JFP **31**, e21, 139 pages, 2021.
doi:10.1017/S0956796821000125

Gradual type theory

MAX S. NEW

Khoury College of Computer Sciences, Northeastern University, Boston, MA 02115, USA
(*e-mail:* maxsnew@umich.edu)

DANIEL R. LICATA

Mathematics and Computer Science, Wesleyan University, Middletown, CT 06459, USA
(*e-mail:* dlicata@wesleyan.edu)

AMAL AHMED

Khoury College of Computer Sciences, Northeastern University, Boston, MA 02115, USA
(*e-mail:* amal@ccs.neu.edu)

Abstract

Gradually typed languages are designed to support both dynamically typed and statically typed programming styles while preserving the benefits of each. Sound gradually typed languages dynamically check types at runtime at the boundary between statically typed and dynamically typed modules. However, there is much disagreement in the gradual typing literature over how to enforce complex types such as tuples, lists, functions and objects. In this paper, we propose a new perspective on the design of runtime gradual type enforcement: runtime type casts exist precisely to ensure the correctness of certain type-based refactorings and optimizations. For instance, for simple types, a language designer might desire that beta-eta equality is valid. We show that this perspective is useful by demonstrating that a cast semantics can be derived from beta-eta equality. We do this by providing an axiomatic account program equivalence in a gradual cast calculus in a logic we call *gradual type theory* (GTT). Based on Levy's call-by-push-value, GTT allows us to axiomatize both call-by-value and call-by-name gradual languages. We then show that we can derive the behavior of casts for simple types from the corresponding eta equality principle and the assumption that the language satisfies a property called *graduality*, also known as the dynamic gradual guarantee. Since we can derive the semantics from the assumption of eta equality, we also receive a useful contrapositive: any observably different cast semantics that satisfies graduality *must* violate the eta equality. We show the consistency and applicability of our axiomatic theory by proving that a contract-based implementation using the lazy cast semantics gives a logical relations model of our type theory, where equivalence in GTT implies contextual equivalence of the programs. Since GTT also axiomatizes the dynamic gradual guarantee, our model also establishes this central theorem of gradual typing. The model is parameterized by the implementation of the dynamic types, and so gives a family of implementations that validate type-based optimization and the gradual guarantee.

1 Introduction

Gradually typed languages (Siek & Taha, 2006; Tobin-Hochstadt & Felleisen, 2008) are designed to support the gradual migration from dynamically typed to statically typed programming with a unified syntax and implementation. This is based on the hypothesis that dynamically typed and statically typed languages have complementary benefits, and are better in different *contexts* in the software development life cycle. Dynamically typed code can be written without conforming to a strict syntactic type discipline, so the programmer can always run their program interactively with minimal effort. This makes dynamically typed languages ideal for prototyping and implementing one-off scripts. Statically typed programs, on the other hand, are checked at compile time for internal consistency, detecting errors before the program even runs and providing mathematically sound reasoning principles that simplify correctness arguments, justify type-based refactorings, enable compiler optimizations and underlie formal software verification. This make statically typed languages ideal for long-term program maintenance and reusable libraries. Gradually typed languages are designed with the perspective that dynamically typed and statically typed styles are useful at different *times* in the software development process, and so enable *gradual migration* from a dynamically typed to a statically typed style. That is, gradual languages support syntax for both static and dynamic typing, and allow for dynamic code to be migrated to a static style simply by adding type annotations. The language should support *gradual* migration in that any mix of statically typed and dynamically typed program modules should be executable as long as the statically typed portions type check. That is, you don't have to migrate the entire codebase from dynamic to static typing before running, or manually implement any glue code for the dynamic and static modules to interoperate.

There are two main approaches to how mixed static–dynamic programs are run. The first approach, which we call "optional typing", is to erase all type information and simply run as if the program were a dynamically typed program. The philosophy of optional typing is that static types are simply a static analysis for catching some bugs. Optional typing is popular in industry languages such as Hack, TypeScript and Flow. The second approach, called "sound gradual typing",[1] which is the focus of this paper, is to insert type casts at the boundary between static and dynamically typed code. These casts will error at runtime if the dynamically typed values do not satisfy the static type specifications. The reason these type casts are inserted are so that in gradually migrating from dynamic to static style, the programmer receives the *reasoning* benefits of static typing: if you have a statically typed variable that is of type $\mathtt{Num}$, then you can be ensured at runtime it really will be a number. This means that if the runtime checks are strong enough, statically typed programs in a sound gradually typed language can receive the same optimizations as in a fully statically typed language.

So the central design decisions for gradually typed language semantics is the semantics of these *runtime type casts* that are inserted at the boundaries between dynamically typed and statically typed code. These runtime checks ensure that typed reasoning principles are valid by checking the types of dynamically typed values at runtime when they flow to statically typed code. For instance, when a statically typed function $f : \mathtt{Num} \to \mathtt{Num}$ is applied to a dynamically typed argument x, the language runtime must check if x is a number, and otherwise raise a dynamic type error. This is usually formalized by translation to an explicitly typed cast calculus where casts are inserted at these static–dynamic boundaries. In this case, the application fx in the source language would elaborate to something like $f(\langle \mathtt{Num} \Leftarrow ?\rangle x)$. Here ? is the type of dynamically typed values, and the cast $\langle \mathtt{Num} \Leftarrow ?\rangle x$ is

[1] Terminology from Takikawa *et al.* (2016).

read as “cast x from ? to `Num`”. The behavior of the cast is then given by the operational semantics of this cast calculus.

While there has been a great deal of research on gradually typed languages and their semantics, there is little agreement on the semantics of these casts, especially when the casts involve more complex types such as tuples, lists, functions and objects. This has led to at least two papers discussing the design space of casts and some trade-offs of their approaches (Siek *et al.*, 2009; Greenman & Felleisen, 2018). The goal of this paper is to promote a new perspective on the design of cast semantics by proposing a *semantic* explanation for cast behavior. Our perspective is that we should base the design of cast semantics on the desired *equational reasoning principles* for the statically typed code. In particular, we will focus on the validity of η equality for simple types, which we will introduce in more detail shortly. Equational reasoning principles formalize when two programs in a language have equivalent behavior. They can be used to justify that program refactorings and optimizations are semantics-preserving. Validity of equational reasoning principles is a particularly useful way to formalize the benefits of static typing because it is directly actionable information: if your language ensures certain equational reasoning principles, then those can be used to justify more optimizations and refactorings. So as a programmer in a sound gradual language migrates from dynamic to static code, we can see that the equational theory becomes richer, in a sense quantifying the idea that the programmer has increased their ability to reason about the code.

We provide evidence that this is a useful perspective on the design of gradual type semantics by showing that some of the semantics of casts are *uniquely determined* by which η principles they satisfy. That is, we can formally derive certain cast semantics from η equality. As a corollary of this theorem, any cast semantics that differs from the ones we derive *must* violate the η reasoning principle, which provides us with specific concrete trade-offs in reasoning that different cast semantics provide.

1.1 Soundness theorems for sound gradual typing

The extent to which these different semantics have been shown to validate type-based reasoning has been limited to syntactic *gradual type soundness* and *blame soundness* theorems. In their general form, these theorems say: “If t is a closed program of type A then it diverges, or reduces to a runtime error blaming dynamically typed code, or reduces to a value that satisfies A.” Since the theorem only describes the result of a single run of the program, it doesn’t *directly* justify any of the optimizations or refactorings that gradual types are supposed to justify. Furthermore, the blame tracking aspect of the theorem is difficult to understand in general since it relies on the notion of blame defined by the operational semantics and doesn’t have an agreed on standard.

We argue that existing gradual type soundness theorems are only indirectly expressing one of the desired properties of the gradual language, which are *program equivalences in the typed portion of the code* that are not valid in the dynamically typed portion. These typed equivalences are essential for ensuring that any reasoning about refactoring or optimization of code that is valid in a fully static setting is also valid for statically typed portions of a gradually typed program. Thus, preserving appropriate typed equivalences—the ones that justify refactoring and optimization—should be one of the criteria that gradually typed languages should satisfy.

In addition to the soundness of type information, we also emphasize that the gradual migration process should be “smooth”. This is captured by the *graduality* principle (originally called the dynamic gradual guarantee Boyland, 2014; Siek *et al.*, 2015). The graduality principle states that migrating from a dynamic to static style should never “interfere” with the results of a computation outside of reporting type errors. As a sports analogy, we can think of the runtime type checks as the “referee” between the interacting static and dynamic portions of code, calling out invalid behaviors but not stepping in and kicking the ball themselves.

More formally, the graduality principle says that given a gradually typed term M, if you make the type information in M more precise (i.e., use dynamic typing less), producing a term M', then M' should have very similar behavior to the original M: either they have the same behavior or they have the same behavior up until the point that M' raises an error.

Eta Laws and their Significance. So what are the program equivalences that hold in statically typed portions of the code but not in dynamically typed portions? Typically, β reductions are valid program equivalences in both statically typed and dynamically typed languages since they directly describe the operational behavior of the program. In general η equality is only satisfied in a typed language, and capture the idea that the behavior of a term is fully determined by the allowed elimination forms for its type.

The η law of the untyped λ-calculus, which states that any λ-term $M \equiv \lambda x.Mx$, is restricted in a typed language to only hold for terms of function type $M : A \to B$ (i.e., λ is the unique/universal way of making an element of the function type). This famously “fails” to hold in call-by-value languages in the presence of effects: if M is a program that prints `"hello"` before returning a function, then M will print *now*, whereas $\lambda x.Mx$ will only print when given an argument. But this can be accommodated with one further modification: the η law is valid in simple call-by-value languages[2] (e.g., SML) if we have a “value restriction” $V \equiv \lambda x.Vx$.

The above illustrates that η/extensionality rules must be stated for each type connective, and be sensitive to the effects/evaluation order of the terms involved. For instance, the η principle for the Boolean type `Bool` *in call-by-value* is that for any term M with a free variable $x : \texttt{Bool}$, M is equivalent to a term that performs an if statement on x:

$$M \equiv \texttt{if}\ x(M[\texttt{true}/x])(M[\texttt{false}/x])$$

If we have an `if` form that is strongly typed (i.e., errors on non-Booleans) then this tells us that it is *safe* to run an if statement on any input of Boolean type (in CBN, by contrast an if statement forces a thunk and so is not necessarily safe). In addition, even if our `if` statement does some kind of coercion, this tells us that the term M only cares about whether x is “truthy” or “falsy” and so a client is free to change, e.g., one truthy value to a different one without changing behavior.

This η principle justifies a number of program optimizations, such as dead-code and common subexpression elimination, and hoisting an if statement outside of the body of a function if it is well scoped:

[2] This does not hold in languages with some intensional feature of functions such as reference equality. We discuss the applicability of our main results more generally in Section 8.

$$\lambda x.\texttt{if}\ \ y\,M\,N \equiv \texttt{if}\ \ y\,(\lambda x.M)\,(\lambda x.N)$$

Any eager datatype, one whose elimination form is given by pattern matching such as $0, +, 1, \times$, `list`, has a similar η principle, which enables similar reasoning, such as proofs by induction. The η principles for lazy types *in call-by-name* support dual behavioral reasoning about lazy functions, records, and streams.

1.2 Which cast semantics?

So what, after all, are the semantics of casts? Here, there is a considerable amount of disagreement in the gradual typing literature. There have been many different proposed semantics of runtime type checking: "transient" cast semantics (Vitousek *et al.*, 2017) only checks the head connective of a type (number, function, list, . . .), "eager" cast semantics (Herman *et al.*, 2010) checks runtime type information on closures, whereas "lazy" cast semantics (Findler & Felleisen, 2002) will always delay a type-check on a function until it is called (and there are other possibilities, see, e.g., Siek *et al.*, 2009; Greenberg, 2015). Let's consider some examples to illustrate the design choices involved.

Example 1: Eager versus Lazy Base Type Casts. Say we start with the following simple dynamically typed expression:

$$(\lambda x.\texttt{true})5$$

In a gradual language based on the style of Siek & Taha (2006), this would be expanded to annotate every subexpression with the dynamic type:

$$((\lambda x : ?.(\texttt{true} :: ?)) :: ?)(5 :: ?)$$

This evaluates to a Boolean `true` tagged at the dynamic type, with no runtime type errors:

$$((\lambda x : ?.(\texttt{true} :: ?)) :: ?)(5 :: ?) \mapsto (\texttt{true} :: ?)[5 :: ?/x] = \texttt{true} :: ?$$

However, let's say the programmer decides to add types to the function $\lambda x.\texttt{true}$ and decides the variable x should be a string. How then should the program

$$((\lambda x : \texttt{String}.\texttt{true}) :: ?)(5 :: ?)$$

evaluate? The graduality principle says that adding types should either result in the same answer or a new type error, so graduality allows for the program to successfully return $\texttt{true} :: ?$ or to introduce a type error. Additionally, any whole-program notion of type soundness would also allow both an error and returning $\texttt{true} :: ?$ since both possibilities satisfy the type ?.

However, despite both of these theorems allowing the possibility of successfully returning 5, in all call-by-value cast semantics that we know of (in which strings and numbers are incompatible types) result in an error saying that 5 does not satisfy the type `String`. Why is this the case? In a gradual language, the annotation $x : \texttt{String}$ should be *actionable* information to the programmer. Knowing that x is a `String` means that string-based operations (getting the length, inspecting characters within its length) are safe to perform on x .

On the other hand, if the language is a *lazy* or *call-by-name* calculus, we argue that the correct behavior is for the program to return `true` without failing. This is because in a lazy language, variables represent delayed ("thunked") computations and in this case x will be bound to a computation that checks if 5 is a string, erroring if it is ever forced. It is appropriate that this check is not run in the typed version of the program because the input x is never *forced*.

So this example shows us that (1) gradual typing can introduce new type errors even when dynamic typing would succeed (2) the semantics of casts should be sensitive to the evaluation order of the language.

Example 2: Eager versus Lazy Function Casts. Function types are central to functional programming, and so have understandably been the main focus of functional gradually typed language semantics. There are at least two common cast semantics for simple functional languages, which we call eager and lazy.

To see the difference, let's consider the following example function:

$$f_d = (\lambda x : \mathtt{String}.\mathtt{string\text{-}length}\, x) :: ? \to ?$$

Here f_d is a just the built-in string-length function η-expanded and cast to be a dynamic to dynamic function, this should be or reduce to a value in most cast calculi. What happens if we erroneously cast f_d to the type $\mathtt{Num} \to \mathtt{Num}$? In lazy cast semantics, function casts like this are given simply by wrapping the function in input and output casts. So in this case

$$f_d :: \mathtt{Num} \to \mathtt{Num}$$

will reduce to a term equivalent to

$$\lambda n : \mathtt{Num}.((\lambda x : \mathtt{String}.\mathtt{string\text{-}length}\, x)(n :: ?)) :: ? :: \mathtt{Num}$$

That is, it will reduce to a function value that if called, will take a number n as input, casts it to ?, applies the original function to that and then casts the result to `Num`. This will always result in an error, because the input n will be cast to the incompatible type `String`.

Eager cast semantics, on the other hand is based on the idea that it is fairly easy to see, if we maintain some runtime type information, that casting f_d to $\mathtt{Num} \to \mathtt{Num}$ will result in a function that always errors, because f_d is a function of type $\mathtt{String} \to \mathtt{Num}$, so the semantics should instead error *immediately* when the cast to $\mathtt{Num} \to \mathtt{Num}$ is applied, rather than returning a function that always errors.

Note again that both behaviors are allowed by type soundness and graduality since they only differ in *when* an error might happen. However, if we inspect what *equational reasoning principles* are valid in the language, we see that $\beta\eta$ equivalence favors *lazy* function semantics.

In particular, in call-by-value, the η law for functions says that any function value f of type $A \to B$ is equivalent to its η expansion:

$$f \cong \lambda x : A.f x$$

In call-by-name, the η law applies to all terms of function type, not just values. Returning to our example, since f_d is a function value of type $? \to ?$, the η equation tells us that

$$f_d \cong \lambda d : ?.f_d\, d$$

Call this term f_η. Then in particular, if the η equation is valid for contextual equivalence, we should have

$$f_d :: \texttt{Num} \to \texttt{Num} \cong f_\eta :: \texttt{Num} \to \texttt{Num}$$

In lazy function semantics, this does indeed hold. However, in eager function semantics, we lose precision in the runtime type information of f_η. f_η is only known to have the type $? \to ?$, whereas f_d is known to have the more precise type $\texttt{String} \to \texttt{Num}$, so $f_\eta :: \texttt{Num} \to \texttt{Num}$ reduces to a function that always errors while $f_d :: \texttt{Num} \to \texttt{Num}$ errors immediately.

While this might seem a rather minor difference, η expansions are quite common in higher order functional libraries, and good functional compilers will perform η contractions to remove the need to construct a closure at runtime. In eager function semantics, these η contractions are not equivalence-preserving transformations: an η contraction might change a successful run of a program with an erroring one. Since most optimizing compilers rely on heuristics to determine how much optimization to perform, this means that whether or not certain η redexes are contracted will decide whether or not their program errors, and so it is probably best for the user that a compiler for eager semantics never contracts an η redex.

Example 3: Eager versus Lazy Product Casts. Compared to functions, products have received less focus in the gradual typing literature, but there is a somewhat similar divide between eager and lazy product semantics.

To illustrate the difference, consider what happens when we cast a pair

$$(5, "\textit{hello}") :: ? \times ? :: \texttt{Num} \times \texttt{Num}$$

In eager product semantics, when we cast a tuple to a product type, evaluation proceeds by casting each component of the tuple, and then reconstructing the tuple with the result of the casts:

$$\texttt{let } (5 :: ? :: \texttt{Num}) = x_1; \texttt{let } (\text{“}\textit{hello}\text{"} :: ? :: \texttt{Num}) = x_2; (x_1, x_2)$$

This then results in an error because the right-hand side of the tuple is a string and not a number.

In *lazy* product semantics, however, a product cast checks *now* if the tuple is a pair, and only checks the components of the tuple when they are projected out. So in this case, the cast above would not error, and if the first component is projected it would also not error, but if the second component were projected there would be a dynamic error raised.

Again graduality allows for both possibilities, but in this case most type soundness theorems would rule out the lazy semantics, though Greenman & Felleisen (2018) discuss alternative “tag soundness” theorems that allow this behavior. However, the lesser known η principle for *eager products* only allows for the eager product semantics. The eager product η equation says that any term M with a free variable p of product type $A_1 \times A_2$ is equivalent to a term that immediately pattern matches on the tuple:

$$p : A_1 \times A_2 \vdash M \cong \texttt{split } p \texttt{ to } (x_1, x_2).M[(x_1, x_2)/p]$$

This implies in particular that pattern matching on any pair value is a *safe* operation, i.e., it never causes an error to be raised. However, in lazy product cast semantics, pattern matching on a pair will project out both sides of the tuple and therefore trigger further casts, so it might produce an error.

For an example of how this might affect a larger program context, consider a program that takes in a tuple and returns a function:

$$\lambda p : A_1 \times A_2 . \lambda x : A' . \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).M$$

A routine refactoring might lift this pattern match higher in the program so that it happens once instead of each time the produced function is called:

$$\lambda p : A_1 \times A_2 . \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).\lambda x : A'.M$$

Depending on the compiler, this might make a big difference in space usage as well if M only uses one component of the pair. In the first case, p in its entirety would be included in the closure, whereas it is easier for an optimizing compiler to see that only x_2 is needed in the closure in the latter.

On the other hand, if we have a *lazy* product type, where each side of the pair is only evaluated when it is projected out, then the lazy product cast semantics *is* more appropriate. In this case, we have the *lazy* product η law which says that any term of product type is equivalent to one where each side is the projection:

$$M \cong (\pi_1 M, \pi_2 M) : A_1 \times A_2$$

When this is true in an effectful language, the pair constructor must be lazy, since otherwise we would be duplicating M's effects. In this case the eager product cast would be *overly strict* because it might error immediately, whereas in a lazy language, the error should be delayed.

Casts from Equations. These examples illustrate that questions of evaluation order and validity of equational reasoning principles should inform the design of gradual typing cast semantics. In fact, if we take a closer look at the semantics of lazy function and eager and lazy product casts, we can see that they are very close to just η expanding the term, except that they introduce some new casts:

$$\text{when}\, f : A \to B, \quad f :: A' \to B' \cong \lambda x' : A'.(f(x' :: A)) :: B'$$

$$p :: A \times B :: A' \times B' \cong \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).(x_1 :: A :: A', x_2 :: B :: B')(\textit{eager})$$

$$p :: A \times B :: A' \times B' \cong ((\pi_1 p) :: A :: A', (\pi_2 p) :: B :: B')(\textit{lazy})$$

This is suggestive of a deeper connection between the cast semantics and the η equations. In fact, in the next section, we are able to show using a novel formulation of the graduality principle that these η principles directly lead to a derivation of the correct corresponding cast semantics. This shows us that the correct eager or lazy behavior of a cast is directly linked to the evaluation order in the intended programming language and in particular to the particular *type* that is involved. We are able to do this for call-by-name and call-by-value evaluation orders by using a more general language

1.3 An axiomatic approach to gradual typing

In this paper, we systematically study the relationship between casts and equational reasoning principles by working with an *axiomatic theory* of gradual program equivalence, a language and logic we call *gradual type theory* (GTT). Gradual type theory is the combination of a language of terms and gradual types with a simple logic for proving program equivalence and *error approximation* (equivalence up to one program erroring when the other does not) results. We use the logic to axiomatize the equational properties we may want in our gradual language, and then we explore what the derivable consequences of those axioms are. The critical benefit of gradual type theory (GTT) is that it can be used both to explore language design questions and to verify behavioral properties of specific programs, such as correctness of optimizations and refactorings.

To get off the ground, we take two properties of the gradual language for granted. First, we assume a compositionality property: that any cast from A to B can be factored through the dynamic type ?, i.e., the cast $\langle B \Leftarrow A\rangle t$ is equivalent to first casting up from A to ? and then down to B: $\langle B \Leftarrow ?\rangle\langle ? \Leftarrow A\rangle t$. These casts often have quite different performance characteristics, but should have the same extensional behavior: of the cast semantics presented in Siek *et al.* (2009), only the partially eager detection strategy violates this principle, and this strategy is not common.

The second property we take for granted is that the language satisfies the graduality property mentioned earlier. Graduality says that if we change the types in a program to be "more precise"—e.g., by changing from the dynamic type to a more precise type such as integers or functions—the program will either produce the same behavior as the original or raise a dynamic type error. Conversely, if a program does not error and some types are made "less precise" then behavior does not change. Graduality is in fact central to our approach so we take some time now to introduce some basic notions. First, we define a "precision" ordering on types: $A \sqsubseteq A'$, read "A is more precise than A'". This ordering is typically generated by a rule that says the dynamic type is the least precise, i.e., $A \sqsubseteq ?$ for any A, and a congruence rule that says all type constructors are monotone in every argument. Notably, this includes the domain and codomain of the function type, differing from subtyping. This ordering is then extended to a "term precision" ordering $t \sqsubseteq t'$ that captures the notion that t is the result of making all of the types in t' more precise. Typically this includes only congruence rules. Then the graduality principle says that if $t \sqsubseteq t'$, that is t is *syntactically* more precise than t', then it is also *semantically* more precise in that its behavior is either the same as that of t', or it results in a dynamic type error. Gradual type theory is based on axiomatizing this *semantic* notion of term precision. It includes a term precision ordering $t \sqsubseteq t'$, but this is interpreted as the semantic idea that t "errors more" than t' rather than the stricter syntactic notion. This semantic notion of error ordering naturally arises in logical relations proofs of graduality (New & Ahmed, 2018; New *et al.*, 2020).

1.4 Technical overview of GTT

The gradual type theory developed in this paper unifies our previous work on operational (logical relations) reasoning for gradual typing in a call-by-value setting (New & Ahmed,

2018) (which did not consider a proof theory), and on an axiomatic proof theory for gradual typing (New & Licata, 2018) in a call-by-name setting (which considered only function and product types, and denotational but not operational models).

In this paper, we develop an axiomatic gradual type theory GTT for a unified language that includes *both* call-by-value/eager types and call-by-name/lazy types (Sections 2, 3), and show that it is sound for contextual equivalence via a logical relations model (Sections 5, 6, 7). Because the η principles for types play a key role in our approach, it is necessary to work in a setting where we can have η principles for both eager and lazy types. We use Levy's Call-by-Push-Value (Levy, 2003) (CBPV), which fully and faithfully embeds both call-by-value and call-by-name evaluation with both eager and lazy datatypes,[3] and underlies much recent work on reasoning about effectful programs (Bauer & Pretnar, 2013; Lindley *et al.*, 2017). GTT can prove results in and about existing call-by-value gradually typed languages, and also suggests a design for call-by-name and full call-by-push-value gradually typed languages.

In prior work (New & Licata, 2018; New & Ahmed, 2018), gradual type casts are decomposed into upcasts and downcasts, as suggested above. A *type precision* relation $A \sqsubseteq A'$ controls which casts exist: a type precision $A \sqsubseteq A'$ induces an upcast from A to A' and a downcast from A' to A. Then, a *term precision* judgement is used for equational/approximational reasoning about programs. Term precision relates two terms whose types are related by type precision, and the upcasts and downcasts are each *specified* by certain term precision judgements holding. This specification axiomatizes only the properties of casts needed to ensure the graduality theorem, and not their precise behavior, so cast reductions can be *proved from it*, rather than stipulated in advance. The specification defines the casts "uniquely up to equivalence", which means that any two implementations satisfying it are behaviorally equivalent.

We generalize this axiomatic approach to call-by-push-value (Section 2), where there are both eager/value types and lazy/computation types. This is both a subtler question than it might at first seem, and has a surprisingly nice answer: we find that upcasts are naturally associated with eager/value types and downcasts with lazy/computation types, and that the modalities relating values and computations induce the downcasts for eager/value types and upcasts for lazy/computation types. Moreover, this analysis articulates an important behavioral property of casts that was proved operationally for call-by-value in New & Ahmed (2018) but missed for call-by-name in New & Licata (2018): upcasts for eager types and downcasts for lazy types are both "pure" in a suitable sense, which enables more refactorings and program optimizations. In particular, we show that these casts can be taken to be (and are essentially forced to be) "complex values" and "complex stacks" (respectively) in call-by-push-value, a standard syntactic notion in CBPV (Levy, 2003) which corresponds to a behavioral property of *thunkability* and *linearity* (Munch-Maccagnoni, 2014) which formalize notions of purity for CBV and CBN. We argue in Section 8 that this property is related to blame soundness.

Our gradual type theory naturally has two dynamic types, a dynamic eager/value type and a dynamic lazy/computation type, where the former can be thought of as a sum of all possible values, and the latter as a product of all possible behaviors. At the language design

[3] The distinction between "lazy" versus "eager" casts above is different than lazy versus eager datatypes.

level, gradual type theory can be used to prove that $\beta\eta$ and graduality are only compatible with specific cast semantics: for value types, the “eager” cast semantics is compatible and for computation types the “lazy” cast semantics (Section 3). These behavioral equivalences can then be used in reasoning about optimizations, refactorings and correctness of specific programs.

1.5 Contract-based models

To show the consistency of GTT as a theory, and to give a concrete operational interpretation of its axioms and rules, we provide a concrete model based on an operational semantics. The model is a *contract* interpretation of GTT in that the “built-in” casts of GTT are translated to ordinary functions in a CBPV language that perform the necessary checks.

To keep the proofs high-level, we break the proof into two steps. First (Sections 5, 6), we translate the axiomatic theory of GTT into an axiomatic theory of CBPV extended with recursive types and an uncatchable error, implementing casts by CBPV code that does contract checking. Then (Section 7), we give an operational semantics for the extended CBPV and define a step-indexed biorthogonal logical relation that interprets the ordering relation on terms as contextual error approximation, which underlies the definition of graduality as presented in New & Ahmed (2018). Combining these theorems gives an implementation of the term language of GTT in which β, η are observational equivalences and the dynamic gradual guarantee is satisfied.

Due to the uniqueness theorems of GTT, the only part of this translation that is not predetermined is the definition of the dynamic types themselves and the casts between “ground” types and the dynamic types. We use CBPV to explore the design space of possible implementations of the dynamic types, and give one that faithfully distinguishes all types of GTT, and another more Scheme-like implementation that implements sums and lazy pairs by tag bits. Both can be restricted to the CBV or CBN subsets of CBPV, but the unrestricted variant is actually more faithful to Scheme-like dynamically typed programming, because it accounts for variable argument functions. Our modular proof architecture allows us to easily prove correctness of β, η and graduality for all of these interpretations.

1.6 Contributions

The main contributions of the paper are as follows.

1. We present Gradual Type Theory in Section 2, a simple axiomatic theory of gradual typing. The theory axiomatizes three simple assumptions about a gradual language: compositionality, graduality and type-based reasoning in the form of η equivalences.
2. We prove many theorems in the formal logic of Gradual Type Theory in Section 3. These include the unique implementation theorems for casts, which show that for each type connective of GTT, the η principle for the type ensures that the casts must implement the lazy contract semantics. Furthermore, we show that upcasts are always pure functions and dually that downcasts are always strict functions, as long as the base type casts are pure/strict.

3. We connect this derivation back to a familiar CBV calculus, showing explicitly that almost all cast reductions are derivable from the simple specification for casts in GTT.
4. To substantiate that GTT is a reasonable axiomatic theory for gradual typing, we construct *models* of GTT in Sections 5, 6 and 7. This proceeds in two stages. First (Section 5), we use call-by-push-value as a typed metalanguage to construct several models of GTT using different recursive types to implement the dynamic types of GTT and interpret the casts as embedding-projection pairs. This extends standard translations of dynamic typing into static typing using type tags: the dynamic value type is constructed as a recursive sum of basic value types, but dually the dynamic computation type is constructed as a recursive *product* of basic computation types. This dynamic computation type naturally models stack-based implementations of variable-arity functions as used in the Scheme language.
5. We then give an operational model of the term precision ordering as contextual error approximation in Sections 6 and 7. To construct this model, we extend previous work on logical relations for error approximation from call-by-value to call-by-push-value (New & Ahmed, 2018), simplifying the presentation in the process.

This article is an extension of a conference publication (New *et al.*, 2019). Compared to the previous paper, we include additional proofs and definitions in all of the technical sections. Additionally, we prove new theorems about *most precise* types in GTT and provide a simple lemma that abstracts over the details of the unique implementation proofs. Finally, we add a new section that connects GTT more concretely to a familiar call-by-value cast calculus to demonstrate more concretely the consequences of the unique implementations theorems.

2 Axiomatic gradual type theory

In this section we introduce the syntax of Gradual Type Theory, an extension of call-by-push-value (Levy, 2003) to support the constructions of gradual typing. First, we introduce call-by-push-value and then describe in turn the gradual typing features: dynamic types, casts and the precision orderings on types and terms.

2.1 Background: Call-by-push-value

We present the syntax of GTT types and terms in Figure 1, and the typing rules in Figure 2. GTT is an extension of CBPV, so we first present CBPV as the unshaded rules in Figure 1. CBPV makes a distinction between *value types* A and *computation types* $\underline{B}$, where value types classify *values* $\Gamma \vdash V : A$ and computation types classify *computations* $\Gamma \vdash M : \underline{B}$. Effects are computations: for example, we might have an error computation $\mho_{\underline{B}} : \underline{B}$ of every computation type, or printing $\texttt{print}\ V; M : \underline{B}$ if $V : \texttt{string}$ and $M : \underline{B}$, which prints V and then behaves as M.

Value Types and Complex Values. The value types include *eager* products 1 and $A_1 \times A_2$ and sums 0 and $A_1 + A_2$, which behave as in a call-by-value/eager language (e.g.,

$$
\begin{array}{rl}
A ::= & ? \mid U\underline{B} \mid 0 \mid A_1 + A_2 \mid 1 \mid A_1 \times A_2 \\
\underline{B} ::= & \underline{¿} \mid \underline{F}A \mid \top \mid \underline{B}_1 \mathbin{\&} \underline{B}_2 \mid A \to \underline{B} \\
T ::= & A \mid \underline{B} \\
\\
V ::= & \langle A' \nwarrow A\rangle V \mid x \mid \texttt{thunk}\ M \mid \texttt{abort}\ V \mid \texttt{inl}\ V \mid \texttt{inr}\ V \mid \texttt{case}\ V\{x_1.V_1 \mid x_2.V_2\} \\
 & \mid () \mid \texttt{split}\ V\ \texttt{to}\ ().V' \mid (V_1, V_2) \mid \texttt{split}\ V\ \texttt{to}\ (x,y).V' \\
\\
 & \langle \underline{B} \swarrow \underline{B}'\rangle M \mid \bullet \mid \mho_{\underline{B}} \mid \texttt{force}\ V \mid \texttt{abort}\ V \mid \texttt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \\
M, S ::= & \mid \texttt{split}\ V\ \texttt{to}\ ().M \mid \texttt{split}\ V\ \texttt{to}\ (x,y).M \\
 & \mid \texttt{ret} V \mid \texttt{bind}\ x \leftarrow M; N \mid \{\} \mid \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} \mid \pi M \mid \pi' M \mid \lambda x : A.M \mid M\ V \\
\\
E ::= & V \mid M \\
\\
\Gamma ::= & \cdot \mid \Gamma, x : A \\
\Delta ::= & \cdot \mid \bullet : \underline{B} \\
\Phi ::= & \cdot \mid \Phi, x \sqsubseteq x' : A \sqsubseteq A' \\
\Psi ::= & \cdot \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'
\end{array}
$$

Fig. 1. GTT type and term syntax.

a pair is only a value when its components are). The notion of value V is more permissive than one might expect, and expressions $\Gamma \vdash V : A$ are sometimes called *complex values* to emphasize this point: complex values include not only closed runtime values, but also open values that have free value variables (e.g., $x : A_1, x_2 : A_2 \vdash (x_1, x_2) : A_1 \times A_2$), and expressions that pattern match on values (e.g., $p : A_1 \times A_2 \vdash \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).(x_2, x_1) : A_2 \times A_1$). Thus, the complex values $x : A \vdash V : A'$ are a syntactic class of "pure functions" from A to A' (though there is no pure function *type* internalizing this judgement), which can be treated like values by a compiler because they have no effects (e.g., they can be freely duplicated or discarded without affecting the program's effects). For each pattern-matching construct (e.g., case analysis on a sum, splitting a pair), we have both an elimination rule whose branches are values (e.g., $\texttt{split}\ p\ \texttt{to}\ (x_1, x_2).V$) and one whose branches are computations (e.g., $\texttt{split}\ p\ \texttt{to}\ (x_1, x_2).M$). To abbreviate the typing rules for both in Figure 2, we use the following convention defined in Figure 1: E for either a complex value or a computation, and T for either a value type A or a computation type $\underline{B}$, and a judgement $\Gamma \mid \Delta \vdash E : T$ for either $\Gamma \vdash V : A$ or $\Gamma \mid \Delta \vdash M : \underline{B}$ (this is a bit of an abuse of notation because Δ is not present in the former). Complex values can be translated away without loss of expressiveness by moving all pattern matching into computations (see Section 6, but they are convenient for us to use to reason about the fact that certain casts (upcasts) are pure.

Shifts. A key notion in CBPV is the *shift* types $\underline{F}A$ and $U\underline{B}$, which mediate between value and computation types: $\underline{F}A$ is the computation type of potentially effectful programs that return a value of type A, while $U\underline{B}$ is the value type of thunked computations of type $\underline{B}$. The introduction rule for $\underline{F}A$ is returning a value of type A ($\texttt{ret}$V), while the elimination rule is

$$\boxed{\Gamma \vdash V : A \text{ and } \Gamma \mid \Delta \vdash M : \underline{B}} \qquad \textsc{UpCast}\ \frac{\Gamma \vdash V : A \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A' \nwarrow A \rangle V : A'} \qquad \textsc{DnCast}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}' \qquad \underline{B} \sqsubseteq \underline{B}'}{\Gamma \mid \Delta \vdash \langle \underline{B} \swarrow \underline{B}' \rangle M : \underline{B}}$$

$$\textsc{Var}\ \frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \textsc{Hole}\ \frac{}{\Gamma \mid \bullet : \underline{B} \vdash \bullet : \underline{B}} \qquad \textsc{Err}\ \frac{}{\Gamma \mid \cdot \vdash \mho_{\underline{B}} : \underline{B}} \qquad U\text{I}\ \frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \vdash \texttt{thunk}\ M : U\underline{B}}$$

$$U\text{E}\ \frac{\Gamma \vdash V : U\underline{B}}{\Gamma \mid \cdot \vdash \texttt{force}\ V : \underline{B}} \qquad \underline{F}\text{I}\ \frac{\Gamma \vdash V : A}{\Gamma \mid \cdot \vdash \texttt{ret} V : \underline{F}A} \qquad \underline{F}\text{E}\ \frac{\Gamma \mid \Delta \vdash M : \underline{F}A \qquad \Gamma, x : A \mid \cdot \vdash N : \underline{B}}{\Gamma \mid \Delta \vdash \texttt{bind}\ x \leftarrow M; N : \underline{B}}$$

$$0\text{E}\ \frac{\Gamma \vdash V : 0}{\Gamma \mid \Delta \vdash \texttt{abort}\ V : T} \qquad +\textsc{Il}\ \frac{\Gamma \vdash V : A_1}{\Gamma \vdash \texttt{inl}\ V : A_1 + A_2} \qquad +\textsc{Ir}\ \frac{\Gamma \vdash V : A_2}{\Gamma \vdash \texttt{inr}\ V : A_1 + A_2}$$

$$+\text{E}\ \frac{\begin{array}{c}\Gamma \vdash V : A_1 + A_2 \\ \Gamma, x_1 : A_1 \mid \Delta \vdash E_1 : T \\ \Gamma, x_2 : A_2 \mid \Delta \vdash E_2 : T\end{array}}{\Gamma \mid \Delta \vdash \texttt{case}\ V\{x_1.E_1 \mid x_2.E_2\} : T} \qquad 1\text{I}\ \frac{}{\Gamma \vdash () : 1} \qquad 1\text{E}\ \frac{\Gamma \vdash V : 1 \qquad \Gamma \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \texttt{split}\ V\ \texttt{to}\ ().E : T}$$

$$\times\text{I}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \times\text{E}\ \frac{\begin{array}{c}\Gamma \vdash V : A_1 \times A_2 \\ \Gamma, x : A_1, y : A_2 \mid \Delta \vdash E : T\end{array}}{\Gamma \mid \Delta \vdash \texttt{split}\ V\ \texttt{to}\ (x,y).E : T} \qquad \to\text{I}\ \frac{\Gamma, x : A \mid \Delta \vdash M : \underline{B}}{\Gamma \mid \Delta \vdash \lambda x : A.M : A \to \underline{B}}$$

$$\to\text{E}\ \frac{\Gamma \mid \Delta \vdash M : A \to \underline{B} \qquad \Gamma \vdash V : A}{\Gamma \mid \Delta \vdash M\ V : \underline{B}} \qquad \top\text{I}\ \frac{}{\Gamma \mid \Delta \vdash \{\} : \top}$$

$$\&\text{I}\ \frac{\Gamma \mid \Delta \vdash M_1 : \underline{B}_1 \qquad \Gamma \mid \Delta \vdash M_2 : \underline{B}_2}{\Gamma \mid \Delta \vdash \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} : \underline{B}_1 \mathbin{\&} \underline{B}_2} \qquad \&\text{E}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \mid \Delta \vdash \pi M : \underline{B}_1} \qquad \&\text{E'}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \mid \Delta \vdash \pi' M : \underline{B}_2}$$

Fig. 2. GTT typing.

sequencing a computation $M : \underline{F}A$ with a computation $x : A \vdash N : \underline{B}$ to produce a computation of a $\underline{B}$ ($\texttt{bind}\ x \leftarrow M; N$). While any closed complex value V is equivalent to an actual value, a computation of type $\underline{F}A$ might perform effects (e.g., printing) before returning a value, or might error or non-terminate and not return a value at all. The introduction and elimination rules for U are written $\texttt{thunk}\ M$ and $\texttt{force}\ V$, and say that computations of type $\underline{B}$ are bijective with values of type $U\underline{B}$. As an example of the action of the shifts, 0 is the empty value type, so $\underline{F}0$ classifies effectful computations that never return, but may perform effects (and then, must e.g., non-terminate or error), while $U\underline{F}0$ is the value type where such computations are thunked/delayed and considered as values. 1 is the trivial value type, so $\underline{F}1$ is the type of computations that can perform effects with the possibility of terminating successfully by returning (), and $U\underline{F}1$ is the value type where such computations are delayed values. $U\underline{F}$ is a monad on value types (Moggi, 1991), while $\underline{F}U$ is a comonad on computation types.

Computation Types. The computation type constructors in CBPV include first the lazy unit $\top$ and lazy product $\underline{B}_1 \mathbin{\&} \underline{B}_2$, which behave as in a call-by-name language (e.g., a component of a lazy pair is evaluated only when it is projected). Functions $A \to \underline{B}$ have a value type as input and a computation type as a result. The equational theory of effects in CBPV computations may be surprising to those familiar only with call-by-value, because at higher computation types effects have a call-by-name-like equational theory. For example, at computation type $A \to \underline{B}$, $\texttt{print}\ c; \lambda x.M = \lambda x.\texttt{print}\ c; M$. Intuitively, the reason is that $A \to \underline{B}$ is not treated as an *observable* type (one where computations are run): the states of the operational semantics are only those computations of type $\underline{F}A$ for some value type A. So the only way to "run" a function computation is to supply it with an argument, and applying both of the above to an argument V is defined to result in $\texttt{print}\ c; M[V/x]$. This does *not* imply that the corresponding equations holds for the call-by-value function type, which we discuss below. As another example, *all* computations are equal at type $\top$, even computations that perform different effects ($\texttt{print}\ c$ versus {} versus $\mho$), because there is by definition *no* way to use a computation of type $\top$ to produce a term of an observable type $\underline{F}A$. Consequently, $U\top$ is isomorphic to 1.

