to sharing this work is that we give a novel relational interpretation of type precision derivations. That is, our logical relation is indexed not by types, but by type precision derivations. For any derivation $A : A_l \subseteq A_r$, we define a relation $V'[A]$ between values of $A_l$ and $A_r$. By taking the reflexivity case $A : A \subseteq A$, we recover the parametricity logical relation. Previous logical relations proofs of graduality defined a logical relation indexed by types, and used casts to define a second relation based on type precision judgments, but the direct relational approach simplifies the proofs and immediately applies to parametricity as well.

### 6.1 Term Precision

To state the graduality theorem, we begin by formalizing the syntactic term precision relation. The intuition behind a precision relation $M \subseteq M'$ is that $M'$ is a (somewhat) dynamically typed term and we have changed some of its type annotations to be more precise, producing $M$. This is one of the main intended use cases for a gradual language: hardening the types of programs over time. Restated in a less directed way, a term $M$ is (syntactically) more precise than $M'$ when the types and annotations in $M$ are more precise than $M'$ and otherwise the terms have the same structure. We formalize this as a judgment $\Gamma_M \vdash_M M_l \subseteq M_r : A \Gamma_r$, where $\Gamma_M$, $\Gamma_M' \vdash A : A_l \subseteq A_r$ is a type precision derivation and $\Gamma_M \vdash \Gamma_r : \Gamma_l \subseteq \Gamma_r$, and $\Gamma_r' : \Gamma_l \subseteq \Gamma_r'$ are precision contexts and $\Gamma_l \vdash M_l : A_l$; $\Gamma_r'$ and $\Gamma_r' \vdash M_r : A_r$; $\Gamma_r'$. A precision context $\Gamma_r$ is like a precision derivation between two contexts: everywhere a type would be in an ordinary context, a precision derivation is used instead.

We show term precision rules for annotations and $V'$ introduction and elimination for the surface language in Figure 14, with full rules in the appendix [New et al. 2020]. The rules are all completely structural: just check that the two terms have the same term constructor and all of the corresponding arguments of the rule are $\sqsubseteq$. As exhibited by the $V'$ elimination rule, the metafunctions $\text{dom}$, $\text{cod}$, $\text{un}V'$, $\text{un}\exists V'$ are extended in the obvious way to work on precision derivations. We define a similar notion of term precision for PolyC$V$. Again we show the rules for casts and $V'$ in Figure 15, the full definition is in the appendix [New et al. 2020]. The main difference is that, following [New et al. 2019], we include four rules involving casts: two for downcasts and two for upcasts. We can summarize all four by saying that if $M_l \subseteq M_r$, then adding a cast to either $M_l$ or $M_r$ still maintains that the left side is more precise than the right, as long as the type on the left is more precise than the right. Semantically, these are the most important term precision rules, as they bridge the worlds of type and term precision.

Then the key lemma is that the elaboration process from PolyG$V$ to PolyC$V$ preserves term precision. The proof, presented in the appendix, follows by induction on term precision proofs [New et al. 2020].

**Lemma 6.1.** If $\Gamma_M \vdash M_l \subseteq M_r : A \Gamma_r$ in the surface language, then $\Gamma_M \vdash M_l^+ \subseteq M_r^+ : A \Gamma_r$. 

6.2 Graduality Theorem

The graduality theorem states that if a term $M$ is syntactically more precise than a term $M'$, then $M$ semantically refines the behavior of $M'$: it may error, but otherwise it has the same behavior as $M'$: if it diverges so does $M'$ and if it terminates at $V$, $M'$ terminates with some $V'$ as well. If we think of $M$ as the result of hardening the types of $M'$, then this shows that hardening types semantically only increases the burden of runtime type checking and doesn’t otherwise interfere with program behavior. We call this operational graduality, as we will consider some related notions later.

**Theorem 6.2 (Operational Graduality).** If $\cdot \vdash M_l \subseteq M_r : A^\Sigma; \cdot$, then either $M^+_l \parallel U$ or both terms diverge $M^+_l \parallel V_l$ and $M^+_r \parallel V_r$.

6.3 Logical Relation

The basic idea of the logical relations proof to proving graduality is to interpret a term precision judgment $\Gamma^E \vdash M_l \subseteq M_r : A^E; \Gamma^\Sigma_0$ in a relational manner. That is, to every type precision derivation $A^E : A_l \subseteq A_r$, we associate a relation $\mathcal{V}[A^E]$ between closed values of types $A_l$ and $A_r$. Then we define a semantic version of the term precision judgment $\Gamma^E \vdash M_l \subseteq M_r \in A^E; \Gamma^\Sigma_0$ which says that given inputs satisfying the relations in $\Gamma^E$, $\Gamma^\Sigma_0$, then either $M_l$ will error, both sides diverge, or $M_l$ and $M_r$ will terminate with values in the relation $\mathcal{V}[A^E]$. We define this relation over CBPV$_{O\Sigma_{\text{sum}}}$ translations of PolyC$^\vee$ terms, rather than PolyC$^\vee$ terms because the operational semantics is simpler.

More precisely, we use the now well established technique of Kripke, step-indexed logical relations [Ahmed et al. 2009]. Because the language includes allocation of fresh type names at runtime, the set of values that should be in the relation grows as the store increases. This is modeled Kripke structure, which indexes the relation by a “possible world” that attaches invariants to the allocated cases. Because our language includes diverging programs (due to the open sum type), we need to use a step-indexed relation that decrements when pattern matching on OSum, and “times out” when the step index hits 0. Finally, following [New and Ahmed 2018; New et al. 2019], to model graduality we need to associate two relations to each type precision derivation: one which times out when the left hand hand term runs out of steps, but allows the right hand side to take any number of steps and vice-versa one that times out when the right runs out of steps.

Figure 16 includes preliminary definitions we need for the logical relation. First, $\text{Atom}_n[A_l, A_r]$ and $\text{CAAtom}_n[A_l, A_r]$ define the world-term-term triples that the relations are defined over. A relation $R \in \text{Rel}_n[A_l, A_r]$ at stage $n$ consists of triples of a world, and a value of type $A_l$ and a value of type $A_r$, (ignore the $n$ for now) such that it is monotone in the world. The world $w \in \text{World}_n$ contains the number of steps remaining $w.j$, the current state of each side $w.X_i$, $w.\Sigma_r$, and finally an interpretation of how the cases in the two stores are related $w.\eta$. An interpretation $\eta \in \text{Interp}_n[\Sigma_i, \Sigma_r]$ consists of a cardinality $\eta.size$ which says how many cases are related and a
function \( \eta.f \) which says which cases are related, i.e., for each \( i \in \eta.size \) it gives a pair of cases, one valid in the left hand store and one in the right. Finally, \( \eta.p \) gives a relation between the types of these two cases. The final side-condition says this association is a **partial bijection**: a case on one side is associated to at most one case on the other side. Staging the relations and worlds is necessary due to a circularity here: a relation is (contravariantly) dependent on the current world, which contains relations. A relation in \( \text{Rel}_w \) is indexed by a \( \text{World}_w \), but a \( \text{World}_n \) contains relations in \( \text{Rel}_w, j \), and \( w.j < n \). In particular, \( \text{World}_0 = \emptyset \), so the definition is well-founded.

The next portion of the figure contains the definition of **world extension** \( w' \supseteq w^2 \), representing the idea that \( w' \) is a possible “future” of \( w \): the step index \( j \) is smaller and the states of the two sides have increased the number of allocated cases, but the old invariants are still there. We define strict extension \( w' \supseteq w \) to mean extension where the step has gotten strictly smaller. This allows us to define the **later relation** \( \triangleright R \) which is used to break circular definitions in the logical relation. Next, we define our notion of non-indexed relation \( \text{Rel}_n \), which is what we quantify over in the interpretation of \( \forall^v, \exists^v \). Then we define the restriction of interpretations and relations to a stage \( n \). An infinitary relation can be “clamped” to any stage \( n \) using \([R]_n \). Finally, we define when two cases are related in an interpretation as \( \eta \models (\sigma_l, \sigma_r, R) \).

The top of Figure 17 contains the definition of the logical relation on values and computations, except for the standard cases for booleans, products and functions, which are included in the appendix [New et al. 2020]. First, we write \( \sim \) as a metavariable that ranges over two symbols: \( < \) which indicates that we are counting steps on the left side, and \( > \) which indicates we are counting steps on the right side. We then define the value relation \( \mathcal{V}^n [\mathcal{A}^2] \gamma \delta \in \text{Rel}_n [\delta_l(A_l), \delta_r(A_r)] \). Here \( \gamma \) maps the free term variables to pairs of values and \( \delta \) maps free type variables to triples of two types and a relation between them. First, the definition for type variables looks up the relation in the relational substitution \( \delta \). Next, two values in \( ? \) are related when they are both injections into OSum, and the “payloads” of the injections are **later** related in the relation \( R \) which the world associates to the corresponding cases. The \( \triangleright \) is used because we count pattern matching on OSum as a step. This also crucially lines up with the fact that pattern matching on the open sum type is the only reduction that consumes a step in our operational semantics. Note that this is a generalization of the logical relation definition for a recursive sum type, where each injection corresponds to a
case of the sum. Here since the sum type is open, we must look in the world to see what cases are allocated. Next, the \( \text{tag}_C(A^E) \) case relates values on the left at some type \( A_l \) and values on the right of type \( ? \). The definition states that the dynamically typed value must be an injection using the tag given by \( G \), and that the payload of that injection must be related to \( V_l \) with the relation given by \( A^E \). This case splits into two because we are pattern matching on a value of the open sum type, and so in the \( > \) case we must decrement because we are consuming a step on the right, whereas in the \( < \) case we do not decrement because we are only counting steps on the left. In the \( V^r X.B^E \) case, two values are related when in any future world, and any relation \( R \in \text{Rel}_{\omega}(A_l, A_r) \), and any pair of cases \( \sigma_l, \sigma_r \) that have \([R]_{\omega,j}\) as their associated relation, if the values are instantiated with \( A_l, \sigma_l \) and \( A_r, \sigma_r \) respectively, then they behave as related computations. The intuition is that values of type \( V^r X.B \) are parameterized by a type \( A \) and a tag for that type \( \sigma \), but the relational interpretation of the two must be the same. This is the key to proving the seal\( _X \) and unseal\( _X \) cases of graduality. The fresh existential is dual in the choice of relation, but the same in its treatment of the case \( \sigma \).

Next, we define the relation on expressions. The two expression relations, \( E^< [A^E] \) and \( E^> [A^E] \) capture the semantic idea behind graduality: either the left expression raises an error, or the two programs have the same behavior: diverge or return related values in \( V^r [A^E] \). However, to account for step-indexing, each is an approximation to this notion where \( E^< [A^E] \) times out if the left side consumes all of the available steps \( w.j \) (where \( (\Sigma, M) \rightarrow^J \) is shorthand for saying it steps to something in \( j \) steps), and \( E^> [A^E] \) times out if the right side consumes all of the available steps. The relation is that when we define the infinitary version of the relations \( V^r [A^E] \) and \( E^> [A^E] \) a the union of all of the level \( n \) approximations.

Next, we give the relational interpretation of environments. The interpretation of the empty environment are empty substitutions with a valid world \( w \). Extending with a value variable \( x : A^E \) means extending \( y \) with a pair of values related by \( V^r [A^E] \). For an abstract type variable \( X \), first \( \delta \) is extended with a pair of types and a relation between them. Then, \( \gamma \) must also be extended with a pair of cases encoding how these types are injected into the dynamic type. Crucially, just as with the \( V^r, \exists^r \) value relations, these cases must be associated by \( w \) to the \( w.j \) approximation of the same relation with which we extend \( \delta \). The interpretation of the known type variables \( X \equiv A^E \) has the same basic structure, the key difference is that rather than using an arbitrary, \( \delta \) is extended with the value relation \( V^r [A^E] \).

With all of that preparation finished, we finally define the semantic interpretation of the graduality judgment \( \Gamma^E \vdash M_l \subseteq M_r \in A^E; \Gamma^E \) in the bottom of Figure 17. First, it says that both \( M_l \subseteq \langle \Gamma^E \rangle \) and \( M_l \subseteq \langle \Gamma^E \rangle \) hold, where we define \( \langle \Gamma^E \rangle \) to mean that for any valid instantiation of the environments (including the preamble \( \Gamma_{\text{pr}} \)), we get related computations. We can then define the “logical” Graduality theorem, that syntactic term precision implies semantic term precision, briefly, \( \vdash \) implies \( \models \).

**Theorem 6.3 (Logical Graduality).** If \( \Gamma^E \vdash M_l \subseteq M_r : A^E; \Gamma^E \), then \( \Gamma^E \models M_l \subseteq M_r \in A^E; \Gamma^E \)

The proof is by induction on the term precision derivation. Each case is proven as a separate lemma in the appendix [New et al. 2020]. The cases of \( V^r, \exists^r \), sealing and unsealing follow because the treatment of type variables between the value and environment relations is the same. In the hide case, the world is extended with \( V^r [A^E] \) as the relation between new cases. The cast cases are the most involved, following by two lemmas proven by induction over precision derivations: one for when the cast is on the left, and the other when the cast is on the right.

Finally, we prove the operational graduality theorem as a corollary of the logical graduality Theorem 6.3 and the adequacy Theorem 5.2. By constructing a suitable starting world \( w_{\text{pr}} \), that allocates the globally known tags, we ensure the operational graduality property holds for the code translated to \( \text{CBPV}_{\text{OSum}} \), and then the simulation theorem implies the analogous property holds for the PolyC' operational semantics.
we can prove "free theorems" that are true in polymorphic languages. These free theorems are says that semantic term precision both ways is plugged into an arbitrary context. The appendix contains a formal definition [New et al. 2020].

6.4 Parametricity and Free Theorems

Our relational approach to proving the graduality theorem is not only elegant, it also makes the theorem more general, and in particular it subsumes the parametricity theorem that we want for the language, because we already assign arbitrary relations to abstract type variables. Then the parametricity theorem is just the reflexivity case of the graduality theorem.

**Theorem 6.4 (Parametricity).** If $\Gamma \vdash M : A; \Gamma'$, then $\Gamma \vdash M^+ \sqsubseteq M^+ : A; \Gamma'$.

To demonstrate that this really is a parametricity theorem, we show that from this theorem we can prove "free theorems" that are true in polymorphic languages. These free theorems are naturally stated in terms of contextual equivalence, the gold standard for operational equivalence of programs, which we define as both programs diverging, erroring, or terminating successfully when plugged into an arbitrary context. The appendix contains a formal definition [New et al. 2020].

To use our logical relation to prove contextual equivalence, we need the following lemma, which says that semantic term precision both ways is sound for PolyG$^\nu$ contextual equivalence.

**Lemma 6.5.** If $\Gamma \vdash M_1 \sqsubseteq M_2 \in A; \Gamma_M$ and $\Gamma \vdash M_2 \sqsubseteq M_1 \in A; \Gamma_M$, then $\Gamma \vdash M_1 \approx^{ctx} M_2 \in A; \Gamma_M$.
which inputs are sealed and which are not, rather than the caller. Because of this, Toro et al. [2019]

whether it terminates, diverges or errors does not depend on the input values. Finally, we give an
A good intuition for this type is that the only possible outputs of the function are sealed values
∀

\[ \lambda : \text{unseal}_X (M[X \equiv A] (\text{seal}_X V_A)) \approx \text{ctx} \lambda : \text{let } y = M[X \equiv B] V_B; V_A \]

\[ M : \forall^\circ X. X \rightarrow X \quad V_A : A \quad V_B : B \]

\[ \lambda : \text{unseal}_X (M[X \equiv A] (\text{seal}_X V_A)) \approx \text{ctx} \lambda : \text{let } y = M[X \equiv B] V_B; V_A \]

NOT = \lambda b : B. \text{if } b \text{ then false else true}

WRAPNOT = \lambda x : X. \text{seal}_X (\text{NOT}(\text{unseal}_X x))

\approx \text{ctx}

\[ \text{pack}^\circ (X \equiv B, (\text{seal}_X \text{true}, (\text{WRAPNOT}, \lambda x : X. \text{unseal}_X x))) \]

\[ \approx \text{ctx} \]

\[ \text{pack}^\circ (X \equiv B, (\text{seal}_X \text{false}, (\text{WRAPNOT}, \lambda x : X. \text{NOT}(\text{unseal}_X x)))) \]

\[ \text{Fig. 18. Free Theorems without ?} \]

We now substantiate that this is a parametricity theorem by proving a few contextual equivalence
results. First we present adaptations of some standard free theorems from typed languages in
Figure 18. The first equivalence shows that the behavior of any term with the "identity function
type" \( \forall^\circ X. X \rightarrow X \) must be independent of the input it is given. We place a \( \lambda \) on each side to
delimit the scope of the X outward. Without the \( X \) (or a similar thunking feature like \( \forall^\circ \) or \( \exists^\circ \)),
the two programs would not have the same (effect) typing. In a more realistic language, this
corresponds to wrapping each side in a module boundary. The next result shows that a function
\( \forall^\circ X. \forall^\circ Y. (X \times Y) \rightarrow (Y \times X) \), if it terminates, must flip the values of the pair, and furthermore
whether it terminates, diverges or errors does not depend on the input values. Finally, we give an
example using existential types. That shows that an abstract "flipper" which uses
false
true
either diverges or errors.

Next, to give a flavor of what kind of relational reasoning is possible in the presence of the
dynamic type, we consider what free theorems are derivable for functions of type \( \forall X. ? \rightarrow X \).
A good intuition for this type is that the only possible outputs of the function are sealed values
that are contained within the dynamically typed input. It is difficult to summarize this in a single
statement, so instead we give the following three examples:

**Theorem 6.6 (\( \forall^\circ X. ? \rightarrow X \) Free Theorems).** Let \( \vdash M : \forall^\circ X. ? \rightarrow X \)

1. For any \( \vdash V : ? \), then \( \lambda : \text{let } y = (\text{unseal}_X (M[X \equiv B] (\text{seal}_X V))) \)

2. For any \( \vdash V : A \) and \( \vdash V' : B 

\lambda : \text{let } y = (\text{unseal}_X (M[X \equiv B] (\text{seal}_X V')) ; V)

3. For any \( \vdash V : A, \vdash V' : B, \vdash V_d : ? 

\lambda : \text{let } y = (\text{unseal}_X (M[X \equiv B] (\text{seal}_X V', V_d))) ; V)

The first example passes in a value \( V \) that does not use the seal \( X \), so we know that the function
cannot possibly return a value of type \( X \). The second example mimics the identity function’s free
theorem. It passes in a sealed value \( V \) and the equivalence shows that \( V' \)'s effects do not depend
on what \( V \) was sealed and the only value that \( V' \) can return is the one that was passed in. The
third example illustrates that there are complicated ways in which sealed values might be passed
as a part of a dynamically typed value, but the principle remains the same: since there is only one
sealed value that’s part of the larger dynamically typed value, it is the only possible return value,
and the effects cannot depend on its actual value. The proof of the first case uses the relational
interpretation that \( X \) is empty. The latter two use the interpretation that \( X \) includes a single value.

Compare this reasoning to what is available in GSF, where the polymorphic function
determines which inputs are sealed and which are not, rather than the caller. Because of this, Toro et al. [2019]
only prove “cheap” theorems involving ? where the polymorphic function is known to be a literal Λ function and not a casted function. As an example, for arbitrary  $M : \forall X. ? \rightarrow X$, consider the application $M \{ B \}(true, false)$. The continuation of this call has no way of knowing if neither, one or both booleans are members of the abstract type $X$. The following examples of possible terms for $M$ illustrate these three cases:

1. $M_1 = (\Lambda X. \lambda x : B \times B. \text{if} \ or \ x \ \text{then} \ U \ \text{else} \ \Omega) :: \forall X.? \rightarrow X$
2. $M_2 = (\Lambda X. \lambda x : X \times B. \text{if} \ \text{snd} \ x \ \text{then} \ \text{fst} \ x \ \text{else} \ \Omega) :: \forall X.? \rightarrow X$
3. $M_3 = (\Lambda X. \lambda x : X \times X. \text{snd} \ x) :: \forall X.? \rightarrow X$

If $M = M_1$, both booleans are concrete so $X$ is empty, but from the inputs the function can determine whether to diverge or error. If $M = M_2$, the first boolean is abstract and the second is concrete, so only the first can inhabit $X$, but the second can be used to determine whether to return a value or not. Finally if $M = M_3$, both booleans are abstract so the function cannot inspect them, but either can be returned. It is unclear what reasoning the continuation has here: it must anticipate every possible way in which the function might decide which values to seal, and so has to consider every dynamically typed value of the instantiating type as possibly abstract and possibly concrete.

7 DISCUSSION AND RELATED WORK

Dynamically typed PolyG$^\nu$ and Design Alternatives. Most gradually typed languages are based on adding types to an existing dynamically typed language, with the static types capturing some feature already existing in the dynamic language that can be migrated to use static typing. PolyG$^\nu$ was designed as a proof-of-concept standalone gradual language, so it might not be clear what dynamic typing features it supports migration of. In particular, since all sealing is explicit, PolyG$^\nu$ does not model migration from programming without seals entirely to programming with them, so its types are relevant to languages that include some kind of nominal data type generation.

The fresh existential types correspond to a particular mode of use of a module system that supports creation of nominal types. The package itself corresponds to a module with a fresh type declaration. Then sealing corresponds to the constructor of the fresh datatype, and unsealing to pattern matching against it. For example, in Racket structs can be used to make fresh nominal types and units provide first-class modules. It would be interesting future work to see if our logical relation can usefully be adapted to Typed Racket’s typed units [Tobin-Hochstadt et al. [n. d.]].

Our fresh polymorphic types are more exotic than the fresh existentials, and don’t clearly correspond to any existing programming features, but they model abstraction over nominal datatypes where the datatype is guaranteed to be freshly generated. One issue with adding this feature to a realistic language is that the outward scoping of type variables may be undesirable, so it is useful to consider alternative designs that achieve the same abstraction principles. One possible design would be to force an ANF-like [Sabry and Felleisen 1992] restriction on instantiations of polymorphic functions, where all instantiations have to be of the form let $M\{X \equiv A\} = f; N$, where $X \equiv A$ is bound in $N$. This makes the scope of the $X$ explicit: it is only bound in $N$. Our translation could easily be modified to accommodate this feature, we chose instead to consider the outward scoping since it makes it easier to compare to programming in the style of System F.

Our use of abstract and known type variables was directly inspired by Neis et al. [2009], who present a language with a fresh type creation mechanism which they show enables parametric reasoning though the language overall does not satisfy a traditional parametricity theorem. This suggests an alternative language design, where $\forall$ and $\exists$ behave normally and we add a newtype facility, analogous to that feature of Haskell, where newtype allocates a new case of the open sum type for each type it creates. Such a language would not have known type variables or PolyG$^\nu$'s
inside-out scoping of type instantiation, but it would also not be parametric by default. Instead, programmers could manually create fresh types and know that they are abstract to other modules.

Since sealing is explicit in PolyG$^\nu$, it does not provide a drop-in replacement for System F, and so the additional syntactic overhead of sealing and unsealing can be quite heavy, especially when using higher-order combinators. For instance a higher-order function composition combinator has type $\forall\nu X.\forall\nu Y.\forall\nu Z.(Y \to Z) \to (X \to Y) \to X \to Z$ and the System F composition $\text{compose}\left(\lambda y : Y.\text{seal}_Z\left(\text{not}\left(\text{unseal}_Y y\right)\right)\right)\left(\lambda x : X.\text{seal}_Y\left(> 0\left(\text{unseal}_X x\right)\right)\right)$

This syntactic overhead can be mitigated via generic wrapping functions using dynamic typing:

\[
\text{wrap } \text{seal}_X \text{unseal}_Z \left(\text{compose}\left(\lambda x : X.\text{seal}_Y\left(> 0\left(\text{unseal}_X x\right)\right)\right)\right)\left(\lambda y : Y.\text{seal}_Z\left(\text{not}\left(\text{unseal}_Y y\right)\right)\right)
\]

But the syntactic overhead cannot be completely removed or done entirely with static typing.

Tag Checking. Siek et al. [2015] claim that graduality demands that tag-checking functions like our $\text{is}(\mathbb{B})$? form must error when applied to sealed values, and used this as a criticism of the design of Typed Racket. However, in our language, $\text{is}(\mathbb{B})$? will simply return $\text{false}$, which matches Typed Racket’s behavior. This is desirable if we are adding types to an existing dynamic language, because typically a runtime tag check should be a safe operation in a dynamically typed language. Explicit sealing avoids this graduality issue, an advantage over previous work.

Logical Relations. Our use of explicit sealing eliminates much of the complexity of prior logical relations [Ahmed et al. 2017; Toro et al. 2019]. To accommodate dynamic conversion and evidence insertion, those relations adopted complex value relations for universal types that in turn restricted the ways in which they could treat type variables. Additionally, we are the first to give a logical relation for fresh existential types, and it is not clear how to adapt the non-standard relation for universals to existentials [Ahmed et al. 2017; Toro et al. 2019].

Next, while we argue that our logical relation more fully captures parametricity than previous work on gradual polymorphism, this is not a fully formal claim. To formalize it, in future work we could show that PolyG$^\nu$ is a model of an effectful variant of an axiomatic parametricity formulation such as Dunphy [2002]; Ma and Reynolds [1991]; Plotkin and Abadi [1993].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their in-depth comments. We also thank Matthias Felleisen for explaining the importance that tag checks be total operations. This material is based on research supported by the National Science Foundation under grants CCF-1910522, CCF-1816837, and CCF-1453796. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Graduality and Parametricity: Together Again for the First Time


Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs (POPL ’15).


Graduality and Parametricity: Together Again for the First Time

\[ \vdash \cdot \quad \vdash \Gamma \quad \Gamma \vdash A \quad \vdash \Gamma \quad \Gamma \vdash A \]
\[ \vdash \Gamma, x : A \quad \vdash \Gamma, X \quad \vdash \Gamma, X \equiv A \]

\[ \Gamma \vdash ? \quad X \in \Gamma \quad \Gamma \vdash \emptyset \quad \Gamma \vdash A_1 \quad \Gamma \vdash A_2 \quad \Gamma \vdash A_1 \times A_2 \quad \Gamma \vdash A_1 \rightarrow A_2 \quad \Gamma \vdash \exists \forall X. A \]
\[ \Gamma, X \vdash A \quad \Gamma \vdash \forall \forall X. A \]

Fig. 19. Well-formedness of Environments, Types

\[ \Gamma \vdash ? : ? \subset ? \quad \Gamma \vdash B : B \subset B \quad X \in \Gamma \subset X \quad \Gamma \vdash A : A \subset G \quad \Gamma \vdash \forall \forall X. A : \exists \forall X. A \subset \exists \forall X. A \]
\[ \Gamma \vdash A : A_1 \subset A_r \quad \Gamma \vdash B : B_1 \subset B_r \quad \Gamma \vdash A \subset A_{1r} \quad \Gamma \vdash B \subset B_{1r} \quad \Gamma \vdash A \times B : A_1 \times B_1 \subset A_r \times B_r \]
\[ \Gamma \vdash A \equiv B : A_1 \equiv A_r \quad \Gamma \vdash \forall \forall X. A \subset \exists \exists X. A \]
\[ \cdot |_{\Gamma} = \cdot \quad (X \equiv A, \Gamma)^{\cdot |_{\Gamma}} = X \equiv A \quad (FV(A) \cap \Gamma^\prime = \emptyset) \]
\[ (X \equiv A, \Gamma)^{\cdot |_{\Gamma}} = \Gamma^{\cdot |_{\Gamma}} \quad (FV(A) \cap \Gamma^\prime \neq \emptyset) \]

Fig. 20. Type Precision in a Precision Context, Restriction of a Precision Context

A SURFACE LANGUAGE

Well-formedness of environments and types is mutually defined:

B TYPE PRECISION

We present the definition of type precision with respect to a precision context in Figure 20.

With this presentation it is easy to see that

1. There is at most one derivation of $\Gamma \vdash A \subset A'$
2. $\vdash ?$ is the most imprecise type.
3. Precision is reflexive.
4. Precision is transitive.

The latter 3 statements are actually ambiguous because we haven’t said what $\Gamma$ is in each situation. We don’t use these operations in their full generality, but ok.

**Lemma B.1.** If $\Gamma \vdash A : A \subset A'$ and $\Gamma \vdash B : A \subset A'$ then $A = B$
\begin{align*}
A \cap ? &= A \\
? \cap B &= B \\
X \cap X &= X \\
\emptyset \cap \emptyset &= \emptyset \\
(A_1 \times A_2) \cap (B_1 \times B_2) &= (A_1 \cap B_1) \times (A_2 \cap B_2) \\
(A_i \rightarrow A_o) \cap (B_i \rightarrow B_o) &= (A_1 \cap B_1) \rightarrow (A_o \cap B_o) \\
(\exists^\nu X.A) \cap (\exists^\nu X.B) &= \exists^\nu X.(A \cap B) \\
(\forall^\nu X.A) \cap (\forall^\nu X.B) &= \forall^\nu X.(A \cap B)
\end{align*}

Fig. 21. Gradual Meets

**Proof.** By induction on derivations. If \( A' \neq ? \), then only one rule applies and the proof follows by inductive hypotheses. If \( A^G = ? : ? \subseteq ? \), the only other rule that could apply is \( \text{tag}_G(A^E) \), which cannot apply because it is easy to see that \( ? \subseteq G \) does not hold for any \( G \). Finally, we need to show that if \( A^E = \text{tag}_G(A^E) \) and \( B^E = \text{tag}_{G'}(A^E) \) then \( G = G' \) because it is easy to see if \( A \subseteq G \) and \( A \subseteq G' \) then \( G = G' \).

This is “identity expansion”, the derivations are used in the parametricity theorem.

**Lemma B.2.** If \( \Gamma \vdash A \) is a well-formed type then \( \Gamma \vdash A : A \subseteq A \).

**Proof.** By induction over \( A \). Every type constructor is punned with its type precision constructor.

This is “cut elimination”:

**Lemma B.3.** If \( \Gamma^E \vdash AB^E : A \subseteq B \) and \( \Gamma^E \vdash BC^E : B \subseteq C \) then we can construct a proof \( \Gamma^E \vdash AC^E : A \subseteq C \).

**Proof.** By induction on \( BC^E \).

1. If \( BC^E = ? : ? \subseteq ? \), then the proof is just \( AB^E \).
2. If \( BC^E = \text{tag}_G(BG^E) \), then \( BG^E : B \subseteq G \) so by inductive hypothesis, there is a proof \( AG^E : A \subseteq G \) and the proof we need is \( \text{tag}_{G'}(AG^E) \).
3. If \( BC^E = BC_1^E \times BC_2^E \), then it must also be the case that \( AB^E = AB_1^E \times AB_2^E \), and then our result is \( AC_1^E \times AC_2^E \) where \( AC_1^E, AC_2^E \) come from the inductive hypothesis.
4. All other cases are analogous to the product.

Next, we (partially) define gradual meets \( A \cap B \) in Figure 21. The meet is undefined if the case is missing.

**Lemma B.4.** For every \( \Gamma \vdash A, B, \Gamma \vdash A \cap B \) and there are precision derivations

1. \( \Gamma \vdash A\cap^E : A \cap B \subseteq A \)
2. \( \Gamma \vdash B\cap^E : A \cap B \subseteq B \)

Such that for any \( \Gamma \vdash C \) with \( \Gamma \vdash CA^E : C \subseteq A \) and \( \Gamma \vdash CB^E : C \subseteq B \), there exists a derivation

\( \Gamma \vdash C\cap^E : C \subseteq A \cap B \)
We present the full cast calculus operational semantics in Figure 22.

Some of the semantics in Figure 7 involve terms with \( \sigma \)s in places we would expect \( X \)s, in particular instantiations, seals, and unseals. These forms come about when we evaluate a hide and substitute \( \sigma \) for \( X \). We also have our aforementioned intermediate form for pack casts. Figure 23 gives the static typing rules for runtime terms. Note that the typing of runtime terms depends on a given \( \Sigma \). To reason about well-typed terms at runtime, we also thread a store through the rules in Figure 5.

We prove a few standard operational lemmas.

**Lemma C.1 (Unique Decomposition).** If \( \Sigma_1 \vdash M_1 : A \), then there exist unique \( E, M_2 \) such that \( M_1 = E[M_2] \).

**Proof.** By induction on \( M_1 \).

**Lemma C.2 (Cast calculus dynamic semantics are deterministic).** If \( \Sigma \vdash M \mapsto \Sigma_1 \vdash M_1 \) and \( \Sigma \vdash M \mapsto \Sigma_2 \vdash M_2 \) then \( \Sigma_1 = \Sigma_2 \) and \( M_1 = M_2 \).

**Proof.** By unique decomposition and definition of the dynamic semantics.

**Lemma C.3 (Progress).** If \( \Sigma_1 \vdash M_1 : A \) then either \( \Sigma_1 \vdash M_1 \mapsto \Sigma_2 \vdash M_2 \), \( M_1 = \emptyset \), or \( M_1 = V \) for some \( V \).

---

Fig. 22. PolyC\(^v\) Dynamic Semantics (full)
\[
\begin{array}{c}
\begin{array}{c}
x : A \in \Gamma \\
\Sigma; \Gamma \vdash x : A ; \\
\Sigma; \Gamma \vdash M : A l ; \Gamma M
\end{array}
\end{array}
\]

Fig. 23. PolyC\textsuperscript{\nu} Typing for Runtime Forms

**Proof.** If \( M_1 = \emptyset \) or \( M_1 \) is a value, we conclude. Otherwise, by Lemma C.1 (unique decomposition), we have unique \( E, M_2 \) such that \( M_1 = E[M_2] \). We proceed by cases on \( M_2 \).

- \( M_2 = \emptyset \) Since \( M_1 \neq \emptyset \), we have \( E \neq [] \), so \( M_1 \mapsto \emptyset \) and we conclude.
- \( M_2 = \text{hide} \ X \equiv B ; M_4 \) Then \( \Sigma_1 \vdash M_2 \mapsto \Sigma_1, \sigma : B \mapsto M_4[\sigma/X] \), so we conclude since \( M_2 \mapsto E[\Sigma_4[\sigma/X]] \).
- \( M_3 = \text{if} \ V \text{ then } M_4 \text{ else } M_5 \) Since \( M_1 \) is well-typed, \( V \vdash \emptyset \) and so, by Lemma F.9 (canonical forms), we may use thanks to Lemma F.2, either \( V = \text{true} \) or \( V = \text{false} \). In the first case, \( M_3 \mapsto M_4 \) and in the second \( M_3 \mapsto M_5 \), so we conclude.

• $M_3 = \text{let } x = V; M_4$ Then $M_3 \mapsto M_4[V/x]$, so we conclude.
• $M_3 = \text{let } (x, y) = V; M_4$ Since $M_1$ is well-typed, $V : A_1 \times A_2$ for some $A_1, A_2$ and so, by Lemma F.9 (canonical forms), $V = (V_1, V_2)$ for some $V_1, V_2$. Then $M_3 \mapsto M_4[V_1/x][V_2/y]$, so we conclude.
• $M_3 = \text{pack}^v(X \equiv B, M_4)$ Then $M_3 \mapsto \text{pack}^v(X \equiv B, [], M_4)$, so we conclude.
• $M_3 = \text{unpack} (X, x) = V; M_4$ Since $M_1$ is well-typed, $V : \exists^v XB$ for some $B$ and so, by Lemma F.9 (canonical forms), $V = \text{pack}^v(X \equiv B, [], M_3)$ for some $M_3$. Then $\Sigma \bowtie M_3 \mapsto \Sigma, \sigma : B \mapsto \text{let } x = M_5[\sigma/X]; M_4[\sigma/X]$, so we conclude.
• $M_3 = V\{\sigma \equiv B\}$ Since $M_1$ is well-typed, $V : \forall^v X A_1$ for some $A_1$ and so, by Lemma F.9 (canonical forms), $V = \lambda x. M_4$ for some $M_4$ or $V = \lambda x. M_4$ for some $A^\exists, V_1$. In the first case, $M_3 \mapsto M_4[\sigma/X]$ and in the second, $M_3 \mapsto \lambda x. M_4[\sigma/X] (V_1 \{\sigma \equiv B\})$.
• $M_3 = V_1 \ V_2$ Since $M_1$ is well-typed, $V : A_1 \rightarrow A_2$ for some $A_1, A_2$ and so, by Lemma F.9 (canonical forms), $V = \lambda x. M_4$ for some $M_4$ or $V = \lambda x. M_4$ for some $A^\exists, V_1$. In the first case, $M_3 \mapsto M_4[V_2/x]$ and in the second, $M_3 \mapsto (A^\exists) (V_1 (A^\exists) V_2)$.
• $M_3 = \text{unseal}_{\sigma}V$ Since $M_1$ is well-typed, $V : \sigma$ and so, by Lemma F.9 (canonical forms), $V = \text{seal}_{\sigma}V_1$ for some $V_1$. Then $M_3 \mapsto V_1$, so we conclude.
• $M_3 = \text{is}(G)? V$ Since $M_1$ is well-typed, $V : \omega$ and so, by Lemma F.9 (canonical forms), $V = \text{in}_{H} V_1$ for some $H, V_1$. If $H = G$, then $M_3 \mapsto \text{true}$, otherwise $M_3 \mapsto \text{false}$.
• $M_3 = \lambda x. M_4$ for some $M_4$ and so, by Lemma F.9 (canonical forms), $V = \text{pack}^v(X \equiv B, [A^\exists \downarrow_2 \ldots, M_4]$ for some $B, A^\exists \downarrow_2 \ldots, M_4$. Then $M_3 \mapsto \text{pack}^v(X \equiv B, [A^\exists \downarrow_2 \ldots, M_4]$.
• $M_3 = (A^\exists \times A^\exists) V$ Since $M_1$ is well-typed, $V : B_1 \times B_2$ for some $B_1, B_2$ and so, by Lemma F.9 (canonical forms), $V = (V_1, V_2)$ for some $V_1, V_2$. Then $M_3 \mapsto ((A^\exists) V_1, (A^\exists) V_2)$.
• $M_3 = (A^\exists) V$ where $A^\exists \in \{\emptyset, \sigma, ?\}$ Then $M_3 \mapsto V$.
• $M_3 = (\text{tag}_G(A^\exists)) V$ Then $M_3 \mapsto \text{in}_{H} (A^\exists) V$.
• $M_3 = (\text{tag}_G(A^\exists)) V$ Since $M_1$ is well-typed, $V : \omega$ and so, by Lemma F.9 (canonical forms), $V = \text{in}_{H} V_1$ for some $H, V_1$. If $H = G$, then $M_3 \mapsto (A^\exists) V$. Otherwise, $M_3 \mapsto \emptyset$.

\[\Box\]

D CBPV

Definition D.1 (Preamble store, substitution). We name the store generated by the preamble $\Sigma_p$, defined as

\[
\Sigma_p = (4, f) \\
f(0) = \emptyset \\
f(1) = U(\text{OSum} \rightarrow \text{FOSum}) \\
f(2) = \text{OSum} \times \text{OSum} \\
f(3) = \exists X. U(\text{Case } X \rightarrow \text{FOSum}) \\
f(4) = U(\forall X. \text{Case } X \rightarrow \text{FOSum})
\]

We define $y_p$ to be a substitution that closes terms with respect to $\Gamma_p$ using the store $\Sigma_p$:

\[
y_p(c_{\text{Bool}}) = 0 \\
y_p(c_{\text{Fun}}) = 1 \\
y_p(c_{\text{Times}}) = 2 \\
y_p(c_{\text{Ex}}) = 3 \\
y_p(c_{\text{All}}) = 4
\]
We give the full translation from PolyC to CBPV in Figure 26. Added to the cases from the paper are the translation for types $\mathbb{B}$ and $A \times B$.

### E TERM PRECISION

Figure 29 shows the full definition of term precision for PolyG. Figures 30 and 31 shows the full definition of term precision for PolyC.

**Lemma E.1 (Casts are Monotone).** If $A^\sqsubseteq : A_1 \sqsubseteq A_2$ and $AB^\sqsubseteq : A_1 \sqsubseteq B_1$ and $AB^\sqsubseteq : A_r \sqsubseteq B_r$ and $B^\sqsubseteq : B_1 \sqsubseteq B_r$, then
Graduality and Parametricity: Together Again for the First Time

\[ S[U] \mapsto U \]

\( \Sigma \vdash S[\text{newcase}_A x; M] \mapsto \Sigma, \sigma : A \mapsto S[M[\sigma/x]] \)

\( S[\text{match inj}_\sigma V \text{ with } \sigma (\text{inj } x.M \mid N)] \mapsto S[M[V/x]] \)

\( S[\text{match inj}_{\sigma_1} V \text{ with } \sigma_2 (\text{inj } x.M \mid N)] \mapsto S[N] \)  

(where \( \sigma_1 \neq \sigma_2 \))

\( S[\text{if true then } M \text{ else } N] \mapsto S[M] \)

\( S[\text{if false then } M \text{ else } N] \mapsto S[N] \)

\( S[\text{let } (x,y) = (V_1,V_2); M] \mapsto S[M[V_1/x,V_2/y]] \)

\( S[\text{force (thunk } M)] \mapsto S[M] \)

\( S[\text{unpack } (X,x) = \text{pack}(A,V); M] \mapsto S[M[A/X,V/x]] \)

\( S[(\lambda X.M)[A]] \mapsto S[M[A/X]] \)

\( S[(\lambda(x:A).M) V] \mapsto S[M[V/x]] \)

\( S[x \leftarrow \text{ret } V; N] \mapsto S[N[V/x]] \)

Fig. 25. CBPV\textsubscript{O}S\textsubscript{um} Operational Semantics

\[
\begin{align*}
\text{[x]} & = \text{ret } x \\
\text{[let } x = M; N]\] & = x \leftarrow [M]; [N] \\
\text{[U}_A]\] & = U \\
\text{[seal}_\alpha M]\] & = [M] \\
\text{[unseal}_\alpha M]\] & = [M] \\
\text{[inj}_G M]\] & = r \leftarrow [M]; \text{ret inj}_{\text{case}(G)} r \\
\text{[is}(G)\text{)? } M]\] & = r \leftarrow [M]; \\
& \text{match r with case}(G)[\text{inj } y.\text{ret true} \mid \text{ret false}] \\
\text{[hide } X \equiv A; M]\] & = \text{newcase}_{[A]} \lambda x : [M] \\
\text{[(A^E)\uparrow M]} & = \text{[A^E]\uparrow \text{[}[M]\]} \\
\text{[true]} & = \text{ret true} \\
\text{[false]} & = \text{ret false} \\
\text{[if } M_1 \text{ then } M_2 \text{ else } M_3]\] & = r \leftarrow [M_1]; \text{if } r \text{ then } [M_2] \text{ else } [M_3] \\
\text{[(M_1,M_2)]} & = x_1 \leftarrow [M_1]; x_2 \leftarrow [M_2]; \text{ret } (x_1,x_2) \\
\text{[let } (x,y) = M; N]\] & = r \leftarrow [M]; \text{let } (x,y) = r; [N] \\
\text{[pack'}(X \equiv A,M)\]] & = \text{ret pack}(A,\text{thunk } (\lambda c_X : \text{Case } A.[M])) \\
\text{[pack'}(X \equiv A',[A_1^E \uparrow n \ldots A_1^E \uparrow 1],M)\]] & = \text{ret pack}(A,\text{thunk } (\lambda c_X : \text{Case } A.M'_n)) \\
& \text{where } M'_n = [M] \\
& \text{and } M'_{n+1} = [A_1^E\uparrow 1][n+1][\text{force } \text{thunk } (\lambda c_X : \text{Case } A'.[M_1]) c_X] \\
\text{[unpack } (X,x) = M; N]\] & = r \leftarrow [M]; \text{unpack } (X,f) = r; \\
& \text{newcase}_x c_X; x \leftarrow (\text{force } f)x; [N] \\
\text{[A'}X.M]\] & = \text{ret } (\text{thunk } (AX.\lambda(c_X : \text{Case } X).[M])) \\
\text{[M}(a \equiv A)\]] & = f \leftarrow [M]; (\text{force } f)[A] \Theta \text{case}(a) \\
\text{[\lambda(x : A).M]} & = \text{ret } \text{thunk } \lambda(x : [A].[M]) \\
\text{[M N]} & = f \leftarrow [M]; a \leftarrow [N]; (\text{force } f) a \\
\end{align*}
\]

Fig. 26. PolyC\textsuperscript{v} term translation

\[
\begin{align*}
\Gamma^C : \Gamma_I \subseteq \Gamma_r & \quad \Gamma^C \vdash A^C : A_I \subseteq A_r \\
(\Gamma^C, x : A^C) : \Gamma_I, x : A_I \subseteq \Gamma_r, x : A_r & \quad (\Gamma^C, X \equiv A^C) : \Gamma_I, X \equiv A_I \subseteq \Gamma_r, X \equiv A_r \\
\Gamma^C : \Gamma_I \subseteq \Gamma_r & \quad (\Gamma^C, X) : \Gamma_I, X \subseteq \Gamma_r, X
\end{align*}
\]

Fig. 27. Type Precision Contexts

\[
\begin{align*}
cod(A^E \rightarrow B^E) & = B^E \\
cod(?) & = ? \\
cod(tag_G(A^E \rightarrow B^E)) & = B^E \\
\pi_1(A^E_0 \times A^E_1) & = A^E_0 \\
\pi_1(?) & = ? \\
\pi_1(tag_{?X!}(A^E_0 \times A^E_1)) & = A^E_1 \\
\text{un} \forall^v(\forall^vX.A^E) & = A^E \\
\text{un} \forall^v(?) & = ? \\
\text{un} \forall^v(tag_{\forall^vX;}(\forall^vX.A^E)) & = A^E \\
\text{un} \exists^v(\exists^vX.A^E) & = A^E \\
\text{un} \exists^v(?) & = ? \\
\text{un} \exists^v(tag_{\exists^vX;}(\exists^vX.A^E)) & = A^E 
\end{align*}
\]

Fig. 28. Metafunctions extended to type precision derivations

(1) If \( \Gamma^C \vdash M_I \subseteq M_r : A^E; \Gamma^C \), then \( \Gamma^C \vdash \langle AB^C_I \rangle^\dagger M_I \subseteq \langle AB^C_I \rangle^\dagger M_r : B^E; \Gamma^C_o \)

(2) If \( \Gamma^C \vdash N_I \subseteq N_r : B^E; \Gamma^C_o \), then \( \Gamma^C \vdash \langle c \rangle \downarrow \text{inserts that } AB^C_I N_I \subseteq \langle AB^C_I \rangle \downarrow N_r : A^E; \Gamma^C_o \)

PROOF. (1) By the following derivation

\[
\begin{align*}
\Gamma^C + M_I & \subseteq M_r : A^E; \Gamma^C_o \\
\Gamma^C & \vdash M_I \subseteq \langle AB^C_I \rangle^\dagger M_r : AB^C_{II}; \Gamma^C_o \\
\Gamma^C & \vdash \langle AB^C_I \rangle^\dagger M_I \subseteq \langle AB^C_I \rangle^\dagger M_r : B^E; \Gamma^C_o
\end{align*}
\]

Where \( AB^C_{II} : A_I \subseteq B_r \), which exists by transitivity lemma B.3.

(2) By the following derivation

\[
\begin{align*}
\Gamma^C + N_I & \subseteq N_r : B^E; \Gamma^C_o \\
\Gamma^C & \vdash \langle AB^C_I \rangle^\dagger N_I \subseteq \langle AB^C_I \rangle^\dagger N_r : AB^C_{II}; \Gamma^C_o \\
\Gamma^C & \vdash \langle AB^C_I \rangle^\dagger N_I \subseteq \langle AB^C_I \rangle^\dagger N_r : A^E; \Gamma^C_o
\end{align*}
\]

Where \( AB^C_{II} : A_I \subseteq B_r \), which exists by transitivity lemma B.3.

\[
\square
\]

**Lemma E.2 (Hide Monotonicity).** If \( \Gamma^C_1 : \Gamma_I \subseteq \Gamma_r \) and \( \Gamma^C_2 : \Gamma_I \subseteq \Gamma_r \) and \( \Gamma_I \subseteq \Gamma_r \) and \( \Gamma_I \subseteq \Gamma_r \) and \( M_I \subseteq M_r \), then

\[
\text{hide } \Gamma_I \subseteq \Gamma_r; M_I \subseteq \text{hide } \Gamma_r \subseteq \Gamma_r; M_r
\]
Graduality and Parametricity: Together Again for the First Time

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C \quad \Gamma^C, \Gamma^C' \vdash B^C : B_l \subseteq B_r \\
\quad \Gamma^C \vdash (M_l :: B_l) \subseteq (M_r :: B_r) : B^C; \Gamma^C' \\
\quad \Gamma^C \vdash x : A^C \in \Gamma^C
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C_M \quad \Gamma^C, \Gamma^C, \Gamma^C_x \vdash N_l \subseteq N_r : B^C; \Gamma^C_N \\
\quad \Gamma^C \vdash x : A^C + N_l \subseteq N_r : B^C; \Gamma^C_N
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C_M \\
\quad \Gamma^C \vdash \text{seal}_X M_l \subseteq \text{seal}_X M_r : \Gamma^C
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash \text{true} \subseteq \text{true} : \mathbb{B}; \\
\quad \Gamma^C \vdash \text{false} \subseteq \text{false} : \mathbb{B};
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C_M \\
\quad \Gamma^C, \Gamma^C_M \vdash N_l \subseteq N_r : B^C; \Gamma^C \\
\quad \Gamma^C, \Gamma^C_M \vdash N_{l_f} \subseteq N_{r_f} : B^C; \Gamma^C_f \\
\quad \Gamma^C \vdash \text{if} M_l \text{ then } N_{l_f} \text{ else } N_{r_f} \vdash B^C \cup B^C_f; \Gamma^C_M, \Gamma^C \\
\quad \Gamma^C, \Gamma^C_M \vdash N_{l_f} \subseteq N_{r_f} : B^C; \Gamma^C_f \\
\quad \Gamma^C, \Gamma^C_M \vdash N_{l_f} \subseteq N_{r_f} : B^C; \Gamma^C_f \\
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C_M \\
\quad \Gamma^C, \Gamma^C, \Gamma^C_x \vdash \pi_0(A^C_x), y : \pi_1(A^C) + N_l \subseteq N_r : B^C; \Gamma^C_N \\
\quad \Gamma^C \vdash \text{let} (x, y) = M_l; N_l \subseteq \text{let} (x, y) = M_r; N_r : B^C; \Gamma^C_M, \Gamma^C_N
\end{align*}
\]

\[
\begin{align*}
\Gamma^C, \Gamma^C \vdash x : A^C + M_l \subseteq M_r : B^C; \Gamma^C' \\
\quad \Gamma^C \vdash A^C : A_l \subseteq A_r \\
\quad \Gamma^C \vdash A^C \to B^C \\
\quad \Gamma^C \vdash \lambda x : A_l. M_l \subseteq \lambda x : A_r. M_r : A^C \to B^C
\end{align*}
\]

\[
\begin{align*}
\Gamma^C, \Gamma^C_M \vdash N_l \subseteq M_r : N_r : B^C; \Gamma^C_N \\
\quad \Gamma^C \vdash M_l \vdash N_l \subseteq M_r : \Gamma^C_M \vdash \text{cod}(A^C); \Gamma^C_M, \Gamma^C_N
\end{align*}
\]

\[
\begin{align*}
\Gamma^C, \Gamma^C, \Gamma^C_M \vdash X : \Gamma^C \vdash \text{pack}^\nu(X \equiv B_l, M_l) \subseteq \text{pack}^\nu(X \equiv B_r, M_r) : \exists \nu.X.A^C
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash \text{unpack}(X, x) = M_l; N_l \subseteq \text{unpack}(X, x) = M_r; N_r : B^C; \Gamma^C_M, \Gamma^C_N \mid X \\
\quad \Gamma^C, \Gamma^C \vdash X \vdash M_l \subseteq M_r : A^C; \Gamma^C_N \\
\quad \Gamma^C, \Gamma^C_M \vdash X \vdash \text{un} \nu^\nu(A^C) + N_l \subseteq N_r : B^C; \Gamma^C_N
\end{align*}
\]

\[
\begin{align*}
\Gamma^C, \Gamma^C \vdash X = M_l; N_l \subseteq \text{unpack}(X, x) = M_r; N_r : B^C; \Gamma^C_M, \Gamma^C_N | X
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash \lambda x : \Gamma^C.M_l \vdash \lambda x : A^C.M_r : \forall \nu.X.A^C
\end{align*}
\]

\[
\begin{align*}
\Gamma^C \vdash M_l \subseteq M_r : A^C; \Gamma^C_M \\
\quad \Gamma^C \vdash B^C : B_l \subseteq B_r \\
\quad \Gamma^C \vdash M_l \{X \equiv B_l\} \subseteq M_r \{X \equiv B_r\} : \text{un} \nu(\nu^\nu(A^C); \Gamma^C_M, X \equiv B^C
\end{align*}
\]

Fig. 29. PolyG\nu Term Precision
Fig. 30. PolyC' Term Precision Part 1

Proof. By induction over $\Gamma^E$, each case is either congruence or trivial application of inductive hypothesis.

Lemma E.3. If $\Gamma^E \vdash M_0 \subseteq M_r : A^E; \Gamma^E'$ in the surface language, then $\Gamma^E \vdash M^+_t \subseteq M^+_r : A^E; \Gamma^E'$

Proof. By induction on term precision derivations.
(1) Cast. By applying lemma E.1 twice.
(2) Var: Immediate
(3) Let: immediate
(4) seal: by the argument for the ascription case.
(5) unseal: There are three cases for $A^E : X, ?$ and tag$_X$(X). The first case is immediate. If $A^E = \Gamma^E$, we need to show

$$\text{unseal}_X \langle \text{tag}_X(X) \rangle \downarrow M^+_t \subseteq \text{unseal}_X \langle \text{tag}_X(X) \rangle \downarrow M^+_r$$

which follows by congruence and lemma E.1. For the final case we need to show

$$\text{unseal}_X M^+_t \subseteq \text{unseal}_X \langle \text{tag}_X(X) \rangle \downarrow M^+_r$$

which follows by congruence for unseal$_X$ and the downcast-right rule.
(6) tag-check is(G)? $M_t \subseteq \text{is}(G) ? M_r$: follows by congruence and lemma E.1.
(7) tru: Immediate
(8) fls: Immediate
(9) if: By if congruence, we need to show the condition and the two branches of the if are ordered.
   - For the condition there are three subcases $M_t \subseteq M_r : A^E$: either $\mathbb{B},$ ? or tag$_E$(B). The ordering follows by the same argument as the unseal$_X$ case.
   - The two branches follow by the same argument. We describe the true branch. We have by inductive hypothesis that $N^+_t \subseteq N^+_r$ and we need to show

$$\langle B^E_{\mathcal{I}_t} \rangle \downarrow \text{hide } \Gamma_{\mathcal{I}_t} \subseteq \Gamma_{\mathcal{I}_t} \cap \Gamma_{\mathcal{F}_t}; N^+_t \subseteq \langle B^E_{\mathcal{I}_r} \rangle \downarrow \text{hide } \Gamma_{\mathcal{I}_r} \subseteq \Gamma_{\mathcal{I}_r} \cap \Gamma_{\mathcal{F}_r}; N^+_r$$

which follows by lemmas E.1 and E.2.
(10) Pair intro: Immediate by inductive hypothesis.
(11) Pair elim: By similar argument to unseal$_X$
(12) By congruence and lemma E.2.
(13) Function application: similar argument to unseal$_X$
(14) $\forall^v$ introduction: by congruence and lemma E.2.
(15) $\forall^v$ elimination: by similar argument to unseal$_X$, and lemma E.2.
(16) $\forall^v$ introduciton: by congruence and lemma E.2.
(17) $\forall^v$ elimination: by similar argument to unseal$_X$ case.

□
Graduality and Parametricity: Together Again for the First Time

\( x : A^\in \in \Gamma^\in \)

\( \Gamma^\in \vdash M_f \subseteq M_r : A^\in ; \Gamma_M \)

\( \Gamma^\in, \Gamma_M, x : A^\in \vdash N_f \subseteq N_r : B^\in ; \Gamma_N \)

\( \Gamma^\in \vdash \text{let } x = M_l; N_l \vdash \text{let } x = M_r; N_r : B^\in ; \Gamma_M, \Gamma_N \)

\( \Gamma^\in \vdash M_f \subseteq M_r : B^\in ; \Gamma'_M, X \equiv A^\in, \Gamma'' \)

\( \Gamma^\in, \Gamma'_M \vdash A^\in : A_f \subseteq A_r \)

\( \Gamma^\in \vdash \text{hide } X \equiv A_l; M_f \subseteq \text{hide } X \equiv A_r; M_r : B^\in ; \Gamma'_M, \Gamma'' \)

\( (X \equiv A^\in) \in \Gamma^\in, \Gamma'_M \)

\( \Gamma^\in, \Gamma'_M \vdash M_f \subseteq M_r : A^\in ; \Gamma'' \)

\( \Gamma^\in \vdash \text{seal}_X M_f \subseteq \text{seal}_X M_r : X ; \Gamma'' \)

\( \Gamma^\in \vdash \text{unseal}_X M_f \subseteq \text{unseal}_X M_r : A^\in ; \Gamma'' \)

\( \Gamma^\in \vdash \text{true } \subseteq \text{true } : B \)

\( \Gamma^\in \vdash M_f \subseteq M_r : \exists X. M_l \)

\( \Gamma^\in, \Gamma'_M \vdash N_{l_1} \subseteq N_{r_1} : B^\in ; \Gamma_{N_1} \)

\( \Gamma^\in, \Gamma'_M \vdash N_{l_f} \subseteq N_{r_f} : B^\in ; \Gamma_{N_1} \)

\( \Gamma^\in \vdash \text{if } M_f \text{ then } N_{l_f} \text{ else } N_{r_f} \subseteq \text{if } M_r \text{ then } N_{r_f} \text{ else } N_{r_f} : B^\in ; \Gamma_{M_1}^\in, \Gamma_{N_1}^\in \)

\( \Gamma^\in \vdash M_{l_1} \subseteq M_1 : A_1^\in \times A_2^\in ; \Gamma_M^\in \)

\( \Gamma^\in, \Gamma'_M \vdash M_{l_2} \subseteq M_2 : A_1^\in ; \Gamma_2^\in \)

\( \Gamma^\in \vdash (M_{l_1}, M_{l_2}) \subseteq (M_{r_1}, M_{r_2}) : A_1^\in \times A_2^\in ; \Gamma_1^\in, \Gamma_2^\in \)

\( \Gamma^\in \vdash \text{let } (x, y) = M_l; N_l \vdash \text{let } (x, y) = M_r; N_r : B^\in ; \Gamma_M^\in, \Gamma_N^\in \)

\( \Gamma^\in, x : A^\in \vdash M_l \subseteq M_r : B^\in ; \)

\( \Gamma^\in, \lambda x : A_l. M_l \subseteq \lambda x : A_l. M_l : A^\in \to B^\in ;\)

\( \Gamma^\in \vdash M_f \subseteq M_r : A^\in \to B^\in ; \Gamma_M^\in \)

\( \Gamma^\in, \Gamma_M^\in \vdash N_l \subseteq M_r : B^\in ; \Gamma_N^\in \)

\( \Gamma^\in \vdash M_l \vdash M_r : A^\in ;\)

\( \Gamma^\in \vdash \text{pack}^\forall (X \equiv B_l, M_l) \subseteq \text{pack}^\forall (X \equiv B_r, M_r) : \exists \forall X. A^\in ;\)

\( \Gamma^\in \vdash \text{unpack} (X, x) = M_l; N_l \subseteq \text{unpack} (X, x) = M_r; N_r : B^\in ; \Gamma_M^\in, \Gamma_N^\in \)

\( \Gamma^\in \vdash M_f \subseteq M_r : \forall X. A^\in ; \Gamma_M^\in \)

\( \Gamma^\in \vdash B^\in \vdash B_l \subseteq B_r \)

\( \Gamma^\in \vdash X_f \vdash M_r : A^\in ;\)

\( \Gamma^\in \vdash A^\forall. X, M_l \subseteq A^\forall. X, M_r : \forall X. A^\in ;\)

Fig. 31. PolyC\^\forall Term Precision Part 2
F SIMULATION

We provide the full proof of simulation here, including all supporting lemmas. As outlined in the paper, this proof revolves around a translation relation that generalizes our translation function.

**Lemma F.1 (Target Language Dynamic Semantics are Deterministic).** If $\Sigma \triangleright M \leftrightarrow \Sigma_1 \triangleright M_1$ and $\Sigma \triangleright M \leftrightarrow \Sigma_2 \triangleright M_2$ then $\Sigma_1 = \Sigma_2$ and $M_1 = M_2$.

**Lemma F.2.** If $\Sigma; \Gamma \vdash M : A; \Gamma'$, then $M \leadsto^CT [\Sigma; \Gamma \vdash M]$.

**Proof.** By induction on $M$. □

The proof of simulation requires knowledge about how substitutions translate. Value substitutions are straightforward due to the definition of our relation. Type substitutions behave differently depending on whether the type variable to be replaced is known.

**Lemma F.3.** If $\Sigma; \Gamma \vdash M \leadsto^CT M' : B; \Gamma'$ and $X \in \Gamma$ and $\sigma : A \in \Sigma$, then $M[\sigma/X] \leadsto^CT M'[\Sigma; \Gamma \vdash A]/X[\sigma/c_X]$.

**Proof.** By induction on the derivation of $M \leadsto^CT M'$. □

**Lemma F.4.** If $\Sigma; \Gamma \vdash M \leadsto^CT M' : B; \Gamma'$ and $X \equiv A \in \Gamma, \Gamma'$ and $\sigma : A \in \Sigma$, then $M[\sigma/X] \leadsto^CT M'[\Sigma; \Gamma \vdash A]/X[\sigma/c_X]$.

**Proof.** By induction on the derivation of $M \leadsto^CT M'$. □

**Lemma F.5.** If $\Sigma; \Gamma, x : A', \Gamma' \vdash M \leadsto^CT M' : A; \Gamma''$ and $\Sigma; \Gamma \vdash V \leadsto^CT \text{ret } V' : A'; \Gamma''$, then $\Sigma; \Gamma, \Gamma' \vdash M[V/x] \leadsto^CT M'[V'/x] : A; \Gamma''$.

**Proof.** By induction on the derivation of $M \leadsto^CT M'$. □

We use a restricted form of reduction that only eliminates bind steps in our proof of simulation. This reduction gives us exactly the leeway we need to deal with the possibility of administrative redexes in our translation.

**Definition F.6 (Bind reduction).** We define $\leftrightarrow_b$ to be the least relation on closed terms such that $S[x \leftarrow \text{ret } V'; M'] \leftrightarrow_b S[M'[V'/x]]$. Note that $\leftrightarrow_b$ is a strict subset of $\leftrightarrow$. We use $\leftrightarrow^*_b$ to mean its reflexive, transitive closure.

**Lemma F.7 (Bind reduction normalization).** Let $\Sigma; \cdot \vdash M_1 : A$. There exists a unique $M_2$ such that $M_1 \leftrightarrow^*_b M_2$ and $M_2$ does not take a bind reduction.