Complex Stacks. Just as the complex values V are a syntactic class of terms that have no effects, CBPV includes a judgement for "stacks" S, a syntactic class of terms that reflect *all* effects of their input. A *stack* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{B}'$ can be thought of as a linear/strict function from $\underline{B}$ to $\underline{B}'$, which *must* use its input which we write as $\bullet$ *exactly* once at the head redex position. We can always use the same variable name $\bullet$ since stacks always have only one input "hole". Also for this reason we sometimes write the substitutions $S[M/\bullet]$ or $S[S'/\bullet]$ as simply $S[M]$ or $S[S']$. Uses of effects can be hoisted out of stacks, because we know the stack will run them exactly once and first. For example, there will be contextual equivalences $S[\mho] = \mho$ and $S[\texttt{print}\ V; M] = \texttt{print}\ V; S[M]$. Just as complex values include pattern matching, *complex stacks* include pattern matching on values and introduction forms for the stack's output type. For example, $\bullet : \underline{B}_1 \mathbin{\&} \underline{B}_2 \vdash \{\pi \mapsto \pi'\bullet \mid \pi' \mapsto \pi\bullet\} : \underline{B}_2 \mathbin{\&} \underline{B}_1$ is a complex stack, even though it mentions $\bullet$ more than once, because running it requires choosing a projection to get to an observable of type $\underline{F}A$, so *each time it is run* it uses $\bullet$ exactly once. Similarly, $\bullet : U\underline{B} \vdash \{\} : \top$ is a (complex) stack despite never using its input, since computations of type $\top$ are dead code and so the evaluation can never be forced. In the equational theory of CBPV, $\underline{F}$ and U are *adjoint*, in the sense that stacks $\bullet : \underline{F}A \vdash S : \underline{B}$ are bijective with values $x : A \vdash V : U\underline{B}$, as both are bijective with computations $x : A \vdash M : \underline{B}$.

To compress the presentation in Figure 2, we use a typing judgement $\Gamma \mid \Delta \vdash M : \underline{B}$ with a "stoup", a typing context Δ that is either empty or contains exactly one assumption $\bullet : \underline{B}$, so $\Gamma \mid \cdot \vdash M : \underline{B}$ is a computation, while $\Gamma \mid \bullet : \underline{B} \vdash M : \underline{B}'$ is a stack. The typing rules for $\top$ and $\&$ treat the stoup additively (it is arbitrary in the conclusion and the same in all premises); for a function application to be a stack, the stack input must occur in the function position. The elimination form for $U\underline{B}$, $\texttt{force}\ V$, is the prototypical non-stack computation (Δ is required to be empty), because forcing a thunk does not use the stack's input.

Embedding Call-by-Value and Call-by-Name. To help understand how CBPV relates to more standard CBV and CBN evaluation orders, we give a brief overview of their translations into CBPV.

First, CBV types A can be translated to CBPV value types A^v. For CBV with $1, \times, 0, +, \to$ all but $\to$ are translated to themselves and the CBV function type $A \to A'$ is translated to $U(A^v \to \underline{F}A'^v)$: a CBV function is a thunk that takes an argument value and returns a result value. Then a CBV expression $x_1 : A_1, \ldots \vdash e : A$ is translated to a computation $x_1 : A_1^v, \ldots \vdash e^v : \underline{F}A^v$. That is, variables in a CBV expression are bound to values, and cbv expression always return a value (or perform effects). The translation is similar to that of monadic form or ANF in that it introduces many bind/return forms to make the evaluation order explicit.

Next, CBN types $\underline{B}$ can be translated to CBPV computation types $\underline{B}^n$. For CBN with $1, \&, +, 0, \to$, the unit 1 and lazy product & are translated to themselves, but the others must introduce thunks appropriately. The CBN function type $\underline{B}_1 \to \underline{B}_2$ is interpreted as $U\underline{B}_1^n \to \underline{B}_2^n$: a CBN function receives its argument as a thunk and then behaves as its output type. The 0 type is interpreted as $U\underline{F}0$: a computation that returns a value of the empty value type. The $+$ type is similarly interpreted as $\underline{F}(U(\underline{B}_1^n) + U(\underline{B}_2^n))$: a CBN computation of sum type returns a tagged thunk of one of the two options. Next, a CBN expression $x_1 : \underline{B}_1, \ldots \vdash e : \underline{B}$ is translated to a computation $x_1 : U(\underline{B}_1^n), \ldots \vdash e^n : \underline{B}^n$. That is, variables in CBN are always bound to thunks but the expression might not be directly observable without being applied to arguments.

Call-by-push-value *subsumes* call-by-value and call-by-name in that these embeddings are *full and faithful*: two CBV or CBN programs are equivalent if and only if their embeddings into CBPV are equivalent, and every CBPV program with a CBV or CBN type can be back-translated (Levy, 2003).

Computation/β and Extensionality/η Principles. We include the standard CBPV β and η principles in a table in Figure E.2 as *order equivalences*. We'll say more about this ordering later, but for now it can simply be considered to mean observationally equivalent. The β principles tell us how computations can be reduced, and are all reductions in the operational semantics to be defined later. They all essentially establish the same pattern: an introduction form followed by an elimination form reduces, binding variables as appropriate. We review some of the CBPV-specific rules. The $U\beta$ rules says forcing a thunk evaluates to the body of the thunk. The $\underline{F}\beta$ rule acts like a let-rule: binding a return of a value substitutes the value in the continuation. For the lazy product rule, the projection selects the appropriate case to run. There are no rules for 0β and $\top\beta$ since they lack an introduction and elimination rule, respectively.

The main advantage of CBPV for our purposes is that it accounts for the η/extensionality principles of both eager/value types and lazy/computation types. While the β rules are true even in untyped calculi, the η principles encode what reasoning the types give us. Intuitively, they axiomatize that the rules of the system are in some sense complete for observing terms of the type. Except for the shifts $U, \underline{F}$, these follow a certain pattern. For

Type	β	η
$+$	$\texttt{case inl } V\{x_1.E_1 \mid \ldots\} \sqsupseteq\sqsubseteq E_1[V/x_1]$ $\texttt{case inr } V\{\ldots \mid x_2.E_2\} \sqsupseteq\sqsubseteq E_2[V/x_2]$	$E[V/x] \sqsupseteq\sqsubseteq \ \texttt{case } V\{x_1.E[\texttt{inl } x_1/x]$ $\mid x_2.E[\texttt{inr } x_2/x]\}$ where $x : A_1 + A_2 \vdash E : T$
0	–	$E[V/x] \sqsupseteq\sqsubseteq \texttt{abort } V$ where $x : 0 \vdash E : T$
$\times$	$\texttt{split } (V_1, V_2) \texttt{ to } (x_1, x_2).E$ $\sqsupseteq\sqsubseteq E[V_1/x_1, V_2/x_2]$	$E[V/x] \sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } (x_1, x_2).$ $E[(x_1, x_2)/x]$ where $x : A_1 \times A_2 \vdash E : T$
1	$\texttt{split } () \texttt{ to } ().E \sqsupseteq\sqsubseteq E$	$E[V/x] \sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } ().E[()/x] : T$ where $x : 1 \vdash E : T$
U	$\texttt{force thunk } M \sqsupseteq\sqsubseteq M$	$V \sqsupseteq\sqsubseteq \texttt{thunk } (\texttt{force } V)$ $V : U\underline{B}$
$\underline{F}$	$\texttt{bind } x \leftarrow \texttt{ret} V; M \sqsupseteq\sqsubseteq M[V/x]$	$S[M] \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow M; S[\texttt{ret} x]$ where $\bullet : \underline{F}A \vdash S : \underline{B}$
$\to$	$(\lambda x : A.M)\ V \sqsupseteq\sqsubseteq M[V/x]$	$N \sqsupseteq\sqsubseteq \lambda x : A.N\ x$ where $N : A \to \underline{B}$
$\&$	$\pi\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M$ $\pi'\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M'$	$N \sqsupseteq\sqsubseteq \{\pi \mapsto \pi N \mid \pi' \mapsto \pi' N\}$ where $N : \underline{B}_1 \mathbin{\&} \underline{B}_2$
$\top$	-	$N \sqsupseteq\sqsubseteq \{\} : \top$ where $N : \top$

Fig. 3. CBPV/GTT computation and extensionality principles.

value type constructors, the η principle tells us something about terms (values, computations and stacks) that have a *free variable* whose type is formed using the type constructor. The η principle says any term E using x is equivalent to one that immediately matches on the variable x and then *only uses the results of the pattern match* to use x. So, for example, the $+\eta$ rule says a term E is equivalent to one that pattern matches on x and then in each case, uses $\texttt{inl } x_l$ or $\texttt{inr } x_r$ in place of x. This equation formalizes the idea that there is *nothing more* to a value of type $A_1 + A_2$ than the information you get out of it from pattern matching.

The computation type constructor η laws are instead about computations (and stacks) whose type is the relevant type constructor. They say that any computation of a given type is equivalent to one formed using the introduction rule, and whose cases derive from applying the elimination rules to the original computation. So for instance, the function type η says that any computation (or stack) is equivalent to one formed by a λ that applies the original computation to the λ parameter. The $\&\eta$ says any computation of lazy product type is equivalent to a pair whose cases derive from applying the appropriate projection to the original. Finally the $\top\eta$ says that any computation of type $\top$ is “dead code” and equivalent to the empty case.

The final two η principles are for the U and $\underline{F}$ type. The $U\ \eta$ is more like a computation type η in that it tells us something about values of type $U\underline{B}$: all of them are equivalent to a thunk that forces the original value. The $\underline{F}\eta$ is similar to the value type η principles in that it tells us about computations $S[M]$ that are a stack applied to a term $M : \underline{F}A$. Any such stack is equivalent to one that binds M to a value and then uses only that value.

2.2 Gradual typing in GTT

Next, we discuss the additions that make CBPV into our gradual type theory GTT.

The Dynamic Type(s). A dynamic type plays a key role in gradual typing, and since GTT has two different kinds of types, we have a new question of whether the dynamic type should be a value type, or a computation type, or whether we should have *both* a dynamic value type and a dynamic computation type. Our modular, type-theoretic presentation of gradual typing allows for any of these choices, and none of the internal theorems we prove about one depend on the presence of the other. However, when we discuss models of the language in Section 5.2 we will mainly discuss models of GTT with *both* dynamic value and computation, and justify why this does not sacrifice much generality. In our models, values of the dynamic value type ? are tagged values of other types, while computations of the dynamic computation type $\underline{¿}$ are instead like objects that can respond to any "method", i.e., can be applied to any sequence of arguments and projections π_i.

We add both ? and $\underline{¿}$ to the grammar of types in Figure 1. We do *not* give introduction and elimination rules for the dynamic types, because we would like constructions in GTT to imply results for many different possible implementations of them. Instead, the terms for the dynamic types will arise from type precision and casts.

2.2.1 Type precision

The *type precision* relation of gradual type theory is written $A \sqsubseteq A'$ and read as "A is more precise than A'"; intuitively, this means that A' supports more behaviors than A. Our previous work (New & Ahmed, 2018; New & Licata, 2018) analyzes this as the existence of an *upcast* from A to A' and a downcast from A' to A which form an embedding-projection pair (*ep pair*) for term error approximation (an ordering where runtime errors are minimal): the upcast followed by the downcast is a no-op, while the downcast followed by the upcast might error more than the original term, because it imposes a runtime type check. Syntactically, type precision is defined (1) to be reflexive and transitive (a preorder), (2) where every type constructor is monotone in all positions and (3) where the dynamic type is greatest in the type precision ordering. This last condition, *the dynamic type is the most dynamic type*, implies the existence of an upcast $\langle ? \leftarrowtail A\rangle$ and a downcast $\langle A \twoheadleftarrow ?\rangle$ for every type A: any type can be embedded into it and projected from it. However, this by design does not characterize ? uniquely—instead, it is open-ended exactly which types exist (so that we can always add more), and some properties of the casts are undetermined; we exploit this freedom in Section 5.2.

This extends in a straightforward way to CBPV's distinction between value and computation types in Figure 4: there is a type precision relation for value types $A \sqsubseteq A'$ and for computation types $\underline{B} \sqsubseteq \underline{B}'$, which (1) each are preorders (VTyRefl, VTyTrans, CTyRefl, CTyTrans), (2) every type constructor is monotone (+Mon, ×Mon, &Mon ,→Mon) where the shifts $\underline{F}$ and U switch which relation is being considered (UMon,

$$\boxed{A \sqsubseteq A' \text{ and } \underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'} \qquad \frac{}{A \sqsubseteq A}\,\textsc{VTyRefl} \qquad \frac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{A \sqsubseteq A''}\,\textsc{VTyTrans} \qquad \frac{}{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'}\,\textsc{CTyRefl} \qquad \frac{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}' \qquad \underline{B}' \mathrel{\underline{\sqsubseteq}} \underline{B}''}{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}''}\,\textsc{CTyTrans}$$

$$\frac{}{A \sqsubseteq ?}\,\textsc{VTyTop} \qquad \frac{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'}{U\underline{B} \sqsubseteq U\underline{B}'}\,U\textsc{Mon} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 + A_2 \sqsubseteq A_1' + A_2'}\,{+}\textsc{Mon} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 \times A_2 \sqsubseteq A_1' \times A_2'}\,{\times}\textsc{Mon}$$

$$\frac{}{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{¿}}\,\textsc{CTyTop} \qquad \frac{A \sqsubseteq A'}{\underline{F}A \mathrel{\underline{\sqsubseteq}} \underline{F}A'}\,F\textsc{Mon} \qquad \frac{\underline{B}_1 \mathrel{\underline{\sqsubseteq}} \underline{B}_1' \qquad \underline{B}_2 \mathrel{\underline{\sqsubseteq}} \underline{B}_2'}{\underline{B}_1 \mathbin{\&} \underline{B}_2 \mathrel{\underline{\sqsubseteq}} \underline{B}_1' \mathbin{\&} \underline{B}_2'}\,\&\textsc{Mon} \qquad \frac{A \sqsubseteq A' \qquad \underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'}{A \to \underline{B} \mathrel{\underline{\sqsubseteq}} A' \to \underline{B}'}\,{\to}\textsc{Mon}$$

$$\boxed{\text{Precision contexts}} \qquad \frac{}{\cdot \ \mathrm{dyn{-}vctx}} \qquad \frac{\Phi \ \mathrm{dyn{-}vctx} \qquad A \sqsubseteq A'}{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \ \mathrm{dyn{-}vctx}}$$

$$\frac{}{\cdot \ \mathrm{dyn{-}cctx}} \qquad \frac{\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'}{(\bullet \sqsubseteq \bullet : \underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}') \ \mathrm{dyn{-}cctx}}$$

Fig. 4. GTT type precision and precision contexts.

FMon) and (3) the dynamic types ? and $\underline{¿}$ are the most dynamic value and computation types, respectively (VTyTop, CTyTop). For example, we have $U(A \to \underline{F}A') \sqsubseteq U(? \to \underline{F}?)$, which is the analogue of $A \to A' \sqsubseteq ? \to ?$ in call-by-value: because $\to$ preserves embedding-retraction pairs, it is monotone, not contravariant, in the domain (New & Ahmed, 2018; New & Licata, 2018).

2.2.2 *Casts*

It is not immediately obvious how to add type casts to CBPV, because CBPV exposes finer judgemental distinctions than previous work considered. However, we can arrive at a first proposal by considering how previous work would be embedded into CBPV. In the previous work on both CBV and CBN (New & Ahmed, 2018; New & Licata, 2018), every type precision judgement $A \sqsubseteq A'$ induces both an upcast from A to A' and a downcast from A' to A. Because CBV types are associated to CBPV value types and CBN types are associated to CBPV computation types, this suggests that each value type precision $A \sqsubseteq A'$ should induce an upcast and a downcast, and each computation type precision $\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'$ should also induce an upcast and a downcast. In CBV, a cast from A to A' typically can be represented by a CBV function $A \to A'$, whose analogue in CBPV is $U(A \to \underline{F}A')$, and values of this type are bijective with computations $x : A \vdash M : \underline{F}A'$, and further with stacks $\bullet : \underline{F}A \vdash S : \underline{F}A'$. This suggests that a *value* type precision $A \sqsubseteq A'$ should induce an embedding-projection pair of *stacks* $\bullet : \underline{F}A \vdash S_u : \underline{F}A'$ and $\bullet : \underline{F}A' \vdash S_d : \underline{F}A$, which allow both the upcast and downcast to a priori be effectful computations. Dually, a CBN cast typically can be represented by a CBN function of type $B \to B'$, whose CBPV analogue is a computation of type $U\underline{B} \to \underline{B}'$, which is equivalent with a computation $x : U\underline{B} \vdash M : \underline{B}'$, and with a value $x : U\underline{B} \vdash V : U\underline{B}'$. This suggests that a *computation* type precision $\underline{B} \mathrel{\underline{\sqsubseteq}} \underline{B}'$ should induce an embedding-projection pair of *values* $x : U\underline{B} \vdash V_u : U\underline{B}'$ and $x : U\underline{B}' \vdash V_d : U\underline{B}$, where both the upcast and the downcast again may a priori be (co)effectful, in the sense that they may not reflect all effects of their input.

However, this analysis ignores an important property of CBV casts in practice: *upcasts* always terminate without performing any effects, and in some systems upcasts are even defined to be values, while only the *downcasts* are effectful (introduce errors). For example, for many types A, the upcast from A to ? is an injection into a sum/recursive type, which is a value constructor. Our previous work on a logical relation for call-by-value gradual typing (New & Ahmed, 2018) proved that all upcasts were pure in this sense as a consequence of the embedding-projection pair properties (but their proof depended on the only effects being divergence and type error). In GTT, we can make this property explicit in the syntax of the casts, by making the upcast $\langle A' \nwarrow A\rangle$ induced by a value type precision $A \sqsubseteq A'$ itself a complex value, rather than computation. On the other hand, many downcasts between value types are implemented as a case analysis looking for a specific tag and erroring otherwise, and so are not complex values.

We can also make a dual observation about CBN casts. The *downcast* arising from $\underline{B} \sqsubseteq \underline{B}'$ has a stronger property than being a computation $x : U\underline{B}' \vdash M : \underline{B}$ as suggested above: it can be taken to be a stack $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}'\rangle \bullet : \underline{B}$, because a downcasted computation evaluates the computation it is "wrapping" exactly once. One intuitive justification for this point of view, which we make precise in Section 5, is to think of the dynamic computation type $\underline{¿}$ as a recursive *product* of all possible behaviors that a computation might have, and the downcast as a recursive type unrolling and product projection, which is a stack. From this point of view, an *upcast* can introduce errors, because the upcast of an object supporting some "methods" to one with all possible methods will error dynamically on the unimplemented ones.

These observations are expressed in the (shaded) UpCast and DnCasts rules for casts in Figure 2: the upcast for a value type precision is a complex value, while the downcast for a computation type precision is a stack (if its argument is). Indeed, this description of casts is simpler than the intuition we began the section with: rather than putting in both upcasts and downcasts for all value and computation type precisions, it suffices to put in only *upcasts* for *value* type precisions and *downcasts* for *computation* type precisions, because of monotonicity of type precision for $U/\underline{F}$ types. The *downcast* for a *value* type precision $A \sqsubseteq A'$, as a stack $\bullet : \underline{F}A' \vdash \langle \underline{F}A \swarrow \underline{F}A'\rangle \bullet : \underline{F}A$ as described above, is obtained from $\underline{F}A \sqsubseteq \underline{F}A'$ as computation types. The upcast for a computation type precision $\underline{B} \sqsubseteq \underline{B}'$ as a value $x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B}\rangle x : U\underline{B}'$ is obtained from $U\underline{B} \sqsubseteq U\underline{B}'$ as value types. Moreover, we will show below that the value upcast $\langle A' \nwarrow A\rangle$ induces a stack $\bullet : \underline{F}A \vdash \ldots : \underline{F}A'$ that behaves like an upcast, and dually for the downcast, so this formulation implies the original formulation above.

We justify this design in two ways in the remainder of the paper. In Section 5, we show how to implement casts by a contract translation to CBPV where upcasts are complex values and downcasts are complex stacks. However, one goal of GTT is to be able to prove things about many gradually typed languages at once, by giving different models, so one might wonder whether this design rules out useful models of gradual typing where casts can have more general effects. In Theorem 3.7, we show instead that our design choice is forced for all casts, as long as the casts between ground types and the dynamic types are values/stacks.

2.2.3 Term precision: Judgements and structural rules

The final piece of GTT is the *term precision* relation, a syntactic judgement that is used for reasoning about the behavioral properties of terms in GTT. To a first approximation, term precision can be thought of as syntactic rules for reasoning about *contextual approximation* relative to errors (not divergence), where $E \sqsubseteq E'$ means that either E errors or E and E' have the same result. However, a key idea in GTT is to consider a *heterogeneous* term precision judgement $E \sqsubseteq E' : T \sqsubseteq T'$ between terms $E : T$ and $E' : T'$ where $T \sqsubseteq T'$—i.e., relating two terms at two different types, where the type on the right is less precise than the type on the left. This judgement structure allows simple axioms characterizing the behavior of casts (New & Licata, 2018) and axiomatizes the graduality property (Siek *et al.*, 2015). Crucially, we include not just the congruence/monotonicity rules typically used in syntactic type precision, but also rules that close the relation under CBPV $\beta\eta$ equality. Here, we break this judgement up into value precision $V \sqsubseteq V' : A \sqsubseteq A'$ and computation precision $M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$. To support reasoning about open terms, the full form of the judgements are as follows:

- $\Gamma \sqsubseteq \Gamma' \vdash V \sqsubseteq V' : A \sqsubseteq A'$ where $\Gamma \vdash V : A$ and $\Gamma' \vdash V' : A'$ and $\Gamma \sqsubseteq \Gamma'$ and $A \sqsubseteq A'$.
- $\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$ where $\Gamma \mid \Delta \vdash M : \underline{B}$ and $\Gamma' \mid \Delta' \vdash M' : \underline{B}'$.

where $\Gamma \sqsubseteq \Gamma'$ is the pointwise lifting of value type precision, and $\Delta \sqsubseteq \Delta'$ is the optional lifting of computation type precision. We write $\Phi : \Gamma \sqsubseteq \Gamma'$ and $\Psi : \Delta \sqsubseteq \Delta'$ as syntax for "zipped" pairs of contexts that are pointwise related by type precision, $x_1 \sqsubseteq x'_1 : A_1 \sqsubseteq A'_1, \ldots, x_n \sqsubseteq x'_n : A_n \sqsubseteq A'_n$, which correctly suggests that one can substitute related terms for related variables. We will implicitly zip/unzip pairs of contexts, and sometimes write, e.g., $\Gamma \sqsubseteq \Gamma$ to mean $x \sqsubseteq x : A \sqsubseteq A$ for all $x : A$ in Γ.

The main point of our rules for term precision is that *there are no type-specific axioms in the definition* beyond the $\beta\eta$-axioms that the type satisfies in a non-gradual language. Thus, adding a new type to gradual type theory does not require any a priori consideration of its gradual behavior in the language definition; instead, this is deduced as a theorem in the type theory. The basic structural rules of term precision in Figure 5 say that it is reflexive and transitive (TmDynRefl, TmDynTrans), that assumptions can be used and substituted for (TmDynVar, TmDynValSubst, TmDynHole, TmDynStkSubst). We also include *congruence* rules for each term constructor, which essentially says that all term constructors are monotone in every subterm. We include the function cases to give an example, the remaining rules are straightforward and are found in the appendix (Figure A.1).

We will often abbreviate a "homogeneous" term precision (where the type or context precision is given by reflexivity) by writing, e.g., $\Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$ for $\Gamma \sqsubseteq \Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$, or $\Phi \vdash V \sqsubseteq V' : A$ for $\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A$, and similarly for computations. The entirely homogeneous judgements $\Gamma \vdash V \sqsubseteq V' : A$ and $\Gamma \mid \Delta \vdash M \sqsubseteq M' : \underline{B}$ can be thought of as a syntax for contextual error approximation (as we prove below). We write $V \sqsupseteq\sqsubseteq V'$ ("equiprecision") to mean term precision relations in both directions (which requires that the types are also equiprecise $\Gamma \sqsupseteq\sqsubseteq \Gamma'$ and $A \sqsubseteq A'$), which is a syntactic judgement for contextual equivalence.

$$\boxed{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \text{ and } \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynRefl}\ \dfrac{}{\Gamma \sqsubseteq \Gamma \mid \Delta \sqsubseteq \Delta \vdash E \sqsubseteq E : T \sqsubseteq T} \qquad \textsc{TmDynVar}\ \dfrac{}{\Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \vdash x \sqsubseteq x' : A \sqsubseteq A'}$$

$$\textsc{TmDynTrans}\ \dfrac{\begin{array}{c}\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash E \sqsubseteq E' : T \sqsubseteq T' \\ \Gamma' \sqsubseteq \Gamma'' \mid \Delta' \sqsubseteq \Delta'' \vdash E' \sqsubseteq E'' : T' \sqsubseteq T''\end{array}}{\Gamma \sqsubseteq \Gamma'' \mid \Delta \sqsubseteq \Delta'' \vdash E \sqsubseteq E'' : T \sqsubseteq T''}$$

$$\textsc{TmDynValSubst}\ \dfrac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \\ \Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash E[V/x] \sqsubseteq E'[V'/x'] : T \sqsubseteq T'}$$

$$\textsc{TmDynHole}\ \dfrac{}{\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynStkSubst}\ \dfrac{\begin{array}{c}\Phi \mid \Psi \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \sqsubseteq \underline{B}_1' \\ \Phi \mid \bullet \sqsubseteq \bullet : \underline{B}_1 \sqsubseteq \underline{B}_1' \vdash M_2 \sqsubseteq M_2' : \underline{B}_2 \sqsubseteq \underline{B}_2'\end{array}}{\Phi \mid \Psi \vdash M_2[M_1/\bullet] \sqsubseteq M_2'[M_1'/\bullet] : \underline{B}_2 \sqsubseteq \underline{B}_2'}$$

$${\to}\textsc{ICong}\ \dfrac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \Psi \vdash \lambda x : A.M \sqsubseteq \lambda x' : A'.M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

$${\to}\textsc{ECong}\ \dfrac{\Phi \mid \Psi \vdash M \sqsubseteq M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \Psi \vdash M\ V \sqsubseteq M'\ V' : \underline{B} \sqsubseteq \underline{B}'}$$

Fig. 5. GTT term precision (structural rules and selected congruence rules).

2.2.4 Term precision axioms

Finally, we assert some term precision axioms that describe the behavior of programs. The cast universal properties at the top of Figure 6, following New & Licata (2018), say that the defining property of an upcast from A to A' is that it is the most precise term of type A' that is less precise than x, a "least upper bound". That is, $\langle A' \leftarrowtail A\rangle x$ is a term of type A' that is less precise than x (the "bound" rule), and for any other term x' of type A' that is less precise than x, $\langle A' \leftarrowtail A\rangle x$ is more precise than x' (the "best" rule). Dually, the downcast $\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \bullet$ is the most dynamic term of type $\underline{B}$ that is more precise than $\bullet$, a "greatest lower bound". These defining properties are entirely independent of the types involved in the casts, and do not change as we add or remove types from the system.

We will show that these defining properties already imply that the shift of the upcast $\langle A' \leftarrowtail A\rangle$ forms a Galois connection/adjunction with the downcast $\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle$, and dually for computation types (see Theorem 3.3). They do not automatically form a Galois insertion/coreflection/embedding-projection pair, but we can add this by the retract axioms in Figure 6. Together with other theorems of GTT, these axioms imply that any upcast followed by its corresponding downcast is the identity (see Theorem 3.4). This specification of casts leaves some behavior undefined: for example, we cannot prove in the theory that $\langle \underline{F}(1+1) \twoheadleftarrow \underline{F}?\rangle\langle ? \leftarrowtail 1\rangle$ reduces to an error. We choose this design because there are valid models in which it is not an error, for instance if the unique value of 1 is represented as the Boolean `true`. In Section 5.2, we show additional axioms that fully characterize the behavior of the dynamic type.

Cast Universal Properties

	Bound	Best
Up	$x : A \vdash x \sqsubseteq \langle A' \nwarrow A\rangle x : A \sqsubseteq A'$	$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A\rangle x \sqsubseteq x' : A'$
Down	$\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}'\rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$	$\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B} \swarrow \underline{B}'\rangle \bullet : \underline{B}$

Retract Axiom

$$x : A \vdash \langle \underline{F}A \swarrow \underline{F}\,?\rangle(\texttt{ret}(\langle ? \nwarrow A\rangle x)) \sqsubseteq \texttt{ret}x : \underline{F}A$$

$$x : U\underline{B} \vdash \langle \underline{B} \swarrow \underline{\text{¿}}\rangle(\texttt{force }(\langle U\underline{\text{¿}} \nwarrow U\underline{B}\rangle x)) \sqsubseteq \texttt{force } x : \underline{B}$$

Error Properties

$$\textsc{ErrBot}\quad \frac{\Gamma' \mid \cdot \vdash M' : \underline{B}'}{\Gamma \sqsubseteq \Gamma' \mid \cdot \vdash \mho \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'} \qquad \textsc{StkStrict}\quad \frac{\Gamma \mid x : \underline{B} \vdash S : \underline{B}'}{\Gamma \mid \cdot \vdash S[\mho_{\underline{B}}] \sqsubseteq \mho_{\underline{B}'} : \underline{B}'}$$

Fig. 6. GTT term precision axioms.

Additionally, for each of the $\beta\eta$ principles phrased in terms of $\sqsupseteq\sqsubseteq$ in Figure E.2 we add two axioms: $\sqsubseteq$ in each direction.

The final axioms assert properties of the runtime error term $\mho$: it is the most precise term (has the fewest behaviors) of every computation type, and all complex stacks are strict in errors, because stacks force their evaluation position. We state the first axiom in a heterogeneous way, which includes congruence $\Gamma \sqsubseteq \Gamma' \vdash \mho_{\underline{B}} \sqsubseteq \mho_{\underline{B}'} : \underline{B} \sqsubseteq \underline{B}'$. Note in particular that at this point none of the rules introduce an error in a term where it was not already present, because we do not presuppose for instance that casting from function type to dynamic to a product type is an error. We consider both how to add axioms like these and how to construct models where these axioms are not satisfied in Section 5.

3 Theorems in gradual type theory

In this section, we show that the axiomatics of gradual type theory determine most properties of casts, which shows that these behaviors of casts are forced in any implementation of gradual typing satisfying graduality and β, η. When elided, proofs are included in the appendix.

3.1 Derived cast rules

As noted above, monotonicity of type precision for U and $\underline{F}$ means that we have the following as instances of the general cast rules:

Lemma 3.1 (Shifted Casts)**.**
The following are derivable:

$$\frac{\Gamma \mid \Delta \vdash M : \underline{F}A' \qquad A \sqsubseteq A'}{\Gamma \mid \Delta \vdash \langle \underline{F}A \swarrow \underline{F}A'\rangle M : \underline{F}A} \qquad\qquad \frac{\Gamma \vdash V : U\underline{B} \qquad \underline{B} \sqsubseteq \underline{B}'}{\Gamma \vdash \langle U\underline{B}' \nwarrow U\underline{B}\rangle V : U\underline{B}'}$$

Proof. They are instances of the general upcast and downcast rules, using the fact that U and $\underline{F}$ are congruences for type precision, so in the first rule $\underline{F}A \sqsubseteq \underline{F}A'$, and in the second, $U\underline{B} \sqsubseteq U\underline{B}'$. □

The cast universal properties in Figure 6 imply the following seemingly more general rules for reasoning about casts:

Lemma 3.2 (Upcast and downcast left and right rules)**.**
The following are derivable:

$$\dfrac{A' \sqsubseteq A'' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \vdash V \sqsubseteq \langle A'' \nwarrow A' \rangle V' : A \sqsubseteq A''}\ \textsc{UpR} \qquad \dfrac{\Phi \vdash V \sqsubseteq V'' : A \sqsubseteq A''}{\Phi \vdash \langle A' \nwarrow A \rangle V \sqsubseteq V'' : A' \sqsubseteq A''}\ \textsc{UpL}$$

$$\dfrac{\underline{B} \sqsubseteq \underline{B}' \qquad \Phi \mid \Psi \vdash M' \sqsubseteq M'' : \underline{B}' \sqsubseteq \underline{B}''}{\Phi \mid \Psi \vdash \langle \underline{B} \swarrow \underline{B}' \rangle M' \sqsubseteq M'' : \underline{B} \sqsubseteq \underline{B}''}\ \textsc{DnL} \qquad \dfrac{\Phi \mid \Psi \vdash M \sqsubseteq M'' : B \sqsubseteq B''}{\Phi \mid \Psi \vdash M \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle M'' : \underline{B} \sqsubseteq \underline{B}'}\ \textsc{DnR}$$

In sequent calculus terminology, in the term precision judgement an upcast is left-invertible, while a downcast is right-invertible, in the sense that any time we have a conclusion with an upcast on the left/downcast on the right, we can without loss of generality apply these rules (this comes from upcasts and downcasts forming a Galois connection). We write the $A \sqsubseteq A'$ and $\underline{B}' \sqsubseteq \underline{B}''$ premises on the non-invertible rules to emphasize that the premise is not necessarily well formed given that the conclusion is.

We did not include explicit congruence rules for casts in Figure A.1 because they are derivable:

Lemma 3.3 (Cast congruence rules)**.**
The following congruence rules for casts are derivable:

$$\dfrac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A'' \nwarrow A \rangle x \sqsubseteq \langle A'' \nwarrow A' \rangle x' : A''} \qquad \dfrac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{x : A \vdash \langle A' \nwarrow A \rangle x \sqsubseteq \langle A'' \nwarrow A \rangle x : A' \sqsubseteq A''}$$

$$\dfrac{\underline{B} \sqsubseteq \underline{B}' \qquad \underline{B}' \sqsubseteq \underline{B}''}{\bullet' \sqsubseteq \bullet'' : \underline{B}' \sqsubseteq \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet' \sqsubseteq \langle \underline{B} \swarrow \underline{B}'' \rangle \bullet'' : \underline{B}}$$

$$\dfrac{\underline{B} \sqsubseteq \underline{B}' \qquad \underline{B}' \sqsubseteq \underline{B}''}{\bullet'' : \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}'' \rangle \bullet'' \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle \bullet'' : \underline{B} \sqsubseteq \underline{B}'}$$

Proof. In all cases, uses the invertible and then non-invertible rule for the cast. For the first rule, by upcast left, it suffices to show $x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle A'' \nwarrow A' \rangle x' : A \sqsubseteq A''$ which is true by upcast right, using $x \sqsubseteq x'$ in the premise. The other cases follow by a similar argument. □

Next, while in GTT we assume the existence of upcast values from value precision and downcast stacks from computation precision, sometimes we can prove that certain terms satisfy the following definition of "downcast value" and "upcast stack".

In GTT, we assert the existence of value upcasts and computation downcasts for derivable type precision relations. While we do not assert the existence of all *value* downcasts and *computation* upcasts, we can define the universal property that identifies a term as such:

Definition 3.1 (Upcast stack/Value downcast)**.**

1. *If $\underline{B} \sqsubseteq \underline{B}'$, a* stack upcast from B to B' *is a stack* $\bullet : \underline{B} \vdash \langle\langle \underline{B}' \leftarrowtail \underline{B} \rangle\rangle \bullet : \underline{B}'$ *that satisfies the computation precision rules of an upcast* $\bullet : \underline{B} \vdash \bullet \sqsubseteq \langle\langle \underline{B}' \leftarrowtail \underline{B} \rangle\rangle \bullet : \underline{B} \sqsubseteq \underline{B}'$ *and* $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \langle\langle \underline{B}' \leftarrowtail \underline{B} \rangle\rangle \bullet \sqsubseteq \bullet' : \underline{B}'$*.*
2. *If $A \sqsubseteq A'$, a* value downcast from A' to A *is a complex value* $x : A' \vdash \langle\langle A \twoheadleftarrow A' \rangle\rangle x : A$ *that satisfies the value precision rules of a downcast* $x : A' \vdash \langle\langle A \twoheadleftarrow A' \rangle\rangle x \sqsubseteq x : A \sqsubseteq A'$ *and* $x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle\langle A \twoheadleftarrow A' \rangle\rangle x' : A$*.*

One convenient application of this is that we can simplify the statement of several properties by "forgetting" that an upcast $\langle A' \leftarrowtail A \rangle$ is a value, and instead using a derivable upcast $\langle\langle \underline{F}A' \leftarrowtail \underline{F}A \rangle\rangle$ as defined in the following (and dually for computation types).

Definition 3.2 (Upcast stacks/Downcast values[4])**.**
If $A \sqsubseteq A'$, then we define

$$\langle\langle \underline{F}A' \leftarrowtail \underline{F}A \rangle\rangle E = \texttt{bind}\, x \leftarrow E\, \texttt{ret} \langle A' \leftarrowtail A \rangle x.$$

which is an upcast stack.
If $\underline{B} \sqsubseteq \underline{B}'$ then we define

$$\langle\langle U\underline{B} \twoheadleftarrow U\underline{B}' \rangle\rangle V = \texttt{thunk}\, (\langle \underline{B} \twoheadleftarrow \underline{B}' \rangle (\texttt{force}\ V))$$

which is a downcast value.

3.2 Type-generic properties of casts

The universal property axioms for upcasts and downcasts in Figure 6 define them *uniquely* up to equiprecision ($\sqsupseteq\sqsubseteq$): anything with the same property is behaviorally equivalent to a cast.

Theorem 3.1 (Specification for Casts is a Universal Property)**.**

1. *If $A \sqsubseteq A'$ and $x : A \vdash V : A'$ is a complex value such that $x : A \vdash x \sqsubseteq V : A \sqsubseteq A'$ and $x \sqsubseteq x' : A \sqsubseteq A' \vdash V \sqsubseteq x' : A'$ then $x : A \vdash V \sqsupseteq\sqsubseteq \langle A' \leftarrowtail A \rangle x : A'$.*
2. *If $\underline{B} \sqsubseteq \underline{B}'$ and $\bullet' : \underline{B}' \vdash S : \underline{B}$ is a complex stack such that $\bullet' : \underline{B}' \vdash S \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}'$ and $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq S : \underline{B}$ then $\bullet' : \underline{B}' \vdash S \sqsupseteq\sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}' \rangle \bullet' : \underline{B}$*

Proof. For the first part, to show $\langle A' \leftarrowtail A \rangle x \sqsubseteq V$, by upcast left, it suffices to show $x \sqsubseteq V : A \sqsubseteq A'$, which is one assumption. To show $V \sqsubseteq \langle A' \leftarrowtail A \rangle x$, we substitute into the second assumption with $x \sqsubseteq \langle A' \leftarrowtail A \rangle x : A \sqsubseteq A'$, which is true by upcast right.

For the second part, to show $S \sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}' \rangle \bullet'$, by downcast right, it suffices to show $S \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}'$, which is one of the assumptions. To show $\langle \underline{B} \twoheadleftarrow \underline{B}' \rangle \bullet' \sqsubseteq S$, we substitute into the second assumption with $\langle \underline{B} \twoheadleftarrow \underline{B}' \rangle \bullet' \sqsubseteq \bullet'$, which is true by downcast left. □

This shows the *specification* for the casts uniquely determines the behavior of the cast. In the next subsection we will show that we can derive the behavior for many casts.

Casts satisfy an identity and composition law:

[4] Readers familiar with category theory should note that these are simply the functorial actions of the $\underline{F}$ and U type constructors.

Theorem 3.2 (Casts (de)composition)**.** *For any* $A \sqsubseteq A' \sqsubseteq A''$ *and* $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$*:*

1. $x : A \vdash \langle A \nwarrow A\rangle x \sqsupseteq\sqsubseteq x : A$
2. $x : A \vdash \langle A'' \nwarrow A\rangle x \sqsupseteq\sqsubseteq \langle A'' \nwarrow A'\rangle\langle A' \nwarrow A\rangle x : A''$
3. $\bullet : \underline{B} \vdash \langle \underline{B} \swarrow \underline{B}\rangle\bullet \sqsupseteq\sqsubseteq \bullet : \underline{B}$
4. $\bullet : \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}''\rangle\bullet \sqsupseteq\sqsubseteq \langle \underline{B} \swarrow \underline{B}'\rangle(\langle \underline{B}' \swarrow \underline{B}''\rangle\bullet) : \underline{B} \sqsubseteq \underline{B}$

In particular, this composition property implies that the casts into and out of the dynamic type are coherent, for example, if $A \sqsubseteq A'$ then $\langle ? \nwarrow A\rangle x \sqsupseteq\sqsubseteq \langle ? \nwarrow A'\rangle\langle A' \nwarrow A\rangle x$.