**Proof.** By induction on the number of binds in $M_1$ not under thunks. If $M_1 = S[x \leftarrow \text{ret } V; N_1]$ then $M_1 \leftrightarrow_b N_1[V/x]$ and we conclude by the inductive hypothesis for $N_1[V/x]$. Otherwise, $M_1$ must not take a bind reduction, so $M_2 = M_1$ and we conclude by reflexivity. □

**Lemma F.8 (Bind reduction confluence).** Let $\Sigma; \cdot \vdash M_1 : A, \Sigma; \cdot \vdash M_2 : A$, and $\Sigma; \cdot \vdash M_3 : A$. If $M_1 \leftrightarrow^*_b M_2$, $M_1 \leftrightarrow^*_b M_3$, and $M_3$ does not take a bind reduction, then $M_2 \leftrightarrow^*_b M_3$. □
Graduality and Parametricity: Together Again for the First Time

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} x \leftarrow \text{ret } V'; M' : A; \Gamma
\]

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} M'[V'/x] : A; \Gamma
\]

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} M' \vdash A; \Gamma'
\]

\[
\Sigma; \Gamma \vdash (M_1, M_2) \rightsquigarrow^{CT} x_1 \leftarrow M'_1; x_2 \leftarrow M'_2; \text{ret } (x_1, x_2) : A_1 \times A_2; \Gamma, \Gamma'
\]

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} M' : B; \Gamma'
\]

\[
\Sigma; \Gamma, \Gamma' \vdash M_1 \rightsquigarrow^{CT} M'_1 : A; \Gamma''
\]

\[
\Sigma; \Gamma, \Gamma' \vdash M_2 \rightsquigarrow^{CT} M'_2 : A; \Gamma''
\]

\[
\Sigma; \Gamma \vdash \text{if } M \text{ then } M_1 \text{ else } M_2 \rightsquigarrow^{CT} r \leftarrow M'; \text{if } r \text{ then } M'_1 \text{ else } M'_2 : A; \Gamma', \Gamma''
\]

\[
\Sigma; \Gamma \vdash \text{true} \rightsquigarrow^{CT} \text{ret true : } B;\cdot
\]

\[
\Sigma; \Gamma \vdash \text{false} \rightsquigarrow^{CT} \text{ret false : } B;\cdot
\]

\[
\Sigma; \Gamma \vdash \text{U} \rightsquigarrow^{CT} \text{U : } A;\cdot
\]

\[
\Sigma; \Gamma \vdash \text{pack}^v(X \equiv A, M) \rightsquigarrow^{CT} \text{ret pack}(A', \text{thunk } \lambda(c_X : \text{Case } A').M') \text{ as } [\Sigma; \Gamma \vdash \exists^v X. B] : \exists^v X. B;\cdot
\]

\[
\Sigma; \Gamma \vdash \text{unpack} (X, x) = M; N \rightsquigarrow^{CT} r \leftarrow M'; \text{unpack} (X, f) = r; \text{newcase}_X c_X; x \leftarrow (\text{force } f) c_X; N'
\]

\[
\Sigma; \Gamma \vdash \text{Ax} \cdot M \rightsquigarrow^{CT} M' : A;\cdot
\]

\[
\Sigma; \Gamma \vdash \Lambda X. M \rightsquigarrow^{CT} \text{ret thunk } \Lambda X. \lambda(c_X : \text{Case } X). M' : \forall^v X. A;\cdot
\]

\[
\Sigma; \Gamma \vdash M[X \equiv A] \rightsquigarrow^{CT} f \leftarrow M'; (\text{force } f)[A] c_X : B; \Gamma', X \equiv A
\]

\[
\Sigma; \Gamma \vdash M[\sigma \equiv A] \rightsquigarrow^{CT} f \leftarrow M'; (\text{force } f)[A] \sigma : B[\sigma/X]; \Gamma'
\]

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} M' : A \rightarrow B; \Gamma'
\]

\[
\Sigma; \Gamma \vdash \text{unseal}_X M \rightsquigarrow^{CT} M' : X; \Gamma'
\]

\[
\Sigma; \Gamma \vdash \text{seal}_X M \rightsquigarrow^{CT} M' : X; \Gamma'
\]

\[
\Sigma; \Gamma \vdash M \rightsquigarrow^{CT} M' : X; \Gamma'
\]

\[
\Sigma; \Gamma \vdash \text{unseal}_X M \rightsquigarrow^{CT} M' : A; \Gamma'
\]

Fig. 32. PolyC^v translation relation
\[ \Sigma; \Gamma \vdash M \rightsquigarrow_{CT} M' : G; \Gamma' \]

\[ \Sigma; \Gamma \vdash \text{inj}_G \ M \rightsquigarrow_{CT} x \leftarrow M' \text{; ret } \text{inj}_{\text{case}(G)} \ x : ?; \Gamma' \]

\[ \Sigma; \Gamma, X \equiv A \vdash M \rightsquigarrow_{CT} M' : B; \quad A' = [\Sigma; \Delta \vdash A] \]

\[ \Sigma; \Gamma \vdash \text{pack}^V (X \equiv A, [\Gamma], M) \rightsquigarrow_{CT} \text{ret pack}(A', \text{thunk } \lambda (c_X : \text{Case } A').M') \text{ as } [\Sigma; \Gamma \vdash \exists X. B \vdash : \exists X. B]. \]

\[ \Sigma; \Gamma \vdash \text{pack}^V (X \equiv A, [A_2 \downarrow \ldots], M) \rightsquigarrow_{CT} \text{ret pack}(A', \text{thunk } \lambda (c_X : \text{Case } A').M') \text{ as } \exists X. A' \downarrow_1 \vdash : \exists X. A' \downarrow_1 \to \text{;} \]

\[ \vdots \]

\[ \vdots \]

\[ \Sigma; \Gamma \vdash (A^E) \downarrow M \rightsquigarrow_{CT} (\exists X. A^E) \downarrow [M'] \vdash A^E ; \Gamma' \quad \Sigma; \Gamma \vdash (A^E) \downarrow M \rightsquigarrow_{CT} (\exists X. A^E) \downarrow [M'] \vdash A^E ; \Gamma' \]

\[ \Sigma; \Gamma \vdash M \rightsquigarrow_{CT} M' : ?; \Gamma' \]

Fig. 33. PolyC\textsuperscript{V} translation relation (Continued)

**Proof.** By induction on \( M_1 \mapsto_{b}^* M_2 \). If \( M_1 = M_2 \), then we conclude since \( M_1 \mapsto_{b}^* M_3 \). Otherwise, by the definition of \( \mapsto_{b}^* \), we have \( M_1 \mapsto_{b} N_1 \mapsto_{b}^* M_2 \). Since \( M_1 \) takes a bind reduction, \( M_1 \neq M_3 \) and so we must have \( M_1 \mapsto_{b} N_2 \mapsto_{b}^* M_3 \). Furthermore, by Lemma C.2 (deterministic semantics), we have \( N_1 = N_2 \). We then conclude by the inductive hypothesis for \( N_2, M_2, M_3 \).

At the core of our simulation argument is the guarantee that, up to some bind reductions, any related terms have a related structure that follows the form of the translation function.

**Lemma F.9 (canonical forms of the translation relation).** If \( \Sigma; \cdot \vdash M \rightsquigarrow_{CT} M' : B \), then either

- \( M = \Lambda X. N_1 \text{ and } M' = \text{ret thunk } \Lambda X. \lambda (c_X : \text{Case } X). N_1' \)
- \( M = N_1 \{ \sigma \equiv A \text{ and } (f \leftarrow N'_1 ; (\text{force } f)[A][\sigma][y_p] \mapsto_{b}^* M'[y_p]] \}
- \( M = \text{hide } X \equiv A; N_1 \text{ and } M' = \text{newcase}_A c_X; N'_1 \)
- \( M = (N_1, N_2) \text{ and } (x_1 \leftarrow N'_1 ; x_2 \leftarrow N'_2) ; \text{ret } (x_1, x_2) [y_p] \mapsto_{b}^* M'[y_p] \)
- \( M = \text{let } (x, y) = N_1; N_2 \text{ and } (r \leftarrow N'_1 ; \text{let } (x, y) = r; N'_2 [y_p]) \mapsto_{b}^* M'[y_p] \)
- \( M = \text{true } \text{and } M' = \text{ret } \text{true} \)
- \( M = \text{false } \text{and } M' = \text{ret } \text{false} \)
- \( M = \text{if } N_1 \text{ then } N_2 \text{ else } N_3 \text{ and } (r \leftarrow N'_1 ; \text{if } r \text{ then } N'_2 \text{ else } N'_3) [y_p] \mapsto_{b}^* M'[y_p] \)
- \( M = \text{let } x = N_1; N_2 \text{ and } (x \leftarrow N'_1 ; N'_2) [y_p] \mapsto_{b}^* M'[y_p] \)
- \( M = U_A \text{ and } M' = U \)
\[
M = \text{inj}_G \ N_1 \text{ and } (x \leftarrow N'_1 ; \text{inj}_{\text{case}(G)} \ x)[\gamma_p] \mapsto_b^* M'[\gamma_p]
\]
\[
M = (A \downarrow) N_1 \text{ and } [\Sigma; \vdash A \downarrow][N'_1][\gamma_p] \mapsto_b^* M'[\gamma_p]
\]
\[
M = \text{is}(G)? \ N_1 \text{ and }
(r \leftarrow M'; \text{match case}(G) \text{ with } r \{ \text{inj } y. \text{ret } \text{true} | \text{ret } \text{false} \})[\gamma_p] \mapsto_b^* M'[\gamma_p]
\]
\[
M = \text{seal}_\sigma N_1 \text{ and } N'_1 \mapsto_b^* M'
\]
\[
M = \text{unseal}_\sigma N_1 \text{ and } N'_1 \mapsto_b^* M'
\]
\[
M = \text{pack}^\nu(X \equiv A, N_1) \text{ and } M' = \text{ret } \text{pack}(A, \text{thunk } \lambda(c_X : \text{Case } A).N'_1) \text{ as } \lfloor B \rfloor
\]
\[
M = \text{pack}^\nu(X \equiv A', [A \downarrow \ldots], N_1) \text{ and } M' = \text{ret } \text{pack}(A, \text{thunk } \lambda(c_X : \text{Case } A).M'_n) \text{ as } \lfloor B \rfloor
\]
\[
\text{where }
M'_0 = N'_1
\]
\[
M'_{i+1} = [\Sigma; \Delta \vdash A]_{i+1}[\text{force } (\text{thunk } \lambda(c_X : \text{Case } A').\lfloor M_i \rfloor)] \ c_X
\]
\[
M = \text{unpack } (X, x) = N_1; N_2 \text{ and } \text{unpack } M \mapsto_b^* M' \text{ where }
M_{\text{unpack}} = r \leftarrow N'_1; \text{unpack } (X, f) = r;
\text{newcase } \begin{cases} x : \text{c}_X; c_X : \text{f} & \text{where } \end{cases}
\]

for some \( N_1, \ldots, N_n \) and \( N'_1, \ldots, N'_n \) where \( N_i \sim_{\text{CT}} N'_i \).

**Proof.** By induction on the derivation of \( M \sim_{\text{CT}} M' \). If the derivation ends in a congruence
from the translation, the conclusion is immediate by reflexivity. We then have only the bind-reduction rule to consider. In this case, we have \( M' = M''[V'/x] \) where \( M \sim_{\text{CT}} x \mapsto \text{ret } V'; M'' \).

By the inductive hypothesis, we have the desired property for \( M \) and \( x \mapsto \text{ret } V'; M'' \). Then, since \((x \leftarrow \text{ret } V'; M'')[\gamma_p] \mapsto_b^* M''[V'/x][\gamma_p] = M'[\gamma_p] \), we have what we need to show by transitivity.

\[\Box\]

**Lemma F.10.** Let \( \Sigma; \vdash V : A \). If \( V \sim_{\text{CT}} M' \), then for any closing \( \gamma M'[\gamma] \mapsto_b^* \text{ret } V'[\gamma] \) and \( V \sim_{\text{CT}} \text{ret } V' \) for some \( V' \).

**Proof.** By induction on the derivation of \( V \sim_{\text{CT}} M' \). All cases are immediate since \( M' = \text{ret } V' \) excluding the case:

\( (V_1, V_2) \sim_{\text{CT}} x_1 \leftarrow N'_1; x_2 \leftarrow N'_2; \text{ret } (x_1, x_2) \) where \( V_1 \sim_{\text{CT}} N'_1 \) and \( V_2 \sim_{\text{CT}} N'_2 \): By the inductive hypothesis for each of these sub-derivations, we have \( N'_1[\gamma] \mapsto_b^* \text{ret } V'_1[\gamma] \) and \( V_i \sim_{\text{CT}} \text{ret } V'_i \). Then, by definition, we have \( M'[\gamma] \mapsto_b^* \text{ret } (V_1, V_2)[\gamma] \) and \( (V_1, V_2) \sim_{\text{CT}} \text{ret } (V_1, V_2) \) as we were required to show.

\[\Box\]

Since our dynamic semantics for PolyC\(^v\) uses evaluation contexts, we need to be able to describe their transformation into CBPV\(_{\text{OSum}}\).

**Lemma F.11 (Translation Context Decomposition).** If \( \Sigma; \vdash M_1 : A \) and \( \Sigma; \vdash E[M_1] \sim_{\text{CT}} M' : B \), then for any closing \( \gamma \), \( S[M'_1][\gamma] \mapsto_b^* M'[\gamma] \) for some \( S \) and \( M'_1 \) such that \( \Sigma; \vdash M_1 \sim_{\text{CT}} M'_1 : A \) and \( E \sim_{\text{CT}} S \).

**Proof.** By induction on the derivation of \( E[M_1] \sim_{\text{CT}} M' \). All cases are straightforward except the bind rule. That case proceeds as follows: we have \( E[M_1] \sim_{\text{CT}} x \mapsto \text{ret } V'_1; M'' \) and, from the inductive hypothesis, some \( S, M'_1 \) such that \( S[M'_1][\gamma] \mapsto_b^* (x \mapsto \text{ret } V'_1; M''')[\gamma] \), \( \Sigma; \vdash M_1 \sim_{\text{CT}} M'_1 : A \), and \( E \sim_{\text{CT}} S \). It then suffices to show that \( S[M'_1][\gamma] \mapsto_b^* M'''[V'_1/x][\gamma] \), which we have by definition.

\[\Box\]
\[
\Sigma, \Gamma \vdash \bullet : A \vdash \left[ \right] \overset{CT}{\Rightarrow} \bullet : A
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash E_1 \overset{CT}{\Rightarrow} S'_1 : A_1 ; \Gamma_1 \quad \Sigma, \Gamma, \Gamma_1 \vdash M_2 \overset{CT}{\Rightarrow} M'_2 : A_2 ; \Gamma_2
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash (E_1, M_2) \overset{CT}{\Rightarrow} x_1 \leftarrow S'_1 ; x_2 \leftarrow M'_2 ; \text{return} (x_1, x_2) : A_1 \times A_2 ; \Gamma_1, \Gamma_2
\]

\[
\Sigma, \Gamma \vdash V_1 \overset{CT}{\Rightarrow} \text{return } V'_1 : A_1 ; \Gamma_1 \quad \Sigma, \Gamma, \Gamma_1 \vdash \bullet : A \vdash E_2 \overset{CT}{\Rightarrow} S'_2 : A_2 ; \Gamma_2
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash (V_1, E_2) \overset{CT}{\Rightarrow} x_2 \leftarrow S'_2 ; \text{return} (V'_1, x_2) : A_1 \times A_2 ; \Gamma_1, \Gamma_2
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash E \overset{CT}{\Rightarrow} S' : \exists \Gamma' \quad \Sigma, \Gamma, \Gamma' \vdash M_1 \overset{CT}{\Rightarrow} M'_1 : A ; \Gamma'' \quad \Sigma, \Gamma, \Gamma' \vdash M_2 \overset{CT}{\Rightarrow} M'_2 : A ; \Gamma''
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash \text{if } E \text{ then } M_1 \text{ else } M_2 \overset{CT}{\Rightarrow} r \leftarrow S' ; \text{if } r \text{ then } M'_1 \text{ else } M'_2 : A ; \Gamma', \Gamma''
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash E \overset{CT}{\Rightarrow} S' : A ; \Gamma' \quad \Sigma, \Gamma, \Gamma', x : A \vdash N \overset{CT}{\Rightarrow} N' : B ; \Gamma''
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash \text{let } x = E ; N \overset{CT}{\Rightarrow} r \leftarrow S' ; \text{let } (x, y) = r ; N' : B ; \Gamma', \Gamma''
\]

\[
\Sigma, \Gamma \vdash \bullet : A \vdash E \overset{CT}{\Rightarrow} S' : \exists^{\forall} X.A ; \Gamma' \quad \Sigma, \Gamma \vdash N \overset{CT}{\Rightarrow} N' : B ; \Gamma''
\]

\[
\begin{aligned}
\Sigma, \Gamma \vdash \bullet : A \vdash \text{unpack } (X, x) = E ; N \overset{CT}{\Rightarrow} & \left( \begin{array}{l}
\text{r} \leftarrow S' ; \\
\text{unpack } (X, f) = r ; \\
\text{newcase}_X c_X ; \\
\text{x} \leftarrow \text{(force } f \text{) cy}_X ; \\
N'
\end{array} \right) : B ; \Gamma', \Gamma''
\end{aligned}
\]

Fig. 34. PolyC$^v$ evaluation context translation relation

**Lemma F.12** (Translation context plug). If $\Sigma_1 \vdash M_1 \overset{CT}{\Rightarrow} M'_1 : A$ and $\Sigma_2 \vdash \bullet : A \vdash E \overset{CT}{\Rightarrow} S : B$, then $\Sigma_1 \cdot \Sigma_2 \vdash E[M_1] \overset{CT}{\Rightarrow} S[M'_1] : B$.

**Proof.** By induction on the derivation of $E \overset{CT}{\Rightarrow} S$. \hfill \square

**Lemma F.13** (Translation stack decomposition). If $\Sigma_1 \vdash M \overset{CT}{\Rightarrow} S[M'_1] : A$ then $M = E[M_1]$ for some $E$ and $M_1$ such that $E \overset{CT}{\Rightarrow} S$ and $M_1 \overset{CT}{\Rightarrow} M'_1$.

**Proof.** By induction on the derivation of $M \overset{CT}{\Rightarrow} S[M'_1]$. \hfill \square

**Lemma F.14** (Bind reduction preserves translation). If $\Sigma_1 \vdash M \overset{CT}{\Rightarrow} M'_1 : A$ and $M'_1[\gamma_p] \mapsto_b M'_2[\gamma_p]$ then $\Sigma_1 \cdot \Sigma_2 \vdash M \overset{CT}{\Rightarrow} M'_2 : A$.

**Proof.** By the definition of $\mapsto_b$, we have that $M'_1 = S[x \leftarrow \text{return } V'; M'_2]$ and $M'_3 = S[M'_2[V'/x]]$ for some $S, V', M'_2$. By Lemma F.13 (stack decomposition), we have $M = E[M_1], E \overset{CT}{\Rightarrow} S$ and $M_3 \overset{CT}{\Rightarrow} x \leftarrow \text{return } V'; M'_3$. We then apply the bind reduction rule, so we have $M_3 \overset{CT}{\Rightarrow} M'_3[V'/x]$ and conclude by Lemma F.12 (translation context plug). \hfill \square

We would like all PolyC$^v$ terms to make progress whenever their corresponding CBPV$_{OSum}$ translation evaluates. However, this is not always the case since some PolyC$^v$ terms step to terms...
\[\Sigma;\Gamma \vdash A' + E \rightarrow^\text{CT} S' : \forall^\text{X} B;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : A' \vdash E[X \equiv A] \rightarrow^\text{CT} S \leftrightarrow S' ; (\text{force } f)[A] \ e_X : B;\Gamma', X \equiv A\]

\[\Sigma;\Gamma \vdash A' + E \rightarrow^\text{CT} S' : \forall^\text{X} B;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : A' + E[\sigma \equiv A] \rightarrow^\text{CT} f \leftrightarrow S' ; (\text{force } f)[A] \ \sigma : B[\sigma/X];\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : A' \vdash E \rightarrow^\text{CT} S' : A \rightarrow B;\Gamma' \quad \Sigma;\Gamma, \Gamma' \vdash N \rightarrow^\text{CT} N' : A;\Gamma''\]

\[\Sigma;\Gamma \mid \bullet : A' + E N \rightarrow^\text{CT} f \leftrightarrow S'; a \leftarrow N' ; (\text{force } f) a : B;\Gamma', \Gamma''\]

\[\Sigma;\Gamma \vdash V \rightarrow^\text{CT} \text{ret } V' : A \rightarrow B;\Gamma' \quad \Sigma;\Gamma, \Gamma' \mid \bullet : A' + E \rightarrow^\text{CT} S' : A;\Gamma''\]

\[\Sigma;\Gamma \mid \bullet : A' + V E \rightarrow^\text{CT} a \leftrightarrow S' ; (\text{force } V') a : B;\Gamma', \Gamma''\]

\[\Sigma;\Gamma \mid \bullet : A' + \text{seal}_X E \rightarrow^\text{CT} S' : X;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : A' \vdash \text{unseal}_X E \rightarrow^\text{CT} S' : A;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : A' \vdash \text{inj}_E \rightarrow^\text{CT} E \rightarrow^\text{CT} \text{inj}_{\text{case}(G)} \rightarrow^\text{CT} X;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : B \vdash E \rightarrow^\text{CT} S' : A;\Gamma' \quad \Sigma;\Gamma \vdash A;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : B \vdash (A;\Gamma) \rightarrow^\text{CT} [\Sigma;\Gamma, \Gamma' \vdash A] \rightarrow^\text{CT} [S'] \rightarrow^\text{CT} A;\Gamma'\]

\[\Sigma;\Gamma \mid \bullet : B \vdash (\text{is}(G)) \rightarrow^\text{CT} (r \leftarrow S' ; \text{match case}(G) \text{ with } r \{\text{inj } y, \text{ret true } | \text{ ret false} \}) ; B;\Gamma'\]

Fig. 35. PolyC\textsuperscript{v} calculus evaluation context translation relation (Continued)

with identical translations. However, the number of such steps is limited by the syntactic size of the term, defined below.

**Definition F.15 (Type/Term size).**

\[
\begin{align*}
|\text{let } x = M; N| &= 1 + |N[M/x]| \\
|\text{tag}_G(A;\Gamma)| &= 2 + |A;\Gamma| \\
|\text{pack}^\textsuperscript{v}(X \equiv A', M)| &= 2 + |M| \\
|C(A;\Gamma)| &= 1 + \Sigma_{M \in \tilde{M}} |M| + \Sigma_{A;\Gamma} |A;\Gamma| \quad \text{otherwise}
\end{align*}
\]

where \(C\) denotes any syntactic term or precision judgment constructor.

**Theorem F.16 (Simulation).** If \(\Sigma;\cdot \vdash M_1 : A;\cdot \) and \(\Sigma;\cdot \vdash M_1 \rightarrow^\text{CT} M'_1 : A;\cdot \) and \(\Sigma \rightarrow M_1 \rightarrow \Sigma' \rightarrow M_2, \) then \(\Sigma;\cdot \vdash M_2 \rightarrow^\text{CT} M'_2 : A;\cdot \) and either \(\Sigma_p;[\Sigma] \rightarrow M'_1[y_p] \leftarrow^\text{+} \Sigma_p;[\Sigma'] \rightarrow M'_2[y_p] \) or \(\Sigma_p;[\Sigma] \rightarrow M'_1[y_p] \leftarrow^\text{+} \Sigma_p;[\Sigma'] \rightarrow M'_2[y_p] \) and \(|M_1| > |M_2|\) for some \(M'_2.\)
Proof. Note that, by unique decomposition, we have some context $E$ and redex $M_3$ such that
$M_1 = E[M_3]$ and either $M_3 = U_B$ or $M_1 \leadsto M_4$ for some $M_4$. By Lemma F.11 (translation context
decomposition), we have some $S, M'_3$ such that $S[M'_3][y_p] \leadsto^* M'_3[y_p]$ where $E \leadsto^C T S$ and $M_3 \leadsto^C T M'_3$. We proceed by cases on $\Sigma \vdash M_1 \leadsto \Sigma \vdash M_2$. In each case, we first use Lemma F.9 (canonical
forms of the translation) to determine the form of $M'_3$.

- Error:

\[
E[U_B] \leadsto U_{type(\Sigma; E[U_B])} \text{ where } E \neq [
\]

We have $M'_3 = U$ and so $S[M'_3][y_p] \leadsto U$. Since $\Sigma; \vdash \cup_{type(\Sigma; M_1)} \leadsto^C T U : type(\Sigma; \vdash M_1)$, we have what we were required to show.

- Instantiation:

\[
E[(\Lambda X. N_1) \{\sigma \equiv B\}] \leadsto E[N_1[\sigma/X]]
\]

Let $B' = [\Sigma; \vdash B]$. We have

\[
(f \leftarrow \text{ret thunk } \Lambda X. \lambda (c_X : \text{Case } X). N'_1; (\text{force } f)[B'] \sigma)[y_p] \leadsto^* M'_3[y_p]
\]

for some $N'_1$ such that $N_1 \leadsto^C T N'_1$. Then, by Lemma F.8 (bind confluence), we have $M'_3[y_p] \leadsto^* ((\text{force thunk } \Lambda X. \lambda (c_X : \text{Case } X). N'_1)[B'] \sigma)[y_p]$ and so by the dynamic semantics, we have the following:

\[
\begin{align*}
M'_3[y_p] & \leadsto^* S[(\text{force thunk } \Lambda X. \lambda (c_X : \text{Case } X). N'_1)[B'] \sigma][y_p] \\
& \leadsto S[(\Lambda X. \lambda (c_X : \text{Case } X). N'_1)[B'] \sigma][y_p] \\
& \leadsto S[(\lambda (c_X : \text{Case } B'). N'_1[B'/X]) \sigma][y_p] \\
& \leadsto S[N'_1[B'/X][\sigma/c_X]][y_p]
\end{align*}
\]

Furthermore, by Lemma F.3, since $\sigma : B \in \Sigma$ and $\Sigma; X \vdash N_1 \leadsto^C T N'_1 : A_1$, we have that $N_1[\sigma/X] \leadsto^C T N'_1[B'/X][\sigma/c_X]$. Then, by Lemma F.12 (translation context plug), we have what we were required to show.

- Hiding:

\[
\Sigma \vdash E[\text{hide } X \equiv B; N] \leadsto \Sigma, \sigma : \text{Case } B \leadsto E[N[\sigma/X]]
\]

We have $M'_3 = \text{newcase}_B c_X; N'$ for some $N'$ such that $N \leadsto^C T N'$. We then have the following by the operational semantics:

\[
\begin{align*}
[\Sigma] & \leadsto S[\text{newcase}_B c_X; N'][y_p] \\
& \leadsto [\Sigma], \sigma : \text{Case } B \leadsto S[N'[\sigma/c_X]][y_p]
\end{align*}
\]

Then by Lemma F.4, since $\Sigma; \vdash N \leadsto^C T N' : A, X \equiv B$, we have $N[\sigma/X] \leadsto^C T N'[\sigma/c_X]$, so we conclude by Lemma F.12 (translation context plug).

- Pack:

\[
E[\text{pack }^\chi(X \equiv A_1, N_1)] \leadsto E[\text{pack }^\chi(X \equiv A_1, [], N_1)]
\]

Note that $\text{pack }^\chi(X \equiv A_1, N_1) \leadsto^C T M'_3$ iff $\text{pack }^\chi(X \equiv A_1, [], N_1) \leadsto^C T M'_3$, so by Lemma F.12 (translation context plug), we may choose $M'_3 = M'_1$. Then, it suffices to show that $|M_1| > |M_2|$. This holds since $|\text{pack }^\chi(X \equiv A_1, N_1)| = 2 + |N_1| > 1 + |N_1| = |\text{pack }^\chi(X \equiv A_1, [], N_1)|$.

- Unpack:

\[
\Sigma \leadsto E[\text{unpack } (X, x) = \text{pack }^\chi(X \equiv A_1, [A^\equiv \uparrow \ldots, N_1]; N_2)] \\
\leadsto \Sigma, \sigma : \text{Case } A_1 \leadsto E[\text{let } x = \langle A^\equiv[\sigma/X] \uparrow \ldots N_1[\sigma/X]; N_2[\sigma/X] \rangle]
\]
Let $A' = \Sigma; \vdash \exists X.B$ and $A'_1 = \Sigma; \vdash A_1$. We have

\[
\begin{align*}
r &\leftarrow \text{ret pack}(A'_1, \text{thunk } \lambda(c_X : \text{Case } A'_1).N'_{cst}[y_p]) \text{ as } A'; \\
\text{unpack } (X, f) &\leftarrow r; \\
\text{newcase}_X c_X; \\
x &\leftarrow (\text{force } f) c_X; \\
N'_2[y_p] &\mapsto_b M'_2[y_p]
\end{align*}
\]

where $N'_{cst} = \left[\Sigma; X \equiv A_1 + A^\mathbb{E}\right] \uparrow \left[\text{force (thunk } \lambda(c_X : \text{Case } A'.(... N'_1 ...)) \text{ c}_X\right]$ and $N_i \sim^{CT} N'_i$. We then have the following by Lemma F.8 (bind confluence) and the operational semantics:

\[
\begin{align*}
&\mapsto^*_{\text{b}} \\
&M'_2[y_p] \\
&\text{unpack } (X, f) = \text{pack}(A'_1, \text{thunk } \lambda(c_X : \text{Case } A'_1).N'_{cst}[y_p]) \text{ as } A'; \\
&\text{newcase}_X c_X; \\
x &\leftarrow (\text{force } f) c_X; \\
N'_2[y_p] &\mapsto_b [\Sigma]\mapsto \text{newcase}_A' c_X; \\
x &\leftarrow (\text{force thunk } \lambda(c_X : \text{Case } A'_1).N'_{cst}[y_p]) c_X; \\
N'_2[y_p][A'/X] &\mapsto_{\Sigma; \sigma : A'_1} \left[\Sigma; \sigma : A'_1\right] x \leftarrow N'_{cst}[y_p][\sigma/c_X]; \\
N'_2[y_p][A'/X][\sigma/c_X] &\mapsto_{\Sigma; \sigma : A'_1} \left[\Sigma; \sigma : A'_1\right] x \leftarrow [\Sigma; X \equiv A_1 + A^\mathbb{E}] \uparrow [\Sigma; X \equiv A_1 + A^\mathbb{E}] \uparrow [\Sigma; X \equiv A_1 + A^\mathbb{E}] \uparrow [... N'_1 ...] \sigma/c_X; N'_2[A'/X][\sigma/c_X].
\end{align*}
\]

Note that, since $\Sigma; X \equiv A_1 + N_1 \sim^{CT} N'_1$, we have $\Sigma; X \equiv A_1 + A^\mathbb{E}\uparrow [... N'_1 ...]$ by the definition of the translation relation for casts. Furthermore, by Lemma F.4, since $\sigma : A_1 \in \Sigma$, we have $\left(\Sigma; X \equiv A_1 + A^\mathbb{E}\right) \uparrow [... N'_1 ...] \sigma/c_X$ and by Lemma F.3, since $\sigma : A_1 \in \Sigma$ and $\Sigma; X \in N_2 \sim^{CT} N'_2 : A_2$, we have $N_2[\sigma/X] \sim^{CT} N'_2[A'/X][\sigma/c_X]$. Thus, we have

\[
\begin{align*}
&\text{let } x = \left(\Sigma; X \equiv A_1 + A^\mathbb{E}\right) \uparrow [... N'_1 ...][\sigma/c_X]; N'_2[A'/X][\sigma/c_X] \\
&\text{so we conclude by Lemma F.12 (translation context plug).}
\end{align*}
\]

- function application

\[
E[(\lambda x : A.N_1) V ] \mapsto E[N_1[V/x]]
\]

We have $(f \leftarrow \text{ret thunk } \lambda x : [\Sigma; \cdot + A].N'_1; a \leftarrow N'_2; \text{force } f a)[y_p] \mapsto_b M'_2[y_p]$ where $N_i \sim^{CT} N'_i$. By Lemma F.10 (value translation), we have some $V'$ such that $N'_2[y_p] \mapsto^*_{\text{b}} \text{ret } V'[y_p]$ and $V \sim^{CT} \text{ret } V'$. Then, by Lemma F.8 (bind confluence), we have $M'_2[y_p] \mapsto^*_{\text{b}} \left(\text{force (thunk } \lambda x : [\Sigma; \cdot + A].N'_1 V'[y_p]. Then $M'_2[y_p] \mapsto^*_{\text{b}} N'_2[V'/x][y_p]$ and by Lemma F.5 (value substitution translation), we have $\Sigma; \cdot + N_1[V/x] \sim^{CT} N'_2[V'/x] : B$. Thus, we conclude by Lemma F.12 (translation context plug).

- If true:

\[
E[\text{if true then } N_1 \text{ else } N_2] \mapsto E[N_1]
\]
We have \((r \leftarrow \text{ret \ true}; \text{if \ true \ then \ N_1' \ else \ N_2')[y_p]) \mapsto_b^* M_1'[y_p]\), where \(N_1 \sim^{CT} N_1'\). We then have the following by Lemma F.7 (bind normalization) and the operational semantics:

\[
\begin{align*}
M_1'[y_p] \\
\mapsto_b^* S[\text{if \ true \ then \ N_1' \ else \ N_2'][y_p] \\
\mapsto S[N_1'][y_p]
\end{align*}
\]

We then conclude by Lemma F.12 (translation context plug) since \(N_1 \sim^{CT} N_1'.\)

- If false:

\[
\begin{align*}
\text{if \ false \ then \ N_1 \ else \ N_2 \mapsto M_2
\end{align*}
\]

We have \((r \leftarrow \text{ret \ false}; \text{if \ false \ then \ N_1' \ else \ N_2')[y_p]) \mapsto_b^* M_2'[y_p]\), where \(N_1 \sim^{CT} N_1'\). We then have the following by Lemma F.7 (bind normalization) and the operational semantics:

\[
\begin{align*}
M_1'[y_p] \\
\mapsto_b^* S[\text{if \ false \ then \ N_1' \ else \ N_2'][y_p] \\
\mapsto S[N_2'][y_p]
\end{align*}
\]

We then conclude by Lemma F.12 (translation context plug) since \(N_2 \sim^{CT} N_2'.\)

- Pair elimination:

\[
\begin{align*}
\text{let} \ (x, y) = (V_1, V_2); N \mapsto N[V_1/x][V_2/y]
\end{align*}
\]

We have \((r \leftarrow M''; \text{let} \ (x, y) = r; N')[y_p] \mapsto_b^* M_3'[y_p]\) where \((V_1, V_2) \sim^{CT} M''\) and \(N \sim^{CT} N'.\) Thus, by Lemma F.10 (value translation), we further have \(M''[y_p] \mapsto_b^* \text{ret} V'[y_p]\) and \((V_1, V_2) \sim^{CT} \text{ret} V'\) for some \(V'.\) Note that by the type of \(V',\) we have \(V' = (V_1', V_2')\) for some \(V_1', V_2'.\)

We then have the following by Lemma F.8 (bind confluence) and the operational semantics:

\[
\begin{align*}
M_1'[y_p] \\
\mapsto_b^* S[\text{let} \ (x, y) = (V_1', V_2'); N'][y_p] \\
\mapsto S[N'[V_1'/x][V_2'/y]][y_p]
\end{align*}
\]

By Lemma F.5, since \(V_1 \sim^{CT} \text{ret} V_1',\) we have \(N[V_1/x][V_2/y] \sim^{CT} N'[V_1'/x][V_2'/y].\) We then conclude by Lemma F.12 (translation context plug).

- Let:

\[
\begin{align*}
\text{let} \ x = V; N_2 \mapsto N_2[V/x]
\end{align*}
\]

We have \((x \leftarrow N_1'; N_2')[y_p] \mapsto_b^* M_3'[y_p]\) where \(V \sim^{CT} N_1'\) and \(N_2 \sim^{CT} N_2'.\) Then, by Lemma F.10 (value translation reduction), we have \(N_1'[y_p] \mapsto_b^* \text{ret} V'[y_p]\) for some \(V'\) such that \(V \sim^{CT} \text{ret} V'.\) Thus, we have the following reduction:

\[
\begin{align*}
S[x \leftarrow N_1'; N_2'][y_p] \mapsto_b^* S[N_2'[V'/x]][y_p]
\end{align*}
\]

By Lemma F.7 (bind normalization) there exists a unique \(M_4'\) such that

\[
S[x \leftarrow N_1'; N_2'][y_p] \mapsto_b^* M_4'[y_p]
\]

and \(M_3'[y_p]\) does not take a bind reduction. Then, by Lemma F.8 (bind reduction confluence), we have \(M_1'[y_p] \mapsto_b^* M_4'[y_p]\) and \(S[N_2'[V'/x]][y_p] \mapsto_b^* M_4'[y_p].\)

Note that, by Lemma F.5 (value substitution translation), we have \(N_2[V/x] \sim^{CT} N_2'[V'/x]\) and thus by Lemma F.12 (translation context plug), we have \(E[N_2[V/x]] \sim^{CT} S[N_2'[V'/x]].\)

Then, by Lemma F.14 (bind reduction preserves translation), we have \(E[N_2[V/x]] \sim^{CT} M_4'.\)

It finally suffices to show that \(|M_1| > |M_2|\), which we have since by definition \(|M_3| = 1 + |N_2[V/x]|\).
• Unseal:
  \[ E[\text{unseal}_\sigma \text{seal}_\sigma V] \rightsquigarrow E[V] \]
  We have \( N'[y]\) \( \mapsto^*_b M'_1[y] \) where \( V \rightsquigarrow^C T \ \ N' \) and so \( S[N'][y] \mapsto^*_b S[M'_1][y] \). By Lemma F.7 (bind normalization) there exists a unique \( M'_2 \) such that \( S[N'][y] \mapsto^*_b M'_2[y] \) and \( M'_2[y] \) does not take a bind reduction. Then, by Lemma F.8 (bind confluence), we have \( S[M'_2][y] \mapsto^*_b M'_2[y] \). Since, by Lemmas F.12 (translation context plug) and F.14 (bind reduction preserves translation), we have \( E[V] \rightsquigarrow^C T M'_4 \), it suffices to show that \( |M'_3| > |M'_2| \).
  This holds since by definition \( |M'_3| = 1 + 1 + |V| \).

• Atomic cast:
  \[ E[\langle A \mapsto \downarrow V \rangle] \mapsto E[V] \text{ where } A \in \{\mathbb{E}, a, ?\} \]
  The reasoning for this case is analogous to the prior case.

• Pair cast:
  \[ E[\langle A_1 \mapsto A_2 \rangle \downarrow (V_1, V_2)] \mapsto E[(\langle A_1 \mapsto \downarrow V_1 \rangle, \langle A_2 \mapsto \downarrow V_2 \rangle)] \]
  We have \( [\Sigma; \vdash A_1 \mapsto A_2] \downarrow [x_1 \gets N'_1; x_2 \gets N'_2; \text{ret} (x_1, x_2)][y] \mapsto^*_b M'_1[y] \) where \( V \rightsquigarrow^C T N'_1 \). Then, by Lemma F.10 (value translation reduction), we have \( N'_1[y] \mapsto^*_b \text{ret} V'_1[y] \) for some \( V'_1, V'_2 \) such that \( V'_1 \rightsquigarrow^C T \text{ret} V'_2 \). Thus, we have the following reductions:

  \[ \begin{align*}
  &\mapsto^*_b [\Sigma; \vdash A_1 \mapsto A_2] \downarrow [\text{ret} (V'_1, V'_2)][y] \\
  &\mapsto^*_b \text{let} (x_1, x_2) = (V'_1, V'_2)[y]; \\
  &\quad x_1' \gets [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} x_1][y]; \\
  &\quad x_2' \gets [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} x_2][y]; \\
  &\quad \text{ret} (x_1', x_2')
  \end{align*} \]

Let \( M'_4[y] \) be the latter term. By Lemma F.8 (bind confluence), we have \( M'_2[y] \mapsto^*_b M'_4[y] \).
  Thus, we have this last reduction:

  \[ \begin{align*}
  &\mapsto^*_b S[M'_2][y] \\
  &\mapsto^*_b S[M'_4][y] \\
  &\mapsto^*_b S[x_1' \gets [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} V'_1]; \\
  &\quad x_2' \gets [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} V'_2]; \\
  &\quad \text{ret} (x_1', x_2')[y]
  \end{align*} \]

And, since \( V'_1 \rightsquigarrow^C T \text{ret} V'_2 \), we have that this result is related to \( M_2 \) by Lemma F.12 (translation context plug), so we conclude.

• Function cast:
  \[ E[\langle (A_1 \to A_2) \downarrow V_1 \rangle] \mapsto E[\langle A_1 \mapsto A_2 \rangle \downarrow (V_1, \langle A_1 \mapsto \downarrow V_2 \rangle)] \]
  We have \( (f \leftarrow [\Sigma; \vdash A_1 \mapsto A_2] \downarrow [N'_1]; a \leftarrow N'_2; \text{force} f a)[y] \mapsto^*_b M'_1[y] \) where \( V \rightsquigarrow^C T N'_1 \). Then, by Lemma F.10 (value translation reduction), we have \( N'_1[y] \mapsto^*_b \text{ret} V'_1[y] \) for some \( V'_1, V'_2 \) such that \( V'_1 \rightsquigarrow^C T \text{ret} V'_2 \). Thus, we have the following reductions:

  \[ \begin{align*}
  &\mapsto^*_b (f \leftarrow [\Sigma; \vdash A_1 \mapsto A_2] \downarrow [N'_1]; a \leftarrow N'_2; \text{force} f a)[y] \\
  &\mapsto^*_b (\text{force} (\text{thunk}Ay):[\Sigma; \Gamma \vdash A_1 \mapsto A_2]) \\
  &\leftarrow [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} y]; [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{force} V'_1 a] V'_1[y] \\
  &\mapsto^*_b (a \leftarrow [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{ret} V'_2]; [\Sigma; \Gamma \vdash A_1 \mapsto A_2] \downarrow [\text{force} V'_1 a])[y]
  \end{align*} \]

Let \( M'_4[y] \) be the latter term. Then we have \( \langle A_1 \mapsto \downarrow V_1 \rangle \mapsto^C T V'_2 \rightsquigarrow^C T M'_4 \) since \( V \rightsquigarrow^C T \text{ret} V'_2 \), applying the bind reduction rule to eliminate the first bind. We then conclude by Lemma F.12 (translation context plug).
• Universal cast:

\[ E[(\forall x. A \overset{\sigma}{\to} N) \{\sigma \equiv A_1]\} \to E[(A \overset{\sigma}{\to} [X]) \downarrow \chi N[\sigma/X]] \]

Let \( A_1 = \lceil A \bigcup A \rceil \). We have

\( x \leftarrow \lceil A \bigcup A \rceil \downarrow [\text{ret thunk} \chi \overset{\sigma}{\to} N \{\sigma \equiv A_1\}] \)

where \( N \to CT N' \). Let \( V' = \text{thunk} \chi \overset{\sigma}{\to} N' \). We then have the following reduction:

\( \to_b (\text{force \ thunk} \chi \overset{\sigma}{\to} N'[\sigma/X] [\text{case} V]) \)

Then, by Lemma F.8 (bind confluence) and the operational semantics, we have

\( S[M'_1]|\gamma_p \to S[\text{force \ thunk} \chi \overset{\sigma}{\to} N'[\sigma/X] [\text{case} V']] |\gamma_p \)

Let \( M'_2 |\gamma_p \) be the latter term. Then, by Lemma F.12 (translation context plug), and since \( \langle A \overset{\sigma}{\to} [X] \rangle |\gamma_p \to CT \gamma_p \langle A' \overset{\sigma}{\to} [X] \rangle |\gamma_p \), which we have by Lemma F.3 (type variable translation substitution) because \( \chi \overset{\sigma}{\to} N |\gamma_p \to CT N' : B \) and \( \sigma : A_1 \in \Sigma \), we may conclude.

• tag upcast:

\[ E[(\text{tag}_G (A \overset{\sigma}{\to} [X])) \downarrow V] \to E[\text{inj}_G \langle A \overset{\sigma}{\to} [X] \rangle \downarrow V] \]

We have \( \lceil A \bigcup A \rceil \downarrow [\text{ret} V'] |\gamma_p \to_b M'_1 |\gamma_p \) where \( V \to CT N' \). Then, by Lemma F.10 (value translation), we have some \( V' \) such that \( N'[\gamma_p] \to_b \text{ret} V'[\gamma_p] \) and \( V \to CT \text{ret} V' \). Thus, we have the following reduction:

\[ \lceil A \bigcup A \rceil \downarrow [\text{ret} V'] |\gamma_p \]

Let the latter term be \( M'_2 |\gamma_p \). By Lemma F.7 (bind normalization) there must exist some \( M'_3 |\gamma_p \) such that \( S[M'_1]|\gamma_p \to_b M'_3 |\gamma_p \) and \( M'_2 |\gamma_p \) does not take a bind reduction. Then, by Lemma F.8 (bind confluence), we have that \( M'_3 |\gamma_p \to_b M'_3 |\gamma_p \). Note that \( E[\text{inj}_G \langle A \overset{\sigma}{\to} [X] \rangle \downarrow V] \to CT S[M'_3] \) by Lemma F.12 (translation context plug). Then, by Lemma F.14 (bind reduction preserves translation), we have \( E[\text{inj}_G \langle A \overset{\sigma}{\to} [X] \rangle \downarrow V] \to CT M'_3 \). Thus, it suffices to show that \( |\text{tag}_G (A \overset{\sigma}{\to} [X]) \downarrow V| > |\text{inj}_G \langle A \overset{\sigma}{\to} [X] \rangle \downarrow V| = 1 + 2 + |A | + |V| \).

• tag downcast:

\[ E[(\text{tag}_G (A \overset{\sigma}{\to} [X])) \downarrow V] \to E[\langle A \overset{\sigma}{\to} [X] \rangle \downarrow V] \]

We have \( \lceil A \bigcup A \rceil \downarrow [r \leftarrow N'; \text{ret} \text{inj}_G(r)] |\gamma_p \to_b M'_3 |\gamma_p \) where \( V \to CT N' \). Then, by Lemma F.10 (value translation), we have some \( V' \) such that \( N'[\gamma_p] \to_b \text{ret} V'[\gamma_p] \) and \( V \to CT \text{ret} V' \). Thus, we have the following reduction:

\[ \lceil A \bigcup A \rceil \downarrow [r \leftarrow N'; \text{ret} \text{inj}_G(r)] |\gamma_p \]

\[ \to_b \text{match case}(G) \text{with} (\text{inj}_G V')\{\text{inj} y.\lceil A \bigcup A \rceil \downarrow \text{ret} y| \cup \} |\gamma_p \]
Let \( M'_4[\gamma_p] \) be the latter term. Then, by Lemma F.8 (bind confluence) and the operational semantics, we have the following reductions:

\[
M'_4[\gamma_p] \\
\mapsto^*_b S[\text{match case}(G) \text{ with } (\text{inj}_{\text{case}(G)} V')\{\text{inj } y.\Sigma; \Gamma \vdash A^\Sigma \downarrow \text{ret } y \mid \emptyset\}] [\gamma_p] \\
\mapsto S[\Sigma; \Gamma \vdash A^\Sigma \downarrow \text{ret } V'] [\gamma_p]
\]

Then, since \( V \leadsto^{CT} \text{ret } V' \), we have \( \langle A^\Sigma \rangle \downarrow V \leadsto^{CT} \Sigma; \Gamma \vdash A^\Sigma \downarrow \text{ret } V'[\gamma_p] \) and we may conclude by Lemma F.12 (translation context plug).

- **Tag downcast error:**

\[
E[(\text{tag}_G (A^\Sigma)) \downarrow \text{inj}_H V] \mapsto E[\mathcal{U}_B] \text{ where } G \neq H
\]

We have \( \Sigma; \Delta \vdash \text{tag}_G (A^\Sigma) \downarrow [r \leftarrow N'; \text{ret } \text{inj}_{\text{case}(H)} r] [\gamma_p] \mapsto^*_b M'_3[\gamma_p] \) where \( V \leadsto^{CT} N' \).

Then, by Lemma F.10 (value translation), we have some \( V' \) such that \( N'[\gamma_p] \mapsto^*_b \text{ret } V'[\gamma_p] \) and \( V \leadsto^{CT} \text{ret } V' \). Thus, we have the following reduction:

\[
\Sigma; \Delta \vdash \text{tag}_G (A^\Sigma) \downarrow [r \leftarrow N'; \text{ret } \text{inj}_{\text{case}(H)} r] [\gamma_p] \\
\mapsto^*_b [\Sigma; \Delta \vdash \text{tag}_G (A^\Sigma) \downarrow [\text{ret } \text{inj}_{\text{case}(H)} V'] [\gamma_p] \\
\mapsto^*_b \text{match case}(G) \text{ with } (\text{inj}_{\text{case}(H)} V')\{\text{inj } y.\Sigma; \Gamma \vdash A^\Sigma \downarrow \text{ret } y \mid \emptyset\}] [\gamma_p]
\]

Let \( M'_3[\gamma_p] \) be the latter term. Then, by Lemma F.8 (bind confluence) and the operational semantics, we have the following reductions:

\[
M'_3[\gamma_p] \\
\mapsto^*_b S[\text{match case}(G) \text{ with } (\text{inj}_{\text{case}(H)} V')\{\text{inj } y.\Sigma; \Gamma \vdash A^\Sigma \downarrow \text{ret } y \mid \emptyset\}] [\gamma_p] \\
\mapsto S[\mathcal{U}] [\gamma_p]
\]

Then, since \( \mathcal{U}_B \leadsto^{CT} \mathcal{U} \), we may conclude by Lemma F.12 (translation context plug).

- **Existential upcast:**

\[
E[\langle \exists^X. A^\Sigma \rangle ] \downarrow \text{pack}^\Sigma (X \equiv A, [A^\Sigma_1 \downarrow \ldots, N_i]) \mapsto E[\text{pack}^\Sigma (X \equiv A, [A^\Sigma_1 \downarrow, A^\Sigma_2 \downarrow \ldots], N_i)]
\]

For some \( N_i' \) such that \( N_i \leadsto^{CT} N_i' \), we have

\[
\Sigma; \cdot \vdash \exists^X. A^\Sigma_1 \downarrow \text{ret pack}(A', \text{thunk } \lambda c_X : \text{Case } A'.N_i' c_X)\] as \( \Sigma; \cdot \vdash \exists^X. A^\Sigma_1 [\gamma_p] \mapsto^*_b M'_3[\gamma_p]
\]

where \( N_i' = \Sigma; X \equiv A' + A^\Sigma_2 \downarrow \downarrow \text{[force (thunk } \lambda c_X : \text{Case } A'.(\ldots N_i' c_X)) \downarrow \downarrow \text{]} \) and \( A' = \Sigma; \cdot \vdash A \).

By Lemma F.8 (bind confluence), we then have the following reductions:

\[
M'_3[\gamma_p] \\
\mapsto^*_b \text{unpack } (Y, f) = \text{pack} (A', \text{thunk } \lambda c_X : \text{Case } A'.N_i' c_X)\] as \( \Sigma; \cdot \vdash \exists^X. A^\Sigma_1 [\gamma_p];
\]

\[
\text{ret pack}(Y, \text{thunk } \lambda c_X : \text{Case } Y; \Sigma; Y \equiv Y + A^\Sigma_2 \downarrow \downarrow \text{[force } f c_X\downarrow \downarrow \text{]}\) as \( \Sigma; \cdot \vdash \exists^X. A^\Sigma_1 [\gamma_p]\]

\[
\mapsto^*_b \text{ret pack}(A', \text{thunk } \lambda c_X : \text{Case } A'.) \]

\[
\Sigma; X \equiv A' + A^\Sigma_1 \downarrow \downarrow \text{[force } (\text{thunk } \lambda c_X : \text{Case } A'.N_i' c_X)\downarrow \downarrow \text{]}\) as \( \Sigma; \cdot \vdash \exists^X. A^\Sigma_1 [\gamma_p]\]

Let the above term be \( M'_4[\gamma_p] \). We know that \( \text{pack}^\Sigma (X \equiv A', [A^\Sigma_1 \downarrow, A^\Sigma_2 \downarrow \ldots], N_i) \leadsto^{CT} M'_4 \) since \( N_i \leadsto^{CT} N_i' \), so we conclude by Lemma F.12 (translation context plug).

- **Existential downcast:**

\[
E[\langle \exists^X. A^\Sigma_1 \rangle ] \downarrow \text{pack}^\Sigma (X \equiv A, [A^\Sigma_2 \downarrow \ldots], N_i) \mapsto E[\text{pack}^\Sigma (X \equiv A, [A^\Sigma_1 \downarrow, A^\Sigma_2 \downarrow \ldots], N_i)]
\]

This case proceeds analogously, replacing \( A^\Sigma_1 \) with \( A^\Sigma_2 \) and vice versa.

- **Tag check true:**

\[
E[\text{is}(G) \downarrow \text{inj}_G V] \mapsto E[\text{true}]
\]
We have
\[(r \leftarrow x \leftarrow N'; \text{ret inj}_{\text{case}(G)} x; \text{match case}(G) \text{ with } r\{\text{inj } y.\text{ret true } | \text{ret false}\})[y_p])\]
\[\mapsto^*_b M_3'[y_p]\]
where \(V \sim_{CT} N'.\) Then, by Lemma F.10 (value translation), we have some \(V'\) such that
\(N'[y_p] \mapsto^*_b \text{ret } V'[y_p]\) and \(V \sim_{CT} \text{ret } V'.\) Thus, we have the following reduction:
\[(r \leftarrow x \leftarrow N'; \text{ret inj}_{\text{case}(G)} x; \text{match case}(G) \text{ with } r\{\text{inj } y.\text{ret true } | \text{ret false}\})[y_p])\]
\[\mapsto^*_b (r \rightarrow \text{ret inj}_{\text{case}(G)} V'; \text{match case}(G) \text{ with } r\{\text{inj } y.\text{ret true } | \text{ret false}\})[y_p])\]
\[\mapsto^*_b (\text{match case}(G) \text{ with } (\text{inj}_{\text{case}(G)} V')\{\text{inj } y.\text{ret true } | \text{ret false}\})[y_p])\]

Then, by Lemma F.8 (bind confluence), we have
\[M_3' \mapsto^*_b (\text{match case}(G) \text{ with } (\text{inj}_{\text{case}(G)} V')\{\text{inj } y.\text{ret true } | \text{ret false}\})[y_p])\]
\[\mapsto \text{ret true}\]

Since \(\text{true} \sim_{CT} \text{ret true}\), we conclude by Lemma F.12 (translation context plug).

- Tag check false:
\[E[\text{is}(G) \? \text{inj}_H V] \mapsto E[\text{false}] \text{ where } G \neq H\]

This case is analogous to the former except that since \(G \neq H\), the translation produces false and since false \(\sim_{CT} \text{false}\), we conclude.

**Lemma F.17 (Multi-step simulation).** If \(\Sigma; \vdash M \sim_{CT} M' : A \) and \(\Sigma \triangleright M \mapsto^* \Sigma_1 \triangleright M_1\), then \(M_1 \sim_{CT} M_1'\) and \(\Sigma_p, [\Sigma] \triangleright M'[y_p] \mapsto^* \Sigma_p, [\Sigma_1] \triangleright M_1'[y_p]\) for some \(M_1'\).

**Proof.** We proceed by induction on the number of steps \(n\) in \(\Sigma \triangleright M \mapsto^* \Sigma_1 \triangleright M_1\).

- \(n = 0\): Then \(M_1 = M\), so we have \(M_1 \sim_{CT} M'\) and \(M'[y_p] \mapsto^* M_1'[y_p]\) by reflexivity.
- \(n = n' + 1\): Then there exists some \(\Sigma_2, M_2\) such that \(\Sigma \triangleright M \mapsto^* \Sigma_2 \triangleright M_2 \mapsto \Sigma_1 \triangleright M_1\). By the inductive hypothesis for \(n\), we then have some \(M_2'\) such that \(M_2 \sim_{CT} M_2'\) and
\[\Sigma_p, [\Sigma] \triangleright M'[y_p] \mapsto^* \Sigma_p, [\Sigma_2] \triangleright M_2'[y_p]\]

Finally, by Theorem F.16 (simulation), since \(M_2 \sim_{CT} M_2'\) and \(\Sigma_2 \triangleright M_2 \mapsto \Sigma_1 \triangleright M_1\), there exists \(M_2'\) such that \(M_1 \sim_{CT} M_2'\) and \(\Sigma_p, [\Sigma_2] \triangleright M_2'[y_p] \mapsto^* \Sigma_p, [\Sigma_1] \triangleright M_1'[y_p]\). Therefore, \(\Sigma_p, [\Sigma] \triangleright M'[y_p] \mapsto^* \Sigma_p, [\Sigma_1] \triangleright M_1'[y_p]\) as we were required to show.

**F.1 Adequacy**

Now that we have established the multi-step simulation theorem, we can prove our desired adequacy theorems that say that we can tell if a PolyC\(^v\) term terminates, errors or diverges by looking at its translation to CBPV\(_{\text{OSum}}\). They follow by our simulation theorem and progress for PolyC\(^v\).

**Corollary F.18.** If \(\Sigma; \vdash M \sim_{CT} M' : A\), and \(\Sigma_p, [\Sigma] \triangleright M'[y_p] \uparrow\), then \(\Sigma \triangleright M \uparrow\).
we have a contradiction.

Assume that $M_i = \emptyset$; By Lemma F.9 (canonical forms), $M_i = \emptyset$ and we have $M'[y_p] \rightarrow^* \emptyset$, but $M'[y_p] \nmid$, so we have a contradiction.

First, we show the full graduality/parametricity logical relation in Figure 36

$\Sigma$ and $M$ for some $M$.

By Lemma F.17 (multi-step simulation), we have some $M_i'$ such that $M_i \rightsquigarrow^C M_i'$ and $M'[y_p] \rightarrow^* M_i'[y_p]$. Then, by Lemma C.3 (progress), $M_i = \emptyset$ or $M_i = V$ for some value $V$ so we have two cases to consider:

- $M_i = \emptyset$: By Lemma F.9 (canonical forms), $M_i = \emptyset$ and we have $M'[y_p] \rightarrow^* \emptyset$, but $M'[y_p] \nmid$, so we have a contradiction.

- $M_i = V$: By Lemma F.10, we have $M'[y_p] \rightarrow^* M_i'[y_p] \rightarrow^* \text{ret } V'$ for some value $V'$, but $M'[y_p] \nmid$, so again we have a contradiction.

$\square$

**Theorem F.19.** If $\Sigma : \vdash M \rightsquigarrow^C M' : A$, and $\Sigma_p, [\Sigma] \triangleright M'[y_p] \Downarrow$, then $\Sigma \triangleright M \Downarrow$.

**Proof.** We proceed by induction first on the number of steps $M'[y_p]$ takes and then on $|M|$. Assume that $\Sigma \triangleright M \iff \Sigma_i \triangleright M_i$ since otherwise, we may conclude. By Theorem F.16 (simulation), we have some $M_i'$ such that $M_i \rightsquigarrow^C M_i'$ and either $\Sigma, [\Sigma] \triangleright M'[y_p] \rightarrow^* \Sigma_p, [\Sigma_i] \triangleright M_i'[y_p]$ or $\Sigma_p, [\Sigma] \equiv M'[y_p] \rightarrow^* \Sigma_p, [\Sigma_i] \equiv M_i'[y_p]$ and $|M| > |M_i|$. Note that since $M'[y_p] \Downarrow$, by Lemma F.1 (target semantics deterministic) we have $M_i'[y_p] \Downarrow$. We then have two cases to consider. If $M' \rightarrow^+ M_i'$, then we conclude by the inductive hypothesis for $M_i'[y_p] \Downarrow$. Otherwise, we have that $|M| > |M_i|$, so we conclude by the inductive hypothesis for $|M_i|$.

$\square$

**Lemma F.20.** If $\Sigma : \vdash M \rightsquigarrow^C M' : A$, and $\Sigma_p, [\Sigma] \triangleright M'[y_p] \Downarrow \emptyset$, then $\Sigma \triangleright M \Downarrow \emptyset_A$.

**Proof.** By Theorem ?? (target termination implies source termination), we have $\Sigma \triangleright M \Downarrow M_R$ for some $M_R$. If $M_R = \emptyset_A$, we conclude. Otherwise, $M_R = \text{ret } V$ for some $V$. We prove this case by contradiction. By Lemma F.17 (multi-step simulation), we have some $\Sigma_i, M_i'$ such that $M_i \rightsquigarrow^C M_i'$ and $\Sigma_i, [\Sigma] \equiv M'[y_p] \rightarrow^* \Sigma_i, [\Sigma_i] \equiv M_i'[y_p]$. Then, by Lemma F.10 (value translation), we then have $M_i'[y_p] \rightarrow^*_b \text{ret } V'[y_p]$. However, by Lemma F.1 (target language deterministic), we have $M_i'[y_p] \Downarrow \emptyset$ and we have a contradiction.

$\square$

**Lemma F.21.** If $\Sigma : \vdash M \rightsquigarrow^C M' : A$, and $\Sigma_p, [\Sigma] \triangleright M'[y_p] \Downarrow \text{ret } V'$, then $\Sigma \triangleright M \Downarrow V$.

**Proof.** By Theorem ?? (target termination implies source termination), we have $\Sigma \triangleright M \Downarrow M_R$ for some $M_R$. If $M_R = V$ for some $V$, we conclude. Otherwise, $M_R = \emptyset_A$. We prove this case by contradiction. By Lemma F.17 (multi-step simulation), we have some $\Sigma_i, M_i'$ such that $M_i \rightsquigarrow^C M_i'$ and $\Sigma_i, [\Sigma] \equiv M'[y_p] \rightarrow^* \Sigma_i, [\Sigma_i] \equiv M_i'[y_p]$. Then, by Lemma F.9 (canonical forms), we then have $M_i'[y_p] \Downarrow M_i'[y_p]$. However, by Lemma F.1 (target language deterministic), we have $M_i'[y_p] \Downarrow \emptyset$ and we have a contradiction.

$\square$

**G \ GRADUALITY AND PARAMETRICITY**

First, we show the full graduality/parametricity logical relation in Figure 36

**Definition G.1.** We say $\gamma, \delta$ are valid instantiations of $\Gamma^2$ in CBPVOSum, written $(\gamma, \delta) \models \Gamma^2$ when:

- For each $i \in (I, r)$, there exists $\Sigma_i$ such that for each $(x : A^i) \in \Gamma^i$, $\Sigma_i \models \gamma(x) : [A_i]$ when $\Gamma^i \rightsquigarrow A^i$.

- For each $x \in \Gamma^e$, $\delta_i(x)$ and $\Sigma_i \models \gamma_i(c_X) : \text{Case } \delta_i(x)$.

- For each $x \in \Gamma^e$, $\delta_r(x) \in \text{Rel}_{\omega}(\delta_i(X), \delta_i(X))$. 

Definition G.2. We define the extension of an interpretation $\eta$ with a new association between seals as

$$
\eta \uplus (\sigma_1, \sigma_r, R) = (\eta.\text{size} + 1, (f, \eta.\text{size} \mapsto (\sigma_1, \sigma_r)), (\rho, \eta.\text{size} \mapsto R))
$$

Logical Lemmas

Lemma G.3. If $R \in \text{Rel}_o[A_l, A_r]$, then $[R]_n \in \text{Rel}_n[A_l, A_r]$.

Proof. Direct by definition. \qed

**Lemma G.4.** \( [\mathcal{V}^{-} \llbracket A^E \rrbracket] y \delta ]_n = [\mathcal{V}^n \llbracket A^E \rrbracket] y \delta \)

**Proof.** Direct by definition.

**Lemma G.5.** If \( R \in \text{Rel}_n[A_I, A_r] \), then \( R \in \text{Rel}_n[A_I, A_r] \)

**Proof.** If \((w, V_l, V_r) \in (\triangleright R)\) and \(w' \equiv w\), then to show \((w', V_l, V_r) \in R\) we need to show that for any \(w'' \sqsubseteq w'\), that \((w'', V_l, V_r) \in R\), but this follows because \((w, V_l, V_r) \in R\) and \(w'' \sqsubseteq w' \equiv w\).

**Lemma G.6.** Let \( \Gamma^E \) be a well-formed context, with \( \Gamma^E \vdash A^E : A_l \sqsubseteq A_r \). If \((y, \delta) \vdash \Gamma^E\), then \( [\mathcal{V}^{-} \llbracket A^E \rrbracket] y \delta \in \text{Rel}_n[A_I, A_r] \).

**Proof.** By induction on \( A^E \).

(1) \( X \): by lemma G.3.