Theorem 3.3 (Casts form Galois Connections)**.** *If* $A \sqsubseteq A'$*, then the following hold*

1. $\bullet' : \underline{FA}' \vdash \langle\langle \underline{FA}' \nwarrow \underline{FA}\rangle\rangle\langle \underline{FA} \swarrow \underline{FA}'\rangle\bullet' \sqsubseteq \bullet' : \underline{FA}'$
2. $\bullet : \underline{FA} \vdash \bullet \sqsubseteq \langle \underline{FA} \swarrow \underline{FA}'\rangle\langle\langle \underline{FA}' \nwarrow \underline{FA}\rangle\rangle\bullet : \underline{FA}$

If $\underline{B} \sqsubseteq \underline{B}'$*, then the following hold*

1. $x : U\underline{B}' \vdash \langle U\underline{B}' \nwarrow U\underline{B}\rangle\langle\langle U\underline{B} \swarrow U\underline{B}'\rangle\rangle x \sqsubseteq x : U\underline{B}'$
2. $x : U\underline{B} \vdash x \sqsubseteq \langle\langle U\underline{B} \swarrow U\underline{B}'\rangle\rangle\langle U\underline{B}' \nwarrow U\underline{B}\rangle x : U\underline{B}$

The retract property says roughly that $x \sqsupseteq\sqsubseteq \langle T' \swarrow T\rangle\langle T' \nwarrow T\rangle x$ (upcast then downcast does not change the behavior), strengthening the $\sqsubseteq$ of Theorem 3.3. In Figure 6, we asserted the retract axiom for casts with the dynamic type. This and the composition property implies the retraction property for general casts:

Theorem 3.4 (Retract Property for General Casts)**.** *If* $A \sqsubseteq A'$ *and* $\underline{B} \sqsubseteq \underline{B}'$*, then*

1. $\bullet : \underline{FA}' \vdash \langle\langle \underline{FA}' \nwarrow \underline{FA}\rangle\rangle\langle \underline{FA} \swarrow \underline{FA}'\rangle\bullet \sqsupseteq\sqsubseteq \bullet : \underline{FA}'$
2. $x : U\underline{B} \vdash \langle\langle U\underline{B} \swarrow U\underline{B}'\rangle\rangle\langle U\underline{B}' \nwarrow U\underline{B}\rangle x \sqsupseteq\sqsubseteq x : U\underline{B}$

3.3 Deriving behavior of casts

We now come to the central technical consequence of the axioms of GTT, that we can *derive* the behavior of most casts from just η principles and our definition of upcasts and downcasts as least upper bounds and greatest lower bounds, respectively. We call these "unique implementation" theorems because they derive an implementation from the specification that by Theorem 3.1 is unique up to observational equivalence: any implementation that satisfies graduality and the associated η principle must be equivalent to the one given here.

Together, the universal property for casts and the η principles for each type imply that the casts must behave as in "wrapping" cast semantics, which we will demonstrate more explicitly in Section 4:

Theorem 3.5 (Cast Unique Implementation Theorem for $+, \times, \to, \&$)**.** *All of the equivalences in Figure 7 are derivable.*

$$\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle s \sqsupseteq\sqsubseteq \texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1' \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2' \nwarrow A_2\rangle x_2)\}$$

$$\begin{aligned}\langle \underline{F}(A_1' + A_2') \swarrow \underline{F}(A_1 + A_2)\rangle\bullet \quad \sqsupseteq\sqsubseteq \quad & \texttt{bind}\ (s : (A_1' + A_2')) \leftarrow \bullet; \texttt{case}\ s \\ & \{x_1'.\ \texttt{bind}\ x_1 \leftarrow (\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle(\texttt{ret} x_1')); \\ & \qquad \texttt{ret}(\texttt{inl}\ x_1) \\ & \mid x_2'.\ \texttt{bind}\ x_2 \leftarrow (\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle(\texttt{ret} x_2'));\ \} \\ & \qquad \texttt{ret}(\texttt{inr}\ x_2)\end{aligned}$$

$$\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle p \sqsupseteq\sqsubseteq \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).(\langle A_1' \nwarrow A_1\rangle x_1, \langle A_2' \nwarrow A_2\rangle x_2)$$

$$\begin{aligned}\langle \underline{F}(A_1' \times A_2') \swarrow \underline{F}(A_1 \times A_2)\rangle\bullet \quad \sqsupseteq\sqsubseteq \quad & \texttt{bind}\ p' \leftarrow \bullet; \texttt{split}\ p'\ \texttt{to}\ (x_1', x_2'). \\ & \texttt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \texttt{ret} x_1'; \\ & \texttt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \texttt{ret} x_2'; \texttt{ret}(x_1, x_2) \\[1ex] \sqsupseteq\sqsubseteq \quad & \texttt{bind}\ p' \leftarrow \bullet; \texttt{split}\ p'\ \texttt{to}\ (x_1', x_2'). \\ & \texttt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \texttt{ret} x_2'; \\ & \texttt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \texttt{ret} x_1'; \texttt{ret}(x_1, x_2)\end{aligned}$$

$$\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\bullet \sqsupseteq\sqsubseteq \{\pi \mapsto \langle \underline{B}_1 \swarrow \underline{B}_1'\rangle \pi\bullet \mid \pi' \mapsto \langle \underline{B}_2 \swarrow \underline{B}_2'\rangle \pi'\bullet\}$$

$$\begin{aligned}& \langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle p \\ & \quad \sqsupseteq\sqsubseteq \texttt{thunk}\quad \{\pi \mapsto \texttt{force}\ (\langle U\underline{B}_1' \nwarrow U\underline{B}_1\rangle(\texttt{thunk}\ \pi(\texttt{force}\ p))) \quad \} \\ & \qquad\qquad\qquad \mid \pi' \mapsto \texttt{force}\ (\langle U\underline{B}_2' \nwarrow U\underline{B}_2\rangle(\texttt{thunk}\ \pi'(\texttt{force}\ p)))\end{aligned}$$

$$\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\bullet \sqsupseteq\sqsubseteq \lambda x.\langle \underline{B} \swarrow \underline{B}'\rangle(\bullet\ (\langle A' \nwarrow A\rangle x))$$

$$\begin{aligned}\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle f \sqsupseteq\sqsubseteq \texttt{thunk}\ (\lambda x'.\ & \texttt{bind}\ x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle(\texttt{ret} x'); \\ & \texttt{force}\ (\langle U\underline{B}' \nwarrow U\underline{B}\rangle(\texttt{thunk}\ ((\texttt{force}\ f)\, x)))\end{aligned}$$

$$\begin{aligned}\sqsupseteq\sqsubseteq \texttt{thunk}\ (\lambda x'.\texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle(\texttt{thunk}\ (\ & \texttt{bind}\ x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle(\texttt{ret} x'); \\ & (\texttt{force}\ f)\, x)))\end{aligned}$$

Fig. 7. Derivable Cast Behavior for $+, \times, \&, \to$

Proof. The proofs are included in the appendix, and use the upcast/downcast lemmas 3.5, 3.6, which we define at the end of this subsection. □

For each value type connective, we derive the semantics of the upcast and the semantics of the corresponding downcast where $\underline{F}$ is applied to the connective. Dually for the computation type connectives we derive the downcast and the upcast where a U is applied. Note that all of the definitions of casts are essentially the same as the definitions of the operational behavior given in the "wrapping" semantics of gradual typing.

Notably, for the eager product $\times$ and the function type $\to$, we derive that two a priori different implementations both satisfy the specification and so are equivalent. Consider first the upcast implementation $\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle V$. We simply pattern match on the input and cast each side:

$$\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle V \sqsupseteq\sqsubseteq \texttt{split}\ V\ \texttt{to}\ (x_1, x_2).(\langle A_1' \nwarrow A_1\rangle x_1, \langle A_2' \nwarrow A_2\rangle x_2)$$

Since upcasts are values, it doesn't matter in which order these two upcasts are done. On the other hand, consider the downcast between $\underline{F}$ of two product types $\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle$. We start by binding the hole to a variable p, and splitting it into its components x_1 and x_2:

$$\texttt{bind}\ p \leftarrow \bullet; \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).M$$

What should M be? The analogous step to the upcast would be to downcast each component of the pair x_1 and x_2 and form a new pair with the results. However unlike the upcast case, downcasts are effectful so we must choose which one to evaluate first. Either the left:

$$M = \texttt{bind}\ \langle A_1 \swarrow A_1' \rangle[\texttt{ret}x_1] \leftarrow y_1; \texttt{bind}\ \langle A_2 \swarrow A_2' \rangle[\texttt{ret}x_2] \leftarrow y_2; \texttt{ret}(y_1, y_2)$$

Or the right:

$$M = \texttt{bind}\ \langle A_1 \swarrow A_1' \rangle[\texttt{ret}x_1] \leftarrow y_1; \texttt{bind}\ \langle A_2 \swarrow A_2' \rangle[\texttt{ret}x_2] \leftarrow y_2; \texttt{ret}(y_1, y_2)$$

Both of these turn out to be equivalent in GTT's inequational theory. It makes sense operationally that these two are equivalent, since all either can do is error. If we were to incorporate blame, then each side might raise a different error but would blame the same party.

There is a similar (non-)choice for the function type, which is intuitively the choice between enforcing domain or codomain first. When upcasting a thunked function type $\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle$, we start by creating a thunk and taking an argument

$$\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle V_f \sqsupseteq\sqsubseteq \texttt{thunk}\ (\lambda y : A'.M)$$

We then have two choices. First, we can downcast the input first, and then upcast a thunk that calls the original function.

$$M = \texttt{bind}\ x \leftarrow \langle \underline{F}A \swarrow \underline{F}A' \rangle[\texttt{ret}y]; \texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle(\texttt{thunk}\ ((\texttt{force}\ V_f)x))$$

Or we can upcast a thunk that will downcast the input itself:

$$M = \texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle(\texttt{thunk}\ (\texttt{bind}\ [\leftarrow \langle \underline{F}A \swarrow \underline{F}A' \rangle; \texttt{ret}y]x(\texttt{force}\ V_f)x))$$

If $\underline{B} = \underline{F}A_o$ and $\underline{B}' = \underline{F}A_o'$, then there is no ambiguity as we clearly must downcast the input first, call the function and then upcast the result, and both are equivalent to the call-by-value function cast:

$$\texttt{bind}\ [\leftarrow \langle \underline{F}A \swarrow \underline{F}A' \rangle; \texttt{ret}y]x\texttt{bind}\ x_o \leftarrow (\texttt{force}\ V_f)x; \texttt{ret}\langle A_o' \nwarrow A_o\rangle x$$

However, if $\underline{B} = A_2 \to \underline{B}_2$ and $\underline{B}' = A_2' \to \underline{B}_2'$ then the function types are $U(A_1 \to A_2 \to \underline{B}_2)$ and $U(A_1' \to A_2' \to \underline{B}_2')$ and correspond to functions of two arguments in call-by-value. Then the choice of enforcing domain or codomain first corresponds to the choice of enforcing argument contracts from left-to-right or right-to-left (or anything in between for further more inputs). As with the product, the orderings turn out to be equivalent.

We can similarly derive cast implementations for the "double shifts":

Theorem 3.6 (Cast Unique Implementation Theorem for $U\underline{F}, \underline{F}U$)**.** *Let $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$.*

1. $x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A\rangle x \sqsupseteq\sqsubseteq \texttt{thunk}\ (\langle\langle \underline{F}A' \nwarrow \underline{F}A\rangle\rangle(\texttt{force}\ x)) : U\underline{F}A'$
2. $\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle \bullet \sqsupseteq\sqsubseteq \texttt{bind}\ x' : U\underline{B}' \leftarrow \bullet; \mathit{ret}(\langle\langle U\underline{B} \swarrow U\underline{B}'\rangle\rangle x)$

Proof. The proofs are in the appendix, but use the upcast/downcast Lemmas 3.5, 3.6. □

While we can prove each of these cases directly, the proofs are fairly repetitive and similar. Instead we package up the proof principle into a couple of lemmas, which abstract over the details of the proof. First, since all of these proof principles are parameterized,

$$\begin{array}{rcl} A+ & ::= & X \\ \underline{B}+ & ::= & \underline{Y} \\ \Theta & ::= & \cdot \mid \Theta, X \ \text{val type} \mid \Theta, \underline{Y} \ \text{comp type} \end{array}$$

Fig. 8. GTT open types.

we need to formally define parameterized types in order to prove our general lemmas. We define these *open* types in Figure 8, which adds value and computation type variables to GTT. We write θ for a substitution of (correctly kinded) type variables and write $\theta \sqsubseteq \theta'$ to mean that type precision holds pointwise. We say a substitution θ instantiates Θ if for each type variable X val type ($\underline{Y}$ comp type), $\theta(X)$ is a closed value type (resp. computation type).

Then we can discuss type constructors as simply types with non-empty Θ. For example,

$$\begin{array}{l} X_1 \ \text{val type}, X_2 \ \text{val type} \vdash X_1 + X_2 \ \text{val type} \\ \underline{Y} \ \text{comp type} \vdash U\underline{Y} \ \text{val type} \\ X_1 \ \text{val type}, X_2 \ \text{val type} \vdash \underline{F}(X_1 + X_2) \ \text{comp type} \end{array}$$

are all open types. It is easy to see that all type constructors are monotone in type precision, because we included a congruence rule for every type constructor in Figure 4:

Lemma 3.4 (Monotonicity of Type Constructors)**.** *For any type constructor Θ val type $\vdash C$, if $\theta \sqsubseteq \theta'$ then $C[\theta] \sqsubseteq C[\theta']$.*

Proof. By induction on C. □

The following lemma gives a method to show a polymorphic value

$$\llparenthesis C[\theta'] \nwarrow C[\theta] \rrparenthesis$$

is an upcast from $C[\theta]$ to $C[\theta']$. It reduces to verification of three properties: well-typedness, monotonicity and identity extension. Of these, only identity extension is nontrivial to prove. This lemma will be used to prove the unique implementation theorems.

Lemma 3.5 (Upcast Lemma)**.** *Let $\Theta \vdash C$ val type be an open value type.*

Suppose $\llparenthesis C[\theta'] \nwarrow C[\theta] \rrparenthesis -$ is a family of values, parameterized by θ, θ' such that

1. *(Well-typedness) For all typing substitutions $\theta \sqsubseteq \theta'$ instantiating Θ,*

$$x : C[\theta] \vdash \llparenthesis C[\theta'] \nwarrow C[\theta] \rrparenthesis x : C[\theta']$$

2. *(Monotonicity) For all substitutions $\theta_l, \theta'_l, \theta_r, \theta'_r$ that instantiate Θ and satisfy $\theta_l \sqsubseteq \theta'_l$, $\theta_l \sqsubseteq \theta_r$, $\theta'_l \sqsubseteq \theta'_r$ and $\theta_r \sqsubseteq \theta'_r$,*

$$x \sqsubseteq x' : C[\theta_l] \sqsubseteq C[\theta_r] \vdash \llparenthesis C[\theta_r] \nwarrow C[\theta_l] \rrparenthesis \sqsubseteq \llparenthesis C[\theta'_r] \nwarrow C[\theta'_l] \rrparenthesis : C[\theta_r] \sqsubseteq C[\theta'_r]$$

3. *(Identity Extension) For all substitutions θ instantiating Θ,*

$$x : C[\theta] \vdash \llparenthesis C[\theta] \nwarrow C[\theta] \rrparenthesis x \sqsupseteq\sqsubseteq x : C[\theta]$$

Then if $\theta \sqsubseteq \theta'$, then $\llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis$ satisfies the universal property of an upcast, so by Theorem 3.1

$$x : C[\theta] \vdash \llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x \sqsupseteq\sqsubseteq \langle C[\theta'] \leftarrowtail C[\theta] \rangle x : C[\theta']$$

Moreover, the left-to-right direction uses only the left-to-right direction of identity extension, and the right-to-left uses only the right-to-left direction.

Proof. First we need to show

$$x \sqsubseteq x' : C[\theta] \sqsubseteq C[\theta'] \vdash \llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x \sqsubseteq x' : C[\theta'].$$

Monotonicity gives that

$$\llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x \sqsubseteq \llparenthesis C[\theta'] \leftarrowtail C[\theta'] \rrparenthesis x' : C[\theta']$$

but by the left-to-right direction of identity extension the right hand side is more precise than x', so transitivity gives the result:

$$\llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x \sqsubseteq \llparenthesis C[\theta'] \leftarrowtail C[\theta'] \rrparenthesis x' \sqsubseteq x'$$

The other direction is similar. To show

$$x : C[\theta] \vdash x \sqsubseteq \llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x : C[\theta] \sqsubseteq C[\theta']$$

By monotonicity, we have

$$\llparenthesis C[\theta] \leftarrowtail C[\theta] \rrparenthesis x \sqsubseteq \llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x : C[\theta] \sqsubseteq C[\theta']$$

so transitivity with the right-to-left direction of identity extension gives the result:

$$x \sqsubseteq \llparenthesis C[\theta] \leftarrowtail C[\theta] \rrparenthesis x \sqsubseteq \llparenthesis C[\theta'] \leftarrowtail C[\theta] \rrparenthesis x$$

Then Theorem 3.1 implies that $\llparenthesis C[\theta'] \leftarrowtail C[\overline{A_i}, \overline{\underline{B}_i}] \rrparenthesis$ is equivalent to $\langle C[\theta'] \leftarrowtail C[\overline{A_i}, \overline{\underline{B}_i}] \rangle$. □

We have then also the exact dual lemma for downcasts:

Lemma 3.6 (Downcast Lemma). *Let $\Theta \vdash \underline{C}$ comp type be an computation type.*

Suppose $\llparenthesis \underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta'] \rrparenthesis -$ is a family of stacks parameterized by θ, θ' satisfying the following properties.

1. *(Well-typedness) For all $\theta \sqsubseteq \theta'$ instantiating Θ*

$$\bullet : \underline{C}[\theta'] \vdash \llparenthesis \underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta'] \rrparenthesis \bullet : \underline{C}[\theta]$$

2. *(Monotonicity) For all substitutions $\theta_l, \theta_l', \theta_r, \theta_r'$ that instantiate Θ and satisfy $\theta_l \sqsubseteq \theta_l'$, $\theta_l \sqsubseteq \theta_r$, $\theta_l' \sqsubseteq \theta_r'$ and $\theta_r \sqsubseteq \theta_r'$,*

$$\bullet \sqsubseteq \bullet : \underline{C}[\theta_r] \sqsubseteq \underline{C}[\theta_r'] \vdash \llparenthesis \underline{C}[\theta_l] \twoheadleftarrow \underline{C}[\theta_r] \rrparenthesis \bullet \sqsubseteq \llparenthesis \underline{C}[\theta_l'] \twoheadleftarrow \underline{C}[\theta_r'] \rrparenthesis \bullet' : \underline{C}[\theta_l] \sqsubseteq \underline{C}[\theta_l']$$

3. *(Identity Extension) For all substitutions θ instantiating Θ,*

$$\bullet : \underline{C}[\theta] \vdash \llparenthesis \underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta] \rrparenthesis \bullet \sqsupseteq\sqsubseteq \bullet : \underline{C}[\theta]$$

Then $(\!|\underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta']|\!)$ satisfies the universal property of a downcast, so by Theorem 3.1

$$\bullet : \underline{C}[\theta'] \vdash (\!|\underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta']|\!)\bullet \sqsupseteq\sqsubseteq \langle \underline{C}[\theta] \twoheadleftarrow \underline{C}[\theta']\rangle\bullet : \underline{C}[\theta]$$

Moreover, the left-to-right direction uses only the left-to-right direction of identity extension, and the right-to-left uses only the right-to-left direction of identity extension.

Proof. The proof is the exact dual of the proof of Lemma 3.5. □

As an example derivation we prove the case for a downcast for function types:

$$\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\bullet \sqsupseteq\sqsubseteq \lambda x.\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle(\bullet\,(\langle A' \leftarrowtail A\rangle x))$$

Here the type constructor is X val type, $\underline{Y}$ comp type $\vdash X \to Y$ comp type. We apply the downcast lemma with the definition being

$$(\!|A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'|\!) = \lambda x.\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle(\bullet\,(\langle A' \leftarrowtail A\rangle x))$$

Then well-typedness clearly holds, and monotonicity follows by congruence for all constructors and Lemma 3.3 for the casts. Finally, for identity extension we need to show

$$\lambda x.\langle \underline{B} \twoheadleftarrow \underline{B}\rangle(\bullet\,(\langle A \leftarrowtail A\rangle x)) \sqsupseteq\sqsubseteq \bullet : A \to \underline{B}$$

First, by the decomposition Theorem 3.2 this is equivalent to

$$\lambda x.\bullet\ x \sqsupseteq\sqsubseteq \bullet : A \to \underline{B}$$

Which is precisely η equivalence for →. The cases for the other connectives proceed similarly.

3.4 Upcasts must be values, downcasts must be stacks

It may seem like an arbitrary choice to define upcasts as values and downcasts as stacks, rather than the a priori more general definition that upcasts from A to A' are effectful terms $x : A \vdash \underline{F}A'$, which is equivalent to assuming that they are given by a stack upcast $\langle \underline{F}A' \leftarrowtail \underline{F}A\rangle$ and dually that computations be given by an a priori nonlinear term $z : U\underline{B}' \vdash U\underline{B}$, which is equivalent to a value downcast $\langle U\underline{B} \twoheadleftarrow U\underline{B}'\rangle$. However, we show now that this choice is essentially *forced* upon us, under the mild assumption that certain "ground" up/downcasts are values/stacks. For this section, we define a *ground type*[5] to be generated by the following grammar:

$$G ::= 1 \mid ? \times ? \mid 0 \mid ? + ? \mid U\underline{¿} \qquad \underline{G} ::= ? \to \underline{¿} \mid \top \mid \underline{¿} \,\&\, \underline{¿} \mid \underline{F}?$$

Let GTT$_G$ be the fragment of GTT where the only primitive casts are those between ground types and the dynamic types, i.e., the cast terms are restricted to $\langle ? \leftarrowtail G\rangle V$, $\langle \underline{F}G \twoheadleftarrow \underline{F}?\rangle$, $\langle \underline{G} \twoheadleftarrow \underline{¿}\rangle E$, $\langle U\underline{¿} \leftarrowtail U\underline{G}\rangle E$.

Lemma 3.7 (Casts are Admissible). *In GTT$_G$, it is admissible that*

1. *for all $A \sqsubseteq A'$ there is a complex value $\langle\!\langle A' \leftarrowtail A\rangle\!\rangle$ satisfying the universal property of an upcast and a complex stack $\langle\!\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\!\rangle$ satisfying the universal property of a downcast*

[5] In gradual typing, "ground" is used to mean a one-level unrolling of a dynamic type, not first-order data.

2. *for all $\underline{B} \sqsubseteq \underline{B}'$ there is a complex stack $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ satisfying the universal property of a downcast and a complex value $\langle\!\langle U\underline{B}' \nwarrow U\underline{B} \rangle\!\rangle$ satisfying the universal property of an upcast.*

Proof. To streamline the exposition above, we stated Theorems 3.2, 3.5 , 3.6 as showing that the "definitions" of each cast are equiprecise with the cast that is a priori postulated to exist (e.g., $\langle A'' \nwarrow A \rangle \sqsupseteq\sqsubseteq \langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle$). However, the proofs factor through Theorem 3.1 and Lemmas 3.5 and 3.6, which show directly that the right-hand sides have the desired universal property—i.e., the stipulation that some cast with the correct universal property exists is not used in the proof that the implementation has the desired universal property. Moreover, the proofs given do not rely on any axioms of GTT besides the universal properties of the "smaller" casts used in the definition and the $\beta\eta$ rules for the relevant types. So these proofs can be used as the inductive steps here, in GTT$_G$.

In the appendix (Definition B.1) we define an alternative type precision relation where casts into dynamic types are factored through ground types, and use that to drive the induction here. □

As discussed in Section 2.2.2, rather than an upcast being a complex value $x : A \vdash \langle A' \nwarrow A \rangle x : A'$, an a priori more general type would be a stack $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$, which allows the upcast to perform effects; dually, an a priori more general type for a downcast $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$ would be a value $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$, which allows the downcast to ignore its argument. The following shows that in GTT$_G$, if we postulate such stack upcasts/value downcasts as originally suggested in Section 2.2.2, then in fact these casts *must* be equal to the action of $U/\underline{F}$ on some value upcasts/stack downcasts, so the potential for effectfulness/nonlinearity affords no additional flexibility.

Theorem 3.7 (Upcasts are Necessarily Values, Downcasts are Necessarily Stacks)**.** *Suppose we extend GTT$_G$ with the following postulated stack upcasts and value downcasts (in the sense of Definition 3.1): For every type precision $A \sqsubseteq A'$, there is a stack upcast $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$, and for every $\underline{B} \sqsubseteq \underline{B}'$, there is a complex value downcast $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$.*

Then there exists a value upcast $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ and a stack downcast $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ such that

$$\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet \sqsupseteq\sqsubseteq (\texttt{bind}\, x : A \leftarrow \bullet; \texttt{ret}\, (\langle\!\langle A' \nwarrow A \rangle\!\rangle x))$$
$$x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x \sqsupseteq\sqsubseteq (\texttt{thunk}\, (\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle (\texttt{force}\, x)))$$

Proof. Lemma 3.7 constructs $\langle\langle A' \nwarrow A \rangle\rangle$ and $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$, so the proof of Theorem 3.6 (which really works for any $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ and $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ with the correct universal properties, not only the postulated casts) implies that the right-hand sides of the above equations are stack upcasts and value downcasts of the appropriate type. Since stack upcasts/value downcasts are unique by an argument analogous to Theorem 3.1, the postulated casts must be equal to these. □

Indeed, the following a priori even more general assumption provides no more flexibility:

Theorem 3.8 (Upcasts are Necessarily Values, Downcasts are Necessarily Stacks II)**.** *Suppose we extend* GTT_G *only with postulated monadic upcasts* $x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A\rangle x : U\underline{F}A'$ *for every* $A \sqsubseteq A'$ *and comonadic downcasts* $\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle \bullet : \underline{F}U\underline{B}$ *for every* $\underline{B} \sqsubseteq \underline{B}'$.

Then there exists a value upcast $\langle\!\langle A' \nwarrow A\rangle\!\rangle$ *such that*

$$x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A\rangle x \sqsupseteq\sqsubseteq \texttt{thunk}\,(\texttt{bind}\, x : A \leftarrow \texttt{force}\, x; \texttt{ret}\,(\langle\!\langle A' \nwarrow A\rangle\!\rangle x))$$

and a stack downcast $\langle\!\langle \underline{B} \swarrow \underline{B}'\rangle\!\rangle$ *such that*

$$\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle \bullet \sqsupseteq\sqsubseteq \texttt{bind}\, x' : U\underline{B}' \leftarrow \bullet;$$
$$\texttt{ret}\,(\texttt{thunk}\,(\langle\!\langle \underline{B} \swarrow \underline{B}'\rangle\!\rangle(\texttt{force}\, x)))$$

In CBV terms, the monadic upcast is like an upcast from A to A' having type $(1 \to A) \to A'$, i.e., it takes a thunked effectful computation of an A as input and produces an effectful computation of an A'.

Proof. Again, Lemma 3.7 constructs $\langle\langle A' \nwarrow A\rangle\rangle$ and $\langle\langle \underline{B} \swarrow \underline{B}'\rangle\rangle$, so the proof of Theorem 3.6 gives the result. □

3.5 Equiprecision and isomorphism

There are two natural notions of equivalence of types in GTT: *equiprecision* and *isomorphism*. We say value types A and A' are equiprecise, written $A \sqsupseteq\sqsubseteq A'$ when they are equivalent in the precision ordering in that $A \sqsubseteq A'$ and $A' \sqsubseteq A$. Computation type precision is defined analogously. In CBPV, the appropriate definition of isomorphism is *pure* value isomorphism between value types and *linear* stack isomorphism between computation types, defined as follows:

Definition 3.3 (Isomorphism)**.**

1. *We write* $A \cong_v A'$ *for a* value isomorphism between A and A', *which consists of two values* $x : A \vdash V' : A'$ *and* $x' : A' \vdash V : A$ *such that* $x : A \vdash V[V'/x'] \sqsupseteq\sqsubseteq x : A$ *and* $x' : A' \vdash V'[V/x] \sqsupseteq\sqsubseteq x' : A'$.
2. *We write* $\underline{B} \cong_c \underline{B}'$ *for a* computation isomorphism between $\underline{B}$ and $\underline{B}'$, *which consists of two stacks* $\bullet : \underline{B} \vdash S' : \underline{B}'$ *and* $\bullet' : \underline{B}' \vdash S : \underline{B}$ *such that* $\bullet : \underline{B} \vdash S[S'/\bullet] \sqsupseteq\sqsubseteq \bullet : \underline{B}$ *and* $\bullet' : \underline{B}' \vdash S'[S/\bullet] \sqsupseteq\sqsubseteq \bullet' : \underline{B}'$.

Note that value and computation isomorphisms are a stronger condition than isomorphism in call-by-value and call-by-name. An isomorphism in call-by-value between types A and A' corresponds to a computation isomorphism $\underline{F}A \cong_c \underline{F}A'$, and dually a call-by-name isomorphism between $\underline{B}$ and $\underline{B}'$ corresponds to a value isomorphism $U\underline{B} \cong_v U\underline{B}'$ (Levy, 2017).

As discussed in our previous work on call-by-name GTT (New & Licata, 2018, 2020), equiprecision is stronger than isomorphism: isomorphism says that the "elements" of the types are in one-to-one correspondence, but equiprecision says additionally that those "elements" are represented in the same way at the dynamic type. To see this formally, first observe:

Theorem 3.9 (Equiprecision implies Isomorphism)**.**

1. *If $A \mathrel{\sqsupseteq\sqsubseteq} A'$, then $\langle A' \leftarrowtail A\rangle$ and $\langle A \leftarrowtail A'\rangle$ form a value isomorphism $A \cong_v A'$.*
2. *If $\underline{B} \mathrel{\sqsupseteq\sqsubseteq} \underline{B}'$, then $\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle$ and $\langle \underline{B}' \twoheadleftarrow \underline{B}\rangle$ form a computation isomorphism $\underline{B} \cong_c \underline{B}'$.*

On the other hand, we should not expect that isomorphism implies equiprecision, since there are many nontrivial isomorphisms that will have different encodings in the dynamic type. For instance there is an isomorphism $U\underline{B} \times 1 \cong_v U\underline{B}$ but the former will typically be represented as a pair of a thunk and a dummy value. For another example, $U(\underline{B}_1 \mathbin{\&} \underline{B}_2) \cong_v U\underline{B}_1 \times U\underline{B}_2$ but the former would typically be represented as a single closure that can be called with either of two methods, whereas the latter will be a pair of two closures each of which implements one of the two methods.

3.6 Most precise types

Though it is common in gradually typed surface languages to have a *most* dynamic type in the form of the dynamic type ?, it is less common to have a *least* dynamic type $\bot$. Having a least dynamic type causes issues with certain definitions. For instance sometimes the type consistency relation $A \sim A'$ is defined as existence of a type more precise than each: $\exists A_l . A_l \sqsubseteq A \wedge A_l \sqsubseteq A'$, but this definition would be trivial given the presence of a most precise type.

We consider here the *semantic* consequences of having a least dynamic/most precise value type $\bot_{\mathrm{v}}$ or computation type $\bot_{\mathrm{c}}$. In either case, the consequences are mild: the most precise value type $\bot_{\mathrm{v}}$ must be isomorphic to 0 while for the most precise computation type $\bot_{\mathrm{c}}$ we cannot derive that $\bot_{\mathrm{c}} \cong \top$, we can prove $U\bot_{\mathrm{c}} \cong U\top$.

In the case of the most precise value type $\bot_{\mathrm{v}}$, we have a pure value $x : \bot_{\mathrm{v}} \vdash \langle A \leftarrowtail \bot_{\mathrm{v}}\rangle x : A$ for every value type A. This suggests that the empty type 0 is a candidate to be $\bot_{\mathrm{v}}$, and in fact we can show the two are isomorphic. To prove this we first recall some general facts about the empty type, in category theoretic terms that it is a *strictly initial* object.

Lemma 3.8 ((strictly) initial object)**.** *All of the following are true.*

1. *For all (value or computation) types T, there exists a unique expression $x : 0 \vdash E : T$. In category-theoretic terms,* 0 *is initial in the category of value types and values.*
2. *For all $\underline{B}$, there exists a unique stack $\bullet : \underline{F}0 \vdash S : \underline{B}$. In category-theoretic terms, $\underline{F}0$ is initial in the category of computation types and stacks.*
3. *Suppose there is a type A with a complex value $x : A \vdash V : 0$. Then V is an isomorphism $A \cong_v 0$. In category-theoretic terms,* 0 *is* strictly *initial.*

Note however that we cannot prove that $\underline{F}0$ is *strictly* initial in the category of stacks. With this lemma in hand, we can show that $\bot_{\mathrm{v}}$ must be value-isomorphic to 0:

Theorem 3.10 (Most Precise Value Type)**.** *If* $\bot_{\mathrm{v}}$ *is a type such that* $\bot_{\mathrm{v}} \sqsubseteq A$ *for all* A*, then in GTT with* 0*,* $\bot_{\mathrm{v}} \cong_v 0$.

Proof. We have the upcast $x : \bot_{\mathrm{v}} \vdash \langle 0 \leftarrowtail \bot_{\mathrm{v}} \rangle x : 0$, so Lemma 3.8 gives the result. □

However, note that unless we already know there is an empty type 0, we see no way to prove that $\bot_{\mathrm{v}}$ is initial in that all terms $x : \bot_{\mathrm{v}} \vdash M$ are equivalent.

Thinking dually, a most precise computation type would have a linear stack $\bullet : \underline{B} \vdash \langle \bot_{\mathrm{c}} \twoheadleftarrow \underline{B} \rangle \bullet : \bot_{\mathrm{c}}$ for every computation type $\underline{B}$, so an obvious candidate would be the lazy unit $\top$, the dual of the empty type. However, the duality here is not perfect and we will only be able to prove the weaker fact that $U\top$ and $U\bot_{\mathrm{c}}$ are isomorphic.

To prove this, we first recall the defining property of $\top$, that it is in category-theoretic terms a *terminal object*, but not provably a *strictly* terminal object, breaking the precise duality with 0.

Lemma 3.9 (Terminal objects)**.**

1. *For any computation type* $\underline{B}$*, there exists a unique stack* $\bullet : \underline{B} \vdash S : \top$*, i.e.,* $\top$ *is a terminal object in the category of computation types and stacks.*
2. *(In any context* Γ*,) there exists a unique complex value* $V : U\top$*, i.e.,* $U\top$ *is a terminal object in the category of value types and values.*
3. *(In any context* Γ*,) there exists a unique complex value* $V : 1$*, i.e.,* 1 *is also a terminal object.*
4. $U\top \cong_v 1$

Note that we cannot show that $\top$ is strictly terminal. Next, we can show that $U\bot_{\mathrm{c}}$ is isomorphic to $U\top$.

Theorem 3.11 (Most Precise Computation Type)**.** *If* $\bot_{\mathrm{c}}$ *is a type such that* $\bot_{\mathrm{c}} \sqsubseteq \underline{B}$ *for all* $\underline{B}$*, and we have a terminal computation type* $\top$*, then* $U\bot_{\mathrm{c}} \cong_v U\top$.

Proof. First, though we can define stacks $\bullet : \top \vdash \langle \bot_{\mathrm{c}} \twoheadleftarrow \top \rangle \bullet : \bot_{\mathrm{c}}$ and $\bullet : \bot_{\mathrm{c}} \vdash \{\} : \top$, we can only prove one direction of the isomorphism:

$$\bullet : \top \vdash \{\}[\langle \bot_{\mathrm{c}} \twoheadleftarrow \top \rangle \bullet / \bullet] = \{\} \sqsupseteq\sqsubseteq \bullet : \top$$

Since $\top$ is not a strict terminal object, the dual of the above argument does not give the other property of a stack isomorphism $\bot_{\mathrm{c}} \cong_c \top$.

On the other hand, we can define values

$$x : U\bot_{\mathrm{c}} \vdash \langle U\top \leftarrowtail U\bot_{\mathrm{c}} \rangle x : U\top$$

$$y : U\top \vdash \langle\langle U\bot_{\mathrm{c}} \twoheadleftarrow U\top \rangle\rangle y : U\bot_{\mathrm{c}}$$

And these do exhibit the isomorphism $U\bot_{\mathrm{c}} \cong_v U\top$. First, by the retract axiom

$$x : U\bot_{\mathrm{c}} \vdash \langle\langle U\bot_{\mathrm{c}} \twoheadleftarrow U\top \rangle\rangle \langle U\top \leftarrowtail U\bot_{\mathrm{c}} \rangle x \sqsupseteq\sqsubseteq x : U\bot_{\mathrm{c}}$$

Types	A	$::=$	$? \mid A \to A \mid 1 \mid A \times A \mid 0 \mid A + A$
Ground types	G	$::=$	$? \to ? \mid 1 \mid ? \times ? \mid 0 \mid ? + ?$
Terms	M, N	$::=$	$\mho \mid x \mid \texttt{let}\ x = M; N \mid \langle A \Leftarrow A \rangle M \mid () \mid \texttt{split}\ M\ \texttt{to}\ ().N$
		$\mid$	$(M, N) \mid \texttt{split}\ M\ \texttt{to}\ (x, y).N \mid \texttt{inl}\ M \mid \texttt{inr}\ M$
		$\mid$	$\texttt{case}\ M\{x_1.N_1 \mid x_2.N_2\} \mid \lambda x : A.M \mid M\, N$
Values	V	$::=$	$\langle ? \Leftarrow G \rangle V \mid \lambda x : A.M \mid () \mid (V, V) \mid \texttt{inl}\ V \mid \texttt{inr}\ V$
Evaluation Contexts	S	$::=$	$\bullet \mid \langle B \Leftarrow A \rangle S \mid \texttt{let}\ x = S; N \mid (S, N) \mid (V, S)$
		$\mid$	$\texttt{split}\ S\ \texttt{to}\ (x, y).N \mid \texttt{inl}\ S \mid \texttt{inr}\ S$
		$\mid$	$\texttt{case}\ S\{x_1.N_1 \mid x_2.N_2\} \mid S\, N \mid V\, S$
Environments	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Substitutions	γ	$::=$	$\cdot \mid \gamma, V/x$

Fig. 9. CBV cast calculus.

and the opposite composite

$$y : U\top \vdash \langle U\top \nwarrow U\bot_c \rangle \langle\!\langle U\bot_c \swarrow U\top \rangle\!\rangle y : U\top$$

is the identity by uniqueness for $U\top$ (Lemma 3.9). □

Given these two Theorems 3.10, 3.11, it is then sensible to ask what are the consequences of *defining* 0 and $\top$ to be most precise types. If this is the case then, like in Section , we can derive what the behavior of their casts would be.

Theorem 3.12. *If* $0 \sqsubseteq A$*, then*

$$z : 0 \vdash \langle A \nwarrow 0 \rangle z \sqsupseteq\sqsubseteq \texttt{absurd}\ z \qquad \bullet : \underline{F}A \vdash \langle \underline{F}0 \swarrow \underline{F}A \rangle \bullet \sqsupseteq\sqsubseteq \texttt{bind}\ \_ \leftarrow \bullet; \mho_{\underline{F}0}$$

If $\top \sqsubseteq \underline{B}$*, then*

$$\bullet : \top \vdash \langle \top \swarrow \underline{B} \rangle \bullet \sqsupseteq\sqsubseteq \{\} \qquad u : U\top \vdash \langle U\underline{B} \nwarrow U\top \rangle u \sqsupseteq\sqsubseteq \texttt{thunk}\ \mho$$

4 Application: Deriving call-by-value operational semantics

To show how GTT can be used to inform the semantics of cast calculi, we show how the uniqueness principles of Theorem 3.1 justify most of the operational behavior of a standard Call-by-value cast calculus. A similar process is possible for call-by-name but we have previously studied this in New & Licata (2018) so we do not cover it here.

4.1 A call-by-value cast calculus

We present the syntax of a typical call-by-value cast calculus in Figure 9, borrowed from previous work (New & Ahmed, 2018), but using syntax for pattern matching that is in line with GTT. We define ground types G to be each of the non-? connectives applied to ?. A big difference from GTT is that casts are not separated into upcasts and downcasts a priori: instead there is a cast $\langle A \Leftarrow A' \rangle M$ for any two types:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \langle A' \Leftarrow A \rangle M : A'}$$

Other than this rule, all other typing rules are those of the simply typed λ calculus (STLC). As in a typical call-by-value calculus, instead of values V being a separate syntactic category from general terms M, they are instead a subset. Values include the ordinary STLC values, and additionally tagged values of the dynamic type $\langle ? \Leftarrow G \rangle V$. We fix the evaluation order by defining *evaluation contexts*, which we write as S since they correspond to CBPV stacks.

Next in Figure 10, we present the operational semantics of our calculus. The first six rules correspond to ordinary CBV reductions so we don't bother to name them. The remaining rules are specific to casts. First, ?ID says that casting from ? to ? is the identity. The next two rules DECOMPUP,DECOMPDN break down complex casts to and from the dynamic type to go through the associated ground type (note that for every type A except ?, there is precisely one ground type G such that $A \sqsubseteq G$). The next two rules TAGMATCH,TAGMISMATCH say that casting a tagged value $\langle ? \Leftarrow G \rangle V$ to a ground type G' succeeds if the tag is the same ($G = G'$) and fails if the tag is different ($G \neq G'$). Finally, the SILLY rule is a catch all that says when casting between two completely unrelated types, the cast fails. The remaining rules give the behavior of casts between two types with the same head connective, implementing the wrapping strategy.

4.2 From CBV to GTT

Our goal is to show that the operational reductions of our CBV cast calculus are in a sense *derivable* from the axioms of GTT. To make this concrete, we will define a type-preserving translation of our CBV calculus terms M into GTT computations M^c and prove that for all but two reduction rules $M \mapsto N$ in the CBV calculus, $M^c \sqsupseteq\sqsubseteq N^c$ is provable in GTT. The only rules that do not follow from the axioms of GTT are those that result in errors: TAGMISMATCH and SILLY. The reason for this is that nothing in GTT encodes the "disjointness" of different type connectives, and so from the perspective of our axiomatics, which types are disjoint is a design decision for the models. We explore in Section 5.2 some alternative design choices for gradual languages.