(2) \( ? \): If \((w, \inj_{\sigma_l} V_l, \inj_{\sigma_r} V_r) \in [\mathcal{V}^{-} \llbracket A^E \rrbracket] y \delta \) and \(w' \equiv w\), then there exists \( R \in \text{Rel}_n[A_I, A_r]\) with \(w, \eta \vdash (\sigma_l, \sigma_r, R)\) and \((w, V_l, V_r) \in \triangleright R\). By definition of \( \equiv \), we have that \(w' \vdash (\sigma_l, \sigma_r, [R]_{w',j})\) so it is sufficient to show \((w, V_l, V_r) \in \triangleright [R]_{w',j}\), which follows by lemma G.5 that later preserves monotonicity.

(3) \( \tagG_{G}(A^E) \): by inductive hypothesis, using lemma G.5 in the \( \triangleright \) case.

(4) \( \triangleleft \): immediate

(5) \( \times \): immediate by inductive hypothesis.

(6) \( \rightarrow \): If \((w, V_l, V_r) \in [\mathcal{V}^n \llbracket A^E \rightarrow B^E \rrbracket] y \delta \) and \(w' \equiv w\). Then given \(w'' \sqsubseteq \equiv \) and \((w'', V'_l, V'_r) \in [\mathcal{V}^n \llbracket A^E \rrbracket] y \delta \), we need to show \((w'', \text{force } V_l' V_r'), \text{force } V_l' V_r', \text{force } V_l', V_r') \in [\mathcal{V}^n \llbracket B^E \rrbracket] y \delta \), but this holds by relatedness of \(V_l, V_r\) since \(w'' \equiv w\) by transitivity of world extension.

(7) \( \forall \psi, \exists \psi \): similar to the \( \rightarrow \) case.

**Lemma G.7.** If \((w, y, \delta) \in \mathcal{G}^{-}[\Gamma^E]\) and \(w' \equiv w\), then \((w', y, \delta) \in \mathcal{G}^{-}[\Gamma^E]\).

**Proof.** By induction on \( \Gamma^E \), uses monotonicity of \( \mathcal{V}^{-}[A^E] \).

**Corollary G.8.** \( \mathcal{V}^{-}[A^E] y \delta \in \text{Rel}_n[A_I, A_r] \)

**Lemma G.9 (Anti-Reduction).** (1) If \( w' \equiv w \) and \((w, \Sigma_l, M_l) \mapsto^{w, j \mapsto w', j} (w', \Sigma_l, M'_l) \) and \((w, \Sigma_r, M_r) \mapsto^{w} (w', \Sigma_r, M'_r) \) and \((w', M'_l, M'_r) \in E^{-}[A^E] y \delta \), then \((w, M_l, M_r) \in E^{-}[A^E] y \delta \).

(2) If \( w' \equiv w \) and \((w, \Sigma_l, M_l) \mapsto^{w, j \mapsto w', j} (w', \Sigma_l, M'_l) \) and \((w, \Sigma_r, M_r) \mapsto^{w} (w', \Sigma_r, M'_r) \) and \((w', M'_l, M'_r) \in E^{-}[A^E] y \delta \), then \((w, M_l, M_r) \in E^{-}[A^E] y \delta \).

**Proof.** We do the \( < \) case, the other is symmetric. By case analysis on \((w', M'_l, M'_r) \in E^{-}[A^E] y \delta \).

(1) If \( w', \Sigma_l, M'_l \mapsto^{w, j \mapsto w, j+1} \), then \((w, \Sigma_l, M_l) \mapsto^{w, j \mapsto w, j+1} \) and \(w, j - w'. j + w'. j + 1 = w, j + 1 \).

(2) If \( w', \Sigma_l, M'_l \mapsto^{w} \Sigma_l, U \), with \( j \leq w, j \), then \((w, \Sigma_l, M_l) \mapsto^{w, j \mapsto w, j + j} \Sigma_l, U \) and \(w, j - w'. j + j \leq w, j \) since \( j \leq w'. j \leq 0 \).

(3) Finally, if there is some \( w'' \equiv w' \) and \((w'', V_l, V_r) \in \mathcal{V}^{-}[A^E] y \delta \) with \(w', \Sigma_l, M'_l \mapsto^{w, j \mapsto w', j} w'' \). \( \Sigma_l, \text{ret } V_l \) and \(w', \Sigma_r, M'_r \mapsto^{w'} \Sigma_r, \text{ret } V_r \), then \((w, \Sigma_l, M_l) \mapsto^{w, j \mapsto w, j \mapsto w, j \mapsto w', j \mapsto w', j \mapsto w, j} \Sigma_l, \text{ret } V_l \) and \(w, \Sigma_r, M_r \mapsto^{w} \Sigma_r, \text{ret } V_r \) and \(w, j - w'. j + w'. j - w', j = w, j - w', j \) so the result holds.

**Lemma G.10 (Pure Anti-Reduction).** If \((w, M'_l, M'_r) \in E^{-}[A^E] y \delta \) and \((w, \Sigma_l, M_l) \mapsto^{0} (w, \Sigma_l, M'_l) \) and \((w, \Sigma_r, M_r) \mapsto^{0} (w, \Sigma_r, M'_r) \), then \((w, M_l, M_r) \in E^{-}[A^E] y \delta \).

**Proof.** Immediate corollary of anti-reduction lemma G.9.
LEMMA G.11 (Pure Forward Reduction). If \((w, M'_l, M'_r) \in E^-[A^\Sigma]y\delta\) and \((w, \Sigma_l, M_l) \mapsto^0\) \((w, \Sigma_r, M_r) \mapsto^0\) \((w, M_l, M_r) \in E^-[A^\Sigma]y\delta\).

Proof. By determinism of evaluation.

LEMMA G.12 (Monadic bind). If \((w, M_l, M_r) \in E^-[A^\Sigma]y\delta\) and for all \(w' \equiv w\), \((w', V_l, V_r) \in V^-[A^\Sigma]y\delta\), \((w', S_l[\text{ret } V_l], S_r[\text{ret } V_r]) \in E^-[B^\Sigma]y\delta\), then \((w, S_l[M_l], S_r[M_r]) \in E^-[B^\Sigma]y\delta\).

Proof. We show the proof for \(E^-[A^\Sigma]\), the > case is symmetric. By case analysis on \((w, M_l, M_r) \in E^-[A^\Sigma]y\delta\).

1. If \(w, \Sigma_l, M_l \mapsto w, j + 1\), then \(w, \Sigma_l, S[M_l] \mapsto w, j + 1\).
2. If \(w, \Sigma_l, M_l \mapsto l\) \(w, \Sigma'_l, U\), then \(w, \Sigma_l, S[M_l] \mapsto l\) \(w, \Sigma'_l, U\).
3. Otherwise there exists \(w'\) and \((w', V_l, V_r) \in V^-[A^\Sigma]y\delta\) with \(w, \Sigma_l, M_l \mapsto w', j - w'j\) \(w', \Sigma_l, \text{ret } V_l\) and \(w, \Sigma_r, M_r \mapsto w', \Sigma_r, \text{ret } V_r\). Then \(w, \Sigma_l, S_l[M_l] \mapsto w', j - w'j\) \(w', \Sigma_l, S_l[\text{ret } V_l]\) and \(w, \Sigma_r, S_r[M_r] \mapsto w', \Sigma_r, S_r[\text{ret } V_r]\), and the result follows by the assumption.

Pure evaluation is monotone.

LEMMA G.13. If \(\Sigma, M \mapsto^* \Sigma, N\), then for any \(\Sigma' \supseteq \Sigma, \Sigma', M \mapsto^* \Sigma', N\).

Clamping

LEMMA G.14. If \((w, V_l, V_r) \in R\) and \(w.j \leq n\), then \((w, V_l, V_r) \in [R]_n\).

Proof. Direct from definition.

Tag-to-type

LEMMA G.15. \(\mathcal{V}_n^-[G]y\delta = [\delta_{\text{ret}}(\text{case}(G))]_n\)

Proof. Direct from definition.

LEMMA G.16 (Weakening). If \(\Gamma^- \vdash A^\Sigma\) and \(\Gamma^- \subseteq \Gamma^-\) and \((w, \gamma, \delta) \in G^-[\Gamma^-]\) and \((w, \gamma', \delta') \in G^-[\Gamma^-]\), where \(\gamma \subseteq \gamma'\) and \(\delta \subseteq \delta'\), then all of the following are true:

\[\mathcal{V}^-[A^\Sigma]y\delta = \mathcal{V}^-[A^\Sigma]y\gamma'\delta'\]
\[E^-[A^\Sigma]y\delta = E^-[A^\Sigma]y\gamma'\delta'\]

Proof. Straightforward, by induction over \(\Gamma^-\).

G.1 Cast Lemmas

To prove the cast left lemma, we need the following lemma that casts always either error or terminate with a well-typed inputs.

LEMMA G.17 (Casts don’t diverge). If \(\Gamma \vdash A^\Sigma : A_l \subseteq A_r\), then for any \(\Sigma\), and \(\Sigma \mid \cdot \vdash \gamma : [\Gamma]\),

1. If \(\Sigma \mid \cdot \vdash V : A_l\), then either \(\Sigma, [\langle A^\Sigma \rangle]_{\text{ret } V_l}[\gamma] \mapsto^* \Sigma, U\) or \(\Sigma, [\langle A^\Sigma \rangle]_{\text{ret } V_l}[\gamma] \mapsto^* \Sigma, \text{ret } V'\).
2. If \(\Sigma \mid \cdot \vdash V : A_r\), then either \(\Sigma, [\langle A^\Sigma \rangle]_{\text{ret } V}[\gamma] \mapsto^* \Sigma, U\) or \(\Sigma, [\langle A^\Sigma \rangle]_{\text{ret } V}[\gamma] \mapsto^* \Sigma, \text{ret } V'\).

Proof. By induction on \(A^\Sigma\).

1. If \(A^\Sigma \in \{?, \top\}\) the cast is trivial.
2. Case \(A^\Sigma = \text{tag}_G(A^\Sigma)\):
(a) The upcast definition expands as follows:
\[
\langle \text{tag}_G(AG^E) \rangle[\text{ret } V][y] = x \rightarrow \langle \text{AG}^E \rangle[\text{ret } V][y]; \text{ret inj}_\sigma x
\]
where \( \sigma = \gamma(\text{case}(G)) \). By inductive hypothesis, \( \langle \text{AG}^E \rangle[\text{ret } V][y] \) either errors (in which case the whole term errors), or runs to a value \( V' \), in which case
\[
x \rightarrow \text{ret } V'; \text{ret inj}_\sigma x \rightarrow^* \text{ret inj}_\sigma V'
\]
(b) The downcast definition expands as follows:
\[
\langle \text{tag}_G(AG^E) \rangle[\text{ret } V][y] = x \rightarrow \text{ret } V; \text{match } x \text{ with } \sigma\{\text{inj } y, \langle AG^E \rangle[\text{ret } y][y] \mid U}\]
Since \( \Sigma \mid .V: \text{OSum}, V = \text{inj}_\sigma V' \) for some \( \sigma' \in \Sigma \).

(i) If \( \sigma' = \sigma \), then
\[
\text{match } \text{inj}_\sigma V' \text{ with } \sigma\{\text{inj } y, \langle AG^E \rangle[\text{ret } y][y] \mid U\} \rightarrow^1 \langle AG^E \rangle[\text{ret } V'][y]
\]
and then it follows by inductive hypothesis with \( AG^E \).

(ii) If \( \sigma' \neq \sigma \), then
\[
\text{match } \text{inj}_\sigma V' \text{ with } \sigma\{\text{inj } y, \langle AG^E \rangle[\text{ret } y][y] \mid U\} \rightarrow^1 U
\]
and the result holds.

(3) If \( A^E = A^E_1 \times A^E_2 \), we consider the downcast case, the upcast is entirely symmetric. First,
\[
\langle A^E_1 \times A^E_2 \rangle[\text{ret } V][y] = x \rightarrow \text{ret } V;
\text{let } (x_1, x_2) = x;
\text{let } (y_1, y_2) = \langle A^E_1 \rangle[\text{ret } x_1][y];
y_2 \rightarrow \langle A^E_2 \rangle[\text{ret } x_2][y];
\text{ret } (y_1, y_2)
\]
Next, since \( V \) is well-typed, \( V = (V_1, V_2) \). Then,
\[
x \rightarrow \text{ret } V;
\text{let } (x_1, x_2) = x;
\text{let } (y_1, y_2) = \langle A^E_1 \rangle[\text{ret } x_1][y];
y_2 \rightarrow \langle A^E_2 \rangle[\text{ret } x_2][y];
\text{ret } (y_1, y_2)
\]
Applying the inductive hypothesis to \( A^E_1 \), either \( \langle A^E_1 \rangle[\text{ret } V_1][y] \) errors (in which case the whole term errors), or it runs to a value \( V'_1 \). Then we need to show
\[
y_2 \leftrightarrow \langle A^E_2 \rangle[\text{ret } V_2][y];
\text{ret } (V'_1, y_2)
\]
errors or terminates. Applying the inductive hypothesis to \( A^E_2 \), either \( \langle A^E_2 \rangle[\text{ret } V_2][y] \) errors (in which case the whole term errors), or it runs to a value \( V'_2 \). Then the whole term runs to \( \text{ret } (V'_1, V'_2) \).

(4) If \( A^E = A^E_1 \rightarrow A^E_2 \), we consider the downcast case (upcast is symmetric). The downcast definition expands as follows:
\[
\langle A^E_1 \rightarrow A^E_2 \rangle[\text{ret } V][y] = f \leftarrow \text{ret } V; \text{ret thunk } \lambda x.\langle A^E_2 \rangle[\text{ret } x][y] \leftarrow \langle A^E_1 \rangle[\text{ret } y][y]; \text{force } f y[y]
\]
Which steps immediately to a value.

(5) If \( A^E = \forall x.A^E_0 \), then it follows by similar reasoning to the function case, that is, it immediately terminates.
(6) If \( A^E = \exists^E X. A^E \), we consider the downcast case (upcast is symmetric). The definition expands as follows:

\[
\begin{align*}
\llbracket \exists^E X. A^E \rrbracket (\llbracket V \rrbracket)[y] & = \text{unpack} (X, x) = \text{ret} V; \text{ret thunk } \lambda c_x. [\llbracket A^E_o \rrbracket][x] c_x \\
\end{align*}
\]

Which steps immediately to a value.

\[ \Box \]

\[ \text{LEMMA G.18 (CAST RIGHT). For any } \Gamma^E : \Gamma, \text{ if } \Gamma^E \vdash AC^E : A \sqsubseteq C \text{ and } \Gamma^E \vdash AB^E : A \sqsubseteq B, \text{ and } \Gamma' \vdash BC^E : B \sqsubseteq C, \text{ then if } (w, y, \delta) \in G^-[\Gamma^E], \]

(1) If \((w, V_l, V_r) \in V^-[AB^E] \gamma \delta\), then \((w, \text{ret } V_l, \llbracket (BC^E) \rrbracket [\llbracket V_r \rrbracket][y_r]) \in E^-[AC^E] \gamma \delta\)

(2) If \((w, V_l, V_r) \in V^-[AC^E] \gamma \delta\), then \((w, \text{ret } V_l, \llbracket (BC^E) \rrbracket [\llbracket V_r \rrbracket][y_r]) \in E^-[AB^E] \gamma \delta\)

\[ \text{PROOF. By induction on } BC^E. \]

(1) If \( BC^E \in \{\emptyset, ?, X\} \), then the cast is trivial.

(2) If \( BC^E = \text{tag}_G(BG^E) \), then \( AC^E = \text{tag}_G(AE^G) \).

(a) For the upcast case, we are given that \((w, V_l, V_r) \in V^-[AB^E] \gamma \delta\) and we need to prove that

\[
(w, \text{ret } V_l, y \leftarrow \llbracket (BG^E) \rrbracket [\llbracket V_r \rrbracket][y_r]; \text{ret inj}_G y) \in E^-[\text{tag}_G(AE^G)] \gamma \delta
\]

By inductive hypothesis, we know

\[
(w, \text{ret } V_l, \llbracket (BG^E) \rrbracket [\llbracket V_r \rrbracket][y_r]) \in E^-[AE^G] \gamma \delta
\]

We then use monadic bind (lemma G.12). Suppose \( w' \models w \) and \((w', V_l', V_r') \in V^-[AE^G] \gamma \delta\).

We need to show that

\[
(w', \text{ret } V_l', y \leftarrow \text{ret } V_r'; \text{ret inj}_G y) \in E^-[\text{tag}_G(AE^G)] \gamma \delta
\]

By anti reduction, it is sufficient to show

\[
(w', V_l, \text{inj}_{y_r(G)}(V_r) \in V^-[\text{tag}_G(AE^G)] \gamma \delta
\]

(i) If \( \sim = < \), we need to show \((w', V_l, V_r) \in V^-[\text{tag}_G(AE^G)] \gamma \delta\), which follows by inductive hypothesis.

(ii) If \( \sim = > \), we need to show \((w', V_l, V_r) \in V^-[\text{tag}_G(AE^G)] \gamma \delta\), that is for any \( w'' \models w'\), \((w'', V_l, V_r) \in V^-[\text{tag}_G(AE^G)] \gamma \delta\) which also follows by inductive hypothesis.

(b) For the downcase, we know \((V_l, V_r) \in V^-[\text{tag}_G(AE^G)] \gamma \delta\). Let \( \sigma_r = y_r(\text{case}(G)) \).

(i) In the \( < \) case, we know \( V_r = \text{inj}_{\sigma_r} V_r' \) and

\[
(w, V_l, V_r') \in V^-[AE^G] \gamma \delta
\]

we need to show

\[
(w, \text{ret } V_l, \text{match } (\text{inj}_{\sigma_r} V_r') \text{ with } \sigma_r \{ \text{inj } x. [\llbracket BG^E \rrbracket [x][y_r] | U] \}) \in E^-[AE^G] \gamma \delta
\]

the right hand side reduces to \([\llbracket BG^E \rrbracket [V''']\), so by anti-reduction it is sufficient to show

\[
(w, \text{ret } V_l, \llbracket (BG^E) \rrbracket [V''']) \in E^-[AE^G] \gamma \delta
\]

which follows by inductive hypothesis.

(ii) In the \( > \) case, we know \( V_r = \text{inj}_{\sigma_r} V_r' \) and

\[
(w, V_l, V_r') \in E^>[AE^G] \gamma \delta
\]

(note the \( > \)). We need to show

\[
(w, \text{ret } V_l, \text{match } (\text{inj}_{\sigma_r} V_r') \text{ with } \sigma_r \{ \text{inj } x. [\llbracket BG^E \rrbracket [x][y_r] | U] \}) \in E^>[AE^G] \gamma \delta
\]

the right hand side takes 1 step to \([\llbracket BG^E \rrbracket [V'']\).

(A) If \( w.j = 0 \), then we are done.
(B) Otherwise, define \( w' = (w, j - 1, w, \Sigma_l, w, \Sigma_r, [w, \eta]_{w,j-1}) \). Then by anti-reduction, it is sufficient to show

\[
(w', \text{ret } V_l, \langle BG_E \rangle_1 \langle V'_l \rangle) \in \mathcal{E}^> \lceil AB_E \rceil_{\nu} \delta
\]

\[
(w', \text{ret } V_l, \langle BG_E \rangle_1 \langle V'_l \rangle) \in \mathcal{E}^> \lceil AB_E \rceil
\]

By inductive hypothesis, it is sufficient to show

\[
(w', V_l, V'_l) \in \mathcal{V}'^> \lceil AG_E \rceil \nu \delta
\]

which follows by assumption because \( w' \sqsubseteq w \).

(3) If \( BC_E = BC_E \times BC_E \), then by precision inversion also \( AC_E = AC_E \times AC_E \) and \( AB_E = AB_E \times AB_E \). We consider the upcast case, the downcast case follows by an entirely analogous argument.

Given \((w, V_l, V_r) \in \mathcal{V}'^> \lceil BC_E \times BC_E \rceil_{\nu} \delta\), we need to show

\[
(w, \text{ret } V_l, \langle (BC_E)_1 \times (BC_E)_2 \rangle \langle \text{ret } V_l \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

Expanding definitions, and applying anti-reduction, this reduces to showing

\[
(w, \text{ret } V_l, \langle (BC_E)_1 \times (BC_E)_2 \rangle \langle \text{ret } V_l \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

\[
(w, \text{ret } V_l, \langle (BC_E)_1 \times (BC_E)_2 \rangle \langle \text{ret } V_l \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

Since \((w, V_l, V_r) \in \mathcal{V}'^> \lceil BC_E \times BC_E \rceil_{\nu} \delta\), we know

\[
V_l = (V_{l1}, V_{l2}) \quad V_r = (V_{r1}, V_{r2})
\]

\[
(V_{l1}, V_{r1}) \in \mathcal{V}'^> \lceil BC_E \rceil_{\nu} \delta \quad (V_{l2}, V_{r2}) \in \mathcal{V}'^> \lceil BC_E \rceil_{\nu} \delta
\]

So after a reduction we need to show

\[
(w, \text{ret } V_l, z_1 \leftarrow \langle (BC_E)_1 \rangle \langle \text{ret } V_{l1} \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

\[
(z_2 \leftarrow \langle (BC_E)_2 \rangle \langle \text{ret } V_{l2} \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

\[
(\text{ret } (z_1, z_2)) \quad (\text{ret } (z_1, z_2))
\]

By forward reduction, it is sufficient to prove the following, (which is amenable to monadic bind):

\[
(w, z \leftarrow \text{ret } V_{l1}, z_1 \leftarrow \langle (BC_E)_1 \rangle \langle \text{ret } V_{r1} \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta \nu \gamma
\]

\[
(z_2 \leftarrow \langle (BC_E)_2 \rangle \langle \text{ret } V_{r2} \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

\[
(\text{ret } (z_1, z_2)) \quad (\text{ret } (z_1, z_2))
\]

We then apply monadic bind with the inductive hypothesis for \( BC_E \). Given \( w' \sqsubseteq w \) and \((V'_{l1}, V'_{r1}) \in \mathcal{V}'^> \lceil AC_E \rceil_{\nu} \delta\) the goal reduces to

\[
(w', z_2 \leftarrow \text{ret } V_{l2}, z_2 \leftarrow \langle (BC_E)_2 \rangle \langle \text{ret } V_{r2} \rangle) \in \mathcal{E}^> \lceil AC_E \times AC_E \rceil_{\nu} \delta
\]

\[
(\text{ret } (V'_{l2}, z_2)) \quad (\text{ret } (V'_{r2}, z_2))
\]

We then apply another monadic bind with the inductive hypothesis for \( BC_E \). Given \( w'' \sqsubseteq w' \) and \((V''_{l2}, V''_{r2}) \in \mathcal{V}'^> \lceil AC_E \rceil_{\nu} \delta \nu \gamma\), the goal reduces to

\[
(w'', (V'_{l1}, V'_{l2}), (V'_{r1}, V'_{r2})) \in \mathcal{V}'^> \lceil AC_E \times AC_E \rceil w'' \gamma
\]

which follows immediately by our assumptions from monadic bind.
(4) If $BC_\sigma^= = BC_\sigma^\uparrow \rightarrow BC_\sigma^\uparrow o$, then by precision inversion also $AC_\sigma^= = AC_\sigma^\uparrow \rightarrow AC_\sigma^\uparrow o$ and $AB_\sigma^= = AB_\sigma^\uparrow \rightarrow AB_\sigma^\uparrow o$. We consider the upcast case, the downcast case follows by an entirely analogous argument.

Given $(V_i, V_r) \in \mathcal{V}^{-}[BC_i^\uparrow \rightarrow BC_o^\uparrow]_\gamma \delta$, we need to show

$$(w, \text{ret } V_i, [(BC_i^\uparrow \rightarrow BC_o^\uparrow)] [\text{ret } V_r][\gamma_r]) \in E^-[AC_i^\uparrow \rightarrow AC_o^\uparrow]_\gamma \delta$$

Expanding definitions, this reduces to showing

$$(w, \text{thunk } (\lambda x. y \leftarrow [(BC_i^\uparrow)] [\text{ret } x][\gamma_r]),) \in \mathcal{V}^{-}[AC_i^\uparrow \rightarrow AC_o^\uparrow]_\gamma \delta$$

where

$$z \leftarrow \text{force } V_i y;$$

$$[(BC_i^\uparrow)] [\text{ret } z][\gamma_r]$$

Let $w' \supset w$ be a future world and $(w', V_{i'}, V_{r'}) \in \mathcal{V}^{-}[AC_i^\uparrow]_\gamma \delta$. Then our goal reduces to showing

$$(w', \text{force } V_i V_{i'}, y \leftarrow [(BC_i^\uparrow)] [\text{ret } V_{i'}][\gamma_r]) \in E^-[AC_o^\uparrow]_\gamma \delta$$

by forward reduction, it is sufficient to show

$$(w', y \leftarrow V_i; y \leftarrow [(BC_i^\uparrow)] [\text{ret } V_{i'}][\gamma_r]) \in E^-[AC_o^\uparrow]_\gamma \delta$$

force $V_i y z \leftarrow \text{force } V_i y;

[(BC_i^\uparrow)] [\text{ret } z][\gamma_r]$$

We then use the inductive hypothesis on $BC_i^\uparrow$ (which applies because of downward-closure) and monadic bind: assume $w'' \supseteq w'$ and $(w'', V_{i''}, V_{r''}) \in \mathcal{V}^{-}[AC_i^\uparrow]_\gamma \delta$. We need to show

$$(w'', \text{force } V_i V_{i''}, z \leftarrow \text{force } V_i V_{i''}) \in E^-[AC_o^\uparrow]_\gamma \delta$$

$$[(BC_i^\uparrow)] [\text{ret } z][\gamma_r]$$

We apply monadic bind again, noting that the applications are related by assumption and downward closure. Assume $w''' \supseteq w''$ and $(w''', V_{i'''}, V_{r'''}) \in \mathcal{V}^{-}[AB_o^\uparrow]_\gamma \delta$. By anti-reduction, the goal reduces to showing

$$(\text{ret } V_o, [(BC_o^\uparrow)] [\text{ret } V_o''][\gamma_r]) \in E^-[AC_o^\uparrow]_\gamma \delta$$

which follows by inductive hypothesis for $BC_o^\uparrow$.

(5) If $BC^= = \forall^X X. BC_o^\uparrow$, then by precision inversion also $AC^= = \forall^X X. AC_o^\uparrow$ and $AB^= = \forall^X X. AB_o^\uparrow$. We consider the upcast case, the downcast case follows by an entirely analogous argument.

Given $(w, V_i, V_r) \in \mathcal{V}^{-}[\forall^X X. BC_o^\uparrow]_\gamma \delta$, we need to show

$$(w, \text{ret } V_i, [\forall^X X. BC_o^\uparrow] [\text{ret } V_r][\gamma_r]) \in E^-[\forall^X AC_o^\uparrow]_\gamma \delta$$

Expanding definitions and applying anti-reduction, this reduces to showing

$$(w, V_i, V_r') \in \mathcal{V}^{-}[\forall^X AC_o^\uparrow]_\gamma \delta$$

where

$$V_r' = \text{thunk } (\Lambda X. \lambda c_X : \text{Case } X. [(BC_o^\uparrow)] [\text{ret } V_r X c_X][\gamma_r])$$

Let $w' \supseteq w, R \in \text{Rel } [A_l, A_r]$, and $w.\eta = (\sigma_l, \sigma_r, [R]_{w, j})$, then we need to show that

$$(w', \text{force } V_l) A_l \sigma_l, (\text{force } V_r') A_r \sigma_r) \in E^-[AC_o^\uparrow]_\gamma \delta'$$

where $\gamma' = (\gamma, c_X \mapsto (\sigma_l, \sigma_r))$ and $\delta' = (\delta, X \mapsto (A_l, A_r, R))$ which reduces in 0 steps to showing

$$(w', \text{force } V_l) A_l \sigma_l, [(BC_o^\uparrow)] [\text{ret } V_r X A_r \sigma_r][\gamma_r]) \in E^-[AC_o^\uparrow]_\gamma \delta'$$
by noting that by definition, \( y'_r = y_r, c_X \mapsto \sigma_r \).

Then, we invoke monadic bind using \((w', (\text{force } V_l) A_l \sigma_l, (\text{force } V_r) A_r \sigma_r) \in E^-[BC_o^\epsilon] y'\delta'\).

Let \( w'' \ni w' \) and \((w'', V_{lo}, V_{ro}) \in V^-[BC_o^\epsilon] y'\delta'\). We then need to show

\[(w'', \text{ret } V_{lo}, [\langle BC_o^\epsilon \rangle_\delta][\text{ret } V_{ro}][y'_r] \in E^-[AC_o^\epsilon] y'\delta')\]

which follows by inductive hypothesis.

(6) If \( BC_o^\epsilon = \exists X. BC_o^\epsilon \), then by precision inversion also \( AC_o^\epsilon = \exists X. AC_o^\epsilon \) and \( AB_o^\epsilon = \exists X. AB_o^\epsilon \).

We consider the upcast case, the downcast case follows by an entirely analogous argument.

Given \((w, V_l, V_r) \in V^-[\exists X. BC_o^\epsilon] y\delta\), we need to show

\[(w, \text{ret } V_l, [\langle \exists X. BC_o^\epsilon \rangle_\delta][\text{ret } V_r][y_r]) \in E^-[\exists X. AC_o^\epsilon] y\delta\]

Expanding definitions and applying anti-reduction, this reduces to showing

\[(w, \text{ret } V_l, \text{unpack } (X, y) = \text{ret } V_l; \text{ret } \text{pack}(X, (\text{thunk } (\lambda c_X : \text{Case } X.[\langle BC_o^\epsilon \rangle_\delta][\text{force } y c_X][y_r]))) \in E^-[\exists X. AC_o^\epsilon] y\delta\]

By definition of \( V^-[\exists X. BC_o^\epsilon] y\delta \), we know

\[V_l = \text{pack}(A_l, V'_l)\]

\[V_r = \text{pack}(A_r, V'_r)\]

and there is an associated relation \( R \in \text{Rel}_{\epsilon}[A_l, A_r] \). Then the goal reduces to showing

\[(\text{pack}(A_l, V'_l), (\text{pack}(A_l, (\text{thunk } (\lambda c_X. [BC_o^\epsilon]_\delta)[(\text{force } V'_r\sigma)\sigma][y'_r])) \in V^-[\exists X. AC_o^\epsilon] y\delta\]

where \( y' = y, c_X \mapsto (\sigma_l, \sigma_r), \) and \( \delta' = \delta, X \mapsto (A_l, A_r, R) \). We use the relatedness assumption and monadic bind again. Then we are given \( w'' \ni w' \), and \((V_{lo}, V_{ro}) \in V^-[AB_o^\epsilon] y'\delta'\) and need to show

\[(w'', \text{ret } V_{lo}, [\langle BC_o^\epsilon \rangle_\delta][\text{ret } V_{ro})] \in E^-[AC_o^\epsilon] y'\delta')\]

which follows by inductive hypothesis.

\[\square\]

**Lemma G.19 (Cast Left).** For any \( \Gamma^\epsilon : \Gamma, \Gamma^\epsilon + AC^\epsilon : A \subseteq C, \Gamma + AB^\epsilon : A \subseteq B, \Gamma^\epsilon + BC^\epsilon : B \subseteq C \) and \((w, y, \delta) \in G^-[\Gamma^\epsilon] \),

1. If \((w, V_l, V_r) \in V^-[BC_o^\epsilon] y\delta\), then \((w, [\langle AB_o^\epsilon \rangle_\delta][\text{ret } V_l][y_l], \text{ret } V_r) \in E^-[AC_o^\epsilon] y\delta\)
2. If \((V_l, V_r) \in V^-[AC_o^\epsilon] y\delta\), then \((w, [\langle AB_o^\epsilon \rangle_\delta][\text{ret } V_l][y_l], \text{ret } V_r) \in E^-[BC_o^\epsilon] y\delta\)

**Proof.** By nested induction on \( AB_o^\epsilon \) and \( AC_o^\epsilon \), i.e., if \( AB_o^\epsilon \) becomes smaller \( AC_o^\epsilon \) can be anything but if \( AC_o^\epsilon \) becomes smaller, then \( AB_o^\epsilon \) must stay the same.

1. If \( AC_o^\epsilon \in \{\exists, AC_o^\epsilon \times AC_o^\epsilon, AC_o^\epsilon \rightarrow AC_o^\epsilon, \forall X. AC_o^\epsilon, \exists X. AC_o^\epsilon \} \), then \( AB_o^\epsilon \) has the same top-level connective, and the proof is symmetric to the case of lemma G.18, which always makes \( AB_o^\epsilon \) and \( AC_o^\epsilon \) smaller in uses of the inductive hypothesis.
2. If \( AC_o^\epsilon = ? \), then also \( AB_o^\epsilon = BC_o^\epsilon = ? \) and the cast is trivial.
3. If \( AC_o^\epsilon = \text{tag}_G(AG_o^\epsilon) \), there are two cases: either \( BC_o^\epsilon = ? \) or \( BC_o^\epsilon = \text{tag}_G(BG_o^\epsilon) \).
   a. If \( BC_o^\epsilon = ? \), then \( AB_o^\epsilon = \text{tag}_G(AG_o^\epsilon) = AC_o^\epsilon \). Define \( \sigma_l = y_l(\text{case}(G)), \sigma_r = y_r(\text{case}(G)) \).
   i. In the upcast case, we know \((w, V_l, V_r) \in V^-[\text{tag}_G(AG_o^\epsilon)] y\delta \). In which case, \( V_r = \text{inj}_{\sigma_r} V'_r \).
(A) In the $<$ case, we know $(w, V_i, V'_i) \in \cal V \uparrow [\cal AG^\Sigma] y \delta$, and we need to show

$$(w, (x \leftarrow [[(\cal AG^\Sigma)] [\text{ret } V_i] [\gamma_l]; \text{ret inj}_{\sigma_l} x], \text{ret inj}_{\sigma_r} V'_i) \in \cal E \uparrow [?]? y \delta$$

which by forward reduction is equivalent to showing

$$(w, (x \leftarrow [[(\cal AG^\Sigma)] [\text{ret } V_i] [\gamma_l]; \text{ret inj}_{\sigma_l} x), x \leftarrow V'_i; \text{ret inj}_{\sigma_r} x) \in \cal E \uparrow [?]? y \delta$$

By inductive hypothesis, we know

$$(w, [[(\cal AG^\Sigma)] [\text{ret } V_i] [\gamma_l], \text{ret } V'_i) \in \delta_R(\text{case } G)$$

so we can apply monadic bind. Let $w' \equiv w$, and $(w', V'_i, V''_i) \in \cal V \uparrow [G] y \delta$. Then we need to show (after applying anti-reduction)

$$(w', \text{inj}_{\sigma_l} V'_i, \text{inj}_{\sigma_r} V''_i) \in \cal V \uparrow [?]? y \delta$$

To do this, we need to give a relation $R$ such that $w' \equiv (\sigma_l, \sigma_r, R)$ and $(w', V'_i, V''_i) \in \triangleright R$. Since $y(\text{case } G) = (\sigma_l, \sigma_r)$, we know $R = [\delta_R(\text{case } G)]_{w'.j}$. And we need to show that for any $w'' \equiv w'$, that $(w'', V'_i, V''_i) \in [R]_{w'.j}$. Which follows by monotonicity because $w'' \equiv w$.

(B) The $>$ case is slightly more complicated. This time we only know we know $(w, V_i, V'_i) \in \triangleright \cal V \uparrow [\cal AG^\Sigma] y \delta$ (note the $>$), and we need to show

$$(w, (x \leftarrow [[(\cal AG^\Sigma)] [\text{ret } V_i] [\gamma_l]; \text{ret inj}_{\sigma_l} x], \text{ret inj}_{\sigma_r} V'_i) \in \cal E \uparrow [?]? y \delta$$

By lemma G.17, we know $[[\cal AG^\Sigma]] [\text{ret } V_i] [\gamma_l]$ either runs to error or terminates. If it runs to an error then our goal holds. Otherwise, let $[[\cal AG^\Sigma]] [\text{ret } V_i] [\gamma_l] \mapsto^* \text{ret } V'_i$. Applying anti-reduction, we need to show

$$(w, \text{inj}_{\sigma_l} V'_i, \text{inj}_{\sigma_r} V'_i) \in \cal V \uparrow [?]? y \delta$$

by the same reasoning as above, we need to show

$$(w, V'_i, V'_i) \in \triangleright [\delta_R(\text{case } G)]_{w.j}$$

Let $w' \equiv w$. We need to show

$$(w', V'_i, V'_i) \in [\delta_R(\text{case } G)]_{w.j}$$

By our assumption, we know

$$(w', V'_i, V'_i) \in \cal V \uparrow [\cal AG^\Sigma] y \delta$$

so by inductive hypothesis, we know

$$(w', [[\cal AG^\Sigma]] [\text{ret } V_i] [\gamma_l], \text{ret } V'_i) \in \cal E \uparrow [G] y \delta = \delta_R(\text{case } G)$$

And since we know $w'.\Sigma_l, [[\cal AG^\Sigma]] [\text{ret } V_i] [\gamma_l] \mapsto^* w'.\Sigma_l, \text{ret } V'_i$ by lemma G.17, this means

$$(w', V'_i, V'_i) \in \cal V \uparrow [G] y \delta = \delta_R(\text{case } G)$$

so the result follows by lemma G.14.

(ii) For the downcast case, we know $(w, V_i, V_e) \in \cal V \uparrow [?]? y \delta$ which means there exists $\sigma_l, \sigma_r, R$ with $w.\eta \equiv (\sigma_l, \sigma_r, R)$ and $V_i = \text{inj}_{\sigma_l} V'_i$ and $V_e = \text{inj}_{\sigma_r} V'_i$ and

$$(w, V'_i, V'_i) \in \triangleright R$$

Expanding the definition of the cast and applying anti-reduction, we need to show

$$(w, \text{match } V_i \text{ with } \sigma_l \{ \text{inj } x. [[\cal AG^\Sigma]] [\text{ret } x] | \cal U \}, \text{ret } V_e) \in \cal E \uparrow [\text{tag}_G(\cal AG^\Sigma)] y \delta$$

If $\gamma_l(\text{case } G) = \sigma_l$, the left side reduces and the result holds. Otherwise, we need to show

$$(w', \text{match } V_i \text{ with } \sigma_l \{ \text{inj } x. [[\cal AG^\Sigma]] [\text{ret } x] | \cal U \} \mapsto^1 [[\cal AG^\Sigma]] \text{ret } V'_i$$
(A) In the > case, we need to show
\[(w, \llbracket \langle AG \rangle \rrbracket \downarrow \text{ret } V'_i, \text{ret } V_r) \in E^>[[\text{tag}_G(AG)]\gamma\delta]\]
Which follows by inductive hypothesis if we can show
\[(w, V'_i, V_r) \in V^>[[\text{tag}_G(G)]\gamma\delta]\]
which reduces to showing
\[(w, V'_i, V'_r) \in \triangleright V^>\{G\}\gamma\delta\]
So let \(w' \equiv w\). We need to show
\[(w', V'_i, V'_r) \in V^>\{G\}\gamma\delta\]
But we know \((w', V'_i, V'_r) \in R\) where \(R = [\delta_R(\text{case}(G))]_{w,j}\), so the result follows by lemma G.15.

(B) In the < case, we check \(w, j\).
- If \(j = 0\), then the left side takes 1 step so the result holds.
- Otherwise, define \(w' = (w, j-1, w, \Sigma_i, w, \Sigma_r, [w, \eta]_{w,j-1})\). Then it is sufficient to show
\[(w', \llbracket \langle AG \rangle \rrbracket \downarrow \text{ret } V'_i, \text{ret } V_r) \in E^<[[\text{tag}_G(AG)]\gamma\delta]\]
To apply the inductive hypothesis we need to show
\[(w', V'_i, \text{ret } V_r) \in V^<[[\text{tag}_G(G)]\gamma\delta]\]
Unrolling definitions, it is sufficient to show
\[(w', V'_i, V'_r) \in V^<\{G\}\gamma\delta\]
But we know already that \((w', V'_i, V'_r) \in R\) where \(R = [\delta_R(\text{case}(G))]_{w,j}\), so the result follows.

(b) Finally, if \(BC^\leq = \text{tag}_G(BC^\leq)\), then \(AC^\leq = \text{tag}_G(AG^\leq)\). We consider the downcast case, the upcast case is entirely symmetric. Let \((w, V_i, V_r) \in V^>[[\text{tag}_G(AG)]\gamma\delta]\). Then we know \(V_r = \text{inj}_{\sigma_r} V'_r\) where \(\sigma_r = \gamma_r(\text{case}(G))\).

(i) If \(\sim = \prec\), we furthermore know \((w, V_i, V'_r) \in V^<\{BG\}\gamma\delta\). We need to show that
\[(w, \llbracket \langle AB \rangle \rrbracket \downarrow \text{ret } V_i][\gamma_j]\text{, ret inj}_{\sigma_i} V'_r) \in E^<[[\text{tag}_G(AG)]\gamma\delta]\]
by forward reduction it is sufficient to show
\[(w, \llbracket \langle AB \rangle \rrbracket \downarrow \text{ret } V_i][\gamma_j]\text{, x} \leftarrow \text{ret } V'_r\text{, ret inj}_{\sigma_i} x) \in E^<[[\text{tag}_G(AG)]\gamma\delta]\]
We know by inductive hypothesis that
\[(w, \llbracket \langle AB \rangle \rrbracket \downarrow \text{ret } V_i][\gamma_j]\text{, ret } V_r)\]
so we can apply monadic bind. Let \(w' \equiv w\) and \((w', V'_i, V''_r) \in V^<\{AG\}\gamma\delta\). Then we need to show (after applying anti-reduction) that
\[(w', V'_i, \text{inj}_{\sigma_i} V''_r) \in V^<\{tag}_G(AG)\}\gamma\delta\]
Which, unrolling the definition, is
\[(w', V'_i, V''_r) \in V^<\{AG\}\gamma\delta\]
which was our assumption.

(ii) If \(\sim = \succ\), we only know \((w, V_i, V'_r) \in \succ V^>\{BG\}\gamma\delta\) (note the \(\succ\)).
\[(w, \llbracket \langle AB \rangle \rrbracket \downarrow \text{ret } V_i][\gamma_j]\text{, ret inj}_{\sigma_i} V'_i) \in E^<[[\text{tag}_G(AG)]\gamma\delta]\]
By lemma G.17, the left hand side either errors or terminates with a value.
(A) If \( w.\Sigma_l, \langle AB^E \rangle_1 \downarrow \langle \text{ret } V_l \rangle [\gamma_l] \rightarrow^* \), then the result holds.
(B) If \( w.\Sigma_l, \langle AB^E \rangle_1 \downarrow \langle \text{ret } V_l \rangle [\gamma_l] \rightarrow^* \), then we need to show that

\[
(w, V'_l, \text{inj}_{\sigma}, V''_l) \in V^\uparrow \Gamma [\text{tag}_G (AG^E)] y \delta \]

Which unrolls to

\[
(w, V'_l, V''_l) \in V^\uparrow [AG^E] y \delta
\]

So let \( w' \vdash w \). We need to show

\[
(w', V'_l, V''_l) \in V^\uparrow [AG^E] y \delta
\]

By inductive hypothesis, we know

\[
(w', \langle AB^E \rangle_1 \downarrow \langle \text{ret } V_l \rangle [\gamma_l], \text{ret } V'_l) \in E^\uparrow [AG^E] y \delta
\]

and so by determinism of evaluation, we know

\[
(w', V'_l, V''_l) \in V^\uparrow [AG^E] y \delta
\]

so the result holds.

\[ \square \]

### G.2 Compatibility Lemmas

**Lemma G.20.**

\[
\Gamma^E_1, x : A^E, \Gamma^E_2 \models x \equiv \sim x \in A^E, \cdot
\]

**Proof.** We need to show

\[
(w, \text{ret } V_l, \text{ret } V_r) \in E^\uparrow [A^E] y \delta
\]

where \( V_l = \gamma_1(x) \). Since both sides are values, it is sufficient to show

\[
(w, \gamma_1(x), \gamma_r(x)) \in V^\uparrow [A^E] y \delta
\]

By definition of \( G^\uparrow [\Gamma^E_1, x : A^E, \Gamma^E_2] \), we know \( y = \gamma_1, x \rightarrow (V_l, V_r), y_2 \) and \( \delta = \delta_1, \delta_2 \) where

\[
(w, \gamma_1, \delta_2) \in G^\uparrow [\Gamma^E_1] \text{ and } (w, V_l, V_r) \in V^\uparrow [A^E] y \delta_1.
\]

Then the result follows because \( V^\uparrow [A^E] y_1 \delta_1 = V^\uparrow [A^E] y \delta \) by Lemma G.16 \( \square \)

**Lemma G.21.** *Generation compatibility.*

\[
\begin{align*}
\Gamma^E_1 \vdash M_l \equiv A_l \in B^E; \Gamma^E_2, X \equiv A^E_1, \Gamma^E_2'' & \quad \Gamma^E_1, \Gamma^E_2' \vdash A_l \subseteq A_r \\
\Gamma^E_1 \vdash \text{hide } X \equiv A_l; M_l \equiv \text{hide } X \equiv A_r; M_r \in B^E; \Gamma^E_2, \Gamma^E_2'' &
\end{align*}
\]

**Proof.** The translation is defined as

\[
\lfloor \text{hide } X \equiv A_l; M_l \rfloor = \text{newcase}_{[A_1]} cX; \lfloor M_l \rfloor
\]

We need to show

\[
(w, \text{newcase}_{[A_1]}[\delta], cX; \lfloor [M_l] [\gamma_l][\delta_l], \text{newcase}_{[A_1][\delta_1]} cX; \lfloor [M_r] [\gamma_r][\delta_1] \rfloor) \in E^\uparrow [B^E] y \delta
\]

Define \( w' = (w, j, (w.\Sigma_l, [A_1] [\delta]), (w.\Sigma_r, [A_r] [\delta]), \eta \equiv (w.\Sigma_l.\text{size}, w.\Sigma_r.\text{size}, \lfloor V^\uparrow [A^E] y \delta \downarrow w.j \rfloor)). \)

Then

\[
w.\Sigma_l.\text{newcase}_{[A_1]}[\delta] cX; \lfloor [M_l] [\gamma_l][\delta_l] \rightarrow^0 w'.\Sigma_l.\lfloor [M_l] [\gamma_l][\delta_l] \rfloor [w'.\Sigma_l.\text{size}/cX] \]

and similarly for the right side

\[
w.\Sigma_r.\text{newcase}_{[A_r]}[\delta] cX; \lfloor [M_r] [\gamma_r][\delta_r] \rightarrow^0 w'.\Sigma_r.\lfloor [M_r] [\gamma_r][\delta_r] \rfloor [w'.\Sigma_r.\text{size}/cX] \]

Then by the anti-reduction lemma G.9, it is sufficient to show

\[
(w', [M_l] [\gamma_l][\delta_l] [w'.\Sigma_l.\text{size}/cX], [M_r] [\gamma_r][\delta_r] [w'.\Sigma_r.\text{size}/cX]) \in E^\uparrow [B^E] y \delta
\]
Define $y' = y, c_X \mapsto (w'.\Sigma_i.size, w'.\Sigma_r.size)$ and $\delta' = \delta, X \mapsto (\delta_l(A_l), \delta_r(A_r), \mathcal{V}^-[A^E]y\delta\delta)$. Then $(w', y', \delta') \in \mathcal{G}^-[\Gamma_p, \Gamma^E, X \equiv A, \Gamma^E']$ since $(w', y, \delta) \in \mathcal{G}^-[\Gamma_p, \Gamma^E, \Gamma^E']$ (by monotonicity lemma G.7). Furthermore, $[M_l] [y_l] [\delta_l] [w'.\Sigma_i.size/c_X] = [M_l] [y'_l] [\delta'_l]$, so the result is equivalent to

$$(w'.[M_l] [y'_l] [\delta'_l], [M_r] [y'_r] [\delta'_r]) \in \mathcal{E}^-[B^E]y\delta$$

which is equivalent to showing

$$(w'.[M_l] [y'_l] [\delta'_l], [M_r] [y'_r] [\delta'_r]) \in \mathcal{E}^-[B^E]y'\delta'$$

by lemma G.16. Finally this result holds by applying the hypothesis. □

**Lemma G.22 (Compatibility: Upcast Left).**

$\Gamma^E \vdash M_l \subseteq M_r \in AC^E; \Gamma^E$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AC^E : A \subseteq C$ \hspace{1cm} $\Gamma, \Gamma' \vdash AB^E : A \subseteq B$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash BC^E : B \subseteq C$

$\Gamma^E \vdash \langle AB^E \rangle \downarrow M_l \subseteq M_r \in BC^E; \Gamma^E'$

**Proof.** We need to show

$$(w, \langle \langle AB^E \rangle \rangle, [[M_l] [y_l] [\delta_l], [M_r] [y_r] [\delta_r]]) \in \mathcal{E}^-[AC^E]y\delta$$

By assumption, we know

$$(w, [[M_l] [y_l] [\delta_l], [M_r] [y_r] [\delta_r]]) \in \mathcal{E}^-[AC^E]y\delta$$

So we apply monadic bind. Let $w' \equiv w$ and $(w', V_l, V_r) \in \mathcal{V}^-[AC^E]y\delta$. We need to show

$$(w', \langle \langle AB^E \rangle \rangle, \text{ret } V_l, \text{ret } V_r) \in \mathcal{E}^-[BC^E]y\delta$$

which follows by lemma G.19. □

**Lemma G.23 (Compatibility: Downcast Left).**

$\Gamma^E \vdash M_l \subseteq M_r \in BC^E; \Gamma^E$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AC^E : A \subseteq C$ \hspace{1cm} $\Gamma, \Gamma' \vdash AB^E : A \subseteq B$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash BC^E : B \subseteq C$

$\Gamma^E \vdash \langle AB^E \rangle \downarrow M_l \subseteq M_r \in AC^E; \Gamma^E'$

**Proof.** By same argument as lemma G.22. □

**Lemma G.24 (Compatibility: Upcast Right).**

$\Gamma^E \vdash M_l \subseteq M_r \in AB^E; \Gamma^E$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AC^E : A \subseteq C$ \hspace{1cm} $\Gamma, \Gamma' \vdash BC^E : B \subseteq C$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AB^E : A \subseteq B$

$\Gamma^E \vdash M_l \subseteq \langle BC^E \rangle \downarrow M_r \in AB^E; \Gamma^E'$

**Proof.** By same argument as lemma G.22, but using lemma G.18. □

**Lemma G.25.**

$\Gamma^E \vdash M_l \subseteq M_r : AC^E; \Gamma^E'$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AC^E : A \subseteq C$ \hspace{1cm} $\Gamma, \Gamma' \vdash BC^E : B \subseteq C$ \hspace{1cm} $\Gamma^E, \Gamma^E' \vdash AB^E : A \subseteq B$

$\Gamma^E \vdash \langle BC^E \rangle \downarrow M_l \subseteq M_r : AB^E; \Gamma^E'$

**Proof.** By same argument as lemma G.22, but using lemma G.18. □

**Lemma G.26.** If $(w, y, \delta) \in \mathcal{G}^-[\Gamma_p^E, X \equiv A^E, \Gamma^E_1]$ of $(w, y, \delta) \in \mathcal{G}^-[\Gamma_p^E, X, \Gamma^E_2]$, then

$$\mathcal{V}^-[X]y\delta = \mathcal{V}^-[A^E]y\delta$$
The result follows immediately from Lemma G.26.

**Lemma G.27. Seal**

\[
\frac{(X \equiv A^E) \in \Gamma^E, \Gamma^E' \quad \Gamma^E \vdash M_t \subseteq M_r : A^E; \Gamma^E'}{
\Gamma^E \vdash \text{seal}_X M_t \subseteq \text{seal}_X M_r : X; \Gamma^E'}
\]

**Proof.** Assume \((M_t[y_1][\delta_1], M_r[y_1][\delta_1]) \in E^{-}[A^E][\gamma_1].\) We need to show \((M_t[y_1][\delta_1], M_r[y_1][\delta_1]) \in E^{-}[X][\gamma_1].\)

This follows immediately from Lemma G.26.

**Lemma G.28. UnSeal**

\[
\frac{(X \equiv A^E) \in \Gamma^E, \Gamma^E' \quad \Gamma^E \vdash M_t \subseteq M_r : X; \Gamma^E'}{
\Gamma^E \vdash \text{unseal}_X M_t \subseteq \text{unseal}_X M_r : A^E; \Gamma^E'}
\]

**Proof.** By same reasoning as the seal case.

**Lemma G.29.**

\[
\Gamma_t \vdash \text{true} \subseteq \text{true} : \emptyset \quad \Gamma_t \vdash \text{false} \subseteq \text{false} : \emptyset
\]

**Proof.** The result \((w, \text{ret true, ret true}) \in E^{-}[\emptyset][\gamma_1][\delta_1] \text{ follows because } (w, \text{true, true}) \in \mathcal{V}^{-}[\emptyset][\gamma_1].\) Similarly for false.

**Lemma G.30.**

\[
\frac{\Gamma^E \vdash M_t \subseteq M_r : \emptyset; \Gamma_0^E \quad \Gamma^E_0 \vdash N_t \subseteq N_r \in B^E; \Gamma^E_0 \quad \Gamma^E_0 \vdash N_{t'} \subseteq N_{r'} \in B^E; \Gamma^E_0}{
\Gamma^E \vdash \text{if } M_t \text{ then } N_t \text{ else } N_{t'} \subseteq \text{if } M_r \text{ then } N_r \text{ else } N_{r'} : B^E; \Gamma^E_0}
\]

**Proof.** Define \(M'_t = [M_t][y_1][\delta_1]\) and similarly for the rest of the subterms. Then we need to show

\[(w, x \leftarrow M'_t; \text{case } x \{x_1.N_{t'} \mid x.f.N_{r'}\}, x \leftarrow M'_r; \text{case } x \{x_1.N_{t'} \mid x.f.N_{r'}\}) \in \mathcal{E}^{-}[B^E][\gamma_1][\delta_1].\]

By assumption and weakening, \((w, M'_t, M'_r) \in \mathcal{E}^{-}[B^E][\gamma_1][\delta_1],\) and we apply monadic bind. Suppose \(w' \equiv w\) and \((w', V_t, V_r) \in \mathcal{V}^{-}[B^E][\gamma_1][\delta_1].\) We need to show

\[(w, x \leftarrow \text{ret } V_t; \text{case } x \{x_1.N_{t'} \mid x.f.N_{r'}\}, x \leftarrow \text{ret } V_r; \text{case } x \{x_1.N_{t'} \mid x.f.N_{r'}\}) \in \mathcal{E}^{-}[B^E][\gamma_1][\delta_1].\]

By definition either \(V_t = V_r = \text{true}\) or \(V_t = V_r = \text{false}.\) WLOG assume it is true. Then

\[x \leftarrow \text{ret } V_t; \text{case } x \{x_1.N_{t'} \mid x.f.N_{r'}\} \mapsto 0 N_{t'}\]

and similarly for the right side. By anti-reduction (lemma G.9), it is sufficient to show

\[(w', N_{t'}, N_{r'}) \in \mathcal{E}^{-}[B^E][\gamma_1][\delta_1].\]

which follows by hypothesis.

**Lemma G.31. Product intro**

\[
\frac{\Gamma^E \vdash M_{00} \subseteq M_0 \in A^E_0; \Gamma_0^E_0 \quad \Gamma^E_0 \vdash M_{11} \subseteq M_1 \in A^E_1; \Gamma_1^E_1}{
\Gamma^E \vdash (M_{01}, M_{10}) \subseteq (M_0, M_1) \in A^E_0 \times A^E_1; \Gamma_0^E_0, \Gamma_1^E_1}
\]
Proof. We need to show that
\[(w, x \leftarrow [M_0]_y [\delta_t], x \leftarrow [M_1]_y [\delta_r], x \leftarrow [M_0]_y [\delta_r], x \leftarrow [M_1]_y [\delta_r]) \in E^{-}[A^E_0 \times A^E_1] \gamma \delta \]
\[y \leftarrow [M_1]_y [\delta_l], y \leftarrow [M_1]_y [\delta_r], y \leftarrow [M_1]_y [\delta_r], y \leftarrow [M_1]_y [\delta_r] \]
\[
\text{ret} (x, y) \quad \text{ret} (x, y)
\]

By inductive hypothesis and weakening, we know \((w, [M_0]_y [\delta_t], [M_0]_y [\delta_r], [\delta_r]) \in E^{-}[A^E_0] \gamma \delta\). Applying monadic bind we get some \(w' \equiv w\) and \((w', V_{l_0}, V_{r_0}) \in V^{-}[A^E_0] \gamma \delta\) and applying anti-reduction, we need to show
\[(w', y \leftarrow [M_1]_y [\delta_t], y \leftarrow [M_1]_y [\delta_r], y \leftarrow [M_1]_y [\delta_r], y \leftarrow [M_1]_y [\delta_r]) \in E^{-}[A^E_0 \times A^E_1] \gamma \delta \]
\[
\text{ret} (V_{l_0}, y) \quad \text{ret} (V_{r_0}, y)
\]

By inductive hypothesis, weakening and monotonicity, we know \((w', [M_1]_y [\delta_t], [M_1]_y [\delta_r], [\delta_r]) \in E^{-}[A^E_1] \gamma \delta\). Applying monadic bind we get some \(w'' \equiv w'\) and \((w'', V_{l_1}, V_{r_1}) \in V^{-}[A^E_1] \gamma \delta\) and applying anti-reduction we need to show
\[(w'', V_{l_1}, V_{r_1}) \in V^{-}[A^E_1] \gamma \delta\]

which follows by assumption and monotonicity. □

Lemma G.32. Product elim
\[
\Gamma^E \vdash M_1 \equiv M_r \in A^E_0 \times A^E_1; \Gamma^E_M \quad \Gamma^E, \Gamma^E_M, x : A^E_0, y : A^E_1 \vdash N_1 \equiv N_r \in B^E; \Gamma^E_N
\]
\[
\Gamma^E \vdash \text{let} (x, y) = M_l; N_1 \equiv \text{let} (x, y) = M_r; N_r \equiv B^E; \Gamma^E_M, \Gamma^E_N
\]

Proof. We need to show
\[(w, [M_1]_y [\delta_t], z \leftarrow [M_r]_y [\delta_r], z \leftarrow [M_r]_y [\delta_r], z \leftarrow [M_r]_y [\delta_r]) \in E^{-}[B^E] \gamma \delta \]
\[
\text{let} (x, y) = z; \quad \text{let} (x, y) = z; \quad [N_1]_y [\delta_t] \quad [N_1]_y [\delta_r] \quad [N_1]_y [\delta_r]
\]

First, by inductive hypothesis and weakening, we know
\[(w, [M_1]_y [\delta_t], [M_r]_y [\delta_r], [\delta_r]) \in E^{-}[A^E_0 \times A^E_1] \gamma \delta\]
Applying monadic bind, we get some \(w' \equiv w\) with \((w', V_{l_0}, V_{r_0}) \in V^{-}[A^E_0] \gamma \delta\) and \((w', V_{l_1}, V_{r_1}) \in V^{-}[A^E_1] \gamma \delta\), and applying anti-reduction we need to show
\[(w', [N_1]_y [\delta_t], [N_1]_y [\delta_r], [N_1]_y [\delta_r], [\delta_r]) \in E^{-}[B^E] \gamma \delta\]

Where we define \(\gamma' = \gamma, x \mapsto (V_{l_0}, V_{r_0}), y \mapsto (V_{l_1}, V_{r_1})\). By weakening, it is sufficient to show
\[(w', [N_1]_y [\delta_t], [N_1]_y [\delta_r], [N_1]_y [\delta_r], [\delta_r]) \in E^{-}[B^E] \gamma' \delta\]

which follows by inductive hypothesis if we can show
\[(w', \gamma', \delta) \in G^{-}[\Gamma_p, \Gamma^E, \Gamma^E_M, \Gamma^E_N, x : A^E_0, y : A^E_1] \gamma' \delta\]
which follows by definition and monotonicity. □

Lemma G.33. Fun intro
\[
\Gamma^E, x : A^E \vdash M_l \equiv M_r \in B^E; \Gamma^E
\]
\[
\Gamma^E \vdash \lambda x : A_l, M_l \equiv \lambda x : A_r, M_r \in A^E \to B^E; \Gamma^E
\]
We need to show
\[
\top \subseteq \gamma
\]
which follows by definition of the value relation and transitivity of \(\gamma\). Suppose \(w' \supseteq w\) and \((w', V_l, V_r) \in \mathcal{V}^{-}[A^\gamma]_\gamma\). We need to show
\[
(w', \text{force} (\text{thunk} \ ((\lambda x : A_r[\delta_r].M'_r)) \ V_l, \text{force} (\text{thunk} \ ((\lambda x : A_r[\delta_r].M_r[\gamma][\delta_r])) \ V_r)) \in \mathcal{E}^{-}[B^\gamma]_\gamma \delta
\]
By anti-reduction it is sufficient to show
\[
(w', [M_l][\gamma'][\delta_l], [M_r][\gamma'][\delta_r]) \in \mathcal{E}^{-}[B^\gamma]_\gamma \delta
\]
where \(\gamma' = \gamma, x \mapsto (V_l, V_r)\). By monotonicity, we have \((w', \gamma', \delta) \in \mathcal{G}^{-}[\Gamma_p, \Gamma_E]_\gamma\) so the result follows by hypothesis. 

**Lemma G.34**

\[
\begin{align*}
\Gamma^E &\vdash M_l \subseteq M_r : A^\gamma \rightarrow B^\gamma; \Gamma^E_M \\
\Gamma^E, \Gamma^E_N &\vdash N_l \subseteq N_r : A^\gamma; \Gamma^E_N
\end{align*}
\]

\[
\Gamma^E \vdash M_l | N_l \subseteq M_r | N_r : B^\gamma; \Gamma^E_M, \Gamma^E_N
\]

**Proof.** Define \(M'_l = [M_l][\gamma][\delta_l]\), etc. We need to show
\[
(w, f \leftarrow M'_l; x \leftarrow N'_l; \text{force} \ f/f \leftarrow M'_r; x \leftarrow N'_r; \text{force} \ f/x) \in \mathcal{E}^{-}[B^\gamma]_\gamma \delta
\]
We apply monadic bind with \((w, M'_l, M'_r) \in \mathcal{E}^{-}[A^\gamma \rightarrow B^\gamma]_\gamma \delta\) which holds by hypothesis and weakening. Suppose \(w' \supseteq w\), with anti-reduction it is sufficient to show
\[
(w', x \leftarrow N'_l; \text{force} \ V_l, x \leftarrow N'_r; \text{force} \ V_r, x) \in \mathcal{E}^{-}[B^\gamma]_\gamma \delta
\]
where \((w', V_l, V_r) \in \mathcal{V}^{-}[A^\gamma \rightarrow B^\gamma]_\gamma \delta\). Then we apply monadic bind with \((w', N'_l, N'_r) \in \mathcal{E}^{-}[A^\gamma]_\gamma \delta\) which holds by hypothesis, weakening and monotonicity. Suppose \(w'' \supseteq w'\); with anti-reduction it is sufficient to show
\[
(w'', \text{force} \ V_l, V'_l, \text{force} \ V_r, V'_r) \in \mathcal{E}^{-}[B^\gamma]_\gamma \delta
\]
Which follows by definition of the value relation and transitivity of \(\supseteq\). 