We define the type and term translation in Figure 11. First, we translate CBV types A to CBPV value types $\underline{F}A$, with the only nontrivial case being the translation of function types. Next the computation type translation is mostly straightforward, making the evaluation order explicit using `bind` $M \leftarrow x; N$. The only nonstandard case is the rule for casts, where as discussed in the Introduction, we define the semantics of all casts to factorize as an upcast to the dynamic type followed by a downcast out of the dynamic type. Finally note that since we are working in CBV, we never need to use the computation dynamic type $\underline{¿}$ because it never appears in the type translation of any CBV type.

Theorem 4.1. *If $M \mapsto N$ by any rule except* TAGMISMATCH *or* SILLY*, then $M^c \sqsupseteq\sqsubseteq N^c$.*

Proof. The proof is in the appendix. Besides some basic lemmas for manipulating substitutions and evaluation contexts, the correspondence of cases is as follows:

$$S[\texttt{let } x = V; N] \mapsto S[N[V/x]] \qquad S[(\lambda x : A.M)\ V] \mapsto S[M[V/x]]$$

$$S[\texttt{split } ()\ \texttt{to}\ ().N] \mapsto S[N] \qquad S[\texttt{split } (V_1, V_2)\ \texttt{to}\ (x_1, x_2).N] \mapsto S[N[V_1/x_1][V_2/x_2]]$$

$$S[\texttt{case inl } V\{x_1.N_1 \mid x_2.N_2\}] \mapsto S[N_1[V/x_1]]$$

$$S[\texttt{case inr } V\{x_1.N_1 \mid x_2.N_2\}] \mapsto S[N_2[V/x_2]]$$

$$\textsc{?Id} \quad S[\langle ? \Leftarrow ?\rangle V] \mapsto S[V] \qquad \textsc{DecompUp} \quad \frac{A \sqsubseteq G \qquad A \neq G}{S[\langle ? \Leftarrow A\rangle V] \mapsto S[\langle ? \Leftarrow G\rangle\langle G \Leftarrow A\rangle V]}$$

$$\textsc{DecompDn} \quad \frac{A \sqsubseteq G \qquad A \neq G}{S[\langle A \Leftarrow ?\rangle V] \mapsto S[\langle A \Leftarrow G\rangle\langle G \Leftarrow ?\rangle V]} \qquad \textsc{TagMatch} \quad S[\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V] \mapsto S[V]$$

$$\textsc{TagMismatch} \quad \frac{G \neq G'}{S[\langle G' \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V] \mapsto \mho} \qquad \textsc{Silly} \quad \frac{A \sqsubseteq G_A \qquad B \sqsubseteq G_B \qquad G_A \neq G_B}{S[\langle B \Leftarrow A\rangle V] \mapsto \mho}$$

$$\rightarrow\textsc{Cast} \quad S[\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle V] \mapsto S[\lambda x : A_1'.\langle A_2' \Leftarrow A_2\rangle(V\ (\langle A_1 \Leftarrow A_1'\rangle x))]$$

$$1\textsc{Cast} \quad S[\langle 1 \Leftarrow 1\rangle()] \mapsto S[()]$$

$$\times\textsc{Cast} \quad S[\langle A_1' \times A_2' \Leftarrow A_1 \times A_2\rangle(V_1, V_2)] \mapsto S[(\langle A_1' \Leftarrow A_1\rangle V_1, \langle A_2' \Leftarrow A_2\rangle V_2)]$$

$$+\textsc{CastL} \quad S[\langle A_1' + A_2' \Leftarrow A_1 + A_2\rangle(\texttt{inl } V)] \mapsto S[\langle A_1' \Leftarrow A_1\rangle V]$$

$$+\textsc{CastR} \quad S[\langle A_1' + A_2' \Leftarrow A_1 + A_2\rangle(\texttt{inr } V)] \mapsto S[\langle A_2' \Leftarrow A_2\rangle V]$$

Fig. 10. CBV cast calculus operational semantics.

1. ?Id and 1Cast follow by the decomposition Theorem 3.2.
2. TagMatch follows by the retract property.
3. The remaining cast cases →Cast, ×Cast, +CastL, +CastR follow by the cases for the corresponding connective in Theorem 3.5.

□

5 Contract models of GTT

To show the soundness of our theory, and demonstrate its relationship to operational definitions of observational equivalence and the gradual guarantee, we develop *models* of

$$
\begin{aligned}
?^{ty} &= ? \\
(A \to A')^{ty} &= U(A^{ty} \to \underline{F}A'^{ty}) \\
1^{ty} &= 1 \\
(A_1 \times A_2)^{ty} &= A_1^{ty} \times A_2^{ty} \\
0^{ty} &= 0 \\
(A_1 + A_2)^{ty} &= A_1^{ty} + A_2^{ty}
\end{aligned}
$$

$$
\begin{aligned}
x^c &= \texttt{ret} x \\
(\texttt{let } x = M; N)^c &= \texttt{bind } x \leftarrow M^c; N^c \\
(\langle A_2 \Leftarrow A_1 \rangle M)^c &= \langle \underline{F}A_2^{ty} \swarrow \underline{F}?\rangle \langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{ty} \rangle\!\rangle [M^c] \\
(\lambda x : A.M)^c &= \texttt{ret}(\texttt{thunk } (\lambda x : A^{ty}.M^c)) \\
(M\,N)^c &= \texttt{bind } f \leftarrow M^c; \texttt{bind } x \leftarrow N^c; \texttt{force } f\, x \\
()^c &= \texttt{ret}() \\
(\texttt{split } S \,\texttt{to}\, ().N)^c &= \texttt{bind } z \leftarrow S^c; \texttt{split } z \,\texttt{to}\, ().N^c \\
(M_1, M_2)^c &= \texttt{bind } x_1 \leftarrow M_1^c; \texttt{bind } x_2 \leftarrow M_2^c; \texttt{ret}(x_1, x_2) \\
(\texttt{split } M \,\texttt{to}\, (x, y).N)^c &= \texttt{bind } z \leftarrow M^c; \texttt{split } z \,\texttt{to}\, (x, y).N^c \\
(\texttt{abort } M)^c &= \texttt{bind } z \leftarrow M^c; \texttt{abort } z \\
(\texttt{inl } M)^c &= \texttt{bind } x \leftarrow M^c; \texttt{retinl } x \\
(\texttt{inr } M)^c &= \texttt{bind } x \leftarrow M^c; \texttt{retinr } x \\
(\texttt{case } M\{x_1.N_1 \mid x_2.N_2\})^c &= \texttt{bind } z \leftarrow M^c; \texttt{case } z\{x_1.N_1^c \mid x_2.N_2^c\}
\end{aligned}
$$

Fig. 11. CBV to GTT translation.

GTT using observational error approximation of a *non-gradual* CBPV calculus. We call this the *contract translation* because it translates the built-in casts of the gradual language into ordinary terms implemented in a non-gradual language. While contracts are typically implemented in a dynamically typed language, our target is typed, retaining type information similarly to manifest contracts (Greenberg *et al.*, 2010). We give some implementations of the dynamic value type in the usual way as a recursive sum of basic value types, i.e., using type tags. We also give some more exotic implementations of the dynamic computation type to demonstrate the design space. These are a kind of dual: a recursive product of basic computation types that we can think of as an "object-oriented" dynamic type that is a universal receiver of any message.

Writing $[\![M]\!]$ for any of the contract translations, the remaining sections of the paper establish two main theorems that give a semantic meaning to the axiomatic term precision relation. First, we will show that if two terms of the same type are equi-dynamic, then their elaborations are observationally equivalent. This gives a simple interpretation of equi-precision for terms. For simplicity, we fix $\underline{F}(1+1)$ as the type of observations. This is fairly arbitrary; we could also have chosen $\underline{F}1$ or $\underline{F}$ applied to any finite datatype and arrive at essentially equivalent results.

Theorem 5.1 (Equi-precision implies Observational Equivalence). *If* $\Gamma \vdash M_1 \sqsupseteq\sqsubseteq M_2 : \underline{B}$*, then for any closing GTT context* $C : (\Gamma \vdash \underline{B}) \Rightarrow (\cdot \vdash \underline{F}(1+1))$*,* $[\![C[M_1]]\!]$ *and* $[\![C[M_2]]\!]$ *have the same behavior: both diverge, both run to an error, or both run to* $\mathtt{true}$ *or both run to* $\mathtt{false}$*.*

Second, we give a semantic meaning to the term precision relation. We interpret it using a kind of observational approximation that is analogous to observational equivalence, but capturing the idea that one side may error. However, there is an additional difficulty which is that while equi-precise terms have the same type, and so can be placed in the same context, in general if $M_1 \sqsubseteq M_2$ then M_1 has a more precise type than M_2, so we cannot necessarily place them in the same context. To overcome this issue, we can insert casts on either term to force them to be of the same type, and then apply a straightforward notion of observational approximation. We formalize this as follows, noting that a "valid interpretation of the dynamic types" will be defined later (Definition 5.2):

Theorem 5.2 (Graduality). *If* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash M_1 \sqsubseteq M_2 : B_1 \sqsubseteq B_2$*, then for any GTT context* $C : (\Gamma_1 \vdash B_1) \Rightarrow (\cdot \vdash \underline{F}(1+1))$*, and any valid interpretation of the dynamic types, either*

1. $[\![C[M_1]]\!] \Downarrow \mho$*, or*
2. $[\![C[M_1]]\!] \Uparrow$ *and* $[\![C[\langle B_1 \twoheadleftarrow B_2\rangle M_2[\langle \Gamma_2 \leftarrowtail \Gamma_1\rangle \Gamma_1]]]\!] \Uparrow$*, or*
3. $[\![C[M_1]]\!] \Downarrow \mathtt{ret} V$*,* $\quad [\![C[\langle B_1 \twoheadleftarrow B_2\rangle M_2[\langle \Gamma_2 \leftarrowtail \Gamma_1\rangle \Gamma_1]]]\!] \Downarrow \mathtt{ret} V$*, and* $V = \mathtt{true}$ *or* $V = \mathtt{false}$*.*

This is not precisely the same as definitions of the gradual guarantee (Siek *et al.*, 2015) that are defined by saying that term precision is an invariant of the operational semantics, since those give a direct theorem about how two programs of different type evaluate. For instance the original gradual guarantee would directly imply that if $M_1 \sqsubseteq M_2$ and M_1 reduces to a value then so does M_2. This can still be derived from our theorem in a more complex way using some additional operational reasoning. First, by our theorem if $\cdot \vdash M_1 \sqsubseteq M_2 : FA_1 \sqsubseteq FA_2$, then

$$\mathtt{bind}\ x \leftarrow M_1; \mathtt{rettrue} \sqsubseteq \mathtt{bind}\ x \leftarrow M_2; \mathtt{rettrue} : F(1+1)$$

Next, it is easy to see from the determinism of the operational semantics (to be introduced later) that for any N, $[\![N]\!]$ reduces to a value if and only if $[\![\mathtt{bind}\ x \leftarrow N; \mathtt{rettrue}]\!]$ reduces to $\mathtt{true}$. So if $[\![M_1]\!]$ reduces to a value, $[\![\mathtt{bind}\ x \leftarrow M_1; \mathtt{rettrue}]\!]$ reduces to $\mathtt{true}$. Then by the third case of the graduality theorem (using the identity context), $[\![\mathtt{bind}\ x \leftarrow M_2; \mathtt{rettrue}]\!]$ reduces to $\mathtt{true}$ and so $[\![M_2]\!]$ reduces to a value.

As a corollary we deduce that the logic of precision is consistent.

Corollary 5.1 (Consistency of GTT). $\cdot \vdash \mathtt{rettrue} \sqsubseteq \mathtt{retfalse} : \underline{F}(1+1)$ *is not provable in GTT.*

Proof. They are distinguished by the identity context. □

We break down this proof into three major steps.

1. (This section) We translate GTT into a statically typed CBPV* language where the casts of GTT are translated to "contracts" in CBPV*: i.e., CBPV terms that implement the runtime type checking. We translate the term precision of GTT to an inequational theory for CBPV. Our translation is parameterized by the implementation of the dynamic types, and we demonstrate several implementations.
2. (Section 6) Next, we eliminate all uses of complex values and stacks from the CBPV language. We translate the complex values and stacks to terms with a proof that they are "pure" (thunkable or linear Munch-Maccagnoni, 2014). This part has little to do with GTT specifically, except that it shows the behavioral property that corresponds to upcasts being complex values and downcasts being complex stacks.
3. (Section 7) Finally, with complex values and stacks eliminated, we give a standard operational semantics for CBPV and define a *logical relation* that is sound and complete with respect to observational error approximation. Using the logical relation, we show that the inequational theory of CBPV is sound for observational error approximation.

By composing these, we get a model of GTT where equiprecision is sound for observational equivalence and an operational semantics that satisfies the graduality theorem.

5.1 Call-by-push-value

Next, in Figure 12, we define the call-by-push-value language CBPV* that will be the target for our contract translations of GTT. We write $+ ::=$ and $- ::=$ to indicate the differences from the grammar in Figure 1. CBPV* is almost a subset of GTT obtained as follows: We remove the casts and the dynamic types $?, \underline{¿}$ (the shaded pieces) from the syntax and typing rules in Figures 1 and 2. There is no type precision, and the inequational theory of CBPV* is the homogeneous fragment of term precision in Figure 5 and Figure A.1 (judgements $\Gamma \vdash E \sqsubseteq E' : T$ where $\Gamma \vdash E, E' : T$, with all the same rules in that figure thus restricted). The inequational axioms are the Type Universal Properties ($\beta\eta$ rules) and Error Properties (with ErrBot made homogeneous) from Figure 6. See the appendix (Figures E.1, E.2, E.3) for an explicit description of these rules. To implement the casts and dynamic types, we *add* general (iso-)*recursive* value types ($\mu X.A$, the fixed point of X val type $\vdash A$ val type) and (iso-)*corecursive* computation types ($\nu \underline{Y}.\underline{B}$, the fixed point of $\underline{Y}$ comp type $\vdash \underline{B}$ comp type). The recursive type $\mu X.A$ is a value type with constructor `roll`, whose eliminator is pattern matching, whereas the corecursive type $\nu \underline{Y}.\underline{B}$ is a computation type defined by its eliminator (`unroll`), with an introduction form that we also write as `roll`. We extend the inequational theory with monotonicity of each term constructor of the recursive types, and with their $\beta\eta$ rules. Note that CBPV* is the axiomatic version of call-by-push-value *with* complex values and stacks, while CBPV , (defined in Section 6) will designate the operational version of call-by-push-value with only operational values and stacks.

$$
\begin{array}{llll}
\text{Value Types} & A & +::= & \mu X.A \mid X \\
 & & -::= & ? \\
\text{Computation Types} & \underline{B} & +::= & \nu \underline{Y}.\underline{B} \mid \underline{Y} \\
 & & -::= & \underline{¿} \\
\text{Values} & V & +::= & \texttt{roll}_{\mu X.A}\ V \\
 & & -::= & \langle A \nwarrow A\rangle V \\
\text{Terms} & M & +::= & \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M \mid \texttt{unroll}\ M \\
 & M & -::= & \langle \underline{B} \swarrow \underline{B}\rangle M \\
\text{Both} & E & +::= & \texttt{unroll}\ V\ \texttt{to roll}\ x.E
\end{array}
$$

$$
\frac{\Gamma \vdash V : A[\mu X.A/X]}{\Gamma \vdash \texttt{roll}_{\mu X.A}\ V : \mu X.A}\ \mu\text{I}
\qquad
\frac{\Gamma \vdash V : \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \texttt{unroll}\ V\ \texttt{to roll}\ x.E : T}\ \mu\text{E}
$$

$$
\frac{\Gamma \mid \Delta \vdash M : \underline{B}[\nu \underline{Y}.\underline{B}]}{\Gamma \mid \Delta \vdash \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M : \nu \underline{Y}.\underline{B}}\ \nu\text{I}
$$

$$
\frac{\Gamma \mid \Delta \vdash M : \nu \underline{Y}.\underline{B}}{\Gamma \mid \Delta \vdash \texttt{unroll}\ M : \underline{B}[\nu \underline{Y}.\underline{B}]}\ \nu\text{E}
\qquad
\frac{\Gamma \vdash V \sqsubseteq V' : A[\mu X.A/X]}{\Gamma \vdash \texttt{roll}\ V \sqsubseteq \texttt{roll}\ V' : \mu X.A}\ \mu\text{ICong}
$$

$$
\frac{\Gamma \vdash V \sqsubseteq V' : \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \mid \Delta \vdash E \sqsubseteq E' : T}{\Gamma \mid \Delta \vdash \texttt{unroll}\ V\ \texttt{to roll}\ x.E \sqsubseteq \texttt{unroll}\ V'\ \texttt{to roll}\ x.E' : T}\ \mu\text{ECong}
$$

$$
\frac{\Gamma \mid \Delta \vdash M \sqsubseteq M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \mid \Delta \vdash \texttt{roll}\ M \sqsubseteq \texttt{roll}\ M' : \nu \underline{Y}.\underline{B}}\ \nu\text{ICong}
$$

$$
\frac{\Gamma \mid \Delta \vdash M \sqsubseteq M' : \nu \underline{Y}.\underline{B}}{\Gamma \mid \Delta \vdash \texttt{unroll}\ M \sqsubseteq \texttt{unroll}\ M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}\ \nu\text{ECong}
$$

$$\boxed{\text{Recursive Type Axioms}}$$

Type	β	η
μ	$\texttt{unroll roll}\ V\ \texttt{to roll}\ x.E \sqsupseteq\sqsubseteq E[V/x]$	$E \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ y.E[\texttt{roll}\ y/x]$ where $x : \mu X.A \vdash E : T$
ν	$\texttt{unroll roll}\ M \sqsupseteq\sqsubseteq M$	$\bullet : \nu \underline{Y}.\underline{B} \vdash \bullet \sqsupseteq\sqsubseteq \texttt{roll unroll}\ \bullet : \nu \underline{Y}.\underline{B}$

Fig. 12. CBPV* types, terms, recursive types (differences from GTT).

5.2 Interpreting the dynamic types

As shown in Theorems 3.2, 3.5, 3.6, almost all of the contract translation is uniquely determined already. However, the interpretation of the dynamic types and the casts between the dynamic types and ground types G and $\underline{G}$ are not determined (they were still postulated in Lemma 3.7). For this reason, our translation is *parameterized* by an interpretation of the dynamic types and the ground casts. By Theorems 3.3, 3.4, we know that these must be *embedding-projection pairs* (ep pairs), which we now define in CBPV*. There are two kinds of ep pairs we consider: those between value types and those between computation types. For the value ep pairs, the embedding models the upcast $\langle A' \nwarrow A\rangle$ and the projection models the downcast $\langle \underline{F}A \swarrow \underline{F}A'\rangle$. For the computation ep pairs, the projection models the downcast $\langle \underline{B} \swarrow \underline{B}'\rangle$ and the embedding models the upcast $\langle U\underline{B}' \nwarrow U\underline{B}\rangle$.

Definition 5.1 (Value and Computation Embedding-Projection Pairs)**.**

1. *A* value ep pair *from A to A' consists of an* embedding *value V_e typed as $x : A \vdash V_e : A'$ and* projection *stack $\bullet : \underline{F}A' \vdash S_p : \underline{F}A$, satisfying the* retraction *and* projection *properties:*

$$x : A \vdash \mathtt{ret}\,x \sqsupseteq\sqsubseteq S_p[\mathtt{ret}\,V_e] : \underline{F}A \qquad \bullet : \underline{F}A' \vdash \mathtt{bind}\ x \leftarrow S_p; \mathtt{ret}\,V_e \sqsubseteq \bullet : \underline{F}A'$$

2. *A* computation ep pair *from $\underline{B}$ to $\underline{B}'$ consists of an* embedding *value $z : U\underline{B} \vdash V_e : U\underline{B}'$ and a* projection *stack $\bullet : \underline{B}' \vdash S_p : \underline{B}$ satisfying* retraction *and* projection *properties:*

$$z : U\underline{B} \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq S_p[\mathtt{force}\ V_e] : \underline{B} \quad w : U\underline{B}' \vdash V_e[\mathtt{thunk}\ S_p[\mathtt{force}\ w]/z] \sqsubseteq w : U\underline{B}'$$

When it is clear from context, we sometimes write $V_e[V']$ for $V_e[V'/x]$.

These are related to more standard notions of embedding-projection pairs as follows. A value ep pair is equivalent to an ep pair between $\underline{F}A$ and $\underline{F}A'$ in the stack category where the embedding is induced by a value $A \vdash A'$. Similarly, a computation ep pair is equivalent to an ep pair between $U\underline{B}$ and $U\underline{B}'$ in that value category where the projection is induced by a stack $\underline{B}' \vdash \underline{B}$. Note that our value ep pairs are equivalent to the notion called a *pre-embedding* in Lindenhovius *et al.* (2019). Readers familiar with Galois connections should note that ep pairs are essentially Galois connections where one of the two orderings is an equivalence.

While this formulation is very convenient in that both kinds of ep pairs are pairs of a value and a stack, the projection properties are sometimes easier to use in the following form:

Lemma 5.1 (Alternative Projection)**.** *If (V_e, S_p) is a value ep pair from A to A' and $\Gamma, y : A' \mid \Delta \vdash M : \underline{B}$, then*

$$\Gamma, x' : A' \vdash \mathtt{bind}\ x \leftarrow S_p[\mathtt{ret}\,x']; M[V_e/y] \sqsubseteq M[x'/y]$$

Similarly, if (V_e, S_p) is a computation ep pair from $\underline{B}$ to $\underline{B}'$, and $\Gamma \vdash M : \underline{B}'$ then

$$\Gamma \vdash V_e[\mathtt{thunk}\ S_p[M]] \sqsubseteq \mathtt{thunk}\ M : U\underline{B}'$$

Using our definition of ep pairs, and using the notion of ground type from Section 3.4 *with* 0 *and* $\top$ *removed*, we define

Definition 5.2 (Dynamic Type Interpretation)**.** *A $?, \underline{¿}$ interpretation ρ consists of (1) a CBPV value type $\rho(?)$, (2) a CBPV computation type $\rho(\underline{¿})$, (3) for each value ground type G except* 0*, a value ep pair $(x.\rho_e(G), \rho_p(G))$ from $[\![G]\!]_\rho$ to $\rho(?)$, and (4) for each computation ground type $\underline{G}$ except $\top$, a computation ep pair $(z.\rho_e(\underline{G}), \rho_p(\underline{G}))$ from $[\![\underline{G}]\!]_\rho$ to $\rho(\underline{¿})$. We write $[\![G]\!]_\rho$ and $[\![\underline{G}]\!]_\rho$ for the interpretation of a ground type, replacing $?$ with $\rho(?)$, $\underline{¿}$ with $\rho(\underline{¿})$, and compositionally otherwise.*

We can leave out 0 and $\top$ since the η laws uniquely determine the upcast $\langle ? \nwarrow 0 \rangle$ and downcast $\langle \top \swarrow \underline{¿} \rangle$.

Next, we show several possible interpretations of the dynamic type that will all give, by construction, implementations that satisfy the gradual guarantee. Our interpretations of the value dynamic type are not surprising. They are the usual construction of the dynamic type using type tags: i.e., a recursive sum of basic value types. On the other hand, our interpretations of the computation dynamic type are less familiar. In duality with the interpretation of ?, we interpret $\underline{¿}$ as a recursive *product* of basic computation types. This interpretation has some analogues in previous work on the duality of computation (Girard, 2001; Zeilberger, 2009), but the most direct interpretation (Definition 5.3) does not correspond to any known work on dynamic/gradual typing. Then we show that a particular choice of which computation types is basic and which are derived produces an interpretation of the dynamic computation type as a type of variable-arity functions whose arguments are passed on the stack, producing a model similar to Scheme without accounting for control effects (Definition 5.6).

5.2.1 *Natural dynamic type interpretation*

Our first dynamic type interpretation is to make the value and computation dynamic types sums and products of the ground value and computation types, respectively. This forms a model of GTT for the following reasons. For the value dynamic type ?, we need a value embedding (the upcast) from each ground value type G with a corresponding projection. The easiest way to do this would be if for each G, we could rewrite ? as a sum of the values that fit G and those that don't: $? \cong G + ?_{-G}$ because of the following lemma.

Lemma 5.2 (Sum Injections are Value Embeddings)**.** *For any A, A', there are value ep pairs from A and A' to $A + A'$ where the embeddings are* `inl` *and* `inr` .

Proof. Define the embedding of A to just be $x.\texttt{inl}\ x$ and the projection to be

$$\texttt{bind}\ y \leftarrow \bullet; \texttt{case}\ y\{\texttt{inl}\ x.\texttt{ret}x \mid \texttt{inr}\ .\mho\}.$$

We show this satisfies retraction and projection in the appendix. □

This shows why the type tag interpretation works: it makes the dynamic type in some sense the minimal type with injections from each G: the sum of all value ground types $? \cong \Sigma_G G$.

The dynamic computation type $\underline{¿}$ can be naturally defined by a dual construction, by the following dual argument. First, we want a computation ep pair from $\underline{G}$ to $\underline{¿}$ for each ground computation type $\underline{G}$. Specifically, this means we want a stack from $\underline{¿}$ to $\underline{G}$ (the downcast) with an embedding. The easiest way to get this is if, for each ground computation type $\underline{G}$, $\underline{¿}$ is equivalent to a lazy product of $\underline{G}$ and "the other behaviors", i.e., $\underline{¿} \cong \underline{G} \,\&\, \underline{¿}_{-\underline{G}}$. Then the embedding on π performs the embedded computation, but on π' raises a type error. The following lemma, dual to Lemma 5.2 shows this forms a computation ep pair:

Lemma 5.3 (Lazy Product Projections are Computation Projections)**.** *For any $\underline{B}, \underline{B}'$, there are computation ep pairs from $\underline{B}$ and $\underline{B}'$ to $\underline{B} \,\&\, \underline{B}'$ where the projections are π and π'.*

Proof. Define the projection for $\underline{B}$ to be π. Define the embedding by $z.\{\pi \mapsto \texttt{force}\ z \mid \pi' \mapsto \mho\}$. Similarly define the projection for $\underline{B}'$. We show this forms an ep pair in the appendix. □

From this, we see that the easiest way to construct an interpretation of the dynamic computation type is to make it a lazy product of all the ground types $\underline{G}$: $\underline{¿} \cong \&_{\underline{G}}\, \underline{G}$. Using recursive types, we can easily make this a definition of the interpretations:

Definition 5.3 (Natural Dynamic Type Interpretation)**.** *We define an interpretation of the dynamic types that satisfies the isomorphisms*

$$\rho(?) \cong 1 + (\rho(?) \times \rho(?)) + (\rho(?) + \rho(?)) + U\rho(\underline{¿})$$
$$\rho(\underline{¿}) \cong (\rho(\underline{¿})\ \&\ \rho(\underline{¿}))\ \&\ (\rho(?) \to \rho(\underline{¿}))\ \&\ \underline{F}\rho(?)$$

with the ep pairs defined as in Lemmas 5.2 and 5.3.

We construct $?, \underline{¿}$ *explicitly using recursive and corecursive types. Specifically, we make the recursion explicit by defining open versions of the types:*

$$X, \underline{Y} \vdash ?_o = 1 + (X \times X) + (X + X) + U\underline{Y} \ \ \textit{val type}$$
$$X, \underline{Y} \vdash \underline{¿}_o = (\underline{Y}\ \&\ \underline{Y})\ \&\ (X \to \underline{Y})\ \&\ \underline{F}X \ \ \textit{comp type}$$

Then we define the types $\rho(?)$, $\rho(\underline{¿})$ *using a standard encoding of mutually recursive types:*

$$\rho(?) = \mu X.?_o[\nu \underline{Y}.\underline{¿}_o/\underline{Y}]$$
$$\rho(\underline{¿}) = \nu \underline{Y}.\underline{¿}_o[\mu X.?_o/X]$$

Then clearly by the roll/unroll isomorphism we get the desired isomorphisms:

$$\rho(?) \cong ?_o[\rho(\underline{¿})/\underline{Y}, \rho(?)/X] = 1 + (\rho(?) \times \rho(?)) + (\rho(?) + \rho(?)) + U\rho(\underline{¿})$$
$$\rho(\underline{¿}) \cong \underline{¿}_c[\rho(?)/X, \rho(\underline{¿})/\underline{Y}] = (\rho(\underline{¿})\ \&\ \rho(\underline{¿}))\ \&\ (\rho(?) \to \rho(\underline{¿}))\ \&\ \underline{F}\rho(?)$$

This dynamic type interpretation is a natural fit for CBPV because the introduction forms for ? are exactly the introduction forms for all of the value types (unit, pairing,`inl`, `inr`, `force`), while elimination forms are all of the elimination forms for computation types (π, π', application and binding); such "bityped" languages are related to Girard (2001), Zeilberger (2009).

Based on this dynamic type interpretation, we can extend GTT to support a truly dynamically typed style of programming, where one can perform case analysis on the dynamic types at runtime, in addition to the type assertions provided by upcasts and downcasts. This extension is given in Figure 13. First, we add a type-case form for the dynamic value type $?E$, allowing us to check what tag a value was constructed with. Then we add a β law ($?\beta$) that says that the injection of a ground type (besides 0) is handled by the corresponding branch. Note that here to save space we abbreviate tag types to just their head connective, so $? \times ?$ is abbreviated as $\times$, etc. And finally for ? we add an η law ($?\eta$) that says that any term with a dynamically typed variable $x : ?$ is equivalent to one that immediately pattern matches on x. We add a similar/dual extension for the dynamic computation type. A dynamic computation is one that can be *used* as any computation type. Its introduction form is a "co-type case" ($\underline{¿}I$) that co-pattern matches on how the computation might be

$$\frac{\begin{array}{c}\Gamma \mid \Delta \vdash V : ? \qquad \Gamma, x_1 : 1 \mid \Delta \vdash E_1 : T \\ \Gamma, x_\times : ? \times ? \mid \Delta \vdash E_\times : T \qquad \Gamma, x_+ : ? + ? \mid \Delta \vdash E_+ : T \qquad \Gamma, x_U : U\underline{\text{¿}} \mid \Delta \vdash E_U : T\end{array}}{\Gamma \mid \Delta \vdash \texttt{tycase}\ V\,\{x_1.E_1 \mid x_\times.E_\times \mid x_+.E_+ \mid x_U.E_U\} : T}\ ?\mathrm{E}$$

$$\frac{G \neq 0}{\texttt{tycase}\ (\langle ? \nwarrow G\rangle V)\,\{x_1.E_1 \mid x_\times.E_\times \mid x_+.E_+ \mid x_U.E_U\} \sqsupseteq\sqsubseteq E_G[V/x_G]}\ ?\beta$$

$$\frac{\Gamma, x : ? \mid \Delta \vdash E : \underline{B}}{\begin{array}{c}E \sqsupseteq\sqsubseteq \\ \texttt{tycase}\ x\,\{x_1.E[\langle ? \nwarrow 1\rangle x_1/x] \mid x_\times.E[\langle ? \nwarrow \times\rangle x_\times/x] \mid x_+.E[\langle ? \nwarrow +\rangle x_+/x] \mid x_U.E[\langle ? \nwarrow U\rangle x_U/x]\}\end{array}}\ ?\eta$$

$$\frac{\Gamma \mid \Delta \vdash M_\to : ? \to \underline{\text{¿}} \qquad \Gamma \mid \Delta \vdash M_\& : \underline{\text{¿}} \mathbin{\&} \underline{\text{¿}} \qquad \Gamma \mid \Delta \vdash M_{\underline{F}} : \underline{F}?}{\Gamma \mid \Delta \vdash \{\& \mapsto M_\& \mid (\to) \mapsto M_\to \mid \underline{F} \mapsto M_{\underline{F}}\} : \underline{\text{¿}}}\ \underline{\text{¿}}I$$

$$\frac{\underline{G} \neq \top}{\langle \underline{G} \swarrow \underline{\text{¿}}\rangle \{\& \mapsto M_\& \mid (\to) \mapsto M_\to \mid \underline{F} \mapsto M_{\underline{F}}\} \sqsupseteq\sqsubseteq M_{\underline{G}}}\ \underline{\text{¿}}\beta$$

$$\bullet : \underline{\text{¿}} \vdash \bullet \sqsupseteq\sqsubseteq \{\& \mapsto \langle \underline{\text{¿}} \mathbin{\&} \underline{\text{¿}} \swarrow \underline{\text{¿}}\rangle \bullet \mid (\to) \mapsto \langle ? \to \underline{\text{¿}} \swarrow \underline{\text{¿}}\rangle \bullet \mid \underline{F} \mapsto \langle \underline{F}? \swarrow \underline{\text{¿}}\rangle \bullet\} \quad (\underline{\text{¿}}\eta)$$

Fig. 13. Natural dynamic type extension of GTT.

used: as a function, lazy product or returner. We add a β law $\underline{\text{¿}}\beta$ that says that projecting a co-type case to a non-$\top$ ground computation type selects the corresponding branch (similarly abbreviating $\underline{\text{¿}} \mathbin{\&} \underline{\text{¿}}$ as &, etc.). And finally, we add an η law $\underline{\text{¿}}\eta$ that says that any dynamically typed computation is equivalent to a co-pattern match.

The axioms we choose might seem to under-specify the dynamic type, but because of the uniqueness of adjoints, the following are derivable.

Lemma 5.4 (Natural Dynamic Type Extension Theorems)**.** *The following are derivable in GTT with the natural dynamic type extension*

$$\langle \underline{F}1 \swarrow \underline{F}?\rangle \mathtt{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_1.\mathtt{ret} x_1 \mid \texttt{else}\ \mho\}$$

$$\langle \underline{F}(? \times ?) \swarrow \underline{F}?\rangle \mathtt{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_\times.\mathtt{ret} x_\times \mid \texttt{else}\ \mho\}$$

$$\langle \underline{F}(? + ?) \swarrow \underline{F}?\rangle \mathtt{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_+.\mathtt{ret} x_+ \mid \texttt{else}\ \mho\}$$

$$\langle \underline{F}U\underline{\text{¿}} \swarrow \underline{F}?\rangle \mathtt{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_U.\mathtt{ret} x_U \mid \texttt{else}\ \mho\}$$

$$\texttt{force}\ \langle U\underline{\text{¿}} \nwarrow U(\underline{\text{¿}} \mathbin{\&} \underline{\text{¿}})\rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \texttt{force}\ V \mid (\to) \mapsto \mho \mid \underline{F} \mapsto \mho\}$$

$$\texttt{force}\ \langle U\underline{\text{¿}} \nwarrow U(? \to \underline{\text{¿}})\rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \mho \mid (\to) \mapsto \texttt{force}\ V \mid \underline{F} \mapsto \mho\}$$

$$\texttt{force}\ \langle U\underline{\text{¿}} \nwarrow U\underline{F}?\rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \mho \mid (\to) \mapsto \mho \mid \underline{F} \mapsto \texttt{force}\ V\}$$

We explore this in more detail with the Scheme-like dynamic type interpretation below.

Next, we easily see that if we want to limit GTT to just the CBV types (i.e., the only computation types are $A \to \underline{F}A'$), then we can restrict the dynamic types as follows:

Definition 5.4 (CBV Dynamic Type Interpretation). *The following is a dynamic type interpretation for the ground types of GTT with the only computation types being the unary call-by-value functions $A \to \underline{F}A'$:*

$$\rho(?) \cong 1 + (\rho(?) + \rho(?)) + (\rho(?) \times \rho(?)) + U\rho(\underline{¿}) \qquad \rho(\underline{¿}) = \rho(?) \to \underline{F}\rho(?)$$

with the straightforward encoding similar to that used in Definition 5.3.

And finally if we restrict GTT to only CBN types (i.e., the only value type is Booleans $1 + 1$), we can restrict the dynamic types as follows:

Definition 5.5 (CBN Dynamic Type Interpretation). *The following a dynamic type interpretation for the ground types of GTT with only Boolean value types:*

$$\rho(?) = 1 + 1 \qquad \rho(\underline{¿}) \cong (\rho(\underline{¿}) \mathbin{\&} \rho(\underline{¿})) \mathbin{\&} (U\rho(\underline{¿}) \to \rho(\underline{¿})) \mathbin{\&} \underline{F}\rho(?)$$

which is easy to encode using corecursive types.

5.2.2 Scheme-like dynamic type interpretation

The above dynamic type interpretations do not correspond to any dynamically typed language used in practice, in part because it includes explicit cases for the "additives", the sum type $+$ and lazy product type &. Normally, these are not included in this way, but rather sums are encoded by making each case use a fresh constructor (using nominal techniques like opaque structs in Racket) and then making the sum the union of the constructors, as argued in Siek & Tobin-Hochstadt (2016). We leave modeling this nominal structure to future work, possibly using the fresh type generation model of New *et al.* (2020), but in minimalist languages, such as simple dialects of Scheme and Lisp, sum types are often encoded *structurally* rather than nominally by using some fixed sum type of *symbols*, also called *atoms*. Then a value of a sum type is modeled by a pair of a symbol (to indicate the case) and a payload with the actual value. We can model this by using the canonical isomorphisms

$$? + ? \cong ((1 + 1) \times ?) \qquad \underline{¿} \mathbin{\&} \underline{¿} \cong (1 + 1) \to \underline{¿}$$

and representing sums as pairs, and lazy products as functions.

The fact that isomorphisms are ep pairs is useful for constructing the ep pairs needed in this Scheme-like dynamic type interpretation.

Lemma 5.5 (Isomorphisms are EP Pairs). *If $x : A \vdash V' : A'$ and $x' : A' \vdash V : A$ are an isomorphism in that $V[V'/x'] \sqsupseteq\sqsubseteq x$ and $V'[V/x] \sqsupseteq\sqsubseteq x'$, then $(x.V', \texttt{bind}\ x' \leftarrow \bullet; \texttt{ret}V)$ are a value ep pair from A to A'. Similarly if $\bullet : \underline{B} \vdash S' : \underline{B}'$ and $\bullet : \underline{B}' \vdash S : \underline{B}$ are an isomorphism in that $S[S'] \equiv \bullet$ and $S'[S] \equiv \bullet$ then $(z.S'[\texttt{force}\ z], S)$ is an ep pair from $\underline{B}$ to $\underline{B}'$.*

So we remove the cases for sums and lazy pairs from the natural dynamic types, and include some atomic type as a case of ?—for simplicity we will just use Booleans. We also do not need a case for 1, because we can identify it with one of the Booleans, say `true`. This leads to the following definition:

Definition 5.6 (Scheme-Like Dynamic Type Interpretation)**.** *We can define a dynamic type interpretation with the following type isomorphisms:*

$$\rho(?) \cong (1+1) + U\rho(\underline{\text{¿}}) + (\rho(?) \times \rho(?)) \qquad \rho(\underline{\text{¿}}) \cong (\rho(?) \to \rho(\underline{\text{¿}})) \mathbin{\&} \underline{F}\rho(?)$$

We construct $?, \underline{\text{¿}}$ *explicitly as follows.*

First define $X:$ *val type* $\vdash \mathtt{Tree}[X]$ *val type to be the type of binary trees:*

$$\mathtt{Tree} = \mu X'.X + (X' \times X')$$

Next, define $X:$ *val type*, $\underline{Y}:$ *comp type* $\vdash \mathtt{VarArg}[X, \underline{Y}]$ *comp type to be the type of variable-arity functions from* X *to* $\underline{Y}$*:*

$$\mathtt{VarArg} = \nu \underline{Y}'.\underline{Y} \mathbin{\&} (X \to \underline{Y}')$$

Then we define an open version of $?, \underline{\text{¿}}$ *with respect to a variable representing the occurrences of* $?$ *in* $\underline{\text{¿}}$*:*

$$X \;\textit{val type} \vdash ?_o = \mathtt{Tree}[(1+1) + U\underline{\text{¿}}_o] \;\textit{val type}$$
$$X \;\textit{val type} \vdash \underline{\text{¿}}_o = \mathtt{VarArg}[\underline{F}X/\underline{Y}] \;\textit{comp type}$$

Then we can define the closed versions using a recursive type:

$$? = \mu X.?_o \qquad \underline{\text{¿}} = \underline{\text{¿}}_o[?]$$

The ep pairs for $\times, U, \underline{F}, \to$ *are clear. To define the rest, first note that there is an ep pair from* $1+1$ *to* $?$ *by Lemma 5.2. Next, we can define* 1 *to be the ep pair to* $1+1$ *defined by the left case and Lemma 5.2, composed with this. The ep pair for* $?+?$ *is defined by composing the isomorphism (which is always an ep pair)* $(?+?) \cong ((1+1) \times ?)$ *with the ep pair for* $1+1$ *using the action of product types on ep pairs (proven as part of Theorem 5.8):* $(?+?) \cong ((1+1) \times ?) \vartriangleleft (? \times ?) \vartriangleleft ?$ *(where we write* $A \vartriangleleft A'$ *to mean there is an ep pair from* A *to* A'*). Similarly, for* $\underline{\text{¿}} \mathbin{\&} \underline{\text{¿}}$*, we use action of the function type on ep pairs (also proven as part of Theorem 5.8):* $\underline{\text{¿}} \mathbin{\&} \underline{\text{¿}} \cong ((1+1) \to \underline{\text{¿}}) \vartriangleleft (? \to \underline{\text{¿}}) \vartriangleleft \underline{\text{¿}}$

If we factor out some of the recursion to use inductive and coinductive types, we get the following isomorphisms:

$$\rho(?) \cong \mathtt{Tree}[(1+1) + U\rho(\underline{\text{¿}})/X] \qquad \rho(\underline{\text{¿}}) \cong \mathtt{VarArg}[\rho(?)/X][\underline{F}\rho(?)/\underline{Y}]$$

That is, a dynamically typed value is a binary tree whose leaves are either Booleans or closures. We think of this as a simple type of S-expressions. Next, a dynamically typed computation is a variable-arity function that is called with some number of dynamically typed value arguments $?$ and returns a dynamically typed result $\underline{F}?$. This captures precisely the function type of Scheme, which allows for variable-arity functions!

What's least clear is *why* the type

$$\mathtt{VarArg}[X][\underline{Y}] = \nu \underline{Y}'.(X \to \underline{Y}') \mathbin{\&} \underline{Y}$$

Should be thought of as a type of variable-arity functions. First consider the infinite unrolling of this type:

$$\texttt{VarArg}[X][\underline{Y}] \simeq \underline{Y} \,\&\, (X \to \underline{Y}) \,\&\, (X \to X \to \underline{Y}) \,\&\, \cdots$$

this says that a term of type $\texttt{VarArg}[X][Y]$ offers an infinite number of possible behaviors: it can act as a function from $X^n \to \underline{Y}$ for any n. Similarly in Scheme, a function can be called with any number of arguments. Finally note that this type is isomorphic to a function that takes a *cons-list* of arguments:

$$\begin{aligned}
&\underline{Y} \,\&\, (X \to \underline{Y}) \,\&\, (X \to X \to \underline{Y}) \,\&\, \cdots \\
&\cong (1 \to \underline{Y}) \,\&\, ((X \times 1) \to \underline{Y}) \,\&\, ((X \times X \times 1) \to \underline{Y}) \,\&\, \cdots \\
&\cong (1 + (X \times 1) + (X \times X \times 1) + \cdots) \to \underline{Y} \\
&\cong (\mu X'.1 + (X \times X')) \to \underline{Y}
\end{aligned}$$

But operationally the type $\texttt{VarArg}[?][\underline{F}?]$ is more faithful model of a Scheme implementation that uses the C-calling convention because all of the arguments are passed individually on the stack, whereas the type $(\mu X.1 + (? \times X)) \to \underline{F}X$ is a function that takes a single argument that is a list. These two are distinguished in Scheme and the "dot args" notation witnesses the isomorphism. GTT differs from Scheme in that it allows the programmer to pop the arguments off the stack one at a time, but there is no difference in expressivity.

Assuming some syntax sugar for recursion and pattern matching we could implement this isomorphism in CBPV as follows (note that this "reverses" the order of the arguments in that the argument on the top of the stack will be at the back of the list, but this difference is just an implementation detail):

$$\begin{aligned}
\texttt{dot-args}\, f\pi' &= \texttt{force}\, f(\texttt{roll inl}\ ()) \\
\texttt{dot-args}\, f\pi x &= \texttt{dot-args}(\texttt{thunk}\ (\lambda xs.\texttt{force}\, f(\texttt{roll}\ (x, xs))))
\end{aligned}$$

The inverse isomorphism can be similarly defined as

$$\begin{aligned}
\texttt{apply}\ (f : U(\nu Y'.X \to \underline{Y'} \,\&\, \underline{Y}))(\texttt{roll inl}\ ()) &= \texttt{force}\, f\pi' \\
\texttt{apply}\ (f : U(\nu Y'.X \to \underline{Y'} \,\&\, \underline{Y}))(\texttt{roll inr}\ (x, xs)) &= \texttt{apply}(\texttt{thunk}\ (\texttt{force}\, f\pi x))xs
\end{aligned}$$

Based on this dynamic type interpretation we can make a "Scheme-like" extension to GTT in Figure 14. First, we add a Boolean type $\mathbb{B}$ with `true`, `false` and if-then-else. Next, we add in the elimination form for ? and the introduction form for $\underline{¿}$. The elimination form for ? is a typed version of Scheme's *match* macro. The introduction form for $\underline{¿}$ is a typed, CBPV version of Scheme's *case-lambda* construct. Finally, we add type precision rules expressing the representations of 1, $A + A$, and $A \times A$ in terms of Booleans that were explicit in the ep pairs used in Definition 5.6.

The reader may be surprised by how *few* axioms we need to add to GTT for this extension: for instance we only define the upcast from 1 to $\mathbb{B}$ and not vice versa, and similarly the sum/lazy pair type isomorphisms only have one cast defined when a priori there are 4 to be defined. Finally for the dynamic types we define β and η laws that use the ground casts as injections and projections respectively, but we don't define the corresponding dual casts (the ones that possibly error).

$$1 \sqsubseteq \mathbb{B} \qquad\qquad A + A \sqsupseteq\sqsubseteq \mathbb{B} \times A \qquad\qquad \underline{B} \mathbin{\&} \underline{B} \sqsupseteq\sqsubseteq \mathbb{B} \to \underline{B}$$

$$\frac{}{\Gamma \vdash \texttt{true}, \texttt{false} : \mathbb{B}}\ \mathbb{B}\mathrm{I} \qquad\qquad \frac{\Gamma \vdash V : \mathbb{B} \qquad \Gamma \vdash E_t : T \qquad \Gamma \vdash E_f : T}{\Gamma \mid \Delta \vdash \texttt{if}\ V\ \texttt{then}\ E_t\ \texttt{else}\ E_f : T}\ \mathbb{B}\mathrm{E}$$

$$\texttt{if true then}\ E_t\ \texttt{else}\ E_f \sqsupseteq\sqsubseteq E_t \qquad\qquad \texttt{if false then}\ E_t\ \texttt{else}\ E_f \sqsupseteq\sqsubseteq E_f$$

$$x : \mathbb{B} \vdash E \sqsupseteq\sqsubseteq \texttt{if}\ x\ \texttt{then}\ E[\texttt{true}/x]\ \texttt{else}\ E[\texttt{false}/x]$$

$$\langle \mathbb{B} \nwarrow 1 \rangle V \sqsupseteq\sqsubseteq \texttt{true} \qquad\qquad \langle \mathbb{B} \times A \nwarrow A + A \rangle \texttt{inl}\ V \sqsupseteq\sqsubseteq (\texttt{true}, V)$$

$$\langle \mathbb{B} \times A \nwarrow A + A \rangle \texttt{inr}\ V \sqsupseteq\sqsubseteq (\texttt{false}, V)$$

$$\pi \langle \underline{B} \mathbin{\&} \underline{B} \swarrow \mathbb{B} \to \underline{B} \rangle M \sqsupseteq\sqsubseteq M\,\texttt{true} \qquad\qquad \pi' \langle \underline{B} \mathbin{\&} \underline{B} \swarrow \mathbb{B} \to \underline{B} \rangle M \sqsupseteq\sqsubseteq M\,\texttt{false}$$

$$\frac{\Gamma \mid \Delta \vdash M_{\to} : ? \to \underline{\text{¿}} \qquad \Gamma \mid \Delta \vdash M_{\underline{F}} : \underline{F}?}{\Gamma \mid \Delta \vdash \{(\to) \mapsto M_{\to} \mid \underline{F} \mapsto M_{\underline{F}}\} : \underline{\text{¿}}}\ \underline{\text{¿}}\mathrm{I}$$

$$\langle \underline{G} \swarrow \underline{\text{¿}} \rangle \{(\to) \mapsto M_{\to} \mid \underline{F} \mapsto M_{\underline{F}}\} \sqsupseteq\sqsubseteq M_{\underline{G}} \quad (\underline{\text{¿}}\beta)$$

$$\bullet : \underline{\text{¿}} \vdash \bullet \sqsupseteq\sqsubseteq \{(\to) \mapsto \langle ? \to \underline{\text{¿}} \swarrow \underline{\text{¿}} \rangle \bullet \mid \underline{F} \mapsto \langle \underline{F}? \swarrow \underline{\text{¿}} \rangle \bullet\} \quad (\underline{\text{¿}}\eta)$$

$$\frac{\begin{array}{c} \Gamma \mid \Delta \vdash V : ? \\ \Gamma, x_{\mathbb{B}} : \mathbb{B} \mid \Delta \vdash E_{\mathbb{B}} : T \\ \Gamma, x_U : U\underline{\text{¿}} \mid \Delta \vdash E_U : T \\ \Gamma, x_{\times} : ? \times ? \mid \Delta \vdash E_{\times} : T \end{array}}{\Gamma \mid \Delta \vdash \texttt{tycase}\ V\ \{x_{\mathbb{B}}.E_{\mathbb{B}} \mid x_U.E_U \mid x_{\times}.E_{\times}\} : T}\ ?\mathrm{E}$$

$$\frac{G \in \{\mathbb{B}, \times, U\}}{\texttt{tycase}\ (\langle ? \nwarrow G \rangle V)\ \{x_{\mathbb{B}}.E_{\mathbb{B}} \mid x_U.E_U \mid x_{\times}.E_{\times}\} \sqsupseteq\sqsubseteq E_G[V/x_G]} \qquad (?\beta)$$

$$\frac{\Gamma, x : ? \mid \Delta \vdash E : \underline{B}}{E \sqsupseteq\sqsubseteq \texttt{tycase}\ x\ \{x_{\mathbb{B}}.E[\langle ? \nwarrow \mathbb{B} \rangle x_{\mathbb{B}}/x] \mid x_{\times}.E[\langle ? \nwarrow \times \rangle x_{\times}/x] \mid x_U.E[\langle ? \nwarrow U \rangle x_U/x]\}}\ ?\eta$$

Fig. 14. Scheme-like extension to GTT.

In fact all of these expected axioms can be *proven* from those we have shown. Again we see the surprising rigidity of GTT: because an $\underline{F}$ downcast is determined by its dual value upcast (and vice versa for U upcasts), we only need to define the upcast as long as the downcast *could* be implemented already. Because we give the dynamic types the universal property of a sum/lazy product type respectively, we can derive the implementations of the "checking" casts. All of the proofs are direct from the uniqueness of adjoints lemma.

Theorem 5.3 (Boolean to Unit Downcast)**.** *In Scheme-like GTT, we can prove*

$$\langle \underline{F}1 \swarrow \underline{F}\mathbb{B} \rangle \bullet \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \bullet; \texttt{if}\ x\ \texttt{then}\ \mathit{ret}()\ \texttt{else}\ \mho$$

$$
\begin{aligned}
&[\![\mathbb{B}]\!] = 1 + 1\\
&[\![\texttt{true}]\!] = \texttt{inl}\ ()\\
&[\![\texttt{false}]\!] = \texttt{inr}\ ()\\
&[\![\texttt{if}\ V\,\texttt{then}\ E_t\,\texttt{else}\ E_f]\!] = \texttt{case}\ [\![V]\!]\{x.[\![E_t]\!] \mid x.[\![E_f]\!]\}\\
&[\![\texttt{tycase}\ x\,\{x_{\mathbb{B}}.E_{\mathbb{B}} \mid x_U.E_U \mid x_\times.E_\times\}]\!] =\\
&\quad \texttt{unroll}\ (x:?)\,\texttt{to}\ \texttt{roll}\ x'.\texttt{unroll}\ x' : \texttt{Tree}[(1+1)+U\underline{\text{¿}}]\,\texttt{to}\ \texttt{roll}\ t.\texttt{case}\ t\\
&\qquad \{l.\texttt{case}\ l\{x_{\mathbb{B}}.[\![E_{\mathbb{B}}]\!] \mid x_U.[\![E_U]\!]\}\\
&\qquad\ \mid x_\times.[\![E_\times]\!]\}\\
&[\![\{(\to)\mapsto M_\to \ \mid \underline{F}\mapsto M_{\underline{F}}\}]\!] = \texttt{roll}_{\nu\underline{Y}.(?\to\underline{Y})\&\underline{F}?}\ \{\pi\mapsto[\![M_\to]\!] \mid \pi'\mapsto[\![M_{\underline{F}}]\!]\}
\end{aligned}
$$

Fig. 15. Scheme-like GTT extension semantics.

Theorem 5.4 (Tagged Value to Sum). *In Scheme-like GTT, we can prove*

$$\langle A + A \nwarrow \mathbb{B}\times A\rangle V \sqsupseteq\sqsubseteq \texttt{split}\ V\,\texttt{to}\ (x,y).\texttt{if}\ x\,\texttt{then}\ \texttt{inl}\ y\,\texttt{else}\ \texttt{inr}\ y$$

and the downcasts are given by Lemma 5.5.

Theorem 5.5 (Lazy Product to Tag Checking Function). *In Scheme-like GTT, we can prove*

$$\langle \mathbb{B}\to\underline{B} \swarrow \underline{B}\,\&\,\underline{B}\rangle\bullet \sqsupseteq\sqsubseteq \lambda x:\mathbb{B}.\texttt{if}\ x\,\texttt{then}\ \pi\bullet\ \texttt{else}\ \pi'\bullet$$

and the upcasts are given by Lemma 5.5.

Theorem 5.6 (Ground Mismatches are Errors). *In Scheme-like GTT we can prove*

$$\langle \underline{F}\mathbb{B} \swarrow \underline{F}?\rangle \mathit{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_{\mathbb{B}}.\mathit{ret} x_{\mathbb{B}} \mid \texttt{else}\ \mho\}$$

$$\langle \underline{F}(?\times ?) \swarrow \underline{F}?\rangle \mathit{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_\times.\mathit{ret} x_\times \mid \texttt{else}\ \mho\}$$

$$\langle \underline{F}U\underline{\text{¿}} \swarrow \underline{F}?\rangle \mathit{ret} V \sqsupseteq\sqsubseteq \texttt{tycase}\ V\,\{x_U.\mathit{ret} x_U \mid \texttt{else}\ \mho\}$$

$$\texttt{force}\ \langle U\underline{\text{¿}} \nwarrow U(?\to\underline{\text{¿}})\rangle V \sqsupseteq\sqsubseteq \{(\to)\mapsto \texttt{force}\ V \mid \underline{F}\mapsto\mho\}$$

$$\texttt{force}\ \langle U\underline{\text{¿}} \nwarrow U\underline{F}?\rangle V \sqsupseteq\sqsubseteq \{(\to)\mapsto\mho \mid \underline{F}\mapsto \texttt{force}\ V\}$$

Next, note that this model gives an example of why the TAGMISMATCH and SILLY rules in Section 4 could not be derived from GTT. In the call-by-value calculus, any cast from a sum type to a product type would fail, but here we have a model where all sum types can be safely cast to $\mathbb{B}\times ?$.

Finally, we note now that all of these axioms are satisfied when using the Scheme-like dynamic type interpretation and extending the translation of GTT into CBPV* given in Section 5.3 with the cases in Figure 15.

$$
\begin{array}{rcl}
x : \llbracket A \rrbracket \vdash \llbracket \langle A' \nwarrow A \rangle \rrbracket : \llbracket A' \rrbracket & \bullet : & \llbracket \underline{B}' \rrbracket \vdash \llbracket \langle \underline{B} \swarrow \underline{B}' \rangle \rrbracket : \llbracket \underline{B} \rrbracket \\
x : 0 \vdash \llbracket \langle A \nwarrow 0 \rangle \rrbracket & = & \texttt{absurd } x \\
\bullet : A \vdash \llbracket \langle \underline{F}0 \swarrow \underline{F}A \rangle \rrbracket & = & \texttt{bind } x \leftarrow \bullet; \mho \\
x : \llbracket ? \rrbracket \vdash \llbracket \langle ? \nwarrow ? \rangle \rrbracket & = & x \\
\bullet : \underline{F}? \vdash \llbracket \langle \underline{F}? \swarrow \underline{F}? \rangle \rrbracket & = & \bullet \\
x : \llbracket G \rrbracket \vdash \llbracket \langle ? \nwarrow G \rangle \rrbracket & = & \rho_{up}(G) \\
\bullet : \underline{F}? \vdash \llbracket \langle \underline{F}G \swarrow \underline{F}? \rangle \rrbracket & = & \rho_{dn}(G) \\
x : \llbracket A \rrbracket \vdash \llbracket \langle ? \nwarrow A \rangle \rrbracket & = & \llbracket \langle ? \nwarrow \lfloor A \rfloor \rangle \rrbracket [\llbracket \langle \lfloor A \rfloor \nwarrow A \rangle \rrbracket / x] \quad (A \notin \{?, \lfloor A \rfloor\}) \\
\bullet : \underline{F}? \vdash \llbracket \langle A \swarrow ? \rangle \rrbracket & = & \llbracket \langle A \swarrow \lfloor A \rfloor \rangle \rrbracket [\llbracket \langle \lfloor A \rfloor \swarrow ? \rangle \rrbracket] \quad (A \notin \{?, \lfloor A \rfloor\}) \\
x : \llbracket A_1 \rrbracket + \llbracket A_2 \rrbracket \vdash \llbracket \langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle \rrbracket & = & \texttt{case } x \\
& & \{x_1 . \llbracket \langle A_1' \nwarrow A_1 \rangle \rrbracket [x_1/x] \\
& & \mid x_2 . \llbracket \langle A_2' \nwarrow A_2 \rangle \rrbracket [x_2/x]\} \\
\bullet : \underline{F}(\llbracket A_1' \rrbracket + \llbracket A_2' \rrbracket) \vdash \llbracket \langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle \rrbracket & = & \texttt{bind } x' \leftarrow \bullet; \texttt{case } x' \\
& & \{x_1' . \texttt{bind } x_1 \leftarrow (\llbracket \langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle \rrbracket [\texttt{ret} x_1']); \texttt{ret} x_1 \\
& & \mid x_2' . \texttt{bind } x_2 \leftarrow (\llbracket \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle \rrbracket [\texttt{ret} x_2']); \texttt{ret} x_2\} \\
x : 1 \vdash \llbracket \langle 1 \nwarrow 1 \rangle \rrbracket & = & x \\
\bullet : \underline{F}1 \vdash \llbracket \langle \underline{F}1 \swarrow \underline{F}1 \rangle \rrbracket & = & \bullet \\
x : \llbracket A_1 \rrbracket \times \llbracket A_2 \rrbracket \vdash \llbracket \langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle \rrbracket & = & \texttt{split } x \texttt{ to } (x_1, x_2). \\
& & (\llbracket \langle A_1' \nwarrow A_1 \rangle \rrbracket [x_1/x], \llbracket \langle A_2' \nwarrow A_2 \rangle \rrbracket [x_2/x]) \\
\bullet : \underline{F}(\llbracket A_1' \rrbracket \times \llbracket A_2' \rrbracket) \vdash \llbracket \langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2') \rangle \rrbracket & = & \texttt{bind } x' \leftarrow \bullet; \texttt{split } x' \texttt{ to } (x_1', x_2'). \\
& & \texttt{bind } x_1 \leftarrow \llbracket \langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle \rrbracket [\texttt{ret} x_1']; \\
& & \texttt{bind } x_2 \leftarrow \llbracket \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle \rrbracket [\texttt{ret} x_2']; \texttt{ret}(x_1, x_2) \\
x : U\underline{F}\llbracket A \rrbracket \vdash \llbracket \langle U\underline{F}A' \nwarrow U\underline{F}A \rangle \rrbracket & = & \texttt{thunk } (\texttt{bind } y \leftarrow \texttt{force } x; \texttt{ret} \llbracket \langle A' \nwarrow A \rangle \rrbracket [y/x]) \\
& & \\
\bullet : \underline{B} \vdash \llbracket \langle \top \swarrow \underline{B} \rangle \rrbracket & = & \{\} \\
x : U\top \vdash \llbracket \langle U\underline{B} \nwarrow U\top \rangle \rrbracket & = & \texttt{thunk } \mho \\
\bullet : \underline{¿} \vdash \llbracket \langle \underline{¿} \swarrow \underline{¿} \rangle \rrbracket & = & \bullet \\
x : U\underline{¿} \vdash \llbracket \langle U\underline{¿} \nwarrow U\underline{¿} \rangle \rrbracket & = & x \\
\bullet : \underline{¿} \vdash \llbracket \langle \underline{G} \swarrow \underline{¿} \rangle \rrbracket & = & \rho_{dn}(\underline{G}) \\
x : U\underline{G} \vdash \llbracket \langle U\underline{¿} \nwarrow U\underline{G} \rangle \rrbracket & = & \rho_{up}(\underline{G}) \\
\bullet : \underline{¿} \vdash \llbracket \langle \underline{B} \swarrow \underline{¿} \rangle \rrbracket & = & \llbracket \langle \underline{B} \swarrow \lfloor \underline{B} \rfloor \rangle \rrbracket [\llbracket \langle \lfloor \underline{B} \rfloor \swarrow \underline{¿} \rangle \rrbracket] \quad (\underline{B} \notin \{\underline{¿}, \lfloor \underline{B} \rfloor\}) \\
x : U\underline{¿} \vdash \llbracket \langle U\underline{¿} \nwarrow U\underline{B} \rangle \rrbracket & = & \llbracket \langle U\underline{¿} \nwarrow U\lfloor \underline{B} \rfloor \rangle \rrbracket [\llbracket \langle U\lfloor \underline{B} \rfloor \nwarrow U\underline{B} \rangle \rrbracket / x] \quad (\underline{B} \notin \{\underline{¿}, \lfloor \underline{B} \rfloor\}) \\
\bullet : \llbracket \underline{B}_1' \rrbracket \,\&\, \llbracket \underline{B}_2' \rrbracket \vdash \llbracket \langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle \rrbracket & = & \{\pi \mapsto \llbracket \langle \underline{B}_1 \swarrow \underline{B}_1' \rangle \rrbracket [\pi \bullet] \mid \pi' \mapsto \llbracket \langle \underline{B}_2 \swarrow \underline{B}_2' \rangle \rrbracket [\pi' \bullet]\} \\
x : U(\llbracket \underline{B}_1 \rrbracket \,\&\, \llbracket \underline{B}_2 \rrbracket) \vdash \llbracket \langle U(\underline{B}_1' \,\&\, \underline{B}_2') \nwarrow U(\underline{B}_1 \,\&\, \underline{B}_2) \rangle \rrbracket & = & \texttt{thunk} \\
& & \{\pi \mapsto \texttt{force } \llbracket \langle \underline{B}_1' \nwarrow \underline{B}_1 \rangle \rrbracket [\texttt{thunk } (\pi \texttt{force } x)] \\
& & \mid \pi' \mapsto \texttt{force } \llbracket \langle \underline{B}_2' \nwarrow \underline{B}_2 \rangle \rrbracket [\texttt{thunk } (\pi' \texttt{force } x)]\} \\
\bullet : \llbracket A' \rrbracket \to \llbracket \underline{B}' \rrbracket \vdash \llbracket \langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle \rrbracket & = & \lambda x : A . \llbracket \langle \underline{B} \swarrow \underline{B}' \rangle \rrbracket [\bullet\, (\llbracket \langle A' \nwarrow A \rangle \rrbracket)] \\
f : U(\llbracket A \rrbracket \to \llbracket U\underline{B} \rrbracket) \vdash \llbracket \langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B}) \rangle \rrbracket & = & \texttt{thunk } \lambda x' : A'. \\
& & \texttt{bind } x \leftarrow \llbracket \langle \underline{F}A \swarrow \underline{F}A' \rangle \rrbracket [\texttt{ret} x']; \\
& & \texttt{force } \llbracket \langle U\underline{B}' \nwarrow U\underline{B} \rangle \rrbracket [\texttt{thunk } ((\texttt{force } f)\, x')/x] \\
\bullet : \underline{F}U\underline{B}' \vdash \llbracket \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}' \rangle \rrbracket & = & \texttt{bind } x' \leftarrow \bullet; \llbracket \langle \underline{B} \swarrow \underline{B}' \rangle \rrbracket [\texttt{force } x']
\end{array}
$$

Fig. 16. Cast to contract translation.

5.3 Contract translation

Having defined the data parameterizing the translation, we now consider the translation of GTT into CBPV* itself. For the remainder of the paper, we assume that we have a fixed dynamic type interpretation ρ, and all proofs and definitions work for any interpretation.

5.3.1 Interpreting casts as contracts

The main idea of the translation is an extension of the dynamic type interpretation to an interpretation of *all* casts in GTT (Figure 16) as contracts in CBPV*, following the definitions in Lemma 3.7. We consider the rules ordered for determining which of possibly overlapping cases to use. We describe a few rules specifically now. The rule for casting a tag type G to and from ? utilizes the assumed dynamic type interpretation ρ. Next, the

$$\frac{A \in \{?, 1, 0\}}{A \sqsubseteq A} \qquad \frac{A \sqsubseteq G}{A \sqsubseteq ?}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}'}{U\underline{B} \sqsubseteq U\underline{B}'} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 + A_2 \sqsubseteq A_1' + A_2'} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 \times A_2 \sqsubseteq A_1' \times A_2'}$$

$$\frac{\underline{B} \in \{\underline{¿}, \top\}}{\underline{B} \sqsubseteq \underline{B}} \qquad \frac{\underline{B} \sqsubseteq \underline{G}}{\underline{B} \sqsubseteq \underline{¿}}$$

$$\frac{A \sqsubseteq A'}{\underline{F}A \sqsubseteq \underline{F}A'} \qquad \frac{\underline{B}_1 \sqsubseteq \underline{B}_1' \qquad \underline{B}_2 \sqsubseteq \underline{B}_2'}{\underline{B}_1 \mathbin{\&} \underline{B}_2 \sqsubseteq \underline{B}_1' \mathbin{\&} \underline{B}_2'} \qquad \frac{A \sqsubseteq A' \qquad \underline{B} \sqsubseteq \underline{B}'}{A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

Fig. 17. Normalized type precision relation.

rule for casting A to ? casts A to its corresponding "tag type" which we write as $\lfloor A \rfloor$. $\lfloor A \rfloor$ is defined as the unique tag type G such that $A \sqsubseteq G$, so $? \times ?$ for any $A \times A'$, etc. The corresponding downcast $\underline{F}?$ to $\underline{F}A$ is defined similarly The rules for sums, products, and $U\underline{F}$ follow from the uniqueness principles proven in the previous section. For the computation type connectives, first, the casts between tag types $\underline{G}$ and $\underline{¿}$ use ρ. Next, the rule for downcasting from $\underline{¿}$ to a non-tag $\underline{B}$ is analogous to the value type case, using an analogous notion of $\lfloor \underline{B} \rfloor$. The cases for $\&, \to, \underline{F}U$ follow the uniqueness theorems. This definition is not obviously total: we need to verify that it covers every possible case where $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$. To prove totality and coherence, we could try induction on the type precision relation of Figure 4, but it is convenient to first give an alternative, normalized set of rules for type precision that proves the same relations, which we do in Figure 17. First, we add reflexivity rules for the base value types. Then we add a rule that says to prove a value type A is more precise than ? it is sufficient to prove it is more precise than a ground type G. This is effectively a limited transitivity rule that allows us to compose a precision proof $A \sqsubseteq G$ with the "primitive" rules for tag types $G \sqsubseteq ?$. We recover the rule $G \sqsubseteq ?$ by composing with the reflexivity proof $G \sqsubseteq G$. Note that there is only one way this can apply since a type can only be more precise than a single tag type. Then we add the congruence rules for the value type constructor $U, +, \times$. Next, we add computation type precision rules similarly: reflexivity for base types, a rule for proving $\underline{¿}$ is the most imprecise type and congruence rules for the computation type constructors.

Lemma 5.6 (Normalized Type Precision is Equivalent to Original)**.** *$T \sqsubseteq T'$ is provable in the normalized typed precision definition iff it is provable in the original typed precision definition.*

Based on normalized type precision, we show

Theorem 5.7. *If $A \sqsubseteq A'$ according to Figure 17, then there is a unique complex value $x : A \vdash [\![\langle A' \leftarrowtail A \rangle]\!] x : A'$ and if $\underline{B} \sqsubseteq \underline{B}'$ according to Figure 17, then there is a unique complex stack $x : \underline{B} \vdash [\![\langle \underline{B}' \leftarrowtail \underline{B} \rangle]\!] x : \underline{B}'$*

5.3.2 *Interpretation of terms*

Next, we extend the translation of casts to a translation of all terms by congruence, since all terms in GTT besides casts are in CBPV*. This satisfies:

Lemma 5.7 (Contract Translation Type Preservation)**.** *If $\Gamma \mid \Delta \vdash E : T$ in GTT, then $\llbracket\Gamma\rrbracket \mid \llbracket\Delta\rrbracket \vdash \llbracket E\rrbracket : \llbracket T\rrbracket$ in CBPV*.*

5.3.3 *Interpretation of term precision*

We have now given an interpretation of the types, terms, and type precision proofs of GTT in CBPV*. To complete this to form a *model* of GTT, we need to give an interpretation of the *term precision* proofs, which is established by the following "axiomatic graduality" theorem. GTT has *heterogeneous* term precision rules indexed by type precision, but CBPV* has only *homogeneous* inequalities between terms, i.e., if $E \sqsubseteq E'$, then E, E' have the *same* context and types. Since every type precision judgement has an associated contract, we can translate a heterogeneous term precision to a homogeneous inequality *up to insertion of contract*. Our next overall goal is to prove the following

Theorem 5.8 (Axiomatic Graduality)**.** *For any dynamic type interpretation,*

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Psi : \Delta \sqsubseteq \Delta' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\llbracket\Gamma\rrbracket \mid \llbracket\Delta'\rrbracket \vdash \llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{B}\rrbracket}$$

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\llbracket\Gamma\rrbracket \vdash \llbracket\langle A' \nwarrow A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] : \llbracket A'\rrbracket}$$

where we define $\llbracket\Phi\rrbracket$ to upcast each variable, and $\llbracket\Delta\rrbracket$ to downcast $\bullet$ if it is non-empty, and if $\Delta = \cdot$, then $M[\llbracket\Delta\rrbracket] = M$. More explicitly,

1. *If $\Phi : \Gamma \sqsubseteq \Gamma'$, then there exists n such that $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma' = x'_1 : A'_1, \ldots, x'_n : A'_n$ where $A_i \sqsubseteq A'_i$ for each $i \leq n$. Then $\llbracket\Phi\rrbracket$ is a substitution from $\llbracket\Gamma\rrbracket$ to $\llbracket\Gamma'\rrbracket$ defined as*

$$\llbracket\Phi\rrbracket = \llbracket\langle A'_1 \nwarrow A_1\rangle\rrbracket x_1/x'_1, \ldots \llbracket\langle A'_n \nwarrow A_n\rangle\rrbracket x_n/x'_n$$

2. *If $\Psi : \Delta \sqsubseteq \Delta'$, then we similarly define $\llbracket\Psi\rrbracket$ as a "linear substitution". That is, if $\Delta = \Delta' = \cdot$, then $\llbracket\Psi\rrbracket$ is an empty substitution and $M[\llbracket\Psi\rrbracket] = M$, otherwise $\llbracket\Psi\rrbracket$ is a linear substitution from $\Delta' = \bullet : \underline{B}'$ to $\Delta = \bullet : \underline{B}$ where $\underline{B} \sqsubseteq \underline{B}'$ defined as*

$$\llbracket\Psi\rrbracket = \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket \bullet / \bullet$$

Relative to previous work on graduality (New & Ahmed, 2018), the distinction between complex value upcasts and complex stack downcasts here guides the formulation of the theorem; e.g., using upcasts in the left-hand theorem would require more thunks/forces. Note that an alternative to using homogeneous inequality up to cast would be to provide a direct logical relations interpretation of the heterogeneous inequality for every pair of types $A \sqsubseteq A'$ (and $\underline{B} \sqsubseteq \underline{B}'$) (New *et al.*, 2020).

We now develop some lemmas on the way toward proving this result. First, we prove that from the basic casts being ep pairs, we can prove that all casts as defined in Figure 16 are ep pairs. Before doing so, we prove the following lemma, which is used for transitivity (e.g., in the $A \sqsubseteq ?$ rule, which uses a composition $A \sqsubseteq \lfloor A \rfloor \sqsubseteq ?$):

Lemma 5.8 (EP Pairs Compose)**.**

1. *If (V_1, S_1) is a value ep pair from A_1 to A_2 and (V_2, S_2) is a value ep pair from A_2 to A_3, then $(V_2[V_1], S_1[S_2])$ is a value ep pair from A_1 to A_3.*
2. *If (V_1, S_1) is a computation ep pair from $\underline{B}_1$ to $\underline{B}_2$ and (V_2, S_2) is a computation ep pair from $\underline{B}_2$ to $\underline{B}_3$, then $(V_2[V_1], S_1[S_2])$ is a computation ep pair from $\underline{B}_1$ to $\underline{B}_3$.*

Lemma 5.9 (Identity EP Pair)**.** $(x.x, \bullet)$ *is an ep pair (value or computation).*

Now, we show that all casts are ep pairs. The proof is a somewhat tedious, but straightforward calculation, and is included in the appendix.

Lemma 5.10 (Casts are EP Pairs)**.**

1. *For any $A \sqsubseteq A'$, the casts $(x.\llbracket\langle A' \nwarrow A\rangle x\rrbracket, \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket)$ are a value ep pair from $\llbracket A\rrbracket$ to $\llbracket A'\rrbracket$.*
2. *For any $\underline{B} \sqsubseteq \underline{B}'$, the casts $(z.\llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle z\rrbracket, \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket)$ are a computation ep pair from $\llbracket\underline{B}\rrbracket$ to $\llbracket\underline{B}'\rrbracket$.*

While tedious, this work pays off greatly in later proofs: this is the *only* proof in the entire development that needs to inspect the definition of a "shifted" cast (a downcast between $\underline{F}$ types or an upcast between U types). All later lemmas have cases for these shifted casts, but *only* use the property that they are part of an ep pair. This is one of the biggest advantages of using an explicit syntax for complex values and complex stacks: the shifted casts are the only ones that non-trivially use effectful terms, so after this lemma is established we only have to manipulate values and stacks, which compose much more nicely than effectful terms. Conceptually, the main reason we can avoid reasoning about the definitions of the shifted casts directly is that any two shifted casts that form an ep pair with the same value embedding/stack projection are equal:

Lemma 5.11 (Embedding determines Projection, and vice versa)**.** *For any value $x : A \vdash V_e : A'$ and stacks $\bullet : \underline{F}A' \vdash S_1 : \underline{F}A$ and $\bullet : \underline{F}A' \vdash S_2 : \underline{F}A$, if (V_e, S_1) and (V_e, S_2) are both value ep pairs, then*

$$S_1 \sqsupseteq\sqsubseteq S_2$$

Similarly for any values $x : U\underline{B} \vdash V_1 : U\underline{B}'$ and $x : U\underline{B} \vdash V_2 : U\underline{B}'$ and stack $\bullet : \underline{B}' \vdash S_p : \underline{B}$, if (V_1, S_p) and (V_2, S_p) are both computation ep pairs then

$$V_1 \sqsupseteq\sqsubseteq V_2$$

The next two lemmas on the way to axiomatic graduality show that Figure 16 translates $\langle A \nwarrow A\rangle$ to the identity and $\langle A'' \nwarrow A'\rangle\langle A' \nwarrow A\rangle$ to the same contract as $\langle A'' \nwarrow A\rangle$, and similarly for downcasts. Intuitively, for all connectives except $\underline{F}, U$, this is because of functoriality of the type constructors on values and stacks. For the $\underline{F}, U$ cases, we will use the corresponding fact about the dual cast, i.e., to prove the $\underline{F}A$ to $\underline{F}A$ downcast is the identity stack, we know by inductive hypothesis that the A to A upcast is the identity, and that the identity stack is a projection for the identity. Therefore Lemma 5.11 implies that the $\underline{F}A$ downcast must be equivalent to the identity. We now discuss these two lemmas and their proofs in detail.

First, we show that the casts from a type to itself are equivalent to the identity. Below, we will use this lemma to prove the reflexivity case of the axiomatic graduality theorem, and to prove a conservativity result, which says that a GTT homogeneous term precision is the same as a CBPV* inequality between their translations.

Lemma 5.12 (Identity Expansion)**.** *For any A and $\underline{B}$,*

$$x : A \vdash [\![\langle A \leftarrowtail A\rangle]\!] \sqsupseteq\sqsubseteq x : A \qquad\qquad \bullet : \underline{B} \vdash [\![\langle \underline{B} \twoheadleftarrow \underline{B}\rangle]\!] \sqsupseteq\sqsubseteq \bullet : \underline{B}$$

Second, we show that a composition of upcasts is translated to the same thing as a direct upcast, and similarly for downcasts. Below, we will use this lemma to translate *transitivity* of term precision in GTT.

Lemma 5.13 (Cast Decomposition)**.** *For any dynamic type interpretation ρ,*

$$\frac{A \sqsubseteq A' \sqsubseteq A''}{x : A \vdash [\![\langle A'' \leftarrowtail A\rangle]\!]_\rho \sqsupseteq\sqsubseteq [\![\langle A'' \leftarrowtail A'\rangle]\!]_\rho[[\![\langle A' \leftarrowtail A\rangle]\!]_\rho] : A''}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''}{\bullet : \underline{B}'' \vdash [\![\langle \underline{B} \twoheadleftarrow \underline{B}''\rangle]\!]_\rho \sqsupseteq\sqsubseteq [\![\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle]\!]_\rho[[\![\langle \underline{B}' \twoheadleftarrow \underline{B}''\rangle]\!]_\rho]}$$

The final lemma before the graduality theorem lets us "move a cast" from left to right or vice versa, via the adjunction property for ep pairs. These arise in the proof cases for `return` and `thunk` , because in those cases the inductive hypothesis is in terms of an upcast (downcast) and the conclusion is in terms of a a downcast (upcast).

Lemma 5.14 (Hom-set formulation of Adjunction)**.** *For any value embedding-projection pair V_e, S_p from A to A', the following are equivalent:*

$$\begin{array}{c} \Gamma \vdash \mathtt{ret} V_e[V] \sqsubseteq M : \underline{F}A' \\ \hline\hline \Gamma \vdash \mathtt{ret} V \sqsubseteq S_p[M] : \underline{F}A \end{array}$$

For any computation ep pair (V_e, S_p) from $\underline{B}$ to $\underline{B}'$, the following are equivalent:

$$\begin{array}{c} \Gamma, z' : U\underline{B}' \vdash M \sqsubseteq S[S_p[\mathtt{force}\ z']] : \underline{C} \\ \hline\hline \Gamma, z : U\underline{B} \vdash M[V_e/z'] \sqsubseteq S[\mathtt{force}\ z] : \underline{C} \end{array}$$

Finally, we prove the axiomatic graduality theorem. In addition to the lemmas above, the main task is to prove the "compatibility" cases which are the congruence cases for introduction and elimination rules. These come down to proving that the casts "commute" with introduction/elimination forms, and are all simple calculations.

Theorem 5.1 (Axiomatic Graduality)**.** *For any dynamic type interpretation, the following are true:*

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Psi : \Delta \sqsubseteq \Delta' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\llbracket \Gamma \rrbracket \mid \llbracket \Delta' \rrbracket \vdash \llbracket M \rrbracket [\llbracket \Psi \rrbracket] \sqsubseteq \llbracket \langle \underline{B} \swarrow \underline{B}' \rangle \rrbracket [\llbracket M' \rrbracket [\llbracket \Phi \rrbracket]] : \llbracket \underline{B} \rrbracket}$$

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\llbracket \Gamma \rrbracket \vdash \llbracket \langle A' \nwarrow A \rangle \rrbracket [\llbracket V \rrbracket] \sqsubseteq \llbracket V' \rrbracket [\llbracket \Phi \rrbracket] : \llbracket A' \rrbracket}$$

As a corollary, we have the following conservativity result, which says that the homogeneous term precisions in GTT are sound and complete for inequalities in CBPV*.

Corollary 5.2 (Conservativity)**.** *If* $\Gamma \mid \Delta \vdash E, E' : T$ *are two terms of the same type in the intersection of GTT and CBPV*, then* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ *is provable in GTT iff it is provable in CBPV*.*

Proof. The reverse direction holds because CBPV* is a syntactic subset of GTT. The forward direction holds by axiomatic graduality and the fact that identity casts are identities. □

6 Complex value/stack elimination

Next, to bridge the gap between the semantic notion of complex value and stack with the more rigid operational notion, we perform a "complexity-elimination" pass.[6] This translation transforms computations using complex values into equivalent ones without them. This demonstrates that complex values do not add any expressive power to the language *operationally*. As an example consider a product upcast $\langle ? \times ? \nwarrow A_1 \times A_2 \rangle V$. By Theorem 3.5, this is equivalent to a pattern match and then tagging both sides of the pair:

$$\texttt{split}\ V\ \texttt{to}\ (x_1, x_2).(\langle ? \nwarrow A_1 \rangle x_1, \langle ? \nwarrow A_2 \rangle x_2)$$

However, this is not a value in the operational sense since it has a $\beta\times$ redex if V is a closed value of product type. So we instead define a translation $V^\dagger$ that turns a value $V : A$ into a computation $V^\dagger : \underline{F}A$ that is equivalent to $\texttt{ret}\, V$. For instance the pair cast would turn into the larger term:

$$\texttt{bind}\ p \leftarrow V^\dagger; \texttt{split}\ p\ \texttt{to}\ (x_1, x_2).\texttt{bind}\ y_1 \leftarrow \langle ? \nwarrow A_1 \rangle {x_1}^\dagger; \texttt{bind}\ y_2 \leftarrow \langle ? \nwarrow A_2 \rangle {x_2}^\dagger;$$
$$\texttt{ret}(y_1, y_2)$$

We see here that this de-complexification pass is akin to translation to A-normal or monadic form.