**Lemma G.35. Forall intro**

\[
\Gamma^E, X \vdash M_l \subseteq M_r : A^\gamma; \Gamma^E \\
\Gamma^E \vdash \Lambda^X.M_l \subseteq \Lambda^X.M_r : \forall^X.A^\gamma; \Gamma^E
\]

**Proof.** Define \(M'_l = [M_l][\gamma][\delta_l]\). It is sufficient to show
\[
(w, \text{thunk} \ ((\Lambda X.\lambda c_X : \text{Case} \ X.M'_l)); \text{thunk} \ ((\Lambda X.\lambda c_X : \text{Case} \ X.M'_r)) \in \mathcal{V}^{-}[\forall^X.A^\gamma]_\gamma \delta
\]
Given \(w' \supseteq w\), and \(R = \text{Rel}(B_l, B_r)\) and \(w'.\eta = ((\sigma_l, \sigma_r, \{R\}.)_{w'\eta})\), we need to show
\[
(w', \text{force} \ \text{thunk} \ ((\Lambda X.\lambda c_X : \text{Case} \ X.M'_l)) B_l \sigma_l \text{force} \ \text{thunk} \ ((\Lambda X.\lambda c_X : \text{Case} \ X.M'_r)) B_r \sigma_r) \in \mathcal{V}^{-}[A^\gamma]_\gamma \delta'
\]
where \(\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)\) and \(\delta' = \delta, X \mapsto (B_l, B_r, R)\). By anti-reduction it is sufficient to show
\[
(w', [M_l][\gamma'][\delta_l], [M_r][\gamma'][\delta_r]) \in \mathcal{V}^{-}[A^\gamma]_\gamma \delta'
\]
which follows by hypothesis if since \((w', \gamma', \delta') \in \mathcal{G}^{-}[\Gamma_p, \Gamma_E, X]\) by definition and monotonicity. 

**Lemma G.36. Forall elim**

\[
\begin{align*}
\Gamma^E \vdash M_l \subseteq M_r : \forall^X.A^\gamma; \Gamma^E_M &\\
\Gamma^E \vdash B^E : B_l \subseteq B_r \\
\Gamma^E \vdash M_l \{X \equiv B_l\} \subseteq M_r \{X \equiv B_r\} : A^\gamma; \Gamma^E, X \equiv B^E
\end{align*}
\]
PROOF. By definition, instantiation translates as
\[
[M_i \{ X \equiv B_i \}] = f \left< [M_i] \text{; force } f \left[ B_i \right] c_X \right>
\]
Define \( M'_i = [M_i] [\gamma_i] [\delta_i] \), then we need to show
\[
(w, f \left< M'_i \text{; force } f \left[ B_i \right] [\delta_i] y_i(c_X), f \left< M'_i \text{; force } f \left[ B_r \right] [\delta_r] y_r(c_X) \right> \in E^{-}[A^\subseteq] \gamma \delta
\]
We know \((w, M'_i, M') \in E^{-}[\nu^\forall X\cdot A^\subseteq] \gamma \delta\) by hypothesis and weakening (lemma G.16), so we can apply monadic bind. Suppose \( w' \equiv w \), then by anti-reduction it is sufficient to show
\[
(w', \text{force } V_t [B_i] [\delta_i] y_i(c_X), \text{force } V_r [B_r] [\delta_r] y_r(c_X)) \in E^{-}[A^\subseteq] \gamma \delta
\]
where \((w', V_t, V_r) \in \nu^\forall [\nu^\forall X\cdot A^\subseteq] \gamma \delta\). Thus it is sufficient to give some relation \( R \in \text{Rel}_n([B_i] [\delta_i], [B_r] [\delta_r]) \) such that \( w' = (y_i(c_X), y_r(c_X), R) \). By definition of \( G^{-}[\Gamma_p, \Gamma^\forall \cdot X, X \equiv B^\subseteq] \), this holds for \( R = \nu^\forall [B^\subseteq] \gamma \delta \). \( \square \)

**Lemma G.37.**
\[
\Gamma^\subseteq, X \equiv B^\subseteq \equiv M_l \subseteq M_r \in A^\subseteq; \quad \Gamma^\subseteq + B^\subseteq : B_l \subseteq B_r
\]
\[
\Gamma^\subseteq \vdash \text{pack}^\forall (X \equiv B_l, M_l) \equiv \text{pack}^\forall (X \equiv B_r, M_r) \in \nu^\forall \nu^\forall X\cdot A^\subseteq;.
\]

**Proof.** Recall the translation is
\[
[\text{pack}^\forall (X \equiv B_l, M_l)] = \text{ret} (\text{pack}([B_i], \text{thunk } \lambda c_X : \text{Case } [B_i] [M_i]) \text{ as } [\nu^\forall \nu^\forall X_A])
\]
where \( \Gamma^\subseteq, X + A^\subseteq : A_l \subseteq A_r \). So it is sufficient to show
\[
(w, \text{pack}([B_i] [\delta_i], \text{thunk } \lambda c_X : \text{Case } ([B_i] [\gamma_i] [\delta_i]) \text{ as } [\nu^\forall \nu^\forall X_A^\subseteq] [\delta_i], \text{pack}([B_r] [\delta_r], \text{thunk } \lambda c_X : \text{Case } ([B_r] [\delta_r]) [M_r] [\gamma_r] [\delta_r])
\]
For the relation, we pick \( R = \nu^\forall [B^\subseteq] \gamma \delta \). Let \( w' \equiv w \) and \( \sigma_l, \sigma_r \) be seals such that \( w'. \eta \vdash (\sigma_l, \sigma_r, [R] w', j) \). Then we need to show
\[
(w, \text{force } (\text{thunk } \lambda c_X : \text{Case } ([B_i] [\delta_i]) [M_l] [\gamma_l] [\delta_l]) \sigma_l, \text{force } (\text{thunk } \lambda c_X : \text{Case } ([B_r] [\delta_r]) [M_r] [\gamma_r] [\delta_r]))
\]
where
\[
\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)
\]
\[
\delta' = \delta, X \mapsto ([B_i] [\delta_i], [B_r] [\delta_r], R)
\]
By anti reduction this reduces to showing
\[
(w', [M_i] [\gamma_i'] [\delta_i'], [M_r] [\gamma_r'] [\delta_r']) \in E^{-}[A^\subseteq] \gamma' \delta'
\]
which follows by assumption. \( \square \)

**Lemma G.38.**
\[
\begin{array}{c}
\Gamma^\subseteq, \Gamma^\subseteq_M \vdash \Gamma^\subseteq_N \quad \Gamma^\subseteq, \Gamma^\subseteq_M, \Gamma^\subseteq_N \vdash B^\subseteq \\
\Gamma^\subseteq \vdash \text{unpack } (X, x) = M_l; N_l \equiv \text{unpack } (X, x) = M_r; N_r \equiv B^\subseteq; \Gamma^\subseteq_M, \Gamma^\subseteq_N
\end{array}
\]

**Proof.** Recall the translation:
\[
[\text{unpack } (X, x) = M_i; N_i] = p \left< [M_i] \text{; unpack } (X, f) = p; \text{newcase}_X c_X; \quad x \left< (\text{force } f c_X); \quad [N_i] \right>
\]

Let \((w, \gamma, \delta) \in G^{-}[\Gamma_p, \Gamma^\subseteq, \Gamma^\subseteq_M, \Gamma^\subseteq_N]. By assumption (and weakening), we know \((w, [M_i] [\gamma_i] [\delta_i], [M_r] [\gamma_r] [\delta_r]) \in E^{-}[\nu^\forall \nu^\forall X_A^\subseteq]\). We apply monadic bind. Let \( w' \equiv w \) and \((w', \nu^\forall v_i, \nu^\forall v_r) \in \nu^\forall [\nu^\forall \nu^\forall X_A^\subseteq]. \)
Then $V \exists \forall l = \text{pack}(A_{XL}, V_{fl})$ and $V \exists \forall r = \text{pack}(A_{XR}, V_{fr})$ with some associated relation $R \in \text{Rel}[A_{XL}, A_{XR}]$. Then by anti-reduction we need to show

$$(w', \text{newcase}_{A_{XL}} c_X; \text{newcase}_{A_{XR}} c_X; \cdots) \in \mathcal{E}^\perp[B^\perp]y\delta$$

$$x \leftarrow (\text{force } V_{fl} c_X); x \leftarrow (\text{force } V_{fr} c_X);$$

$$[N_l][y_l][\delta_l]; [N_r][y_r][\delta_r]$$

Define

$$w'' = (w', j, (w'.\Sigma_l, A_{XL}), (w'.\Sigma_r, A_{XR}), w'.\eta \equiv (A_{XL}, A_{XR}, [R]_{w, j}))$$

$$\gamma' = \gamma, c_X \mapsto (w'.\Sigma_l.size, w'.\Sigma_r.size)$$

$$\delta' = \delta, X \mapsto (A_{XL}, A_{XR}, R)$$

Then by anti-reduction it is sufficient to show

$$(w'', (x \leftarrow (\text{force } V_{fl} \sigma_l); [N_l][y_l'][\delta_l']), (x \leftarrow (\text{force } V_{fr} \sigma_r); [N_r][y_r'][\delta_r']))) \in \mathcal{E}^\perp[B^\perp]y'\delta'$$

By assumption (and monotonicity), $(w'', \text{force } V_{fl} \sigma_l, \text{force } V_{fr} \sigma_r) \in \mathcal{E}^\perp[A^\perp]y'\delta'$, so we can apply monadic bind. Let $w''' \equiv w''$ and $(w'''', V_{fl}, V_{fr}) \in \mathcal{V}^\perp[A^\perp]y''\delta''$. Then after anti-reduction we need to show

$$(w''', [N_l][y''''][\delta_l'], [N_r][y''''][\delta_r']) \in \mathcal{E}^\perp[B^\perp]y''\delta''$$

where $y'' = y'$, $x \mapsto (V_{fl}, V_{fr})$. By weakening this is equivalent to

$$(w''', [N_l][y''''][\delta_l'], [N_r][y''''][\delta_r']) \in \mathcal{E}^\perp[B^\perp]y''\delta''$$

which follows by assumption.

\[ G.3 \text{ Proof of Graduality} \]

\textbf{Lemma G.39.} If $C : (\Gamma \vdash \cdots : A; \Gamma_o) \Rightarrow (\Gamma' \vdash \cdots : B; \Gamma'_o)$ and $\Gamma \vdash M_l \sqsubseteq M_r \subseteq A; \Gamma_o$, then $\Gamma' \vdash C[M_l] \sqsubseteq C[M_r] \subseteq B; \Gamma'_o$.

\textbf{Proof.} By induction on $C$, using a corresponding compatibility lemma in each case.

\[ G.40 \text{ Definition of Contextual Error Approximation} \]

\textbf{Definition G.40.} Let $\Gamma \vdash M_l : A; \Gamma_M$ and $\Gamma \vdash M_2 : A; \Gamma_M$. Then we say $M_l$ (contextually) error approximates $M_2$, written $\Gamma \vdash M_l \sqsubseteq^{ctx} M_2 \in A; \Gamma_M$ when for any context $C : (\Gamma \vdash A; \Gamma_M) \Rightarrow (\cdots \vdash B; \cdot)$, all of the following hold:

1. If $C[M_l] \uparrow$ then $C[M_2] \uparrow$
2. If $C[M_l] \downarrow V_1$ then there exists $V_2$ such that $C[M_2] \downarrow V_2$.

\textbf{Definition G.41.} If $\Gamma \vdash M_1 : A; \Gamma_M$ and $\Gamma \vdash M_2 : A; \Gamma_M$, then $M_1$ and $M_2$ are contextually equivalent, $\Gamma \vdash M_1 \approx^{ctx} M_2 \in A; \Gamma_M$, when for any context $C : (\Gamma \vdash A; \Gamma_M) \Rightarrow (\cdot \vdash B; \cdot)$, both diverge $C[M_1], C[M_2] \downarrow$, or error $C[M_1], C[M_2] \downarrow \cup$ or terminate successfully $C[M_1] \downarrow V_1, C[M_2] \downarrow V_2$.

From syntactic type safety, it is clear that mutual error approximation implies equivalence:

\textbf{Lemma G.42.} If $\Gamma \vdash M_1 \sqsubseteq^{ctx} M_2 : A; \Gamma_M$ and $\Gamma \vdash M_2 \sqsubseteq^{ctx} M_1 : A; \Gamma_M$, then $\Gamma \vdash M_1 \approx^{ctx} M_2 : A; \Gamma_M$.

\textbf{Proof.} The first two cases are direct. For the third case, we know by type safety that there are only 3 possibilities for a closed term’s behavior: $C[M_1] \uparrow$, $C[M_1] \downarrow V_1$ or $C[M_1] \downarrow \cup$. If $C[M_1] \downarrow \cup$, then it is not the case that $C[M_1] \uparrow$ or $C[M_1] \downarrow V_1$, but by the first two cases that means that it is not the case that $C[M_2] \uparrow$ or $C[M_1] \downarrow V$, so it must be the case that $C[M_2] \downarrow \cup$. The opposite direction follows by symmetry.
To prove the soundness of the logical relation with respect to error approximation, we first need to construct for each step-index \( n \) a world \( w^\gamma_p(n) \) to hold the invariants for the cases generated in the preamble of a whole program, a substitution \( \delta^\gamma_p \) to give the relational interpretation of \( \gamma_p \), and a "binary" version \( \gamma^2_p \) of the preamble substitution \( \gamma_p \).

**Definition G.43 (Preamble World, Relational Substitution).**

\[
\eta^\gamma_p(n) = \emptyset \sqcup (\mathbb{B}, \mathbb{B}, \mathcal{V}^{-}[\mathbb{B}][\emptyset]) \sqcup (\text{OSum} \times \text{OSum}, \text{OSum} \times \text{OSum}, \mathcal{V}^{-}[\emptyset \times \emptyset][\emptyset])
\]
\[
\square \quad (U(\text{OSum} \rightarrow \text{OSum}), U(\text{OSum} \rightarrow \text{OSum}), \mathcal{V}^{-}[\emptyset \rightarrow \emptyset][\emptyset])
\]
\[
\square \quad (\exists X. U(\text{Case} \ X \rightarrow \text{FSum}), \exists X. U(\text{Case} \ X \rightarrow \text{FSum}), \mathcal{V}^{-}[\exists^{\prime} X. \emptyset][\emptyset])
\]
\[
\square \quad (U(\forall X. \text{Case} \ X \rightarrow \text{FSum}), \forall X. U(\text{Case} \ X \rightarrow \text{FSum}), \mathcal{V}^{-}[\forall^{\prime} X. \emptyset][\emptyset])
\]
\[
w^\gamma_p(n) = (n, \Sigma_p, \Sigma_p, \eta^\gamma_p(n))
\]
\[
\delta^\gamma = \emptyset, \text{Bool} \mapsto (\mathbb{B}, \mathbb{B}, \mathcal{V}^{-}[\mathbb{B}][\emptyset]), \text{Times} \mapsto (\text{OSum} \times \text{OSum}, \text{OSum} \times \text{OSum}, \mathcal{V}^{-}[\emptyset \times \emptyset][\emptyset]),
\]
\[
\text{Fun} \mapsto (U(\text{OSum} \rightarrow \text{OSum}), U(\text{OSum} \rightarrow \text{OSum}), \mathcal{V}^{-}[\emptyset \rightarrow \emptyset][\emptyset]),
\]
\[
\text{Ex} \mapsto (\exists X. U(\text{Case} \ X \rightarrow \text{FSum}), \exists X. U(\text{Case} \ X \rightarrow \text{FSum}), \mathcal{V}^{-}[\exists^{\prime} X. \emptyset][\emptyset]),
\]
\[
\text{All} \mapsto (U(\forall X. \text{Case} \ X \rightarrow \text{FSum}), \forall X. U(\text{Case} \ X \rightarrow \text{FSum}), \mathcal{V}^{-}[\forall^{\prime} X. \emptyset][\emptyset])
\]

\[
\gamma^2_p(x) = (\gamma_p(x), \gamma_p(x))
\]

The crucial property is that these together satisfy \( G^-[\Gamma_p] \):

**Lemma G.44 (Validity of Preamble World).** For every \( n \), \((w^\gamma_p(n), \gamma^2_p, \delta^\gamma_p) \in G^-[\Gamma_p]\).

**Proof.** Clear by definition. \( \square \)

**Lemma G.45.**

\[
\Gamma \vdash M_l \subseteq M_2 \in A; \Gamma_M
\]

**Proof.** Let \( C \) be an appropriately typed closing context. By the congruence Lemma G.39 we know

\[
\cdot \vdash C[M_l] \subseteq C[M_2] \in B; \cdot
\]

By Lemma G.44, we know that

\[
(w^\gamma_p(n), [C[M_l]][\gamma_p], [C[M_2]][\gamma_p]) \in E^-[B^{-}]\gamma_p^2\delta^\gamma_p
\]

(noting that \([C[M_l]][\gamma_p][\delta^\gamma] = [C[M_l]][\gamma_p])).

- If \( C[M_l] \uparrow \), then by the simulation Theorem F.16 we know \( \Sigma_p, [C[M_l]][\gamma_p] \uparrow \). Then, by adequacy for divergence Corollary F.18, to show \( C[M_2] \uparrow \), it is sufficient to show that \( \Sigma_p, [C[M_2]][\gamma_p] \uparrow \). We will show that for every \( n \), \( \Sigma_p, [C[M_2]][\gamma_p] \uparrow \rightarrow^n \Sigma_n, N_n \) for some \( \Sigma_n, N_n \). We know

\[
(w^\gamma_p(n), [C[M_l]][\gamma_p], [C[M_2]][\gamma_p]) \in E^-[B^{-}]\gamma \delta
\]

we proceed by the cases of \( E^-[B^{-}]\gamma \delta \)

- If \( w^\gamma_p(n, \Sigma_r, [C[M_2]][\gamma_p]) \rightarrow^{n-j} \) we are done because \( w^\gamma_p(n, \Sigma_r = \Sigma_p \) and \( w^\gamma_p(n, j = n) \).

- If \( w^\gamma_p(n, \Sigma_l, [C[M_1]][\gamma_p]) \rightarrow^* \Sigma', U \) we have a contradiction because \( \Sigma_p, [C[M_1]][\gamma_p] \uparrow \).

- If \( w^\gamma_p(n, \Sigma_l, [C[M_1]][\gamma_p]) \rightarrow^* \Sigma', \text{ret } V_1 \), we also have a contradiction for the same reason.

The two casts cases are precisely dual. Things to prove:

- If \( C[M_1] \Downarrow V_1 \) then by simulation we know \( \Sigma_p, [C[M_1]] \mapsto^* \Sigma', \text{ret } V'_1 \) for some \( n, V_1 \). Furthermore, to show \( C[M_2] \Downarrow V_2 \) it is sufficient by simulation to show \( \Sigma_p, [C[M_2]] \mapsto^* \text{ret } V'_2 \). We know 

\[
(w_p^\prec(n), [C[M_1]][\gamma_p], [C[M_2]][\gamma_p]) \in \mathcal{E} [B^\prec] \gamma \delta \]

we proceed by the cases of \( \mathcal{E} [B^\prec] \gamma \delta \).

- If \( w_p^\prec(n).\Sigma_t, [C[M_1]][\gamma_p] \mapsto w_p^\prec(n).\Sigma_t, [C[M_1]][\gamma_p] \mapsto^* U \), this contradicts the fact that \( \Sigma_p, [C[M]] \mapsto^n \Sigma', \text{ret } V'_1 \).

- Otherwise, \( w_p^\prec(n).\Sigma_t, [C[M_2]][\gamma_p] \mapsto^* \Sigma', \text{ret } V'_2 \), so the result holds because \( w_p^\prec(n).\Sigma_t = \Sigma_p \).

Finally we prove Lemma G.39 that states that semantic term precision is a congruence.

**Proof.** By induction over \( C \). In ever non-cast case we use the corresponding compatibility rule. The two cases are precisely dual.

- If \( C = \langle A^E \rangle \uparrow C' \), where \( \Gamma, \Gamma'_o \vdash A^E : A_l \subseteq A_r \), then we need to show 

\[
\Gamma' \vdash \langle A^E \rangle \uparrow C'[M_l] \subseteq \langle A^E \rangle \uparrow C'[M_r] \subseteq A_r; \Gamma'_o
\]

First we use the upcast-left compatibility rule, meaning we need to show 

\[
\Gamma' \vdash C'[M_l] \subseteq \langle A^E \rangle \uparrow C'[M_r] \subseteq A_r; \Gamma'_o
\]

Next, we use the upcast-right compatibility rule, meaning we need to show 

\[
\Gamma' \vdash C'[M_l] \subseteq C'[M_r] \subseteq A_l; \Gamma'_o
\]

which follows by inductive hypothesis.

- If \( C = \langle A^E \rangle \downarrow C' \), where \( \Gamma, \Gamma'_o \vdash A^E : A_l \subseteq A_r \), then we need to show 

\[
\Gamma' \vdash \langle A^E \rangle \downarrow C'[M_l] \subseteq \langle A^E \rangle \downarrow C'[M_r] \subseteq A_r; \Gamma'_o
\]

First we use the downcast-right compatibility rule, meaning we need to show 

\[
\Gamma' \vdash \langle A^E \rangle \downarrow C'[M_l] \subseteq C'[M_r] \subseteq A_r; \Gamma'_o
\]

Next, we use the downcast-left compatibility rule, meaning we need to show 

\[
\Gamma' \vdash C'[M_l] \subseteq C'[M_r] \subseteq A_r; \Gamma'_o
\]

which follows by inductive hypothesis.

**G.4 Free Theorems**

Most of the free theorems are stated in terms of contextual equivalence. We define contextual equivalence of PolyG\(^v\) terms to mean contextual equivalence of their elaborations into PolyC\(^v\) terms. By lemma G.42, we can prove a contextual equivalence \( M_1 \simeq_{ctx} M_2 \) by proving contextual error approximation both ways (\( M_1 \simgleq_{ctx} M_2 \) and \( M_2 \simgleq_{ctx} M_1 \)). We can prove contextual error approximation by proving logical relatedness by soundness of the logical relation for open terms (Lemma G.45), which is defined in terms of the two logical relations \( \triangleleft, \triangleright \), giving us technically 4 things to prove: \( \subseteq, \simgleq, \triangleright \) and \( \triangleleft \). However, these cases are all very similar so we show only one case and the other cases follow by essentially symmetric arguments.
Standard Free Theorems.

\[ M : \forall \forall X.X \rightarrow X \quad V_A : A \quad V_B : B \]
\[
\lambda_\_ : ? \text{unseal}_X(M\{X \equiv A\}(\text{seal}_X V_A)) \approx^\text{ctx} \lambda_\_ : ? \text{let } y = M\{X \equiv B\} V_B; V_A
\]

Theorem G.46.

Proof. There are 4 cases: \( \subseteq, \subseteq, > \\cap \subseteq \) but they are all by a similar argument. Let \((w, y, \delta) \in \mathcal{G}^{-\Gamma_p, \Gamma}\). We need to show

\[
(w, \text{ret thunk } \lambda_\_ : ?, \text{ret thunk } \lambda_\_ : ?) \in \mathcal{E}^{-A}[y\delta
\text{newcase}_{[A][\delta]} c_X;
[\text{unseal}_X(M\{X \equiv A\}(\text{seal}_X V_A))] [y\delta] [\text{let } y = M\{X \equiv B\} V_B; V_A][yr][\delta_r]
\]

Given some \(w' \supseteq w\) and (irrelevant) values \((w', V_{dum1}, V_{dum2})\), we need to show (after applying anti-reduction) that

\[
(w', \text{newcase}_{[A][\delta]} c_X;)
\text{newcase}_{[A][\delta]} c_X;
[\text{unseal}_X(M\{X \equiv A\}(\text{seal}_X V_A))] [y\delta] [\text{let } y = M\{X \equiv B\} V_B; V_A][yr][\delta_r]
\]

Each side allocates a new case and we need to pick a relation with which to instantiate it. By Lemmas F.10 and F.2, we have that \([V_A][y\delta] [\text{let } y = M\{X \equiv B\} V_B; V_A][yr][\delta_r]\)

There are 4 cases:

1. \(\text{newcase}_{[A][\delta]} c_X;\)
2. \(\text{newcase}_{[A][\delta]} c_X;\)
3. \(\text{newcase}_{[A][\delta]} c_X;\)
4. \(\text{newcase}_{[A][\delta]} c_X;\)

Then we define \(w'' \equiv w\) to be the world extended with \([R]_{w', j}:\)

\[
w'' = (w', j, w'.\Sigma_l, [A][\delta_l], w'.\Sigma_r[\delta_r], w'.\eta \uplus (w'.\Sigma_l.size, w'.\Sigma_r.size, [R]_{w', j})
\]

Then clearly \(w'' \supseteq w\) and

\[
w'.\Sigma_l, \text{newcase}_{[A][\delta_l]} c_X;
\implies^0 w''.\Sigma_l, [\text{unseal}_X(M\{X \equiv A\}(\text{seal}_X V_A))] [y\delta]
\]

and similarly for the right hand side:

\[
w'.\Sigma_r, \text{newcase}_{[A][\delta_r]} c_X;
\implies^0 w''.\Sigma_r, [\text{let } y = M\{X \equiv B\} V_B; V_A][y\delta]
\]

where \(\gamma' = \gamma, c_X \mapsto (w'.\Sigma_l.size, w'.\Sigma_r.size).\) Expanding definitions (and noting that \(c_X\) is free in \(M\)), we need to show

\[
(w'', S_l[[M][y\delta]], S_r[[M][\gamma]] [\delta_{\gamma}]) \in \mathcal{E}^{-[A][\gamma][\delta_{\gamma}]
\]

where

\[
S_l = f \leftarrow (z \leftarrow \bullet; \text{force } z([A][\delta_l]) \gamma'(c_X));
\]
\[
x \leftarrow [V_A][y\delta];
\]
\[
\text{force } f \leftarrow
\]

and

\[
S_r = y \leftarrow (f \leftarrow (z \leftarrow \bullet; \text{force } z([B][\delta_l]) \gamma'(c_X)));
\]
\[
x \leftarrow [V_B][y\delta];
\]
\[
\text{force } f \leftarrow
\]

We apply monadic bind. Let \( w''' \equiv w'' \) and \((w''', V_l, V_r) \in \mathcal{V}^-[\forall X. X \to X] y \delta \). After anti-reduction we need to show

\[
(w''', f \leftarrow \text{force } V_l([A][[\delta_l]]) y'_1(c_X); S'_r \leftarrow \text{force } V_r([B][[\delta_l]]) y'_2(c_X); x \leftarrow [V_A][y][\delta_l]) \in \mathcal{E}^-[A] y \delta
\]

where

\[
S'_r = y \leftarrow \bullet; [V_A][y][\delta_l] \in \mathcal{E}^-[A] y \delta
\]

By weakening it is equivalent to showing the terms are in the relation \( \mathcal{E}^-[A] y \delta' \) where \( \delta' = ([A][\delta_l], [B][\delta_l], R) \). By definition of \( \mathcal{V}^-[\forall X. X \to X] \), we have

\[
(w''', \text{force } V_l([A][\delta_l]) y'_1(c_X), \text{force } V_r([B][\delta_l]) y'_2(c_X)) \in \mathcal{E}^-[X \to X] y \delta'
\]

So we apply monadic bind. Let \( w''''' \equiv w''' \) and let \((w''''', V_{fl}, V_{fr}) \in \mathcal{V}^-[X \to X] y \delta' \). Applying anti-reduction, we need to show

\[
(w''''', \text{force } V_{fl} V_{Al}, y \leftarrow \text{force } V_{fr} V_{Br}; [V_A][y][\delta_l]) \in \mathcal{E}^-[A] y \delta'
\]

By definition of \( \mathcal{V}^-[X \to X] \), \((w''''', \text{force } V_{fl} V_{Al}, \text{force } V_{fr} V_{Br}) \in \mathcal{E}^-[X] y \delta' \) if \((w''''', V_{Al}, V_{Br}) \in \mathcal{V}^-[X] y \delta' \) which holds because \((w'''', V_{Al}, V_{Br}) \in R \). By Lemmas F.10 and F.2, we have that \([V_A][y][\delta_l] \mapsto^0 \text{ret } V_{Al} \) for some \( V_{Al} \). Applying monadic bind a final time we get \( w_{fin} \equiv w'''' \) and \((w_{fin}, V'_{Al}, V'_{Br}) \in \mathcal{V}^-[X] y \delta' \) and after anti-reduction need to show that

\[
(w_{fin}, \text{ret } V'_{Al}, \text{ret } V_{Al}) \in \mathcal{E}^-[A] y \delta'
\]

By weakening this is equivalent to

\[
(w_{fin}, \text{ret } V'_{Al}, \text{ret } V_{Al}) \in \mathcal{E}^-[A] y \delta
\]

By the definition of \( \mathcal{V}^-[X] y \delta' \), \( V'_{Al} = V_{Al} \), so this follows by reflexivity; specifically that \( \Gamma \vdash V_A \equiv V_A \in A; \ldots \). This is because, by the definition of \( \mathcal{E}^-[A] y \delta \), since \( V_A[y][\delta_l] \mapsto^0 \text{ret } V_{Al} \) and \( V_A[y][\delta_l] \mapsto^0 \text{ret } V_{Al} \), those values are related.

\[\square\]

Theorem G.47.

\[
M : \forall X. \forall Y. (X \times Y) \to (Y \times X) \quad V_A : A 
V_B : B
\]

\[\lambda \_ : \text{?let } (y, x) = (M(X \equiv A)\{Y \equiv B\} (\text{seal}_X V_A, \text{seal}_Y V_B)); (\text{unseal}_Y x, \text{unseal}_x y) \]

\[\lambda \_ : \text{?let } (y, x) = (M(X \equiv B)\{Y \equiv A\} (\text{seal}_X V_B, \text{seal}_Y V_A)); (\text{unseal}_Y y, \text{unseal}_x x) \]

**Proof.** We show the \( \sqsubseteq \) case, the \( \sqsupseteq \) case is symmetric. Let \((w, y, \delta) \in \mathcal{G}^-[\Gamma_F, \Gamma] \). Define the following terms

\[
N_l = [\text{let } (y, x) = (M(X \equiv A)\{Y \equiv B\} (\text{seal}_X V_A, \text{seal}_Y V_B)); (\text{unseal}_x x, \text{unseal}_Y y)]
\]

\[
N_r = [\text{let } (y, x) = (M(X \equiv B)\{Y \equiv A\} (\text{seal}_X V_B, \text{seal}_Y V_A)); (\text{unseal}_Y y, \text{unseal}_x x)]
\]

Then we need to show for any \( w_1 \equiv w \) that

\[
(w_1, \text{newcase}_A c_X; \text{newcase}_B c_Y; N_l[y][\delta_l], \text{newcase}_B c_X; \text{newcase}_A c_Y; N_r[y][\delta_r]) \in \mathcal{E}^-[A \times B] y \delta
\]
By Lemmas F.10 and F.2, we have that \([V_A][y_1][\delta_1] \rightarrow^0 \text{ret } V_{A1}\) and \([V_B][y_1][\delta_1] \rightarrow^0 \text{ret } V_{B1}\) for some \(V_{A1}\) and \(V_{B1}\). Define

\[
A_l = [A] [\delta_l]
\]
\[
B_l = [B] [\delta_l]
\]
\[
R_X = \{(w, V_{A1}, V_{B1}) \in \text{Atom}[A_l, B_r] \mid w \sqsupseteq w_1\}
\]
\[
R_Y = \{(w, V_{B1}, V_{A1}) \in \text{Atom}[B_l, A_r] \mid w \sqsupseteq w_1\}
\]
\[
w_2 = (w_1, l, (w_1, \Sigma_l, A_l, B_l), (w_1, \Sigma_r, B_r, A_r), (w_1, q) \sqsupseteq (A_l, B_r, [R_X]_{w_1, j}) \sqsupseteq (B_l, A_r, [R_Y]_{w_1, j}))
\]
\[
\sigma_{Xl} = w_1, \Sigma_l, \text{size}
\]
\[
\sigma_{Yl} = w_1, \Sigma_l, \text{size} + 1
\]
\[
\gamma' = y, c_X \mapsto (\sigma_{Xl}, \sigma_{Xr}), c_Y \mapsto (\sigma_{Yl}, \sigma_{Yr})
\]
\[
\delta' = \delta, X \mapsto (A_l, B_r, R_X), Y \mapsto (B_l, A_r, R_Y)
\]

Then it is sufficient to show that

\[
(w_2, N_l[y_1'][\delta_1'], N_r[y_1'][\delta_1']) \in \mathcal{E}^- [A \times B] Y \delta
\]

which is equivalent by weakening to

\[
(w_2, N_l[y_1'][\delta_1'], N_r[y_1'][\delta_1']) \in \mathcal{E}^- [A \times B] Y' \delta'
\]