So if complex values add no operational expressive power, why do we use them at all? The reason is that they greatly simplify using the inequational theory to reason about casts. Using complex values, all upcasts are values, and so can be substituted for variables. On the other hand, if upcasts are computations, then (1) we cannot substitute them directly for

[6] Levy (2003) provides a similar complexity-elimination pass, but does not prove the inequality preservation that we require here, so we give an alternative, but equivalent translation for which this property is easy to verify.

$$
\begin{array}{lcl}
V & ::= & x \mid \texttt{roll}_{\mu X.A}\ V \mid \texttt{inl}\ V \mid \texttt{inr}\ V \mid () \mid (V_1, V_2) \mid \texttt{thunk}\ M \\
M & ::= & \mho_{\underline{B}} \mid \texttt{let}\ x = V; M \mid \texttt{unroll}\ V\ \texttt{to roll}\ x.M \mid \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M \mid \texttt{unroll}\ M \mid \texttt{abort}\ V \mid \\
 & & \texttt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \mid \texttt{split}\ V\ \texttt{to}\ ().M \mid \texttt{split}\ V\ \texttt{to}\ (x,y).M \mid \texttt{force}\ V \mid \\
 & & \texttt{ret}V \mid \texttt{bind}\ x \leftarrow M; N \mid \lambda x : A.M \mid M\ V \mid \{\} \mid \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} \mid \pi M \mid \pi' M \\
S & ::= & \bullet \mid \texttt{bind}\ x \leftarrow S; M \mid S\ V \mid \pi S \mid \pi' S \mid \texttt{unroll}_{\nu \underline{Y}.\underline{B}}\ S
\end{array}
$$

Fig. 18. Operational CBPV syntax.

variables without renormalizing the term and (2) if the variable occurs zero times or more than once, we do not know a priori if such a substitution would be semantics-preserving. This second reason provides the main difficulty in proving that the de-complexification pass preserves the inequational theory. To prove this we need to show that for any complex value V, the de-complexified computation $V^\dagger$ is in some sense a "pure" computation. This notion of purity in CBPV is called *thunkability* (Führmann, 1999; Munch-Maccagnoni, 2014). In the inequational theory of CBPV, this is defined by saying that a term $M : \underline{F}A$ is thunkable if running M to a value and then duplicating its value is the same as running M every time we need its value. Formally, we define it as

Definition 6.1 (Thunkable Computation)**.** *A computation* $\Gamma \vdash M : \underline{F}A$ *is* thunkable *if*

$$\Gamma \vdash \mathit{ret}(\texttt{thunk}\ M) \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow M; \mathit{ret}(\texttt{thunk}\ (\mathit{ret} x)) : \underline{F}U\underline{F}A$$

Since we also use complex *stacks* in the equational theory, we also need to define a de-complexification pass for stacks that takes a stack $\bullet : \underline{B} \vdash S : \underline{B}'$ to a computation with a free variable $x : U\underline{B} \vdash S^\dagger[x] : \underline{B}'$ that is equivalent to $S[\texttt{force}\ x]$. Similarly, to prove the de-complexification preserves the inequational theory, we need a semantic property that all de-complexified complex stacks satisfy that is a dual to thunkability called *linearity* (Munch-Maccagnoni, 2014). Intuitively, a term $\Gamma \vdash M : \underline{B}'$ is linear in a variable $x : U\underline{B} \in \Gamma$ if it acts like a term that immediately forces x once and then never forces x again. This is described in the CBPV inequational theory as follows: if we have a double thunk $z : U\underline{F}U\underline{B}$, then either we can force it now and pass the result to M as x, or we can just run M with a thunk that will force z each time x is forced—but if M forces x exactly once, these two are the same.

Definition 6.2 (Linear Term)**.** *A term* $\Gamma \vdash M : \underline{C}$ *is* linear in $x : U\underline{B} \in \Gamma$ *if*

$$\Gamma, z : U\underline{F}U\underline{B} \vdash \texttt{bind}\ x \leftarrow \texttt{force}\ z; M \sqsupseteq\sqsubseteq M[\texttt{thunk}\ (\texttt{bind}\ x \leftarrow (\texttt{force}\ z); \texttt{force}\ x)]$$

Now, let's define de-complexification. First, the syntax of operational CBPV, the target of the de-complexification translation as shown in Figure 1 (unshaded), but with recursive types added as in Section 5.1, and with values and stacks restricted as shown in Figure 18.

In CBPV, values include only introduction forms, as usual for values in operational semantics, and CBPV stacks consist only of elimination forms for computation types (the syntax of CBPV enforces an A-normal form, where only values can be pattern matched on, so `case` and `split` are not evaluation contexts in the operational semantics).

The *de-complexification* procedure is defined as follows.

Definition 6.3 (De-complexification). *We define decomplexification of values, stacks and computations recursively as follows:*

$$
\begin{array}{rcl}
\bullet^\dagger & = & \mathtt{force}\ z \\
(\mathit{ret} V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathit{ret} x \\
(M\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; M^\dagger\ x \\
(\mathtt{force}\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathtt{force}\ x \\
(\mathtt{absurd}\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathtt{absurd}\ x \\
(\mathtt{case}\ V \{x_1.E_1 \mid x_2.E_2\})^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathtt{case}\ x \{x_1.{E_1}^\dagger \mid x_2.{E_2}^\dagger\} \\
(\mathtt{split}\ V\, \mathtt{to}\ ().E)^\dagger & = & \mathtt{bind}\ w \leftarrow V^\dagger ; \mathtt{split}\ w\, \mathtt{to}\ ().E^\dagger \\
(\mathtt{split}\ V\, \mathtt{to}\ (x,y).E)^\dagger & = & \mathtt{bind}\ w \leftarrow V^\dagger ; \mathtt{split}\ w\, \mathtt{to}\ (x,y).E^\dagger \\
(\mathtt{unroll}\ V\, \mathtt{to}\ \mathtt{roll}\ x.E)^\dagger & = & \mathtt{bind}\ y \leftarrow V^\dagger ; \mathtt{unroll}\ y\, \mathtt{to}\ \mathtt{roll}\ x.E^\dagger
\end{array}
$$

$$
\begin{array}{rcl}
x^\dagger & = & \mathit{ret} x \\
(\mathtt{inl}\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathit{ret}\mathtt{inl}\ x \\
(\mathtt{inr}\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathit{ret}\mathtt{inr}\ x \\
()^\dagger & = & \mathit{ret}() \\
(V_1, V_2)^\dagger & = & \mathtt{bind}\ x_1 \leftarrow {V_1}^\dagger ; \mathtt{bind}\ x_2 \leftarrow {V_2}^\dagger ; \mathit{ret}(x_1, x_2) \\
(\mathtt{thunk}\ M)^\dagger & = & \mathit{ret}\mathtt{thunk}\ M^\dagger \\
(\mathtt{roll}\ V)^\dagger & = & \mathtt{bind}\ x \leftarrow V^\dagger ; \mathtt{roll}\ x
\end{array}
$$

The translation is type-preserving and the identity from CBPV*'s point of view

Lemma 6.1 (De-complexification De-complexifies). *For any CBPV* term $\Gamma \mid \Delta \vdash E : T$, $E^\dagger$ is a term of CBPV satisfying $\Gamma, \Delta^\dagger \vdash E^\dagger : T^\dagger$ where $\cdot^\dagger = \cdot$ $(\bullet : \underline{B})^\dagger = z : U\underline{B}$, $\underline{B}^\dagger = \underline{B}$, $A^\dagger = \underline{F}A$.*

Lemma 6.2 (De-complexification is Identity in CBPV*). *Considering CBPV as a subset of CBPV* we have*

1. *If $\Gamma \mid \cdot \vdash M : \underline{B}$ then $M \sqsupseteq\sqsubseteq M^\dagger$.*
2. *If $\Gamma \mid \Delta \vdash S : \underline{B}$ then $S[\mathtt{force}\ z] \sqsupseteq\sqsubseteq S^\dagger$.*
3. *If $\Gamma \vdash V : A$ then $\mathit{ret} V \sqsupseteq\sqsubseteq V^\dagger$.*

Furthermore, if M, V, S are in CBPV, the proof holds in CBPV.

Finally, we need to show that the translation preserves inequalities ($E^\dagger \sqsubseteq E'^\dagger$ if $E \sqsubseteq E'$), where the target inequational theory is given by restricting GTT to the homogeneous fragment and adding monotonicity and $\beta\eta$ rules for recursive types (see appendix for details). In particular, the thunkability/linearity properties are needed to prove the preservation of the η principles for value types and the strictness of complex stacks with respect to errors under decomplexification.

We need a few lemmas about thunkables and linears to prove that complex values become thunkable and complex stacks become linear. We show them in detail here because many of them correspond to program optimizations that are valid for thunkable/linear terms and therefore apply to upcasts and downcasts.

First, the following lemma is useful for optimizing programs with thunkable subterms. Intuitively, since a thunkable has "no effects" it can be reordered past any other effectful binding. Führmann (1999) calls a morphism that has this property *central* (after the center of a group, which is those elements that commute with every element of the whole group).

Lemma 6.3 (Thunkables are Central)**.** *If* $\Gamma \vdash M : \underline{F}A$ *is thunkable and* $\Gamma \vdash N : \underline{F}A'$ *and* $\Gamma, x : A, y : A' \vdash N' : \underline{B}$*, then*

$$\texttt{bind}\ x \leftarrow M; \texttt{bind}\ y \leftarrow N; N' \sqsupseteq\sqsubseteq \texttt{bind}\ y \leftarrow N; \texttt{bind}\ x \leftarrow M; N'$$

Next, we show thunkables are closed under composition and that return of a value is always thunkable. This allows us to easily build up bigger thunkables from smaller ones.

Lemma 6.4 (Thunkables compose)**.** *If* $\Gamma \vdash M : \underline{F}A$ *and* $\Gamma, x : A \vdash N : \underline{F}A'$ *are thunkable, then*

$$\texttt{bind}\ x \leftarrow M; N$$

is thunkable.

Lemma 6.5 (Return is Thunkable)**.** *If* $\Gamma \vdash V : A$ *then* $\texttt{ret}V$ *is thunkable.*

Proof. By $\underline{F}\beta$:

$$\texttt{bind}\ x \leftarrow \texttt{ret}V; \texttt{retthunk}\ \texttt{ret}x \sqsupseteq\sqsubseteq \texttt{retthunk}\ \texttt{ret}V$$

□

And we can then prove the desired property for complex values:

Lemma 6.6 (Complex Values Simplify to Thunkable Terms)**.** *If* $\Gamma \vdash V : A$ *is a (possibly) complex value, then* $\Gamma \vdash V^{\dagger} : \underline{F}A$ *is thunkable.*

Dually, we have that a stack out of a force is linear and that linears are closed under composition, so we can easily build up bigger linear morphisms from smaller ones.

Lemma 6.7 (Force to a stack is Linear)**.** *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$*, then* $\Gamma, x : U\underline{B} \vdash S[\texttt{force}\ x] : \underline{B}$ *is linear in x.*

Proof.

$$\begin{aligned} S[\texttt{force thunk}\ (\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)] &\sqsupseteq\sqsubseteq S[(\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)] \\ &\qquad (U\beta) \\ &\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{force}\ z; S[\texttt{force}\ x] \\ &\qquad (\underline{F}\eta) \end{aligned}$$

□

Lemma 6.8 (Linear Terms Compose)**.** *If* $\Gamma, x : U\underline{B} \vdash M : \underline{B}'$ *is linear in* x *and* $\Gamma, y : \underline{B}' \vdash N : \underline{B}''$ *is linear in* y*, then* $\Gamma, x : U\underline{B} \vdash N[\texttt{thunk}\ M/y] :$

Lemma 6.9 (Complex Stacks Simplify to Linear Terms)**.** *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$ *is a (possibly) complex stack, then* $\Gamma, z : U\underline{B} \vdash (S)^{\dagger} : \underline{C}$ *is linear in* z*.*

Composing this with the previous translation from GTT to CBPV* shows that *GTT value type upcasts are thunkable and computation type downcasts are linear*.

Since the translation takes values and stacks to terms, it cannot preserve substitution up to equality. Rather, we get the following, weaker notion that says that the translation of a syntactic substitution is equivalent to an effectful composition.

Lemma 6.10 (Compositionality of De-complexification)**.**

1. *If* $\Gamma, x : A \mid \Delta \vdash E : T$ *and* $\Gamma \vdash V : A$ *are complex terms, then*

$$(E[V/x])^{\dagger} \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow V^{\dagger}; E^{\dagger}$$

2. *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$ *and* $\Gamma \mid \Delta \vdash M : \underline{B}$*, then*

$$(S[M])^{\dagger} \sqsupseteq\sqsubseteq S^{\dagger}[\texttt{thunk}\ M^{\dagger}/z]$$

Finally we conclude with our desired theorem, that de-complexification preserves the precision relation.

Theorem 6.1 (De-complexification preserves Precision)**.** *If* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ *then* $\Gamma, \Delta^{\dagger} \vdash E^{\dagger} \sqsubseteq E'^{\dagger} : T^{\dagger}$

Proof. By induction over precision derivations. Details are in the appendix. □

As a corollary, we also get the following conservativity result that says that precision in CBPV with complex values and stacks coincides with CBPV without them. This shows that complex values and stacks can be viewed as simply a convenient way to manipulate thunkable and linear terms and the calculus is not fundamentally different from CBPV.

Corollary 6.1 (Complex CBPV is Conservative over CBPV)**.** *If* M, M' *are terms in CBPV and* $M \sqsubseteq M'$ *is provable in CBPV* then* $M \sqsubseteq M'$ *is provable in CBPV.*

Proof. Because de-complexification preserves precision, $M^{\dagger} \sqsubseteq M'^{\dagger}$ in simple CBPV. Then it follows because de-complexification is equivalent to identity (in CBPV):

$$M \sqsupseteq\sqsubseteq M^{\dagger} \sqsubseteq M'^{\dagger} \sqsupseteq\sqsubseteq M'$$

□

7 Operational model of GTT

In this section, we complete our model construction for GTT by providing an operational semantics and a semantic interpretation of the error ordering $\sqsubseteq$ based on observational approximation. First, we define a mostly standard CBPV operational semantics, which in turn provides an operational semantics of GTT by the elaborations defined in Sections 5

$$\begin{array}{rcl}
S[\mho] & \mapsto^{0} & \mho \\
S[\texttt{case inl } V\{x_1.M_1 \mid x_2.M_2\}] & \mapsto^{0} & S[M_1[V/x_1]] \\
S[\texttt{case inr } V\{x_1.M_1 \mid x_2.M_2\}] & \mapsto^{0} & S[M_2[V/x_2]] \\
S[\texttt{split } (V_1, V_2)\, \texttt{to}\, (x_1, x_2).M] & \mapsto^{0} & S[M[V_1/x_1, V_2/x_2]] \\
S[\texttt{unroll roll}_A\ V\, \texttt{to roll}\ x.M] & \mapsto^{1} & S[M[V/x]] \\
S[\texttt{force thunk}\ M] & \mapsto^{0} & S[M] \\
S[\texttt{let}\ x = V; M] & \mapsto^{0} & S[M[V/x]] \\
S[\texttt{bind}\ x \leftarrow \texttt{ret}\, V; M] & \mapsto^{0} & S[M[V/x]] \\
S[(\lambda x : A.M)\ V] & \mapsto^{0} & S[M[V/x]] \\
S[\pi\{\pi \mapsto M \mid \pi' \mapsto M'\}] & \mapsto^{0} & S[M] \\
S[\pi'\{\pi \mapsto M \mid \pi' \mapsto M'\}] & \mapsto^{0} & S[M'] \\
S[\texttt{unroll roll}_{\underline{B}}\ M] & \mapsto^{1} & S[M]
\end{array}$$

$$\frac{}{M \Mapsto^{0} M} \qquad \frac{M_1 \mapsto^{i} M_2 \qquad M_2 \Mapsto^{j} M_3}{M_1 \Mapsto^{i+j} M_3}$$

Fig. 19. CBPV operational semantics.

and 6. Then, we define a notion of observational approximation that captures the core semantic idea of graduality: $M \sqsubseteq N$ when they have the same behavior, except M sometimes errors when N does not. Finally, we prove graduality by showing that our term precision syntax is sound for observational approximation. To prove this last point, we construct a more flexible semantic formulation of the error ordering: a step-indexed biorthogonal logical relation for CBPV. This section is necessarily fairly technical, especially Section 7.3, which concerns the logical relation and will be most useful for those interested in logical relations for graduality and CBPV more generally.

7.1 Call-by-push-value operational semantics

We use a small-step operational semantics for CBPV as shown in Figure 19. Note that for this definition, V and S represent simple values and stacks as shown in Figure 18, not the more general complex values and stacks. The single-step semantics $M \mapsto^{i} N$ is, if we ignore the i, the ordinary small-step semantics of CBPV as found in Levy (2003), but written in a Felleisen-Hieb style using stacks in place of evaluation contexts. The index i, used later in our step-indexed logical relation, is used to count the reductions that unroll uses of recursive or corecursive types: those reductions cost 1 step while others are free (0 steps). We then define a quantitative version of the reflexive, transitive closure $M \Mapsto^{i} N$ where the reflexivity step costs 0 and a chain of reductions adds the cost of each step. Note that in the remainder of this section we will only ever need to reduce closed computations of type $\underline{F}A$.

We can then observe the following standard operational properties. (We write $M \mapsto N$ with no index when the index is irrelevant.)

Lemma 7.1 (Reduction is Deterministic)**.** *If $M \mapsto M_1$ and $M \mapsto M_2$, then $M_1 = M_2$.*

Lemma 7.2 (Subject Reduction)**.** *If $\cdot \vdash M : \underline{F}A$ and $M \mapsto M'$ then $\cdot \vdash M' : \underline{F}A$.*

Lemma 7.3 (Progress)**.** *If $\cdot \vdash M : \underline{F}A$ then one of the following holds:*

$$M = \mho \qquad\qquad M = \mathtt{ret} V \text{ with } V : A \qquad\qquad \exists M'.\, M \mapsto M'$$

The standard progress-and-preservation properties allow us to define the "final result" of a computation as follows:

Corollary 7.1 (Possible Results of Computation)**.** *For any* $\cdot \vdash M : \underline{F}(1+1)$*, one of the following is true:*

$$M \Uparrow \qquad M \Downarrow \mho \qquad M \Downarrow \mathtt{ret\,true} \qquad M \Downarrow \mathtt{ret\,false}$$

Proof. We define $M \Uparrow$ to hold when if $M \Mapsto^i N$ then there exists N' with $N \mapsto N'$. For the terminating results, we define $M \Downarrow R$ to hold if there exists some i with $M \Mapsto^i R$. Then we prove the result by coinduction on execution traces. If $M \in \{\mho, \mathtt{ret\,true}, \mathtt{ret\,false}\}$ then we are done, otherwise by progress, $M \mapsto M'$, so we need only observe that each of the cases above is preserved by $\mapsto$. □

Definition 7.1 (Results)**.** *A result is one of* $\Omega, \mho, \mathtt{ret\,true}$*. We denote results by* R*. We define the result of a closed program* $\cdot \vdash M : \underline{F}(1+1)$ *as follows, justified by by Corollary 7.1.*

$$result(M) = \begin{cases} \Omega & \text{if } M \Uparrow \\ R & \text{if } M \Downarrow R \end{cases}$$

7.2 Observational equivalence and approximation

Next, we define observational equivalence and approximation in CBPV. The (standard) definition of observational equivalence is that we consider two terms (or values) to be equivalent when replacing one with the other in any program text produces the same overall resulting computation. In Figure 20 we define a context C be a term/value/stack with a single $[\cdot]$ as some subterm/value/stack, and define a typing $C : (\Gamma \vdash \underline{B}) \Rightarrow (\Gamma' \vdash \underline{B}')$ to hold when for any $\Gamma \vdash M : \underline{B}$, $\Gamma' \vdash C[M] : \underline{B}'$ (and similarly for values/stacks). Using contexts, we can lift any relation on *results* to relations on open terms, values and stacks.

Definition 7.2 (Contextual Lifting)**.** *Given any relation* $\sim \subseteq Result \times Result$*, we can define its* observational lift $\sim^{ctx}$ *to be the typed relation defined by*

$$\Gamma \mid \Delta \vDash E \sim^{ctx} E' \in T = \forall C : (\Gamma \mid \Delta \vdash T) \Rightarrow (\cdot \vdash \underline{F}2).\ result(C[E]) \sim result(C[E'])$$

The contextual lifting $\sim^{\mathrm{ctx}}$ inherits much structure of the original relation $\sim$ as the following lemma shows. This justifies calling $\sim^{\mathrm{ctx}}$ a contextual preorder when $\sim$ is a preorder (reflexive and transitive) and similarly a contextual equivalence when $\sim$ is an equivalence (preorder and symmetric).

Lemma 7.4 (Contextual Lifting preserves Preorder, Equivalence properties)**.** *If* $\sim$ *is reflexive, symmetric or transitive, then for each typing,* $\sim^{ctx}$ *is reflexive, symmetric or transitive as well, respectively.*

$$\begin{array}{rcl}
C_V & ::= & [\cdot] \mid \texttt{roll}_{\mu X.A}\ C_V \mid \texttt{inl}\ C_V \mid \texttt{inr}\ C_V \mid (C_V, V) \mid (V, C_V) \mid \texttt{thunk}\ C_M \\
\\
C_M & ::= & [\cdot] \mid \texttt{let}\ x = C_V; M \mid \texttt{let}\ x = V; C_M \mid \texttt{unroll}\ C_V\ \texttt{to roll}\ x.M \\
 & \mid & \texttt{unroll}\ V\ \texttt{to roll}\ x.C_M \mid \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ C_M \mid \texttt{unroll}\ C_M \mid \texttt{abort}\ C_V \\
 & \mid & \texttt{case}\ C_V\{x_1.M_1 \mid x_2.M_2\} \mid \texttt{case}\ V\{x_1.C_M \mid x_2.M_2\} \mid \texttt{case}\ V\{x_1.M_1 \mid x_2.C_M\} \\
 & \mid & \texttt{split}\ C_V\ \texttt{to}\ ().M \mid \texttt{split}\ V\ \texttt{to}\ ().C_M \mid \texttt{split}\ C_V\ \texttt{to}\ (x,y).M \\
 & \mid & \texttt{split}\ V\ \texttt{to}\ (x,y).C_M \mid \texttt{force}\ C_V \mid \texttt{ret} C_V \mid \texttt{bind}\ x \leftarrow C_M; N \\
 & \mid & \texttt{bind}\ x \leftarrow M; C_M \mid \lambda x : A.C_M \mid C_M\ V \mid M\ C_V \\
 & \mid & \{\pi \mapsto C_M \mid \pi' \mapsto M_2\} \mid \{\pi \mapsto M_1 \mid \pi' \mapsto C_M\} \mid \pi C_M \mid \pi' C_M \\
C_S & ::= & \pi C_S \mid \pi' C_S \mid S\ C_V \mid C_S\ V \mid \texttt{bind}\ x \leftarrow C_S; M \mid \texttt{bind}\ x \leftarrow S; C_M
\end{array}$$

Fig. 20. CBPV contexts.

In the remainder of the paper we work only with relations that are at least preorders so we write $\trianglelefteq$ rather than $\sim$.

The most famous use of lifting is for observational equivalence, which is the lifting of equality of results ($=^{\text{ctx}}$), and we will show that $\sqsupseteq\sqsubseteq$ proofs in GTT imply observational equivalences. However, as shown in New & Ahmed (2018), the graduality property is defined in terms of an observational *approximation* relation $\sqsubseteq$ that places $\mho$ as the least element, and every other element as a maximal element. Note that this is *not* the standard notion of observational approximation, which we write $\preceq$, which makes Ω a least element and every other element a maximal element. To distinguish these, we call $\sqsubseteq$ *error* approximation and $\preceq$ *divergence* approximation. We present these graphically (with two more) in Figure 21.

The goal of this section is to prove that a symmetric equality $E \sqsupseteq\sqsubseteq E'$ in CBPV (i.e., $E \sqsubseteq E'$ and $E' \sqsubseteq E$) implies contextual equivalence $E =^{\text{ctx}} E'$ and that inequality in CBPV $E \sqsubseteq E'$ implies error approximation $E \sqsubseteq^{\text{ctx}} E'$, proving graduality of the operational model:

$$\frac{\Gamma \mid \Delta \vdash E \sqsupseteq\sqsubseteq E' : T}{\Gamma \mid \Delta \vDash E =^{\text{ctx}} E' \in T} \qquad \frac{\Gamma \mid \Delta \vdash E \sqsubseteq E' : T}{\Gamma \mid \Delta \vDash E \sqsubseteq^{\text{ctx}} E' \in T}$$

Because we have non-well-founded μ/ν types, we use a *step-indexed logical relation* to prove properties about the contextual lifting of certain preorders $\trianglelefteq$ on results. In step-indexing, the *infinitary* relation given by $\trianglelefteq^{\text{ctx}}$ is related to the set of all of its *finitary approximations* $\trianglelefteq^i$, which "time out" after observing i steps of evaluation and declare that the terms *are* related. This means that the original relation is only recoverable from the finite approximations if Ω is always related to another element: if the relation is a preorder, we require that Ω is a *least* element.

We call such a preorder a *divergence preorder*.

Definition 7.3 (Divergence Preorder)**.** *A preorder on results $\trianglelefteq$ is a divergence preorder if $\Omega \trianglelefteq R$ for all results R.*

But this presents a problem, because *neither* of our intended relations ($=$ and $\sqsubseteq$) is a divergence preorder; rather both have Ω as a *maximal* element. However, there is a standard "trick" for subverting this obstacle in the case of contextual equivalence (Ahmed,

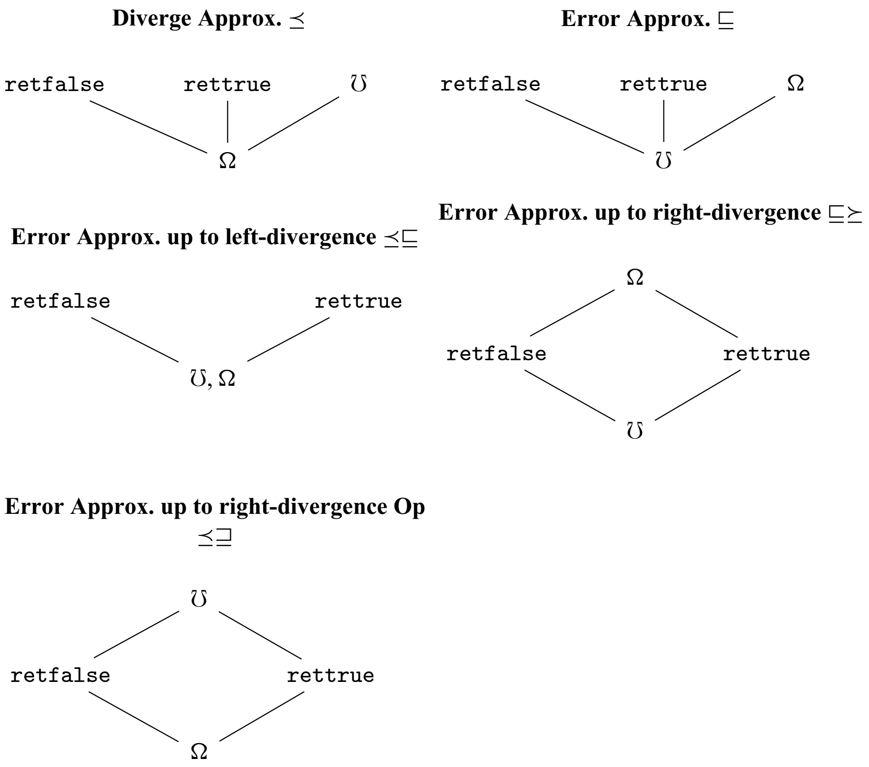
Diverge Approx. ≼
retfalse
rettrue
℧
Ω
Error Approx. ⊑
retfalse
rettrue
Ω
℧
Error Approx. up to left-divergence ≼⊑
retfalse
rettrue
℧, Ω
Error Approx. up to right-divergence ⊑≽
Ω
retfalse
rettrue
℧
Error Approx. up to right-divergence Op ≼⊒
℧
retfalse
rettrue
Ω

Fig. 21. Result orderings.

2006): we notice that we can define equivalence as the symmetrization of divergence approximation, i.e., $M =^{\text{ctx}} N$ if and only if $M \preceq^{\text{ctx}} N$ and $N \preceq^{\text{ctx}} M$, and since $\preceq$ has Ω as a least element, we can use a step-indexed relation to prove it. As shown in New & Ahmed (2018), a similar trick works for error approximation, but since $\sqsubseteq$ is *not* an equivalence relation, we decompose it rather into two *different* orderings: error approximation up to divergence on the left $\preceq\sqsubseteq$ and error approximation up to divergence on the right $\sqsubseteq\succeq$, also shown in Figure 21. Note that $\preceq\sqsubseteq$ is a preorder, but not a poset because $\mho, \Omega$ are order equivalent but not equal. Then clearly $\preceq\sqsubseteq$ is a divergence preorder and the *opposite* of $\sqsubseteq\succeq$, written $\preceq\sqsupseteq$ is a divergence preorder.

Then we can completely reduce the problem of proving $=^{\text{ctx}}$ and $\sqsubseteq^{\text{ctx}}$ results to proving results about divergence preorders by the following characterizations, which can also be seen as alternative definitions of $\preceq\sqsubseteq$, $\sqsubseteq\succeq$.

Lemma 7.5 (Decomposing Result Preorders). *Let R, S be results.*

1. *$R = S$ if and only if $R \sqsubseteq S$ and $S \sqsubseteq R$.*
2. *$R = S$ if and only if $R \preceq S$ and $S \preceq R$.*
3. *$R \preceq\sqsubseteq S$ iff $R \sqsubseteq S$ or $R \preceq S$.*
4. *$R \sqsubseteq\succeq S$ iff $R \sqsubseteq S$ or $R \succeq S$.*

Which easily extends to similar facts about the contextual liftings.

Corollary 7.2 (Contextual Decomposition)**.** *Let R^o be defined as the transpose relation $xR^o y = yRx$, then*

1. $=^{ctx} \Leftrightarrow \preceq^{ctx} \wedge((\preceq)^{ctx})^\circ$
2. $=^{ctx} \Leftrightarrow \sqsubseteq^{ctx} \wedge((\sqsubseteq)^{ctx})^\circ$
3. $\sqsubseteq^{ctx} \Leftrightarrow \preceq\sqsubseteq^{ctx} \wedge((\preceq\sqsupseteq)^{ctx})^\circ$

7.3 CBPV step-indexed logical relation

Next, we turn to the problem of proving results about $E \trianglelefteq^{\mathrm{ctx}} E'$ where $\trianglelefteq$ is a divergence preorder. Dealing directly with a contextual preorder is practically impossible, so instead we develop an alternative formulation as a logical relation that is much easier to use. Fortunately, we can apply standard logical relations techniques to provide an alternate definition *inductively* on types. However, since we have non-well-founded type definitions using μ and ν, our logical relation will also be defined inductively on a *step index* that times out when we've exhausted our step budget. To bridge the gap between the indexed logical relation and the divergence preorder we care about, we define the "finitization" of a divergence preorder to be a relation between *programs* and *results*: the idea is that a program approximates a result R at index i if it reduces to R in less than i steps or it reduces at least i times.

Definition 7.4 (Finitized Preorder)**.** *Given a divergence preorder $\trianglelefteq$, we define the* finitization *of $\trianglelefteq$ to be, for each natural number i, a relation between programs and results*

$$\trianglelefteq^i \subseteq \{M \mid \cdot \vdash M : \underline{F}(1+1)\} \times \mathit{Results}$$

defined by

$$M \trianglelefteq^i R = (\exists M'.\, M \Mapsto^i M') \vee (\exists (j < i). \exists R_M.\, M \Mapsto^j R_M \wedge R_M \trianglelefteq R)$$

Note that in this definition, unlike in the definition of divergence, we only count non-well-founded steps. This makes it slightly harder to establish the intended equivalence $M \trianglelefteq^\omega R$ if and only if $\mathrm{result}(M) \trianglelefteq R$, but makes the logical relation theorem stronger: it proves that diverging terms must use recursive types of some sort and so any term that does not use them terminates. This issue would be alleviated if we had proved type safety by a logical relation rather than by progress and preservation.

However, the following properties of the indexed relation can easily be established. First, a kind of "transitivity" of the indexed relation with respect to the original preorder, which is key to proving transitivity of the logical relation.

Lemma 7.6 (Indexed Relation is a Module of the Preorder)**.** *If $M \trianglelefteq^i R$ and $R \trianglelefteq R'$ then $M \trianglelefteq^i R'$*

Proof. If $M \Mapsto^i M'$ then there's nothing to show, otherwise $M \Mapsto^{j<i} \mathrm{result}(M)$ so it follows by transitivity of the preorder: $\mathrm{result}(M) \trianglelefteq R \trianglelefteq R'$. □

Then we establish a few basic properties of the finitized preorder.

Lemma 7.7 (Downward Closure of Finitized Preorder)**.** *If $M \trianglelefteq^i R$ and $j \leq i$ then $M \trianglelefteq^j R$.*

Proof. Since $M \trianglelefteq^i R$, either $M \Mapsto^i M_i$ or there exists $k < i, R_M$ with $M \Mapsto^k R_M$ and $R_M \trianglelefteq R$.

1. If $M \Mapsto^i M_i$ then there exists some M_j with $M \Mapsto^j M_j$ since $j \leq i$.
2. Otherwise, $M \Mapsto^k R_M$ where $k < i$ and $R_M \trianglelefteq R$. If $k \geq j$, then we "time out" and $M_j M_j$ for some intermediate M_j, otherwise the second case of $\trianglelefteq^j$ applies. □

Lemma 7.8 (Triviality at 0)**.** *For any $\cdot \vdash M : \underline{F}(1+1)$, $M \trianglelefteq^0 R$*

Proof. Because $M \Mapsto^0 M$ □

Lemma 7.9 (Result (Anti-)reduction)**.** *If $M \Mapsto^i N$ then $\mathit{result}(M) = \mathit{result}(N)$.*

Lemma 7.10 (Anti-reduction)**.** *If $M \trianglelefteq^i R$ and $N \Mapsto^j M$, then $N \trianglelefteq^{i+j} R$*

Proof.

1. If $M \Mapsto^i M'$ then $N \Mapsto^{i+j} M'$
2. If $M \Mapsto^{k<i}$ result(M) then $N \Mapsto^{k+j}$ result(M) and result$(M) =$ result(N) and $k + j < i + j$. □

Next, we define the (closed) *logical* preorder for values and stacks by induction on types and the index i in Figure 22. First, we discuss the value relation. For every natural number i and value type A we define a relation $\trianglelefteq^{\log}_{A,i}$ between closed values of type A. Two values should be related when any use of them would result in related behaviors. The relation is defined in a type-directed fashion, the intuition being that we relate two positive values when they are built up in the same way: i.e., they have the same introduction form and their subterms are related. First, the empty type 0 is associated to the empty relation, because there are no closed values. Next, values of a sum type $A + A'$ are related if they are the same case, and their components are related. The unique unit value is related to itself. Pairs are related if both subcomponents are related. For values of recursive type $\mu X.A$, it would not be well founded to say that they are related if their unrolled values are related at the same index i, because this would be at a larger type $A[\mu X.A/X]$. Instead, we also decrease the index by 1 when we relate to subcomponents. This relation bottoms out and has any well-typed values as related if the index is 0. Finally U is treated differently from the other value types because it is the only one not eliminated by pattern matching. A thunk $V : U\underline{B}$ can only be used by forcing it. By the definition of the operational semantics, this only ever occurs in the step $S[\mathtt{force}\ V]$, so (ignoring indices for a moment), we should define $V_1 \trianglelefteq V_2$ to hold in this case when, given any $S_1 \trianglelefteq S_2$, the result of $S_2[\mathtt{force}\ V_2]$ is approximated by $S_1[\mathtt{force}\ V_1]$. To incorporate the indices, we have to quantify over $j \leq i$ in this definition because we need to know that the values are related in all futures, including ones where some other part of the term has been reduced (consuming some

$$\begin{aligned}
\trianglelefteq^{\log}_{A,i} &\subseteq \{\cdot \vdash V : A\} \times \{\cdot \vdash V : A\} \\
V_1 \trianglelefteq^{\log}_{0,i} V_2 &= \bot \\
\texttt{inl}\ V_1 \trianglelefteq^{\log}_{A+A',i} \texttt{inl}\ V_2 &= V_1 \trianglelefteq^{\log}_{A,i} V_2 \\
\texttt{inr}\ V_1 \trianglelefteq^{\log}_{A+A',i} \texttt{inr}\ V_2 &= V_1 \trianglelefteq^{\log}_{A',i} V_2 \\
() \trianglelefteq^{\log}_{1,i} () &= \top \\
(V_1, V_1') \trianglelefteq^{\log}_{A\times A',i} (V_2, V_2') &= V_1 \trianglelefteq^{\log}_{A,i} V_2 \wedge V_1' \trianglelefteq^{\log}_{A',i} V_2' \\
\texttt{roll}_{\mu X.A}\ V_1 \trianglelefteq^{\log}_{\mu X.A,i} \texttt{roll}_{\mu X.A}\ V_2 &= i = 0 \vee V_1 \trianglelefteq^{\log}_{A[\mu X.A/X],i-1} V_2 \\
V_1 \trianglelefteq^{\log}_{U\underline{B},i} V_2 &= \forall j \le i.\ \forall S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2.\ S_1[\texttt{force}\ V_1] \trianglelefteq^{j} \text{result}(S_2[\texttt{force}\ V_2])
\end{aligned}$$

$$\begin{aligned}
\trianglelefteq^{\log}_{\underline{B},i} &\subseteq \{\cdot \mid \bullet : \underline{B} \vdash S : \underline{F}(1+1)\} \times \{\cdot \mid \bullet : \underline{B} \vdash S : \underline{F}(1+1)\} \\
S_1[\bullet V_1] \trianglelefteq^{\log}_{A\to\underline{B},i} S_1[\bullet V_2] &= V_1 \trianglelefteq^{\log}_{A,i} V_2 \wedge S_1 \trianglelefteq^{\log}_{\underline{B},i} S_2 \\
S_1[\pi_1 \bullet] \trianglelefteq^{\log}_{\underline{B}\&\underline{B}',i} S_2[\pi_1 \bullet] &= S_1 \trianglelefteq^{\log}_{\underline{B},i} S_2 \\
S_1[\pi_2 \bullet] \trianglelefteq^{\log}_{\underline{B}\&\underline{B}',i} S_2[\pi_2 \bullet] &= S_1 \trianglelefteq^{\log}_{\underline{B}',i} S_2 \\
S_1 \trianglelefteq^{\log}_{\top,i} S_2 &= \bot \\
S_1[\texttt{unroll}\ \bullet] \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B},i} S_2[\texttt{unroll}\ \bullet] &= i = 0 \vee S_1 \trianglelefteq^{\log}_{\underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}],i-1} S_2 \\
S_1 \trianglelefteq^{\log}_{\underline{F}A,i} S_2 &= \forall j \le i.\ \forall V_1 \trianglelefteq^{\log}_{A,j} V_2.\ S_1[\texttt{ret}\, V_1] \trianglelefteq^{j} \text{result}(S_2[\texttt{ret}\, V_2])
\end{aligned}$$

$$\begin{aligned}
\cdot \trianglelefteq^{\log}_{\cdot,i} \cdot &= \top \\
\gamma_1, V_1/x \trianglelefteq^{\log}_{\Gamma,x:A,i} \gamma_2, V_2/x &= \gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2 \wedge V_1 \trianglelefteq^{\log}_{A,i} V_2
\end{aligned}$$

Fig. 22. Logical relation from a preorder $\trianglelefteq$.

steps). From a mathematical perspective, this quantification over smaller indices is crucial for ensuring the relation is downward closed.

Next, we define when two *stacks* are related. We define the relation only for two "closed" stacks, which both have the same type of their *hole* $\underline{B}$ and both have "output" the observation type $\underline{F}(1 + 1)$. The reason is that in evaluating a program M, steps always occur as $S[N] \mapsto S[N']$ where S is a stack of this form. An intuition for the relation is that for negative types, two stacks are related when they start with the same elimination form and the remainder of the stacks are related. Two function stacks are related if they both apply the hole to related values, and then apply related stacks to that result (analogous to the value product). Two product stacks are related if they make the same projection and then use the result in related ways (analogous to the value sum). There are no stacks out of the $\top$ type, so the relation is empty (analogous to the empty value type). For ν, we handle the step indices in the same way as for μ, saying both stacks unroll the hole and then use it in related ways, decrementing the index. Finally, for $\underline{F}A$, a stack $S[\bullet : \underline{F}A]$ is strict in its input and waits for its input to evaluate down to a value $\texttt{ret}\, V$, so two stacks with $\underline{F}A$ holes are related when in any future world, they produce related behavior when given related values (analogous to the U type).

Readers interested in logical relations should note that the quantification over related stacks in the U relation and corresponding quantification over related values in the $\underline{F}$ relation are instances of a general construction known as the *orthogonal* of a relation (Pitts &

Stark, 1998), and such logical relations are often called *biorthogonal*. One advantage of the CBPV language is that it makes the use of orthogonality in logical relations *explicit* in the type structure, analogous to the benefits of using Nakano's *later* modality (Nakano, 2000) for step indexing (which we ironically do not do). For instance, in a typical biorthogonal logical relation for CBV, the CBV function type relation has a somewhat complex definition involving both orthogonals. The presence of both orthogonals is nicely explained by the CBPV translation of the CBPV function type: $U(A \to \underline{F}A')$, which uses both the U and $\underline{F}$ types, and unfolding the definition of $\trianglelefteq^{\text{log}}_{U(A\to\underline{F}A'),i}$ will produce something essentially the same as the usual CBV function relation.

Finally, we extend the definition to contexts to *closing substitutions* pointwise: two closing substitutions for Γ are related at i if they are related at i for each $x : A \in \Gamma$. Note that the definition is well founded using the lexicographic ordering on (i, A) and $(i, \underline{B})$: either the type reduces and the index stays the same or the index reduces.

The logical preorder for open terms is defined as usual by quantifying over all related closing substitutions, but also over all stacks to the observation type $\underline{F}(1 + 1)$:

Definition 7.5 (Logical Preorder)**.** *For a divergence preorder $\trianglelefteq$, its step-indexed logical preorder is*

1. $\Gamma \vDash M_1 \trianglelefteq^{log}_i M_2 \in \underline{B}$ *iff for every* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_2$ *and* $S_1 \trianglelefteq^{log}_{\underline{B},i} S_2$,
$$S_1[M_1[\gamma_1]] \trianglelefteq^{i} \mathit{result}(S_2[M_2[\gamma_2]]).$$
2. $\Gamma \vDash V_1 \trianglelefteq^{log}_i V_2 \in A$ *iff for every* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_2$,
$$V_1[\gamma_1] \trianglelefteq^{log}_{A,i} V_2[\gamma_2].$$
3. $\Gamma \mid \underline{B} \vDash S_1 \trianglelefteq^{log}_i S_2 \in \underline{B}'$ *iff for every* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_2$ *and* $S'_1 \trianglelefteq^{log}_{\underline{B}',i} S'_2$,
$$S'_1[S_1[\gamma_1]] \trianglelefteq^{log}_{\underline{B},i} S'_2[S_2[\gamma_2]]).$$

We next want to prove that the logical preorder is a congruence relation, i.e., the fundamental lemma of the logical relation. This requires the easy lemma, that the relation on closed terms and stacks is downward closed.

Lemma 7.11 (Logical Relation Downward Closure)**.** *For any type T, if $j \leq i$ then $\trianglelefteq^{log}_{T,i} \subseteq \trianglelefteq^{log}_{T,j}$*

Next, we show the fundamental theorem:

Theorem 7.1 (Logical Preorder is a Congruence)**.** *For any divergence preorder, the logical preorder $E \trianglelefteq^{log}_i E'$ is a congruence relation, i.e., it is closed under applying any value/term/stack constructors to both sides.*

Proof. The proof is straightforward, with one case for each term former, and is included in the appendix. □

As a direct consequence we get the reflexivity of the relation:

Corollary 7.3 (Reflexivity)**.** *For any* $\Gamma \vdash M : \underline{B}$*, and* $i \in \mathbb{N}$, $\Gamma \vDash M \trianglelefteq^{log}_{i} M \in \underline{B}$.

Therefore we have the following *strengthening* of the progress-and-preservation type soundness theorem: because $\trianglelefteq^{i}$ only counts unrolling steps, terms that never use μ or ν types (for example) are guaranteed to terminate.

Corollary 7.4 (Unary LR)**.** *For every program* $\cdot \vdash M : \underline{F}(1+1)$ *and* $i \in \mathbb{N}$, $M \trianglelefteq^{i}$ *result*(M)

Proof. By reflexivity, $\cdot \vDash M \trianglelefteq^{i} M \in \underline{F}(1+1)$ and by definition $\bullet \trianglelefteq^{\log}_{\underline{F}(1+1),i} \bullet$, so unrolling definitions we get $M \trianglelefteq^{i} \text{result}(M)$. □

Using reflexivity, we prove that the indexed relation between terms and results recovers the original preorder in the limit as $i \to \omega$. We write $\trianglelefteq^{\omega}$ to mean the relation holds for every i, i.e., $\trianglelefteq^{\omega} = \bigcap_{i \in \mathbb{N}} \trianglelefteq^{i}$.

Corollary 7.5 (Limit Lemma)**.** *For any divergence preorder* $\trianglelefteq$, *result*$(M) \trianglelefteq R$ *iff* $M \trianglelefteq^{\omega} R$.

Corollary 7.6 (Logical implies Contextual)**.** *If* $\Gamma \vDash E \trianglelefteq^{log}_{\omega} E' \in \underline{B}$ *then* $\Gamma \vDash E \trianglelefteq^{ctx} E' \in \underline{B}$.

Proof. Let C be a closing context. By congruence, $C[M] \trianglelefteq^{\log}_{\omega} C[N]$, so using empty environment and stack, $C[M] \trianglelefteq^{\omega} \text{result}(C[N])$ and by the limit lemma, we have $\text{result}(C[M]) \trianglelefteq \text{result}(C[N])$. □

This establishes that our logical relation can prove graduality, so it only remains to show that our *inequational theory* implies our logical relation. Having already validated the congruence rules and reflexivity, we validate the remaining rules of transitivity, error, substitution, and $\beta\eta$ for each type constructor. Other than the $\mho \sqsubseteq M$ rule, all of these hold for any divergence preorder.

For transitivity, with the unary model and limiting lemmas in hand, we can prove that all of our logical relations (open and closed) are transitive in the limit. To do this, we first prove the following kind of "quantitative" transitivity lemma, and then transitivity in the limit is a consequence.

Lemma 7.12 (Logical Relation is Quantitatively Transitive)**.**

1. *If* $V_1 \trianglelefteq^{log}_{A,i} V_2$ *and* $V_2 \trianglelefteq^{log}_{A,\omega} V_3$*, then* $V_1 \trianglelefteq^{log}_{A,i} V_3$
2. *If* $S_1 \trianglelefteq^{log}_{\underline{B},i} S_2$ *and* $S_2 \trianglelefteq^{log}_{\underline{B},\omega} S_3$*, then* $S_1 \trianglelefteq^{log}_{\underline{B},i} S_3$

Lemma 7.13 (Logical Relation is Quantitatively Transitive (Open Terms))**.**

1. *If* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_2$ *and* $\gamma_2 \trianglelefteq^{log}_{\Gamma,\omega} \gamma_3$*, then* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_3$
2. *If* $\Gamma \vDash M_1 \trianglelefteq^{log}_{i} M_2 \in \underline{B}$ *and* $\Gamma \vDash M_2 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$*, then* $\Gamma \vDash M_1 \trianglelefteq^{log}_{i} M_3 \in \underline{B}$.
3. *If* $\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_2 \in A$ *and* $\Gamma \vDash V_2 \trianglelefteq^{log}_{\omega} V_3 \in A$*, then* $\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_3 \in A$.
4. *If* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{i} S_2 \in \underline{B}'$ *and* $\Gamma \mid \bullet : \underline{B} \vDash S_2 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$*, then* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{i} S_3 \in \underline{B}'$.

Corollary 7.7 (Logical Relation is Transitive in the Limit)**.**

1. *If* $\Gamma \vDash M_1 \trianglelefteq^{log}_{\omega} M_2 \in \underline{B}$ *and* $\Gamma \vDash M_2 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$*, then* $\Gamma \vDash M_1 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$.
2. *If* $\Gamma \vDash V_1 \trianglelefteq^{log}_{\omega} V_2 \in A$ *and* $\Gamma \vDash V_2 \trianglelefteq^{log}_{\omega} V_3 \in A$*, then* $\Gamma \vDash V_1 \trianglelefteq^{log}_{\omega} V_3 \in A$.
3. *If* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{\omega} S_2 \in \underline{B}'$ *and* $\Gamma \mid \bullet : \underline{B} \vDash S_2 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$*, then* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$.

Next, we verify the β, η equivalences hold as orderings each way.

Lemma 7.14 (β, η)**.** *For any divergence preorder, the* β, η *laws are valid for* $\trianglelefteq^{log}_{\omega}$

And that the logical relation behaves well is closed under substitution of related terms.

Lemma 7.15 (Substitution Principles)**.** *For any divergence preorder* $\trianglelefteq$*, the following are valid*

1. $$\frac{\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_2 \in A \qquad \Gamma, x : A \vDash V_1' \trianglelefteq^{log}_{i} V_2' \in A'}{\Gamma \vDash V_1'[V_1/x] \trianglelefteq^{log}_{i} V_2'[V_2/x] \in A'}$$

2. $$\frac{\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_2 \in A \qquad \Gamma, x : A \vDash M_1 \trianglelefteq^{log}_{i} M_2 \in \underline{B}}{\Gamma \vDash M_1[V_1/x] \trianglelefteq^{log}_{i} M_2[V_2/x] \in \underline{B}}$$

For errors, the strictness axioms hold for any $\trianglelefteq$, but the axiom that $\mho$ is a least element is specific to the definitions of $\preceq\sqsubseteq, \sqsubseteq\succeq$

Lemma 7.16 (Error Rules)**.** *For any divergence preorder* $\trianglelefteq$ *and appropriately typed* S, M*,*

$$S[\mho] \trianglelefteq^{log}_{\omega} \mho \qquad \mho \trianglelefteq^{log}_{\omega} S[\mho] \qquad \mho \preceq\sqsubseteq^{log}_{\omega} M \qquad M \preceq\sqsupseteq^{log}_{\omega} \mho$$

The lemmas we have proved cover all of the inequality rules of CBPV, so applying them with $\trianglelefteq$ chosen to be $\preceq\sqsubseteq$ and $\preceq\sqsupseteq$ gives

Lemma 7.17 ($\preceq\sqsubseteq$ and $\sqsubseteq\succeq$ are Models of CBPV)**.** *If* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : \underline{B}$ *then* $\Gamma \mid \Delta \vDash E \preceq\sqsubseteq^{\omega} E' \in \underline{B}$ *and* $\Gamma \mid \Delta \vDash E' \preceq\sqsupseteq^{\omega} E \in \underline{B}$.

Because logical implies contextual equivalence, we can conclude with the main theorem:

Theorem 7.2 (Contextual Approximation/Equivalence Model CBPV)**.** *If* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ *then* $\Gamma \mid \Delta \vDash E \sqsubseteq^{ctx} E' \in T$.
If $\Gamma \mid \Delta \vdash E \sqsupseteq\sqsubseteq E' : T$ *then* $\Gamma \mid \Delta \vDash E =^{ctx} E' \in T$.

8 Discussion and related work

In this paper, we have given a logic for reasoning about gradual programs in a mixed call-by-value/call-by-name language, shown that the axioms uniquely determine almost all of the contract translation implementing runtime casts, and shown that the axiomatics is sound for contextual equivalence/approximation in an operational model.

In immediate future work, we believe it is straightforward to add inductive/coinductive types and obtain similar unique cast implementation theorems. For instance, casting a list's element type should necessarily be equivalent to mapping the corresponding cast over the list:

$$\langle \mathtt{list}(A') \leftarrowtail \mathtt{list}(A) \rangle \sqsupseteq\sqsubseteq \mathtt{map}\langle A' \leftarrowtail A \rangle$$

In particular, the equations for inductive/recursive types should rule out "shallow" cast semantics that for example for lists only checks if the value is a list, but not immediately what its elements are.

Additionally, since more efficient cast implementations such as optimized cast calculi (the lazy variant in Herman *et al.*, 2010) and threesome casts (Siek & Wadler, 2010), are equivalent to the lazy contract semantics, they should also be models of GTT, and if so we could use GTT to reason about program transformations and optimizations in them.

Optimizations. In this paper, we have created an inequational theory for reasoning about gradually typed programs, with the primary purpose being to prove the graduality theorem and our uniqueness principles. Since order equivalence in our theory is sound for contextual equivalence, this system in principle could be used to justify optimizations of gradually typed programs or even optimized implementations of languages. While a full study of optimization is out of scope, we point out how thunkability of upcasts and linearity of downcasts is relevant to many optimizations. Thunkability is a very useful property for an optimizing compiler: it might more commonly be called "purity" in optimization literature. It means upcasts can be moved or duplicated freely: hoisted out of or lowered into a loop or closure for instance. Similarly, linearity is very useful for a compiler for a lazy language: it might more commonly be called "strictness" in lazy optimization literature. When a function is known to be strict in an argument, it can be optimized to instead take that argument by value, avoiding a costly thunk allocation at each call-site. Both of these properties (thunkability/purity and linearity/strictness) are useful for compilers to know and require significant program analyses to detect. A compiler for a gradual language should be able to augment these program analyses with this useful information about upcasts and downcasts.

Applicability of Cast Uniqueness Principles. The cast uniqueness principles given in Theorem 3.5 are theorems in the formal logic of Gradual Type Theory, and so there is a question of to what languages the theorem applies. The theorem applies to any *model* of gradual type theory, such as the models we have constructed using call-by-push-value given in Sections 5, 6, 7. We conjecture that simple call-by-value and call-by-name gradual languages are also models of GTT, by extending the translation of call-by-push-value into call-by-value and call-by-name in the appendix of Levy's monograph (Levy, 2003). In order for the theorem to apply, the language must validate an appropriate version of the η principles for the types. So, for example, a call-by-value language that has reference equality of functions does *not* validate even the value-restricted η law for functions, and so the case for functions does not apply. It is a well-known issue that in the presence of pointer equality of functions, the lazy semantics of function casts is not compatible with the graduality property, and our uniqueness theorem provides a different perspective on this phenomenon (Findler *et al.*, 2004; Strickland *et al.*, 2012; Siek *et al.*, 2015). However, we note that the cases of the uniqueness theorem for each type connective are completely *modular*: they rely only on the specification of casts and the β, η principles for the particular connective, and not on the presence of any other types, even the dynamic types. So even if a call-by-value language may have reference equality functions, if it has the η principle for strict pairs, then the pair cast must be that of Theorem 3.5.

Next, we consider the applicability to non-eager languages. Analogous to call-by-value, our uniqueness principle should apply to simple *call-by-name* gradual languages, where full η equality for functions is satisfied, but η equality for Booleans and strict pairs requires a "stack restriction" dual to the value restriction for call-by-value function η. We are not aware of any call-by-name gradual languages, but there is considerable work on *contracts* for non-eager languages, especially Haskell (Hinze *et al.*, 2006; Xu *et al.*, 2009). However, we note that Haskell is *not* a call-by-name language in our sense for two reasons. First, Haskell uses call-by-need evaluation where results of computations are memoized. However, when only considering Haskell's effects (error and divergence), this difference is not observable so this is not the main obstacle. The bigger difference between Haskell and call-by-name is that Haskell supports a `seq` operation that enables the programmer to force evaluation of a term to a value. This means Haskell violates the function η principle because Ω will cause divergence under `seq`, whereas $\lambda x.\Omega$ will not. This is a crucial feature of Haskell and is a major source of differences between implementations of lazy contracts, as noted in Degen *et al.* (2012). We can understand this difference by using a different translation into call-by-push-value: what Levy calls the "lazy paradigm", as opposed to call-by-name (Levy, 2003). Simply put, connectives are interpreted as in call-by-value, but with the addition of extra thunks UF, so for instance, the lazy function type $A \to B$ is interpreted as $UFU(UFA \to FB)$ and the extra UFU here is what causes the failure of the call-by-name η principle. With this embedding and the uniqueness theorem, GTT produces a definition for lazy casts, and the definition matches the work of Xu *et al.* (2009) when restricting to nondependent contracts.

Comparing Soundness Principles for Cast Semantics. Greenman & Felleisen (2018) gives a spectrum of differing syntactic type soundness theorems for different semantics of gradual typing. Our work here is complementary, showing that certain program equivalences can only be achieved by certain cast semantics.

Degen *et al.* (2012) give an analysis of different cast semantics for contracts in lazy languages, specifically based on Haskell, i.e., call-by-need with `seq`. They propose two properties "meaning preservation" and "completeness" that they show are incompatible and identify which contract semantics for a lazy language satisfy which of the properties. The meaning preservation property is closely related to graduality: it says that evaluating a term with a contract either produces blame or has the same observable effect as running the term without the contract. Meaning preservation rules out overly strict contract systems that force (possibly diverging) thunks that wouldn't be forced in a non-contracted term. Completeness, on the other hand, requires that when a contract is attached to a value that it is *deeply* checked. The two properties are incompatible because, for instance, a pair of a diverging term and a value can't be deeply checked without causing the entire program to diverge. Using Levy's embedding of the lazy paradigm into call-by-push-value their incompatibility theorem should be a consequence of our main theorem in the following sense. We showed that any contract semantics departing from the implementation in Theorem 3.5 must violate η or graduality. Their completeness property is inherently eager, and so must be different from the semantics GTT would provide, so either the restricted η or graduality fails. However, since they are defining contracts within the language, they satisfy the restricted η principle provided by the language, and so it must be graduality, and therefore meaning preservation that fails.

Axiomatic Casts. Henglein's work on dynamic typing also uses an axiomatic semantics of casts, but axiomatizes behavior of casts at each type directly, whereas we give a uniform definition of all casts and derive implementations for each type (Henglein, 1994). Because of this, the theorems proven in that paper are more closely related to our model construction in Section 5. More specifically, many of the properties of casts needed to prove Theorem 5.8 have direct analogues in Henglein's work, such as the coherence theorems. Finally, we note that our assumption of compositionality, i.e., that all casts can be decomposed into an upcast followed by a downcast, is based on Henglein's analysis, where it was proven to hold in his coercion calculus.

Gradual Typing Frameworks. In this work, we have applied a method of "gradualizing" axiomatic type theories by adding in precision orderings and adding dynamic types, casts and errors by axioms related to the precision orderings. This is similar in spirit to two recent frameworks for designing gradual languages: Abstracting Gradual Typing (AGT) (Garcia *et al.*, 2016) and the Gradualizer (Cimini & Siek, 2016, 2017). All of these approaches start with a typed language and construct a related gradual language. A major difference between our approach and those is that our work is based on axiomatic semantics and so we take into account the equality principles of the typed language, whereas Gradualizer is based on the typing and operational semantics and AGT is based on the type safety proof of the typed language. Furthermore, our approach produces not just a single language, but also an axiomatization of the structure of gradual typing and so we can prove results about many languages by proving theorems in GTT. The downside to this is that our approach doesn't directly provide an operational semantics for the gradual language, whereas for AGT this is a semi-mechanical process and for Gradualizer, completely automated. Finally, we note that neither system always agrees with the design that provides the desired η principles. AGT produces the "eager" semantics for function types, while the Gradualizer produces the "lazy" semantics for call-by-value products. It isn't clear how to modify either system to change the semantics.

Blame. We do not give a treatment of runtime blame reporting, but we argue that the observation that upcasts are thunkable and downcasts are linear is directly related to blame soundness (Tobin-Hochstadt & Felleisen, 2006; Wadler & Findler, 2009) in that if an upcast were *not* thunkable, it should raise positive blame and if a downcast were *not* linear, it should raise negative blame. First, consider a potentially effectful stack upcast of the form $\langle \underline{F}A' \nwarrow \underline{F}A \rangle$. If it is not thunkable, then in our logical relation this would mean there is a value $V : A$ such that $\langle \underline{F}A' \nwarrow \underline{F}A\rangle(\texttt{ret}\,V)$ performs some effect. Since the only observable effects for casts are dynamic type errors, $\langle \underline{F}A' \nwarrow \underline{F}A\rangle(\texttt{ret}\,V) \mapsto \mho$, and we must decide whether the positive party or negative party is at fault. However, since this is call-by-value evaluation, this error happens unconditionally on the continuation, so the continuation never had a chance to behave in such a way as to prevent blame, and so we must blame the positive party. Dually, consider a value downcast of the form $\langle U\underline{B} \swarrow U\underline{B}'\rangle$. If it is not linear, that would mean it forces its $U\underline{B}'$ input either never or more than once. Since downcasts should refine their inputs, it is not possible for the downcast to use the argument twice, since, e.g., printing twice does not refine printing once. So if the cast is not linear, that means it fails without ever forcing its input, in which case it knows nothing about the positive party and so must blame the negative party. In future work, we plan to investigate extensions of GTT with more than one $\mho$ with different blame labels, and an axiomatic account of a blame-aware observational equivalence.

Denotational and Category-Theoretic Models. We have presented certain concrete models of GTT using ordered CBPV with errors, in order to efficiently arrive at a concrete operational interpretation. It may be of interest to develop a more general notion of model of GTT for which we can prove soundness and completeness theorems, as in New & Licata (2018). A model would be a strong adjunction between double categories where one of the double categories has all "companions" and the other has all "conjoints", corresponding to our upcasts and downcasts. Then the contract translation should be a construction that takes a strong adjunction between two categories and makes a strong adjunction between double categories where the ep pairs are "Kleisli" ep pairs: the upcast is has a right adjoint, but only in the Kleisli category and vice versa the downcast has a left adjoint in the co-Kleisli category.

Furthermore, the ordered CBPV with errors should also have a sound and complete notion of model, and so our contract translation should have a semantic analogue as well.

Gradual Session Types. Gradual session types (Igarashi *et al.*, 2017) share some similarities to GTT, in that there are two sorts of types (values and sessions) with a dynamic value type and a dynamic session type. However, their language is not *polarized* in the same way as CBPV, so there is not likely an analogue between our upcasts always being between value types and downcasts always being between computation types. Instead, we might reconstruct this in a polarized session type language (Pfenning & Griffith, 2015). The two dynamic types would then be the "universal sender" and "universal receiver" session types.

Dynamically Typed Call-by-Push-Value. Our interpretation of the dynamic types in call-by-push-value suggests a design for a Scheme-like language with a value and computation distinction. This may be of interest for designing an extension of Typed Racket that efficiently supports CBN or a Scheme-like language with codata types. While the definition of the dynamic computation type by a lazy product may look strange, we argue that it is no stranger than the use of its dual, the sum type, in the definition of the dynamic value type. That is, in a truly dynamically typed language, we would not think of the dynamic type as being built out of some sum type construction, but rather that it is the *union* of all of the ground value types, and the union happens to be a *disjoint* union and so we can model it as a sum type. In the dual, we don't think of the computation dynamic type as a *product*, but instead as the *intersection* of the ground computation types. Thinking of the type as unfolding:

$$\underline{¿} = \underline{F}\underline{¿} \wedge (? \to \underline{F}?) \wedge (? \to ? \to \underline{F}?) \wedge \cdots$$

This says that a dynamically typed computation is one that can be invoked with any finite number of arguments on the stack, a fairly accurate model of implementations of Scheme that pass multiple arguments on the stack.

Dependent Contract Checking. We also plan to explore using GTT's specification of casts in a dependently typed setting, building on work using Galois connections for casts between dependent types (Dagand *et al.*, 2018; Eremondi *et al.*, 2019), and work on effectful dependent types based a CBPV-like judgement structure (Ahman *et al.*, 2016; Pédrot & Tabareau, 2020).

Acknowledgments

The authors would like to thank Ron Garcia, Kenji Maillard and Gabriel Scherer for helpful discussions about this work. We thank the anonymous POPL 2019 reviewers for helpful feedback on this article. This material is based on research sponsored by the National Science Foundation under grants CCF-1910522, CCF-1816837, CCF-1453796 and the United States Air Force Research Laboratory under agreement numbers FA9550-15-1-0053 and FA9550-16-1-0292. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

Conflicts of Interest

None.

References

Ahman, D., Ghani, N. & Plotkin, G. D. (2016) Dependent types and fibred computational effects. In Foundations of Software Science and Computation Structures, pp. 36–54.

Ahmed, A. (2006) Step-indexed syntactic logical relations for recursive and quantified types. In European Symposium on Programming (ESOP), pp. 69–83.

Bauer, A. & Pretnar, M. (2013) An effect system for algebraic effects and handlers. In *Algebra and Coalgebra in Computer Science*, pp. 1–16. Berlin, Heidelberg: Springer.

Boyland, J. (2014) The problem of structural type tests in a gradually typed language. In 21st Workshop on Foundations of Object-Oriented Languages.

Cimini, M. & Siek, J. G. (2016) The gradualizer: A methodology and algorithm for generating gradual type systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'16.

Cimini, M. & Siek, J. G. (2017) Automatically generating the dynamic semantics of gradually typed languages. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017, pp. 789–803.

Dagand, P.-È., Tabareau, N. & Tanter, È. (2018) Foundations of dependent interoperability. *J. Funct. Program.* **28**, e9.

Degen, M., Thiemann, P. & Wehr, S. (2012) The interaction of contracts and laziness. *Higher-Order Symb. Comput.* **25**, 85–125.

Eremondi, J., Tanter, È. & Garcia, R. (2019) Approximate normalization for dependent gradual types. In International Conference on Functional Programming (ICFP), Berlin, Germany.

Findler, R. B. & Felleisen, M. (2002) Contracts for higher-order functions. In International Conference on Functional Programming (ICFP), pp. 48–59.

Findler, R. B., Flatt, M. & Felleisen, M. (2004) Semantic casts: Contracts and structural subtyping in a nominal world. In European Conference on Object-Oriented Programming (ECOOP).

Führmann, C. (1999) Direct models of the computational lambda-calculus. *Electr. Notes Theor. Comput. Sci.* **20**, 245–292.

Garcia, R., Clark, A. M. & Tanter, É. (2016) Abstracting gradual typing. In ACM Symposium on Principles of Programming Languages (POPL).

Girard, J.-Y. (2001) Locus solum: From the rules of logic to the logic of rules. *Math. Struct. Comput. Sci.* **11**(3), 301–506.

Greenberg, M. (2015) Space-efficient manifest contracts. In ACM Symposium on Principles of Programming Languages (POPL), pp. 181–194.

Greenberg, M., Pierce, B. C. & Weirich, S. (2010) Contracts made manifest. In ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain.

Greenman, B. & Felleisen, M. (2018) A spectrum of type soundness and performance. In International Conference on Functional Programming (ICFP), St. Louis, Missouri.

Henglein, F. (1994) Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.* **22**(3), 197–230.

Herman, D., Tomb, A. & Flanagan, C. (2010) Space-efficient gradual typing. *Higher-Order Symb. Comput.* **23**, 167.

Hinze, R., Jeuring, J. & Löh, A. (2006) Typed contracts for functional programming. In International Symposium on Functional and Logic Programming (FLOPS).

Igarashi, A., Thiemann, P., Vasconcelos, V. T. & Wadler, P. (2017) Gradual session types. *Proc. ACM Program. Lang.* **1**(ICFP), 38:1–38:28.

Levy, P. B. (2003) *Call-by-Push-Value: A Functional/Imperative Synthesis*. Springer.

Levy, P. B. (2017) Contextual isomorphisms. In ACM Symposium on Principles of Programming Languages (POPL).

Lindenhovius, B., Mislove, M. & Zamdzhiev, V. (2019) Mixed linear and non-linear recursive types. *Proc. ACM Program. Lang.* **3**(ICFP) 11, 1–29.

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017. ACM, pp. 500–514.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.

Munch-Maccagnoni, G. (2014) Models of a non-associative composition. In Foundations of Software Science and Computation Structures, pp. 396–410.

Nakano, H. (2000) A modality for recursion. In IEEE Symposium on Logic in Computer Science (LICS), Santa Barbara, California.

New, M. S. & Ahmed, A. (2018) Graduality from embedding-projection pairs. In International Conference on Functional Programming (ICFP), St. Louis, Missouri.

New, M. S. & Licata, D. R. (2018) Call-by-name gradual type theory. In FSCD.

New, M. S. & Licata, D. R. (2020) Call-by-name gradual type theory. In LMCS.

New, M. S., Licata, D. R. & Ahmed, A. (2019) Gradual type theory. In ACM Symposium on Principles of Programming Languages (POPL), Cascais, Portugal.

New, M. S., Jamner, D. & Ahmed, A. (2020) Graduality and parametricity: Together again for the first time. In ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana.

Pédrot, P.-M. & Tabareau, N. (2020) The fire triangle: How to mix substitution, dependent elimination, and effects. In ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana.

Pfenning, F. & Griffith, D. (2015) Polarized substructural session types (invited talk). In International Conference on Foundations of Software Science and Computation Structures (FoSSaCS).

Pitts, A. & Stark, I. (1998) Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, Gordon, A. & Pitts, A. (eds), Publications of the Newton Institute, Cambridge University Press, pp. 227–273.

Siek, J., Garcia, R. & Taha, W. (2009) Exploring the design space of higher-order casts. In European Symposium on Programming (ESOP). Berlin, Heidelberg: Springer-Verlag, pp. 17–31.

Siek, J. & Tobin-Hochstadt, S. (2016) The recursive union of some gradual types. In *A List of Successes that can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, LNCS, Springer, vol. 9600.

Siek, J., Vitousek, M., Cimini, M. & Boyland, J. T. (2015) Refined criteria for gradual typing. In 1st Summit on Advances in Programming Languages. SNAPL 2015.

Siek, J. G. & Taha, W. (2006) Gradual typing for functional languages. In Scheme and Functional Programming Workshop (Scheme), pp. 81–92.

Siek, J. G. & Wadler, P. (2010) Threesomes, with and without blame. In ACM Symposium on Principles of Programming Languages (POPL). ACM, pp. 365–376.

Strickland, T. S., Tobin-Hochstadt, S., Findler, R. B. & Flatt, M. (2012) Chaperones and impersonators: Run-time support for reasonable interposition. In ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Tucson, Arizona.

Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J. & Felleisen, M. (2016) Is sound gradual typing dead? In ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida.

Tobin-Hochstadt, S. & Felleisen, M. (2006) Interlanguage migration: From scripts to programs. In Dynamic Languages Symposium (DLS), pp. 964–974.

Tobin-Hochstadt, S. & Felleisen, M. (2008) The design and implementation of typed scheme. In ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California.

Vitousek, M. M., Swords, C. & Siek, J. G. (2017) Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In ACM Symposium on Principles of Programming Languages (POPL), Paris, France.

Wadler, P. & Findler, R. B. (2009) Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pp. 1–16.

Xu, D. N., Peyton Jones, S. & Claessen, K. (2009) Static contract checking for haskell. In ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia.

Zeilberger, N. (2009) *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph.D. thesis, Carnegie Mellon University.

A Term precision congruence rules

The full congruence rules for GTT are found in Figure A.1. Note that we need not add congruence rules for $\mho$ or upcasts/downcasts since they are already derivable.

$$\textsc{+ILCong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A_1 \sqsubseteq A_1'}{\Phi \vdash \texttt{inl}\ V \sqsubseteq \texttt{inl}\ V' : A_1 + A_2 \sqsubseteq A_1' + A_2'}
\qquad
\textsc{+IRCong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A_2 \sqsubseteq A_2'}{\Phi \vdash \texttt{inr}\ V \sqsubseteq \texttt{inr}\ V' : A_1 + A_2 \sqsubseteq A_1' + A_2'}$$

$$\textsc{+ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : A_1 + A_2 \sqsubseteq A_1' + A_2' \\ \Phi, x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1' \mid \Psi \vdash E_1 \sqsubseteq E_1' : T \sqsubseteq T' \\ \Phi, x_2 \sqsubseteq x_2' : A_2 \sqsubseteq A_2' \mid \Psi \vdash E_2 \sqsubseteq E_2' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \texttt{case}\ V\{x_1.E_1 \mid x_2.E_2\} \sqsubseteq \texttt{case}\ V\{x_1'.E_1' \mid x_2'.E_2'\} : T'}
\qquad
\textsc{0ECong}\quad \frac{\Phi \vdash V \sqsubseteq V' : 0 \sqsubseteq 0}{\Phi \mid \Psi \vdash \texttt{abort}\ V \sqsubseteq \texttt{abort}\ V' : T \sqsubseteq T'}$$

$$\textsc{1ICong}\quad \frac{}{\Phi \vdash () \sqsubseteq () : 1 \sqsubseteq 1}
\qquad
\textsc{1ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : 1 \sqsubseteq 1 \\ \Phi \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \texttt{split}\ V\ \texttt{to}\ ().E \sqsubseteq \texttt{split}\ V\ \texttt{to}\ ().'E' : T \sqsubseteq T'}$$

$$\times\textsc{ICong}\quad \frac{\begin{array}{c}\Phi \vdash V_1 \sqsubseteq V_1' : A_1 \sqsubseteq A_1' \\ \Phi \vdash V_2 \sqsubseteq V_2' : A_2 \sqsubseteq A_2'\end{array}}{\Phi \vdash (V_1, V_2) \sqsubseteq (V_1', V_2') : A_1 \times A_2 \sqsubseteq A_1' \times A_2'}
\qquad
\to\textsc{ICong}\quad \frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \Psi \vdash \lambda x : A.M \sqsubseteq \lambda x' : A'.M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

$$\times\textsc{ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : A_1 \times A_2 \sqsubseteq A_1' \times A_2' \\ \Phi, x \sqsubseteq x' : A_1 \sqsubseteq A_1', y \sqsubseteq y' : A_2 \sqsubseteq A_2' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \texttt{split}\ V\ \texttt{to}\ (x, y).E \sqsubseteq \texttt{split}\ V'\ \texttt{to}\ (x', y').E' : T \sqsubseteq T'}$$

$$\to\textsc{ECong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \Psi \vdash M\ V \sqsubseteq M'\ V' : \underline{B} \sqsubseteq \underline{B}'}
\qquad
U\textsc{ICong}\quad \frac{\Phi \mid \cdot \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \vdash \texttt{thunk}\ M \sqsubseteq \texttt{thunk}\ M' : U\underline{B} \sqsubseteq U\underline{B}'}$$

$$U\textsc{ECong}\quad \frac{\Phi \vdash V \sqsubseteq V' : U\underline{B} \sqsubseteq U\underline{B}'}{\Phi \mid \cdot \vdash \texttt{force}\ V \sqsubseteq \texttt{force}\ V' : \underline{B} \sqsubseteq \underline{B}'}
\qquad
F\textsc{ICong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \cdot \vdash \texttt{ret} V \sqsubseteq \texttt{ret} V' : \underline{F A} \sqsubseteq \underline{F A'}}$$

$$F\textsc{ECong}\quad \frac{\begin{array}{c}\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{F A} \sqsubseteq \underline{F A'} \\ \Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \cdot \vdash N \sqsubseteq N' : \underline{B} \sqsubseteq \underline{B}'\end{array}}{\Phi \mid \Psi \vdash \texttt{bind}\ x \leftarrow M; N \sqsubseteq \texttt{bind}\ x' \leftarrow M'; N' : \underline{B} \sqsubseteq \underline{B}'}
\qquad
\top\textsc{ICong}\quad \frac{}{\Phi \mid \Psi \vdash \{\} \sqsubseteq \{\} : \top \sqsubseteq \top}$$

$$\&\textsc{ICong}\quad \frac{\Phi \mid \Psi \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \sqsubseteq \underline{B}_1' \qquad \Phi \mid \Psi \vdash M_2 \sqsubseteq M_2' : \underline{B}_2 \sqsubseteq \underline{B}_2'}{\Phi \mid \Psi \vdash \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} \sqsubseteq \{\pi \mapsto M_1' \mid \pi' \mapsto M_2'\} : \underline{B}_1 \mathbin{\&} \underline{B}_2 \sqsubseteq \underline{B}_1' \mathbin{\&} \underline{B}_2'}$$

$$\&\textsc{ECong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B}_1 \mathbin{\&} \underline{B}_2 \sqsubseteq \underline{B}_1' \mathbin{\&} \underline{B}_2'}{\Phi \mid \Psi \vdash \pi M \sqsubseteq \pi M' : \underline{B}_1 \sqsubseteq \underline{B}_1'}
\qquad
\&\textsc{E'Cong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B}_1 \mathbin{\&} \underline{B}_2 \sqsubseteq \underline{B}_1' \mathbin{\&} \underline{B}_2'}{\Phi \mid \Psi \vdash \pi' M \sqsubseteq \pi' M' : \underline{B}_2 \sqsubseteq \underline{B}_2'}$$

Fig. A.1. GTT term precision (congruence rules).

B Proofs for Section 3

Proof of Lemma 3.2. *Proof.* For upcast left, substitute V' into the axiom $x \sqsubseteq \langle A'' \nwarrow A' \rangle x : A' \sqsubseteq A''$ to get $V' \sqsubseteq \langle A'' \nwarrow A' \rangle V'$, and then use transitivity with the premise.

For upcast right, by transitivity of

$$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x' : A' \sqsubseteq A' \qquad x' \sqsubseteq x'' : A' \sqsubseteq A'' \vdash x' \sqsubseteq x'' : A' \sqsubseteq A''$$

we have

$$x \sqsubseteq x'' : A \sqsubseteq A'' \vdash \langle A' \nwarrow A\rangle x \sqsubseteq x'' : A' \sqsubseteq A''$$

Substituting the premise into this gives the conclusion.

For downcast left, substituting M' into the axiom $\langle \underline{B} \swarrow \underline{B}'\rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$ gives $\langle \underline{B} \swarrow \underline{B}'\rangle M \sqsubseteq M$, and then transitivity with the premise gives the result.

For downcast right, transitivity of

$$\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \qquad \bullet' \sqsubseteq \bullet'' : \underline{B}' \sqsubseteq \underline{B}'' \vdash \bullet' \sqsubseteq \langle \underline{B}' \swarrow \underline{B}''\rangle \bullet''$$

gives $\bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}'' \vdash \bullet \sqsubseteq \langle \underline{B}' \swarrow \underline{B}''\rangle \bullet''$, and then substitution of the premise into this gives the conclusion. □

Proof of Theorem 3.2. *Proof.* We use Theorem 3.1 in all cases, and show that the right-hand side has the universal property of the left.

1. Both parts expand to showing $x \sqsubseteq x : A \sqsubseteq A \vdash x \sqsubseteq x : A \sqsubseteq A$, which is true by assumption.
2. First, we need to show $x \sqsubseteq \langle A'' \nwarrow A'\rangle(\langle A' \nwarrow A\rangle x) : A \sqsubseteq A''$. By upcast right, it suffices to show $x \sqsubseteq \langle A' \nwarrow A\rangle x : A \sqsubseteq A'$, which is also true by upcast right.
   For $x \sqsubseteq x'' : A \sqsubseteq A'' \vdash \langle A'' \nwarrow A'\rangle(\langle A' \nwarrow A\rangle x) \sqsubseteq x''$, by upcast left twice, it suffices to show $x \sqsubseteq x'' : A \sqsubseteq A''$, which is true by assumption.
3. Both parts expand to showing $\bullet : \underline{B} \vdash \bullet \sqsubseteq \bullet : \underline{B}$, which is true by assumption.
4. To show $\bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}'' \vdash \bullet \sqsubseteq \langle \underline{B} \swarrow \underline{B}'\rangle(\langle \underline{B}' \swarrow \underline{B}''\rangle \bullet)$, by downcast right (twice), it suffices to show $\bullet : \underline{B} \sqsubseteq \bullet'' : \underline{B}'' \vdash \bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}''$, which is true by assumption. Next, we have to show $\langle \underline{B} \swarrow \underline{B}'\rangle(\langle \underline{B}' \swarrow \underline{B}''\rangle \bullet) \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}''$, and by downcast left, it suffices to show $\langle \underline{B}' \swarrow \underline{B}''\rangle \bullet \sqsubseteq \bullet : \underline{B}' \sqsubseteq \underline{B}''$, which is also true by downcast left. □

Proof of Theorem 3.3. *Proof.*

1. By η for F types, $\bullet' : \underline{F}A' \vdash \bullet' \sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet'; \texttt{ret}x' : \underline{F}A'$, so it suffices to show
   $$\texttt{bind}\ x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle \bullet'; \texttt{ret}(\langle A' \nwarrow A\rangle x) \sqsubseteq \texttt{bind}\ x' : A' \leftarrow \bullet'; \texttt{ret}x'$$
   By congruence, it suffices to show $\langle \underline{F}A \swarrow \underline{F}A'\rangle \bullet' \sqsubseteq \bullet' : \underline{F}A \sqsubseteq \underline{F}A'$, which is true by downcast left, and $x \sqsubseteq x' : A \sqsubseteq A' \vdash \texttt{ret}(\langle A' \nwarrow A\rangle x) \sqsubseteq \texttt{ret}x' : A'$, which is true by congruence for ret, upcast left, and the assumption.
2. By η for F types, it suffices to show
   $$\bullet : \underline{F}A \vdash \texttt{bind}\ \bullet \leftarrow x; \texttt{ret}x \sqsubseteq \texttt{bind}\ x \leftarrow \bullet; \langle \underline{F}A \swarrow \underline{F}A'\rangle(\texttt{ret}(\langle A' \nwarrow A\rangle x)) : \underline{F}A$$
   so by congruence,
   $$x : A \vdash \texttt{ret}x \sqsubseteq \langle \underline{F}A \swarrow \underline{F}A'\rangle(\texttt{ret}(\langle A' \nwarrow A\rangle x))$$
   By downcast right, it suffices to show
   $$x : A \vdash \texttt{ret}x \sqsubseteq (\texttt{ret}(\langle A' \nwarrow A\rangle x)) : \underline{F}A \sqsubseteq \underline{F}A'$$

and by congruence

$$x : A \vdash x \sqsubseteq ((\langle A' \leftarrowtail A\rangle x)) : A \sqsubseteq A'$$

which is true by upcast right.

3. By η for U types, it suffices to show

   $$x : U\underline{B}' \vdash \langle U\underline{B}' \leftarrowtail U\underline{B}\rangle(\texttt{thunk }(\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\texttt{force } x)) \sqsubseteq \texttt{thunk }(\texttt{force } x) : U\underline{B}'$$

   By upcast left, it suffices to show

   $$x : U\underline{B}' \vdash (\texttt{thunk }(\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\texttt{force } x)) \sqsubseteq \texttt{thunk }(\texttt{force } x) : U\underline{B} \sqsubseteq U\underline{B}'$$

   and by congruence

   $$x : U\underline{B}' \vdash \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\texttt{force } x \sqsubseteq \texttt{force } x : \underline{B} \sqsubseteq \underline{B}'$$

   which is true by downcast left.