Next, note that by reflexivity,

\[
(w_2, [M][y_1'][\delta_1'], [M][y_1'][\delta_1']) \in \mathcal{E}^- [\forall^\forall X. \forall^\forall Y. (X \times Y) \rightarrow (Y \times X)] Y' \delta'
\]

and

\[
N_l = [S_l][[M]][y_1'][\delta_1']
\]

for stacks

\[
S_l = \text{let } (y, x) = (\bullet \{X \equiv A\} \{Y \equiv B\} (\text{seal}_X V_A, \text{seal}_Y V_B)); (\text{unseal}_x X, \text{unseal}_y Y)
\]
\[
S_r = \text{let } (y, x) = (\bullet \{X \equiv B\} \{Y \equiv A\} (\text{seal}_X V_B, \text{seal}_Y V_A)); (\text{unseal}_y Y, \text{unseal}_x X)
\]

So we can apply monadic bind. Let \(w_3 \sqsupseteq w_2\) and \((w_3, V_{x'l}, V_{y'r}) \in \mathcal{E}^- [\forall^\forall X. \forall^\forall Y. (X \times Y) \rightarrow (Y \times X)] Y' \delta'\). Then by anti-reduction it is sufficient to show

\[
(w_3, [S_{l'}][\text{force } V_{x'l}] A_l \sigma_{Xl}, [S_{l'}'][\text{force } V_{y'r}] B_r \sigma_{Xr}) \in \mathcal{E}^- [A \times B] Y' \delta'
\]

for stacks

\[
S_l' = \text{let } (y, x) = (\bullet \{X \equiv B\} (\text{seal}_X V_A, \text{seal}_Y V_B)); (\text{unseal}_x X, \text{unseal}_y Y)
\]
\[
S_r' = \text{let } (y, x) = (\bullet \{X \equiv A\} (\text{seal}_X V_B, \text{seal}_Y V_A)); (\text{unseal}_y Y, \text{unseal}_x X)
\]

By definition, we know \((w_3, \text{force } V_{x'l} A_l \sigma_{Xl}, \text{force } V_{y'r} B_r \sigma_{Xr}) \in \mathcal{E}^- [\forall^\forall Y. (X \times Y) \rightarrow (Y \times X)] Y' \delta', so we can apply monadic bind again. Let \(w_4 \sqsupseteq w_3\) and \((w_4, V_{x'l'} V_{y'r'}) \in \mathcal{E}^- [\forall^\forall Y. (X \times Y) \rightarrow (Y \times X)] Y' \delta'\). Then after anti-reduction we need to show

\[
(w_4, [S_{l''}][\text{force } V_{x'l''} B_l \sigma_{Yl}], [S_{l'''}][\text{force } V_{y'r'} A_r \sigma_{Yr}]) \in \mathcal{E}^- [A \times B] Y' \delta'
\]

for stacks

\[
S_l'' = \text{let } (y, x) = (\bullet (\text{seal}_X V_A, \text{seal}_Y V_B)); (\text{unseal}_x X, \text{unseal}_y Y)
\]
\[
S_r'' = \text{let } (y, x) = (\bullet (\text{seal}_X V_B, \text{seal}_Y V_A)); (\text{unseal}_y Y, \text{unseal}_x X)
\]
Similarly to before, we know (w₅, force V⁺ᵥₜ (Vᵥₜ, A, σᵥₜ) ∈ E⁻[(X × Y) → (Y × X)] Y′δ′) so we can apply monadic bind again. Let w₅ ⊇ w_r and (w₅, Vᵥₙ, Vₛ) ∈ V⁻[(X × Y) → (Y × X)] Y′δ′. Next, note that

\[
\begin{aligned}
&[(\text{seal}_X Vₐ, \text{seal}_Y V₇)] [y'_r] [δ'_r] \mapsto 0 \operatorname{ret} (Vₐ, V₇) \\
&[(\text{seal}_X V₇, \text{seal}_Y Vₐ)] [y'_r] [δ'_r] \mapsto 0 \operatorname{ret} (V₇, Vₐ)
\end{aligned}
\]

So by anti-reduction we need to show

\[(w₅, [S''][\text{force } V⁺ᵥₜ (Vₐ, V₇)], [S''][\text{force } V⁺ᵥₙ (V₇, Vₐ)]) ∈ E⁻[A × B] Y′δ′\]

where

\[
\begin{aligned}
S'' = \text{let } (y, x) = •; (\text{unseal}_X x, \text{unseal}_Y y) \\
S''' = \text{let } (y, x) = •; (\text{unseal}_Y y, \text{unseal}_X x)
\end{aligned}
\]

To show that

\[(w₅, \text{force } V⁺ᵥₜ (Vₐ, V₇), \text{force } V⁺ᵥₙ (V₇, Vₐ)) ∈ E⁻[Y × X] Y′δ′\]

It is sufficient to show

\[(w₅, (Vₐ, V₇), (V₇, Vₐ)) ∈ E⁻[X × Y] Y′δ′\]

which follows because (w₅, Vₐ, V₇) ∈ Rₓ and (w₅, V₇, Vₐ) ∈ Rᵧ. Then we can apply monadic bind a final time. Let w₅ₗ ∈ w₅ and (w₅ₗ, Vₕ, V₉) ∈ V⁻[Y × X] Y′δ′. Then we need to show

\[(w₅ₗ, [S''][\text{ret } Vₙ], [S''][\text{ret } V₉]) ∈ E⁻[A × B] Y′δ′\]

First, by definition, it must be the case that Vₙ = (Vₙ, Vₐ) and V₉ = (Vₐ, V₇). Then by anti-reduction we need to show

\[(w₅ₗ, \text{ret } (Vₐ, V₇), \text{ret } (V₇, Vₐ)) ∈ E⁻[A × B] Y′δ′\]

which means we need to show

\[(w₅ₗ, Vₐ, V₇) ∈ V⁻[A] Y′δ′\]

and

\[(w₅ₗ, V₇, Vₐ) ∈ V⁻[B] Y′δ′\]

which follows by reflexivity and monotonicity.

\[\square\]

**Lemma G.48.**

\[
\begin{aligned}
\text{pack}^s(X ≡ B, (\text{seal}_X \text{true}, (\text{NOT}, λx : X.\text{unseal}_X x))) \\
\approx \text{ctx} \operatorname{pack}^s(X ≡ B, (\text{seal}_X \text{false}, (\text{NOT}, λx : X.\text{NOT}(\text{unseal}_X x))))
\end{aligned}
\]

**Proof.** We do the Ξ~ case, the ~Ξ case is symmetric. Let \((w, y, δ) ∈ G⁻[Γ_p]\). The goal reduces to showing

\[
\begin{aligned}
\left(\begin{array}{l}
\text{pack}(\text{thunk } λc_X : \text{Case } X, [(\text{seal}_X \text{true}, (\text{NOT}, (λx : X.\text{unseal}_X x)))]), \\
\text{pack}(\text{thunk } λc_X : \text{Case } X, [(\text{seal}_X \text{false}, (\text{NOT}, (λx : X.\text{NOT}(\text{unseal}_X x)))]))
\end{array}\right) ∈ V⁻[Γ_p] \left(\begin{array}{l}
\exists y' X X × ((X → X) × (X → B)) Y δ
\end{array}\right)
\]

The relation we pick is \( R = \{(w', \text{true, false}) \in \text{Atom}([\mathbb{B}, \mathbb{B}]) \cup \{(w', \text{false, true}) \in \text{Atom}([\mathbb{B}, \mathbb{B}])\} \) Then we need to show for any future \( w' \ni w \) and \( w' \ni (\sigma_l, \sigma_r, [R]_{w', j}) \), that

\[
\begin{align*}
&\left( w', \right. \\
&\quad (\text{force thunk } \lambda \chi \mathcal{X} : \text{Case } X.\left[ \left( \text{seal}_X \text{true, } \text{NOT, } \lambda \chi : X.\text{unseal}_X \chi \right) \right] \sigma_l), \\
&\left( \text{force thunk } \lambda \chi \mathcal{X} : \text{Case } X.\left[ \left( \text{seal}_X \text{true, } \text{NOT, } \lambda \chi : X.\text{NOT} \left( \text{unseal}_X \chi \right) \right) \right] \sigma_r) \\
&\in \mathcal{E}^{-}[X \times (X \rightarrow X) \times (X \rightarrow \mathbb{B})] \psi' \delta'
\end{align*}
\]

where

\[
\begin{align*}
\psi' &= \psi, \epsilon_X \mapsto (\sigma_l, \sigma_r) \\
\delta' &= \delta, X \mapsto (\mathbb{B}, \mathbb{B}, R)
\end{align*}
\]

And after applying anti-reduction, by monadic bind, we need to show the following 3 things:

\[
\begin{align*}
&\left( w', \text{true, false} \right) \in \mathcal{V}^{-}[X] \psi' \delta' \\
&\left( w', \text{NOT} [\psi'] \sigma_\gamma \sigma_\delta \right) \in \mathcal{E}^{-}[X \rightarrow X] \psi' \delta'
\end{align*}
\]

(1) First, \((w', \text{true, false}) \in \mathcal{V}^{-}[X] \psi' \delta'\) follows directly from the definition of \( \delta_R (X) = R \).

(2) Second, let \( w'' \ni w' \) and \((w'', V_l, V_r) \in \mathcal{V}^{-}[X] \psi' \delta'\). Then we need to show

\[
\begin{align*}
&\left( w'', \text{force } V_{\text{NOT}} [\psi'_l] [\gamma'_l] V_l, \text{force } V_{\text{NOT}} [\psi'_r] [\gamma'_r] V_r \right) \in \mathcal{E}^{-}[X] \psi' \delta'
\end{align*}
\]

where \([\text{NOT}] = \text{ret } V_{\text{NOT}}\). There are two cases: either \( V_l = \text{true} \) and \( V_r = \text{false} \) or vice-versa. In either case, NOT swaps the two values and the result holds.

(3) Finally, let \( w'' \ni w' \) and \((w'', V_l, V_r) \in \mathcal{V}^{-}[X] \psi' \delta'\). Then we need to show,

\[
\begin{align*}
&\left( w'', \text{force } V_{f_1} V_l, \text{force } V_{f_r} V_r \right) \in \mathcal{E}^{-}[\mathbb{B}] \psi' \delta'
\end{align*}
\]

where \([\lambda \chi : X.\text{unseal}_X \chi] = \text{ret } V_{f_1} \) and \([\lambda \chi : X.\text{NOT } (\text{unseal}_X \chi)] = \text{ret } V_{f_r}\). By definition of \( R \), either \( V_l = \text{true} \) and \( V_r = \text{false} \) or vice-versa. In either case, both sides evaluate to \( \text{ret } V_l \), and we need to show

\[
\begin{align*}
&\left( w'', V_l, V_r \right) \in \mathcal{V}^{-}[\mathbb{B}] \psi' \delta'
\end{align*}
\]

which follows by definition.

\[ \square \]

**Free Theorems with the Dynamic Type.**

**Theorem G.49.** *If \( M : \forall X. ? \rightarrow X \) and \( \vdash A \) and \( \vdash V : ? \), then either*

\[
\left( (\lambda_\_ : ?.\text{unseal}_X (M\{X \equiv A\} V)) \text{true} \right)^+ \]

*or*

\[
\left( (\lambda_\_ : ?.\text{unseal}_X (M\{X \equiv A\} V)) \text{true} \right)^+ \rightarrow^* \mathbb{U}
\]

**Proof.** Define \( \mathcal{N} = (\lambda_\_ : ?.\text{unseal}_X (M\{X \equiv A\} V)) \text{true} \) By the adequacy lemmas ??,??, it is sufficient to show that for every \( n \in \mathbb{N} \), either

\[
\Sigma_p \triangleright \left[ \left( \mathcal{N} \right)^+ \right] [\gamma_p] \rightarrow^n
\]

*or*

\[
\Sigma_p \triangleright \left[ \left( \mathcal{N} \right)^+ \right] [\gamma_p] \rightarrow^\leq n \mathbb{U}
\]

Unraveling definitions, we get

\[
\Sigma_p ; \left[ \left( (\lambda_\_ : ?.\text{unseal}_X (M\{X \equiv A\} V)) \text{true} \right)^+ \right] \rightarrow^* \Sigma_p ; \text{newcase}_{A} \epsilon_X ; \left[ \left( (M\{X \equiv A\} V)^+ \right) [\gamma_p] \right] \]

*\rightarrow \Sigma ; \left[ \left( (M\{X \equiv A\} V)^+ \right) [\gamma_p, \epsilon_X \mapsto \sigma] \right]
\]

where \( \sigma = \Sigma_\cdot \text{size} \), \( \Sigma = \Sigma_\cdot \sigma \).

Next, define \( y = \gamma_p, c_X \mapsto (\sigma, \sigma) \), \( w = (w^\rightarrow (n), j, \Sigma, \Sigma, w, \eta \mapsto ([A], [A], \emptyset)) \) and \( \delta = \delta^<_p, X \mapsto ([A], [A], \emptyset) \), noting that we use the empty relation \( \emptyset \) as the interpretation of \( X \).

To prove the claim, it is sufficient to show that

\[
(w, \left[ \left( (M\{ X = A \} \equiv V) \right)^+ \right] \left[ \eta_i \right], \left[ \left( (M\{ X = A \} \equiv V) \right)^+ \right] \left[ \eta_r \right]) \in \mathcal{E}^-[X] y \delta
\]

since \( \mathcal{V}^-[X] y \delta = \emptyset \) and therefore the case where the two sides reduce to values is impossible.

First, since \( (w, y_p^\rightarrow, \delta^<_p) \in \mathcal{G}^-[\Pi_p] \), by reflexivity and weakening we know \( (w, \left[ (M)^+ \right] \left[ \eta_i \right], \left[ (M)^+ \right] \left[ \eta_r \right]) \in \mathcal{E}^-[\mathcal{V}^\rightarrow ? \rightarrow X] y \delta \).

We apply monadic bind. Assume \( w' \equiv w \) and \( (w', V_{\ell,l}, V_{\ell,r}) \in \mathcal{V}^-[\mathcal{V}^\rightarrow ? \rightarrow X] y \delta \), then by anti-reduction we need to show that

\[
(w', \left[ S \right] \left[ \text{force } V_{\ell,l} \left[ [A] \right] \sigma \right] \left[ \eta_i \right] \left[ \delta_l \right], \left[ S \right] \left[ \text{force } V_{\ell,r} \left[ [A] \right] \sigma \right] \left[ \eta_r \right] \left[ \delta_r \right]) \in \mathcal{E}^-[\mathcal{X}] y \delta
\]

where

\[
S = \bullet V
\]

Next, since \( w', \eta \models (\sigma, \sigma, \emptyset) \) and \( \delta(X) = \left( [A], [A], \emptyset \right) \), we have \( (w', \text{force } V_{\ell,l} \left[ [A] \right] \sigma, \text{force } V_{\ell,l} \left[ [A] \right] \sigma) \in \mathcal{E}^-[\mathcal{?} \rightarrow X] y \delta \).

Again we apply monadic bind. Assume \( w'' \equiv w' \) and \( (w'', V_{\ell,l}, V_{\ell,r}) \in \mathcal{V}^-[\mathcal{?} \rightarrow X] y \delta \). Then by Lemma F.10, we know \( \Sigma \models [V^+] \left[ \eta_i \right] \mapsto^* \Sigma \rightsquigarrow \text{ret } V_{x} \left[ \eta_i \right] \), so applying anti-reduction it suffices to show

\[
(w'', \text{force } V_{\ell,l} V_{x} \left[ \eta_i \right], \text{force } V_{\ell,r} V_{x} \left[ \eta_r \right]) \in \mathcal{E}^-[\mathcal{X}] y \delta
\]

for which it suffices to show that

\[
(w'', \text{force } V_{\ell,l} \left[ \eta_i \right], \text{force } V_{\ell,r} \left[ \eta_r \right]) \in \mathcal{V}^-[\mathcal{?}] y \delta
\]

for which it suffices to show that

\[
(w'', \left[ V^+ \right] \left[ \eta_i \right], \left[ V^+ \right] \left[ \eta_r \right]) \in \mathcal{E}^-[\mathcal{?}] y \delta
\]

which by weakening is equivalent to showing

\[
(w'', \left[ V^+ \right] \left[ \eta_p \right], \left[ V^+ \right] \left[ \eta_r \right]) \in \mathcal{E}^-[\mathcal{?}] y^2 \delta^<_p
\]

which follows by reflexivity and the fact that \( (w'', y^2 \delta^<_p, \delta^<_p) \in \mathcal{G}^-[\Pi_p] \).

\[\square\]

**Theorem G.50.** For any \( \cdot \vdash A, B \) and \( V_A : A \) and \( V_B : B \),

\[
\lambda_\cdot : ? \cdot \text{unseal}_X(M\{ X = A \} \text{ seal}_X V_A) \approx \text{ctx}
\]

\[
\lambda_\cdot : ? \cdot \text{let } y = (\text{unseal}_X(M\{ X = B \} \text{ seal}_X V_B)) ; V_A : ? \rightarrow A
\]

**Proof.** We show one direction of the 4 cases, the others are analogous. Let \((w, \gamma, \delta) \in \mathcal{G}^-[\Pi_p] \).

We need to show

\[
(w, \text{ret thunk } \lambda_\cdot : ?, \text{newcase}[A] c_X; [M\{ X = A \} \left( \{?\} \downarrow \langle \text{tag}_X(X) \rangle \downarrow \langle \text{seal}_X V_A \rangle \rangle] \left[ \eta_i \right] \left[ \delta_l \right]) \in \mathcal{E}^-[A] y \delta
\]

\[
\text{ret thunk } \lambda_\cdot : ?, \text{newcase}[A] c_X; [\text{let } y = M\{ X = B \} \left( \{?\} \downarrow \langle \text{tag}_X(X) \rangle \downarrow \langle \text{seal}_X V_B \rangle \rangle ; V_A \left[ \eta_r \right] \left[ \delta_r \right]
\]
Given some \( w' \equiv w \) and (irrelevant) values \((w', V_{duml}, V_{dumr})\), we need to show (after applying anti-reduction) that

\[
\begin{align*}
(w', \text{newcase}_{\lambda} c_X; & \quad [M\{X \equiv A\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_A))] \gamma_1[\delta_i] \\
& \quad \in \mathcal{E}^{-}[\lambda] y \delta \\
\text{newcase}_{\lambda} c_X; & \quad [\text{let } y = M\{X \equiv B\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_B)); V_A] \gamma_r[\delta_r]
\end{align*}
\]

Each side allocates a new case and we need to pick a relation with which to instantiate it. By Lemmas F.10 and F.2, we have that \([V_A] \gamma_1[\delta_i] \mapsto^0 \text{ret } V_{AI}\) and \([V_B] \gamma_r[\delta_r] \mapsto^0 \text{ret } V_{Br}\) for some \( V_{AI} \) and \( V_{Br} \). As in the ordinary identity function proof, we define \( R \) to be the “singleton” relation:

\[
R = \{(w, V_{AI}, V_{Br}) \in \text{Atom}[[\lambda] \gamma_1, [\lambda] \gamma_r] \mid w \equiv w'\}
\]

Then we define \( w'' \) to be the world extended with \([R]_{w',j} \):

\[
w'' = (w', j. w'. \Sigma_l, [\lambda] \gamma_1, w'. \Sigma_r \gamma_2, w'. \eta \equiv (w'. \Sigma_l. \text{size}, w'. \Sigma_r. \text{size}, [R]_{w',j})
\]

Then clearly \( w'' \equiv w' \) and

\[
w'. \Sigma_l. \text{newcase}_{\lambda} [\lambda] \gamma_1 c_X;
\]

\[
[M\{X \equiv A\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_A))] \gamma_1[\delta_i]
\]

\[
\mapsto^0 w''. \Sigma_l. [M\{X \equiv A\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_A))] \gamma'_1[\delta_i]
\]

and similarly for the right hand side:

\[
w'. \Sigma_r. \text{newcase}_{\lambda} [\lambda] \gamma_1 c_X;
\]

\[
[\text{let } y = M\{X \equiv B\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_B)); V_A] \gamma_r[\delta_r]
\]

\[
\mapsto^0 w''. \Sigma_r. [\text{let } y = M\{X \equiv B\} ((\_)? \downarrow \langle \text{tag}_X(X) \rangle) (\text{seal}_X(V_B)); V_A] \gamma'_r[\delta_r]
\]

where \( \gamma' = y. c_X \mapsto (w'. \Sigma_l. \text{size}, w'. \Sigma_r. \text{size}) \). Expanding definitions (and noting that \( c_X \) is free in \( M \)), we need to show

\[
(w'', S_1[[M] \gamma'_1[\delta_i]], S_2[[M] \gamma'_2[\delta_r]]) \in \mathcal{E}^{-}[\lambda] y \delta
\]

where

\[
S_1 = f \leftarrow (z \leftarrow \bullet; \text{force } z([\lambda] \gamma_1) \gamma'_1(c_X)); \\
x \leftarrow [[\text{tag}_X(X)] \gamma_1[\delta_i][V_A] \gamma_1[\delta_i]]; \\
\text{force } f \ x
\]

and

\[
S_2 = y \leftarrow \begin{cases} 
\left(f \leftarrow (z \leftarrow \bullet; \text{force } z([\lambda] \gamma_1) \gamma'_1(c_X)); \\
x \leftarrow [[\text{tag}_X(X)] \gamma_r[\delta_r][V_B] \gamma_r[\delta_r]]; \\
\text{force } f \ x \right) : [V_A] \gamma_r[\delta_r] \end{cases}
\]
We apply monadic bind. Let \( w''' \equiv w'' \) and \((w''', V_l, V_r) \in \mathcal{V}^-[\forall \! X. ? \to X] \gamma \delta \). After anti-reduction we need to show

\[
(w''' \quad f \leftarrow \text{force } V_l([A][\delta_1]) \gamma'_1(c_X); \quad \begin{array}{c}
x \leftarrow \mathcal{V}([\text{tag}_X(X)]) [\gamma][\delta_1][[V_A][\gamma]][\delta_1]; \\
f \leftarrow \mathcal{V}([\text{tag}_X(X)]) [\gamma][\delta_1][[V_A][\gamma]][\delta_1]; \\
f \leftarrow \text{force } f \ x
\end{array}
\]

\[
\in \mathcal{E}^-[A] \gamma \delta
\]

where

\[
S'_l = y \leftarrow \bullet; [V_A][\gamma][\delta_1] \in \mathcal{E}^-[A] \gamma \delta
\]

By weakening it is equivalent to showing the terms are in the relation \( \mathcal{E}^-[A] \gamma' \delta' \) where \( \delta' = ([A][\delta_1], [B][\delta_1], R) \). By definition of \( \mathcal{V}^-[\forall \! X. ? \to X] \), we have

\[
(w''', \text{force } V_l([A][\delta_1]) \gamma'_1(c_X), \text{force } V_r([B][\delta_1]) \gamma'_1(c_X)) \in \mathcal{E}^-[? \to X] \gamma' \delta'
\]

So we apply monadic bind. Let \( w'''' \equiv w''' \) and let \((w''', V_{fl}, V_{fr}) \in \mathcal{V}^-[? \to X] \gamma' \delta' \). Applying anti-reduction, we need to show

\[
(w'''' \quad \text{force } V_{fl} (\text{inj}_{\sigma_l} V_{Al}), y \leftarrow \text{force } V_{fr} (\text{inj}_{\sigma_r} V_{Br}); [V_A][\gamma][\delta_1]) \in \mathcal{E}^-[A] \gamma' \delta'
\]

By definition of \( \mathcal{V}^-[? \to X] \), we can prove \((w'''', \text{force } V_{fl} (\text{inj}_{\sigma_l} V_{Al}), \text{force } V_{fr} (\text{inj}_{\sigma_r} V_{Br})) \in \mathcal{E}^-[X] \gamma' \delta' \) by showing that \((w''', \text{inj}_{\sigma_l} V_{Al}, \text{inj}_{\sigma_r} V_{Br}) \in \mathcal{V}^-[?] \gamma' \delta' \). This holds because \((w'''' \in \mathcal{V}_{Al}, V_{Br}) \in [R]_{w'''} \eta = (\sigma_l, \sigma_r, [R]_{w'''}). \)

By Lemmas 10 and 2, we have that \([V_A][\gamma][\delta_1] \mapsto \text{ret } V_{Ar} \) for some \( V_{Ar} \). Applying monadic bind a final time we get \( w_{fin} \equiv w'''' \) and \((w_{fin}, V'_{Al}, V'_{Br}) \in \mathcal{V}^-[X] \gamma' \delta' \) and after anti-reduction need to show that

\[
(w_{fin}, \text{ret } V'_{Al}, \text{ret } V_{Ar}) \in \mathcal{E}^-[A] \gamma' \delta'
\]

By weakening this is equivalent to

\[
(w_{fin}, \text{ret } V'_{Al}, \text{ret } V_{Ar}) \in \mathcal{E}^-[A] \gamma \delta
\]

By the definition of \( \mathcal{V}^-[X] \gamma' \delta' \), \( V'_{Al} = V_{Al} \), so this follows by reflexivity; specifically that \( \Gamma \vdash V_A \subseteq V_A \in \mathcal{V}_{Al} \). This is because, by the definition of \( \mathcal{E}^-[A] \gamma \delta \), since \( V_A[\gamma][\delta_1] \mapsto \text{ret } V_{Al} \) and \( V_A[\gamma][\delta_1] \mapsto \text{ret } V_{Ar} \), those values are related.

\[\square\]

**Theorem G.51.** For any \( \vdash A, B \) and \( V_A : A, V_B : B, \vdash V_d : ? \),

\[
\lambda_\vdash \text{unseal}_X(M(X \equiv A) (\text{seal}_X V_A, V_d)) \approx \text{clx}
\]

\[
\lambda_{\vdash} \text{let } y = (\text{unseal}_X(M(X \equiv B) (\text{seal}_X V_B, V_d))); V_A
\]

**Proof.** Again, we show one direction of the 4 cases, the others are analogous. Let \((w, y, \gamma, \delta) \in \mathcal{G}^-[\Gamma_p] \). We need to show

\[
(w, \text{ret thunk } \lambda_{\vdash}; \text{newcase}_{[A]} c_X; \text{[M}X \equiv A\text{]}[\text{tag}_X? (X \times ?)] [\text{seal}_X V_A, V_d][][\gamma][\delta_1]; \quad \text{ret thunk } \lambda_{\vdash}; \text{newcase}_{[A]} c_X; \text{[let } y = M(X \equiv B) [\text{tag}_X? (X \times ?)] [\text{seal}_X V_B, V_d); V_A][\gamma][\delta_1] \)
\]
Given some \( w' \equiv w \) and (irrelevant) values \((w', V_{duml}, V_{dumr})\), we need to show (after applying anti-reduction) that
\[
(w', \text{newcase}_{[\Lambda]} c_X; M\{X \equiv A\} (\langle?\rangle \downarrow \langle \text{tag}_{\land X}(X \times ?) \rangle \uparrow (\text{seal}_X V_A), V_d) [Y_1] \downarrow \delta] \in \mathcal{E}^- [\Lambda] y \delta
\]
\[
\text{newcase}_{[\Lambda]} c_X; [\text{let } y = M\{X \equiv B\} (\langle?\rangle \downarrow \langle \text{tag}_{\land X}(X \times ?) \rangle \uparrow (\text{seal}_X V_B), V_d); V_A] [Y_r] \downarrow \delta_r
\]

Each side allocates a new case and again we to pick a relation with which to instantiate it. Again, by Lemmas F.10 and F.2, we have that \([V_A] [Y_1] \downarrow \delta \mapsto^0 \text{ret } V_{Al} \) and \([V_B] [Y_r] \downarrow \delta_r \mapsto^0 \text{ret } V_{Br} \) for some \( V_{Al} \) and \( V_{Br} \). As in the ordinary identity function proof, we define \( R \) to be the "singleton" relation:
\[
R = \{(w, V_{Al}, V_{Br}) \in \text{Atom}[\Lambda][\delta_1], [B][\delta_r] | w \equiv w'\}
\]

Then we define \( w'' \) to be the world extended with \( [R]_{w.j} \):
\[
w'' = (w', j, w'.\Sigma_l, [\Lambda][\delta_1], w'.\Sigma_r[\delta_r], w'.\eta \oplus (w'.\Sigma_l.\text{size}, w'.\Sigma_r.\text{size}, [R]_{w.j})
\]

Then clearly \( w'' \equiv w' \) and
\[
w'.\Sigma_l, \text{newcase}_{[\Lambda]}[\delta_1] c_X;
\]
\[
M\{X \equiv A\} (\langle?\rangle \downarrow \langle \text{tag}_{\land X}(X \times ?) \rangle \uparrow (\text{seal}_X V_A), V_d) [Y_1] \downarrow \delta_1
\]

\[
\mapsto^0 w''.\Sigma_l, [\text{let } y = M\{X \equiv B\} (\langle?\rangle \downarrow \langle \text{tag}_{\land X}(X \times ?) \rangle \uparrow (\text{seal}_X V_B), V_d); V_A] [Y_r] \downarrow \delta_r
\]

and similarly for the right hand side:
\[
w'.\Sigma_r, \text{newcase}_{[\Lambda]}[\delta_r] c_X;
\]
\[
\mapsto^0 w''.\Sigma_r, [\text{let } y = M\{X \equiv B\} (\langle?\rangle \downarrow \langle \text{tag}_{\land X}(X \times ?) \rangle \uparrow (\text{seal}_X V_B), V_d); V_A] [Y_r] \downarrow \delta_r
\]

where \( \gamma' = y, c_X \mapsto (w'.\Sigma_l.\text{size}, w'.\Sigma_r.\text{size}) \). Following the same argument as the previous cases, using monadic bind several times, we need to show that for some \( w'''' \equiv w'' \) and \( (w''', V_{fl}, V_{fr}) \in \mathcal{V}^- [\Lambda \rightarrow X] y' \delta' \), that
\[
(w'''', \text{force } V_{fl} \text{ inj}_{Y_l(c\text{times})} (\text{inj}_{\sigma_l} V_{Al}, V_{dl}), y \leftarrow \text{force } V_{fr} \text{ inj}_{Y_r(c\text{times})} (\text{inj}_{\sigma_r} V_{Br}, V_{dr}); [V_A][Y_r][\delta_r]) \in \mathcal{E}^- [\Lambda][\delta_1]
\]

By definition of \( \mathcal{V}^- [\Lambda \rightarrow X] \), we can prove
\[
(w''', \text{force } V_{fl} \text{ inj}_{Y_l(c\text{times})} (\text{inj}_{\sigma_l} V_{Al}, V_{dl}), \text{force } V_{fr} \text{ inj}_{Y_r(c\text{times})} (\text{inj}_{\sigma_r} V_{Br}, V_{dr})) \in \text{Erel}_p X y' \delta'
\]
by showing that
\[
(w''', \text{inj}_{Y_l(c\text{times})} (\text{inj}_{\sigma_l} V_{Al}, V_{dl}), \text{inj}_{Y_r(c\text{times})} (\text{inj}_{\sigma_r} V_{Br}, V_{dr})) \in \mathcal{V}^- [\Lambda][\delta_1]
\]

since \( (w''', y, \delta) \in \mathcal{G}^- [\Gamma_p] \), which holds because \( (w''', V_{Al}, V_{Br}) \in R \) and \( (w''', V_{dl}, V_{dr}) \in \mathcal{E}^- [\text{dyn}][\delta_1]
\]

So again we apply monadic bind, receiving some \( w'''' \equiv w''' \) and \( (w'''', V_{Xl}, V_{Xr}) \in \mathcal{V}^- [X] y' \delta' \) and we need to show (after applying anti-reduction) that
\[
(w'''', \text{ret } V_{Xl}; [V_A][Y_l][\delta_1]) \in \mathcal{E}^- [\Lambda][\delta_1]
\]
But by definition of \( \mathcal{V}^- [X] y' \delta' \), we know that \( V_{Xl} = V_{Al} \) and that \([V_A][Y_l][\delta_1] \mapsto^0 \text{ret } V_{Al} \), so it is sufficient to show
\[
(w'''', [V_A][Y_l][\delta_1], [V_A][Y_r][\delta_r]) \in \mathcal{E}^- [\Lambda][\delta_1]
\]
which follows by reflexivity.