4. By η for U types, it suffices to show

   $$x : U\underline{B} \vdash \texttt{thunk }(\texttt{force } x) \sqsubseteq \texttt{thunk }(\langle B \twoheadleftarrow B'\rangle(\texttt{force }(\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle x))) : U\underline{B}$$

   and by congruence

   $$x : U\underline{B} \vdash (\texttt{force } x) \sqsubseteq (\langle B \twoheadleftarrow B'\rangle(\texttt{force }(\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle x))) : \underline{B}$$

   By downcast right, it suffices to show

   $$x : U\underline{B} \vdash (\texttt{force } x) \sqsubseteq (\texttt{force }(\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle x)) : \underline{B} \sqsubseteq \underline{B}'$$

   and by congruence

   $$x : U\underline{B} \vdash x \sqsubseteq (\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle x) : \underline{B} \sqsubseteq \underline{B}'$$

   which is true by upcast right. □

Proof of Theorem 3.4. *Proof.* We need only to show the $\sqsubseteq$ direction, because the converse is Theorem 3.3.

1. Substituting $\texttt{ret}(\langle A' \leftarrowtail A\rangle x)$ into Theorem 3.3's

   $$\bullet : \underline{F}A \vdash \bullet \sqsubseteq \texttt{bind } x \leftarrow \bullet; \langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle(\texttt{ret}(\langle A' \leftarrowtail A\rangle x)) : \underline{F}A$$

   and β-reducing gives

   $$x : A \vdash \texttt{ret}(\langle A' \leftarrowtail A\rangle x) \sqsubseteq \langle \underline{F}A \twoheadleftarrow \underline{F}?\rangle(\texttt{ret}(\langle ? \leftarrowtail A'\rangle\langle A' \leftarrowtail A\rangle x))$$

   Using this, after η-expanding $\bullet : \underline{F}A$ on the right and using congruence for bind, it suffices to derive as follows:

   $$\begin{array}{lll}
   \langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle(\texttt{ret}(\langle A' \leftarrowtail A\rangle x)) & \sqsubseteq & \text{congruence} \\
   \langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\langle \underline{F}A' \twoheadleftarrow \underline{F}?\rangle(\texttt{ret}(\langle ? \leftarrowtail A'\rangle\langle A' \leftarrowtail A\rangle x)) & \sqsubseteq & \text{composition} \\
   \langle \underline{F}A \twoheadleftarrow \underline{F}?\rangle(\texttt{ret}(\langle ? \leftarrowtail A\rangle x)) & \sqsubseteq & \text{retract axiom for } \langle ? \leftarrowtail A\rangle \\
   \texttt{ret}x & &
   \end{array}$$

2. After using η for U and congruence, it suffices to show

   $$x : U\underline{B} \vdash \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle(\texttt{force }(\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle x)) \sqsubseteq \texttt{force } x : \underline{B}$$

Substituting $x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B}\rangle x : U\underline{B}'$ into Theorem 3.3's

$$x : U\underline{B}' \vdash x \sqsubseteq \texttt{thunk}\ (\langle B' \twoheadleftarrow \underline{\text{¿}}\rangle(\texttt{force}\ (\langle U\underline{\text{¿}} \nwarrow U\underline{B}'\rangle x))) : U\underline{B}'$$

gives

$x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B}\rangle x \sqsubseteq \texttt{thunk}\ (\langle B' \twoheadleftarrow \underline{\text{¿}}\rangle(\texttt{force}\ (\langle U\underline{\text{¿}} \nwarrow U\underline{B}'\rangle\langle U\underline{B}' \nwarrow U\underline{B}\rangle x))) :$
$U\underline{B}'$

So we have

$$\begin{array}{lll}
\langle B \twoheadleftarrow B'\rangle(\texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle x) & \sqsubseteq & \\
\langle B \twoheadleftarrow B'\rangle\texttt{force}\ (\texttt{thunk}\ (\langle B' \twoheadleftarrow \underline{\text{¿}}\rangle(\texttt{force}\ (\langle U\underline{\text{¿}} \nwarrow U\underline{B}'\rangle\langle U\underline{B}' \nwarrow U\underline{B}\rangle x)))) & \sqsubseteq & \beta \\
\langle B \twoheadleftarrow B'\rangle(\langle B' \twoheadleftarrow \underline{\text{¿}}\rangle(\texttt{force}\ (\langle U\underline{\text{¿}} \nwarrow U\underline{B}'\rangle\langle U\underline{B}' \nwarrow U\underline{B}\rangle x))) & \sqsubseteq & \text{composition} \\
\langle B \twoheadleftarrow \underline{\text{¿}}\rangle(\texttt{force}\ (\langle U\underline{\text{¿}} \nwarrow U\underline{B}\rangle x)) & \sqsubseteq & \text{retract axiom for } \langle \underline{B} \twoheadleftarrow \underline{\text{¿}}\rangle \\
\texttt{ret}x & \sqsubseteq & \text{composition}
\end{array}$$

□

Proof of Theorem 3.5. *Proof.*

1. Sums upcast. We use Lemma 3.5 with the type constructor X_1 val type, X_2 val type $\vdash X_1 + X_2$ val type. Suppose $A_1 \sqsubseteq A_1'$ and $A_2 \sqsubseteq A_2'$ and let

$$s : A_1 + A_2 \vdash \langle\!\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle\!\rangle s : A_1' + A_2'$$

stand for

$$\texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1' \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2' \nwarrow A_2\rangle x_2)\}$$

This clearly satisfies the typing requirement and monotonicity.
Finally, for identity extension, we need to show

$$\texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1 \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2 \nwarrow A_2\rangle x_2)\} \sqsupseteq\sqsubseteq s$$

which is true because $\langle A_1 \nwarrow A_1\rangle$ and $\langle A_2 \nwarrow A_2\rangle$ are the identity, and using "weak η" for sums, $\texttt{case}\ s\{x_1.\texttt{inl}\ x_1 \mid x_2.\texttt{inr}\ x_2\} \sqsupseteq\sqsubseteq x$, which is the special case of the η rule in Figure 6 for the identity complex value:

$$\begin{array}{rl}
\texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1 \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2 \nwarrow A_2\rangle x_2)\} & \sqsupseteq\sqsubseteq \\
\texttt{case}\ s\{x_1.\texttt{inl}\ (x_1) \mid x_2.\texttt{inr}\ (x_2)\} & \sqsupseteq\sqsubseteq \\
s &
\end{array}$$

2. Sums downcast. We use the downcast lemma with X_1 val type, X_2 val type $\vdash \underline{F}(X_1 + X_2)$ comp type. Let

$$\bullet' : \underline{F}(A_1' + A_2') \vdash \langle\!\langle \underline{F}(A_1 + A_2) \twoheadleftarrow \underline{F}(A_1' + A_2')\rangle\!\rangle \bullet' : \underline{F}(A_1 + \underline{A}_2)$$

stand for

$$\begin{array}{l}
\texttt{bind}\ (s : (A_1' + A_2')) \leftarrow \bullet; \\
\quad \texttt{case}\ s\{x_1'.\texttt{bind}\ x_1 \leftarrow (\langle \underline{F}A_1 \twoheadleftarrow \underline{F}A_1'\rangle(\texttt{ret}x_1')); \texttt{ret}(\texttt{inl}\ x_1) \mid \ldots\}
\end{array}$$

(where, as in the theorem statement, inr branch is analogous). This clearly satisfies typing and monotonicity.

Finally, for identity extension, we show

$$\begin{array}{ll}
\texttt{bind}\ (s:(A_1+A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.\texttt{bind}\ x_1 \leftarrow ((\langle \underline{F}A_1 \swarrow \underline{F}A_1\rangle(\texttt{ret}x_1)); \texttt{ret}(\texttt{inl}\ x_1) \mid \ldots\} & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ (s:(A_1+A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.\texttt{bind}\ x_1 \leftarrow ((\texttt{ret}x_1)); \texttt{ret}(\texttt{inl}\ x_1) \mid \ldots\} & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ (s:(A_1+A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.\texttt{ret}(\texttt{inl}\ x_1) \mid x_2.\texttt{ret}(\texttt{inr}\ x_2)\} & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ (s:(A_1+A_2)) \leftarrow \bullet; \texttt{ret}s & \sqsupseteq\sqsubseteq \\
\bullet &
\end{array}$$

using the downcast identity, β for $\underline{F}$ types, η for sums, and η for $\underline{F}$ types.

3. Eager product upcast. We use Lemma 3.5 with the type constructor X_1 val type, X_2 val type $\vdash X_1 \times X_2$ val type. Let

$$p : A_1 \times A_2 \vdash \langle\!\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle\!\rangle s : A_1' \times A_2'$$

stand for

$$\texttt{split}\ p\,\texttt{to}\ (x_1, x_2).(\langle A_1' \nwarrow A_1\rangle x_1, \langle A_2' \nwarrow A_2\rangle x_2)$$

which clearly satisfies the typing requirement and monotonicity.
Finally, for identity extension, using η for products and the fact that $\langle A \nwarrow A\rangle$ is the identity, we have

$$\texttt{split}\ p\,\texttt{to}\ (x_1, x_2).(\langle A_1 \nwarrow A_1\rangle x_1, \langle A_2 \nwarrow A_2\rangle x_2) \sqsupseteq\sqsubseteq \texttt{split}\ p\,\texttt{to}\ (x_1, x_2).(x_1, x_2) \sqsupseteq\sqsubseteq p$$

4. Eager product downcast.
We use the downcast lemma with X_1 val type, X_2 val type $\vdash \underline{F}(X_1 \times X_2)$ comp type. Let

$$\bullet' : \underline{F}(A_1' \times A_2') \vdash \langle\!\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2') \rangle\!\rangle \bullet' : \underline{F}(A_1 \times \underline{A}_2)$$

stand for

$$\texttt{bind}\ p' \leftarrow \bullet; \texttt{split}\ p'\ \texttt{to}\ (x_1', x_2').\texttt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \texttt{ret}x_1';$$

$$\texttt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \texttt{ret}x_2'; \texttt{ret}(x_1, x_2)$$

which clearly satisfies the typing requirement and monotonicity.
Finally, for identity extension, we show

$$\begin{array}{ll}
\texttt{bind}\ p \leftarrow \bullet; \texttt{split}\ p\,\texttt{to}\ (x_1, x_2).\texttt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1\rangle \texttt{ret}x_1; & \\
\texttt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \texttt{ret}x_2; \texttt{ret}(x_1, x_2) & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ p \leftarrow \bullet; \texttt{split}\ p\,\texttt{to}\ (x_1, x_2).\texttt{bind}\ x_1 \leftarrow \texttt{ret}x_1; \texttt{bind}\ x_2 \leftarrow \texttt{ret}x_2; \texttt{ret}(x_1, x_2) & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ p \leftarrow \bullet; \texttt{split}\ p\,\texttt{to}\ (x_1, x_2).\texttt{ret}(x_1, x_2) & \sqsupseteq\sqsubseteq \\
\texttt{bind}\ p \leftarrow \bullet; \texttt{ret}p & \sqsupseteq\sqsubseteq \\
\bullet &
\end{array}$$

using the downcast identity, β for $\underline{F}$ types, η for eager products, and η for $\underline{F}$ types.
An analogous argument works if we sequence the downcasts of the components in the opposite order:

$$\texttt{bind}\ p' \leftarrow \bullet; \texttt{split}\ p'\ \texttt{to}\ (x_1', x_2').\texttt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \texttt{ret}x_2';$$

$$\texttt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \texttt{ret}x_1'; \texttt{ret}(x_1, x_2)$$

(the only facts about downcasts used above are congruence and the downcast identity), which shows that these two implementations of the downcast are themselves equiprecise.

5. Lazy product downcast. We use Lemma 3.6 with the type constructor $\underline{Y}_1$ comp type, $\underline{Y}_2$ comp type $\vdash \underline{Y}_1 \mathbin{\&} \underline{Y}_2$ val type. Let

$$\bullet' : \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \vdash \langle\!\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1 \mathbin{\&} \underline{B}_2 \rangle\!\rangle \bullet' : \underline{B}_1 \mathbin{\&} \underline{B}_2$$

stand for

$$\{\pi \mapsto \langle \underline{B}_1 \swarrow \underline{B}'_1 \rangle \pi \bullet' \mid \pi' \mapsto \langle \underline{B}_2 \swarrow \underline{B}'_2 \rangle \pi' \bullet' \}$$

which clearly satisfies the typing requirement and monotonicity.
For identity extension, we have, using $\langle \underline{B} \swarrow \underline{B} \rangle$ is the identity and η for &,

$$\{\pi \mapsto \langle \underline{B}_1 \swarrow \underline{B}_1 \rangle \pi \bullet \mid \pi' \mapsto \langle \underline{B}_2 \swarrow \underline{B}_2 \rangle \pi' \bullet \} \sqsupseteq\sqsubseteq \{\pi \mapsto \pi \bullet \mid \pi' \mapsto \pi' \bullet\} \sqsupseteq\sqsubseteq \bullet$$

6. Lazy product upcast.
We use Lemma 3.5 with the type constructor $\underline{Y}_1$ comp type, $\underline{Y}_2$ comp type $\vdash U(\underline{Y}_1 \mathbin{\&} \underline{Y}_2)$ val type. Let

$$p : U(\underline{B}_1 \mathbin{\&} \underline{B}_2) \vdash \langle\!\langle U(\underline{B}_1 \mathbin{\&} \underline{B}_2) \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2) \rangle\!\rangle p : U(\underline{B}'_1 \mathbin{\&} \underline{B}'_2)$$

stand for

$$\texttt{thunk}\ \{\pi \mapsto \texttt{force}\ (\langle U\underline{B}'_1 \nwarrow U\underline{B}_1 \rangle (\texttt{thunk}\ \pi(\texttt{force}\ p))) \mid \pi' \mapsto \texttt{force}\ (\langle U\underline{B}'_2 \nwarrow U\underline{B}_2 \rangle\}$$

$$(\texttt{thunk}\ \pi'(\texttt{force}\ p)))$$

which clearly satisfies the typing requirement and monotonicity.
Finally, for identity extension, using η for *times*, β and η for U types, and the fact that $\langle A \nwarrow A \rangle$ is the identity, we have

$$\begin{array}{rl}
\texttt{thunk}\ \{\pi \mapsto \texttt{force}\ (\langle U\underline{B}_1 \nwarrow U\underline{B}_1 \rangle (\texttt{thunk}\ \pi(\texttt{force}\ p))) \mid \pi' \mapsto \texttt{force}\ (\langle U\underline{B}_2 \nwarrow U\underline{B}_2 \rangle\} & \\
(\texttt{thunk}\ \pi'(\texttt{force}\ p))) & \sqsupseteq\sqsubseteq \\
\texttt{thunk}\ \{\pi \mapsto \texttt{force}\ (\texttt{thunk}\ \pi(\texttt{force}\ p)) \mid \pi' \mapsto \texttt{force}\ (\texttt{thunk}\ \pi'(\texttt{force}\ p))\} & \sqsupseteq\sqsubseteq \\
\texttt{thunk}\ \{\pi \mapsto \pi(\texttt{force}\ p) \mid \pi' \mapsto \pi'(\texttt{force}\ p)\} & \sqsupseteq\sqsubseteq \\
\texttt{thunk}\ (\texttt{force}\ p) & \sqsupseteq\sqsubseteq \\
p &
\end{array}$$

7. Function downcast.
We use Lemma 3.6 with the type constructor X val type, $\underline{Y}$ comp type $\vdash X \to \underline{Y}$ comp type. Let

$$\bullet' : A' \to \underline{B}' \vdash \langle\!\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle\!\rangle \bullet' : A \to \underline{B}$$

stand for

$$\lambda x. \langle \underline{B} \swarrow \underline{B}' \rangle (\bullet\ (\langle A' \nwarrow A \rangle x))$$

which clearly satisfies the typing requirement and monotonicity.
For identity extension, we have, using $\langle A \nwarrow A \rangle$ and $\langle \underline{B} \swarrow \underline{B} \rangle$ are the identity and η for $\to$,

$$\lambda x. \langle \underline{B} \swarrow \underline{B} \rangle (\bullet\ (\langle A \nwarrow A \rangle x)) \sqsupseteq\sqsubseteq \lambda x. (\bullet\ (x)) \sqsupseteq\sqsubseteq \bullet$$

8. Function upcast. We use Lemma 3.5 with the type constructor X val type, $\underline{Y}$ comp type $\vdash U(\underline{X} \to \underline{Y})$ val type. Suppose $A \sqsubseteq A'$ as value types and $\underline{B} \sqsubseteq \underline{B}'$ as computation types and let

$$p : U(A \to \underline{B}) \vdash \langle\!\langle U(A \to \underline{B}) \nwarrow U(A \to \underline{B}) \rangle\!\rangle p : U(A' \to \underline{B}')$$

stand for

$$\mathtt{thunk}\ (\lambda x'.\mathtt{bind}\ x \leftarrow \langle \underline{FA} \swarrow \underline{FA'}\rangle(\mathtt{ret} x'); \mathtt{force}\ (\langle U\underline{B'} \nwarrow U\underline{B}\rangle(\mathtt{thunk}\ (\mathtt{force}\ (f)\,x))))$$

which clearly satisfies the typing requirement and monotonicity.
Finally, for identity extension, using η for $\to$, β for F types and β/η for U types, and the fact that $\langle \underline{B} \nwarrow \underline{B}\rangle$ and $\langle A \swarrow A\rangle$ are the identity, we have

$$\begin{aligned}
\mathtt{thunk}\ (\lambda x.\mathtt{bind}\ x \leftarrow \langle \underline{FA} \swarrow \underline{FA}\rangle(\mathtt{ret} x); \mathtt{force}\ (\langle U\underline{B} \nwarrow U\underline{B}\rangle) \\
(\mathtt{thunk}\ (\mathtt{force}\ (f)\,x))) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{thunk}\ (\lambda x.\mathtt{bind}\ x \leftarrow (\mathtt{ret} x); \mathtt{force}\ (\mathtt{thunk}\ (\mathtt{force}\ (f)\,x))) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{thunk}\ (\lambda x.\mathtt{force}\ (\mathtt{thunk}\ (\mathtt{force}\ (f)\,x))) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{thunk}\ (\lambda x.(\mathtt{force}\ (f)\,x)) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{thunk}\ (\mathtt{force}\ (f)) &\quad \sqsupseteq\sqsubseteq \\
f &
\end{aligned}$$

9. $z : 0 \vdash \langle A \nwarrow 0\rangle z \sqsupseteq\sqsubseteq \mathtt{absurd}\ z : A$ is immediate by η for 0 on the map $z : 0 \vdash \langle A \nwarrow 0\rangle z : A$. □

Proof of Theorem 3.6. *Proof.*

1. We apply the upcast lemma with the type constructor X val type $\vdash U\underline{FX}$ val type. The term $\mathtt{thunk}\ (\langle\!\langle \underline{FA'} \nwarrow \underline{FA}\rangle\!\rangle(\mathtt{force}\ x))$ has the correct type and clearly satisfies monotonicity. Finally, for identity extension, we have

$$\begin{aligned}
\mathtt{thunk}\ (\langle\!\langle \underline{FA} \nwarrow \underline{FA}\rangle\!\rangle(\mathtt{force}\ x)) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{thunk}\ ((\mathtt{force}\ x)) &\quad \sqsupseteq\sqsubseteq \\
x &
\end{aligned}$$

   using η for U types and the identity principle for $\langle\!\langle \underline{FA} \nwarrow \underline{FA}\rangle\!\rangle$ (proved analogously to Theorem 3.2).
2. We use the downcast lemma with $\underline{Y}$ comp type $\vdash \underline{FUY}$ comp type, where $\mathtt{bind}\ x' : U\underline{B'} \leftarrow \bullet; \mathtt{ret}(\langle\!\langle U\underline{B} \swarrow U\underline{B'}\rangle\!\rangle x)$ clearly satisfies typing and monotonicity.
   Finally, for identity extension, we have

$$\begin{aligned}
\mathtt{bind}\ x : \underline{B} \leftarrow \bullet; \mathtt{ret}(\langle\!\langle \underline{B} \swarrow \underline{B}\rangle\!\rangle x) &\quad \sqsupseteq\sqsubseteq \\
\mathtt{bind}\ x : \underline{B} \leftarrow \bullet; \mathtt{ret}(x) &\quad \sqsupseteq\sqsubseteq \\
\bullet &
\end{aligned}$$

   using the identity principle for $\langle\!\langle \underline{B} \swarrow \underline{B}\rangle\!\rangle$ (proved analogously to Theorem 3.2) and η for F types. □

The admissibility theorem for casts in GTT$_G$, is proved by induction over a restricted form of type precision derivations.

Definition B.1 (Ground type precision)**.** *Let $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq' \underline{B'}$ be the relations defined by the rules in Figure 4 with the axioms $A \sqsubseteq ?$ and $\underline{B} \sqsubseteq \underline{¿}$ restricted to ground types—i.e., replaced by $G \sqsubseteq ?$ and $\underline{G} \sqsubseteq \underline{¿}$.*

Lemma B.1. *For any type A, $A \sqsubseteq' ?$. For any type $\underline{B}$, $\underline{B} \sqsubseteq' \underline{\text{¿}}$.*

Proof. By induction on the type. For example, in the case for $A_1 + A_2$, we have by the inductive hypothesis $A_1 \sqsubseteq' ?$ and $A_2 \sqsubseteq' ?$, so $A_1 + A_2 \sqsubseteq' ? + ? \sqsubseteq ?$ by congruence and transitivity, because $? + ?$ is ground. In the case for $\underline{F}A$, we have $A \sqsubseteq ?$ by the inductive hypothesis, so $\underline{F}A \sqsubseteq \underline{F}? \sqsubseteq \underline{\text{¿}}$. □

Lemma B.2 ($\sqsubseteq$ and $\sqsubseteq'$ agree)**.** *$A \sqsubseteq A'$ iff $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq \underline{B}'$ iff $\underline{B} \sqsubseteq' \underline{B}'$*

Proof. The "if" direction is immediate by induction because every rule of $\sqsubseteq'$ is a rule of $\sqsubseteq$. To show $\sqsubseteq$ is contained in $\sqsubseteq'$, we do induction on the derivation of $\sqsubseteq$, where every rule is true for $\sqsubseteq'$, except $A \sqsubseteq ?$ and $\underline{B} \sqsubseteq \underline{\text{¿}}$, and for these, we use Lemma B.1. □

Proof of Lemma 3.7 (cont.).

Proof. By induction on type precision $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq' \underline{B}'$.

(We chose not to make this more explicit above, because we believe the equational description in a language with all casts is a clearer description of the results, because it avoids needing to hypothesize terms that behave as the smaller casts in each case.)

We show a few representative cases:

In the cases for $G \sqsubseteq ?$ or $\underline{G} \sqsubseteq \underline{\text{¿}}$, we have assumed appropriate casts $\langle ? \nwarrow G\rangle$ and $\langle \underline{F}G \swarrow \underline{F}?\rangle$ and $\langle \underline{G} \swarrow \underline{\text{¿}}\rangle$ and $\langle U\underline{\text{¿}} \nwarrow U\underline{G}\rangle$.

In the case for identity $A \sqsubseteq A$, we need to show that there is an upcast $\langle\langle A \nwarrow A\rangle\rangle$ and a downcast $\langle\langle \underline{F}A \swarrow \underline{F}A\rangle\rangle$ The proof of Theorem 3.2 shows that the identity value and stack have the correct universal property.

In the case where type precision was concluded by transitivity between $A \sqsubseteq A'$ and $A' \sqsubseteq A''$, by the inductive hypotheses we get upcasts $\langle\langle A' \nwarrow A\rangle\rangle$ and $\langle\langle A'' \nwarrow A'\rangle\rangle$, and the proof of Theorem 3.2 shows that defining $\langle\langle A'' \nwarrow A\rangle\rangle$ to be $\langle\langle A'' \nwarrow A'\rangle\rangle\langle\langle A' \nwarrow A\rangle\rangle$ has the correct universal property. For the downcast, we get $\langle\langle \underline{F}A \swarrow \underline{F}A'\rangle\rangle$ and $\langle\langle \underline{F}A' \swarrow \underline{F}A''\rangle\rangle$ by the inductive hypotheses, and the proof of Theorem 3.2 shows that their composition has the correct universal property.

In the case where type precision was concluded by the congruence rule for $A_1 + A_2 \sqsubseteq A'_1 + A'_2$ from $A_i \sqsubseteq A'_i$, we have upcasts $\langle\langle A'_i \nwarrow A_i\rangle\rangle$ and downcasts $\langle\langle \underline{F}A_i \swarrow \underline{F}A'_i\rangle\rangle$ by the inductive hypothesis, and the proof of Theorem 3.2 shows that the definitions given there have the desired universal property.

In the case where type precision was concluded by the congruence rule for $\underline{F}A \sqsubseteq \underline{F}A'$ from $A \sqsubseteq A'$, we obtain by induction an *upcast* $A \sqsubseteq A'$ and a downcast $\langle\langle \underline{F}A \swarrow \underline{F}A'\rangle\rangle$. We need a *downcast* $\langle\langle \underline{F}A \swarrow FA'\rangle\rangle$, which we have, and an *upcast* $\langle\langle U\underline{F}A \swarrow U\underline{F}A'\rangle\rangle$, which is constructed as in Theorem 3.6. □

Proof of Theorem 3.9. *Proof.*

1. We have upcasts $x : A \vdash \langle A' \nwarrow A\rangle x : A'$ and $x' : A' \vdash \langle A \nwarrow A'\rangle x' : A$. For the composites, to show $x : A \vdash \langle A \nwarrow A'\rangle\langle A' \nwarrow A\rangle x \sqsubseteq x$ we apply upcast left twice, and conclude $x \sqsubseteq x$ by assumption. To show, $x : A \vdash x \sqsubseteq \langle A \nwarrow A'\rangle\langle A' \nwarrow A\rangle x$, we have $x : A \vdash x \sqsubseteq \langle A' \nwarrow A\rangle x : A \sqsubseteq A'$ by upcast right, and therefore $x : A \vdash x \sqsubseteq \langle A \nwarrow$

$A'\rangle\langle A' \nwarrow A\rangle x : A \sqsubseteq A$ again by upcast right. The other composite is the same proof with A and A' swapped.

2. We have downcasts $\bullet : \underline{B} \vdash \langle \underline{B} \swarrow \underline{B}'\rangle\bullet : \underline{B}'$ and $\bullet : \underline{B}' \vdash \langle \underline{B}' \swarrow \underline{B}\rangle\bullet : \underline{B}$.
   For the composites, to show $\bullet : \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B}' \swarrow \underline{B}\rangle\langle \underline{B} \swarrow \underline{B}'\rangle\bullet$, we apply downcast right twice, and conclude $\bullet \sqsubseteq \bullet$. For $\langle \underline{B}' \swarrow \underline{B}\rangle\langle \underline{B} \swarrow \underline{B}'\rangle\bullet \sqsubseteq \bullet$, we first have $\langle \underline{B} \swarrow \underline{B}'\rangle\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$ by downcast left, and then the result by another application of downcast left. The other composite is the same proof with $\underline{B}$ and $\underline{B}'$ swapped. □

Proof of Lemma 3.8.

Proof.

1. Take E to be $x : 0 \vdash \texttt{abort}\ x : T$. Given any E', we have $E \sqsupseteq\sqsubseteq E'$ by the η principle for 0.
2. Take S to be $\bullet : \underline{F}0 \vdash \texttt{bind}\ x \leftarrow \bullet; \texttt{abort}\ x : \underline{B}$. Given another S', by the η principle for F types, $S' \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \bullet; S'[\texttt{ret} x]$. By congruence, to show $S \sqsupseteq\sqsubseteq S'$, it suffices to show $x : 0 \vdash \texttt{abort}\ x \sqsupseteq\sqsubseteq S[\texttt{ret} x] : \underline{B}$, which is an instance of the previous part.
3. We have $y : 0 \vdash \texttt{abort}\ y : A$. The composite $y : 0 \vdash V[\texttt{abort}\ y/x] : 0$ is equiprecise with y by the η principle for 0, which says that any two complex values with domain 0 are equal.
   The composite $x : A \vdash \texttt{abort}\ V : A$ is equiprecise with x, because
   $$x : A, y : A, z : 0 \vdash x \sqsupseteq\sqsubseteq \texttt{abort}\ z \sqsupseteq\sqsubseteq y : A$$
   where the first is by η with $x : A, y : A, z : 0 \vdash E[z] := x : A$ and the second with $x : 0, y : 0 \vdash E[z] := y : A$ (this depends on the fact that 0 is "distributive", i.e., $\Gamma, x : 0$ has the universal property of 0). Substituting $\texttt{abort}\ V$ for y and V for z, we have $\texttt{abort}\ V \sqsupseteq\sqsubseteq x$. □

Proof of Lemma 3.9.

Proof.

1. Take $S = \{\}$. The η rule for $\top$, $\bullet : \top \vdash \bullet \sqsupseteq\sqsubseteq \{\} : \top$, under the substitution of $\bullet : \underline{B} \vdash S : \top$, gives $S \sqsupseteq\sqsubseteq \{\}[S/\bullet] = \{\}$.
2. Take $V = \texttt{thunk}\ \{\}$. We have $x : U\top \vdash x \sqsupseteq\sqsubseteq \texttt{thunk}\ \texttt{force}\ x \sqsupseteq\sqsubseteq \texttt{thunk}\ \{\} : U\top$ by the η rules for U and $\top$.
3. Take $V = ()$. By η for 1 with $x : 1 \vdash E[x] := () : 1$, we have $x : 1 \vdash () \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ ().: 1$. By η fro 1 with $x : 1 \vdash E[x] := x : 1$, we have $x : 1 \vdash x \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ ()..$ Therefore $x : 1 \vdash x \sqsupseteq\sqsubseteq () : 1$.
4. We have maps $x : U\top \vdash () : 1$ and $x : 1 \vdash \texttt{thunk}\ \{\} : U\top$. The composite on 1 is the identity by the previous part. The composite on $\top$ is the identity by part (2). □

Proof of Theorem 3.12.

Proof.

1. $x : 0 \vdash \langle A \nwarrow 0\rangle x \sqsupseteq\sqsubseteq \texttt{abort}\ x : A$ is immediate by η for 0.

2. First, to show $\bullet : \underline{F}A \vdash \texttt{bind}\ \_ \leftarrow \bullet; \mho \sqsubseteq \langle \underline{F}0 \swarrow \underline{F}A\rangle\bullet$, we can η-expand the right-hand side into $\texttt{bind}\ x : A \leftarrow \bullet; \langle \underline{F}0 \swarrow \underline{F}A\rangle\texttt{ret}x$, at which point the result follows by congruence and the fact that type error is minimal, so $\mho \sqsubseteq \langle \underline{F}0 \swarrow \underline{F}A\rangle\texttt{ret}x$. Second, to show $\bullet : \underline{F}A \vdash \langle \underline{F}0 \swarrow \underline{F}A\rangle\bullet \sqsubseteq \texttt{bind}\ \_ \leftarrow \bullet; \mho$, we can η-expand the left-hand side to $\bullet : \underline{F}A \vdash \texttt{bind}\ y \leftarrow \langle \underline{F}0 \swarrow \underline{F}A\rangle\bullet; \texttt{ret}y$, so we need to show
$$\bullet : \underline{F}A \vdash \texttt{bind}\ y : 0 \leftarrow \langle \underline{F}0 \swarrow \underline{F}A\rangle\bullet; \texttt{ret}y \sqsubseteq \texttt{bind}\ y' : A \leftarrow \bullet; \mho : \underline{F}0$$
We apply congruence, with $\bullet : \underline{F}A \vdash \langle \underline{F}0 \swarrow \underline{F}A\rangle\bullet \sqsubseteq \bullet : 0 \sqsubseteq A$ by the universal property of downcasts in the first premise, so it suffices to show
$$y \sqsubseteq y' : 0 \sqsubseteq A \vdash \texttt{ret}y \sqsubseteq \mho_{\underline{F}0} : \underline{F}0$$
By transitivity with $y \sqsubseteq y' : 0 \sqsubseteq A \vdash \mho_{\underline{F}0} \sqsubseteq \mho_{\underline{F}0} : \underline{F}0 \sqsubseteq \underline{F}0$, it suffices to show
$$y \sqsubseteq y : 0 \sqsubseteq 0 \vdash \texttt{ret}y \sqsubseteq \mho_{\underline{F}0} : \underline{F}0$$
But now both sides are maps out of 0, and therefore equal by Lemma 3.8.
3. The downcast is immediate by η for $\top$, Lemma 3.9.
4. First,
$$u : U\top \vdash \texttt{thunk}\ \mho \sqsubseteq \texttt{thunk}\ (\texttt{force}\ (\langle U\underline{B} \nwarrow U\top\rangle u)) \sqsupseteq\sqsubseteq \langle U\underline{B} \nwarrow U\top\rangle u : U\underline{B}$$
by congruence, η for U, and the fact that error is minimal. Conversely, to show
$$u : U\top \vdash \langle U\underline{B} \nwarrow U\top\rangle u \sqsubseteq \texttt{thunk}\ \mho : U\underline{B}$$
it suffices to show
$$u : U\top \vdash u \sqsubseteq \texttt{thunk}\ \mho_{\underline{B}} : U\top \sqsubseteq U\underline{B}$$
by the universal property of an upcast. By Lemma 3.9, any two elements of $U\top$ are equiprecise, so in particular $u \sqsupseteq\sqsubseteq \texttt{thunk}\ \mho_{\top}$, at which point congruence for thunk and $\mho_{\top} \sqsubseteq \mho_{\underline{B}} : \top \sqsubseteq \underline{B}$ gives the result. □

C Proofs for Section 4

To reason about substitution and plugging in evaluation contexts in the correctness proofs, we additionally define a *value* translation that directly translates CBV values to GTT values and a *stack* translation that directly translates CBV evaluation contexts to GTT stacks in Figure C.1

We then prove a few correctness principles for these with respect to the term translation.

Lemma C.1. $V^c \sqsupseteq\sqsubseteq \texttt{ret}V^v$

Proof. By induction on V.

- $\langle ? \Leftarrow G\rangle V$:
$$\begin{aligned} (\langle ? \Leftarrow G\rangle V)^c &= \langle F? \swarrow F?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}G^{ty}\rangle\!\rangle V^c && \text{(defn.)}\\ &\sqsupseteq\sqsubseteq \langle\!\langle \underline{F}? \nwarrow \underline{F}G^{ty}\rangle\!\rangle V^c && \text{(Theorem 3.2)}\\ &= \texttt{bind}\ x \leftarrow V^c; \texttt{ret}\langle ? \nwarrow G^{ty}\rangle x && \text{(defn.)}\end{aligned}$$

$$
\begin{aligned}
(\langle ? \Leftarrow G\rangle V)^v &= \langle ? \nwarrow G^{ty}\rangle V\\
(\lambda x : A.M)^v &= \texttt{thunk}\ (\lambda x : A^{ty}.M^c)\\
()^v &= ()\\
(V_1, V_2)^v &= (V_1^v, V_2^v)\\
(\texttt{inl}\ \ V)^v &= \texttt{inl}\ \ V^v\\
(\texttt{inr}\ \ V)^v &= \texttt{inr}\ \ V^v
\end{aligned}
$$

$$
\begin{aligned}
\bullet^s &= \bullet\\
(\texttt{let}\ x = S; N)^s &= \texttt{bind}\ x \leftarrow S^s; N^c\\
(\langle A_2 \Leftarrow A_1\rangle S)^s &= \texttt{bind}\ x \leftarrow S^s; \langle FA_2^{ty} \swarrow F?\rangle(\texttt{ret}\langle ? \nwarrow A_1^{ty}\rangle x)\\
(S\ N)^s &= \texttt{bind}\ f \leftarrow S^s; \texttt{bind}\ x \leftarrow N^c; \texttt{force}\ f\ x\\
(V\ S)^s &= \texttt{bind}\ x \leftarrow S^s; \texttt{force}\ \ V^v\ x\\
(\texttt{split}\ S\ \texttt{to}\ ().N)^s &= \texttt{bind}\ z \leftarrow S^s; \texttt{split}\ z\ \texttt{to}\ ().N^c\\
(S_1, M_2)^s &= \texttt{bind}\ x_1 \leftarrow S_1^c; \texttt{bind}\ x_2 \leftarrow M_2^c; \texttt{ret}(x_1, x_2)\\
(V_1, S_2)^s &= \texttt{bind}\ x_2 \leftarrow S_2^c; \texttt{ret}(V_1^v, x_2)\\
(\texttt{split}\ S\ \texttt{to}\ (x,y).N)^s &= \texttt{bind}\ z \leftarrow S^s; \texttt{split}\ z\ \texttt{to}\ (x,y).N^c\\
(\texttt{abort}\ \ S)^s &= \texttt{bind}\ z \leftarrow S^s; \texttt{abort}\ z\\
(\texttt{inl}\ \ S)^s &= \texttt{bind}\ x \leftarrow S^s; \texttt{retinl}\ x\\
(\texttt{inr}\ \ S)^s &= \texttt{bind}\ x \leftarrow S^s; \texttt{retinr}\ x\\
(\texttt{case}\ \ S\{x_1.N_1 \mid x_2.N_2\})^s &= \texttt{bind}\ z \leftarrow S^s; \texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\}
\end{aligned}
$$

Fig. C.1. CBV value and stack translations.

$$
\begin{aligned}
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{ret}\, V^v; \texttt{ret}\langle ? \nwarrow G^{ty}\rangle x && \text{(defn.)}\\
&\sqsupseteq\sqsubseteq \texttt{ret}\langle ? \nwarrow G^{ty}\rangle V^v && (F\beta)\\
&= \texttt{ret}(\langle ? \Leftarrow G\rangle V)^v && \text{(defn.)}
\end{aligned}
$$

- $\lambda x : A.M$: immediate by reflexivity.
- $()$: immediate by reflexivity.
- (V_1, V_2):

$$
\begin{aligned}
(V_1, V_2)^c &= \texttt{bind}\ x_1 \leftarrow V_1^c; \texttt{bind}\ x_2 \leftarrow V_2^c; \texttt{ret}(x_1, x_2) && \text{(definition)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1 \leftarrow \texttt{ret}\, V_1^v; \texttt{bind}\ x_2 \leftarrow \texttt{ret}\, V_2^v; \texttt{ret}(x_1, x_2) && \text{(I.H., twice)}\\
&\sqsupseteq\sqsubseteq \texttt{ret}(V_1, V_2) && (F\beta\ \text{twice})
\end{aligned}
$$

- $\texttt{inl}\ \ V$:

$$
\begin{aligned}
(\texttt{inl}\ \ V)^c &= \texttt{bind}\ x \leftarrow V^c; \texttt{retinl}\ x && \text{(definition)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{ret}\, V^v; \texttt{retinl}\ x && \text{(I.H.)}\\
&\sqsupseteq\sqsubseteq \texttt{retinl}\ \ V^v && (F\beta)
\end{aligned}
$$

- $\texttt{inr}\ \ V$: similar to $\texttt{inl}$ case. □

Lemma C.2. $(M[V/x])^c \sqsupseteq\sqsubseteq M^c[V^v/x]$

Proof. By induction on M. All cases but variable are by congruence and inductive hypothesis.

- $M = x$:

$$\begin{aligned}
(x[V/x])^c &= V^c && \text{(def. substitution)}\\
&\sqsupseteq\sqsubseteq \texttt{ret}\, V^v && \text{(Lemma C.1)}\\
&= (\texttt{ret} x)[V^v/x] && \text{(def. substitution)}\\
&= (x^c)[V^v/x] && \text{(def. substitution)}
\end{aligned}$$

- $M = y \neq x$:

$$\begin{aligned}
(y[V/x])^c &= y^c && \text{(def. subst.)}\\
&\sqsupseteq\sqsubseteq \texttt{ret} y && \text{(def.)}\\
&\sqsupseteq\sqsubseteq (\texttt{ret} y)[V/x] && \text{(def. subst.)}
\end{aligned}$$

□

Lemma C.3. $(S[M])^c \sqsupseteq\sqsubseteq S^s[M^c]$

Proof. By induction on S. Most cases are straightforward by congruence and induction hypothesis. We show the other cases.

- $S = V\,S$:

$$\begin{aligned}
((V\,S)[M])^c &= (V\,(S[M]))^c && \text{(defn. plugging)}\\
&= \texttt{bind}\ f \leftarrow V^c; \texttt{bind}\ x \leftarrow (S[M])^c; \texttt{force}\ f\ x && \text{(defn.)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ f \leftarrow \texttt{ret}\, V^v; \texttt{bind}\ x \leftarrow (S[M])^c; \texttt{force}\ f\ x && \text{(Lemma C.1)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow (S[M])^c; \texttt{force}\ V^v\ x && (F\beta)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow S^s[M^c]; \texttt{force}\ V^v\ x && \text{(I.H.)}\\
&\sqsupseteq\sqsubseteq (\texttt{bind}\ x \leftarrow S^s; \texttt{force}\ V^v\ x)[M^c] && \text{(defn. of plug)}\\
&= (V\,S)^s[M^c]
\end{aligned}$$

- $S = (V_1, S_2)$

$$\begin{aligned}
((V_1, S_2)[M])^c &= (V_1, S_2[M])^c && \text{(defn. plugging)}\\
&= \texttt{bind}\ x_1 \leftarrow V_1^c; \texttt{bind}\ x_2 \leftarrow S_2[M]^c; \texttt{ret}(x_1, x_2) && \text{(defn.)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1 \leftarrow \texttt{ret}\, V_1^v; \texttt{bind}\ x_2 \leftarrow S_2[M]^c; \texttt{ret}(x_1, x_2)\\
& && \text{(Lemma C.1)}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_2 \leftarrow S_2[M]^c; \texttt{ret}(V_1^v, x_2) && (F\beta)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_2 \leftarrow S_2^s[M^c]; \texttt{ret}(V_1^v, x_2) && \text{(I.H.)}\\
&= (\texttt{bind}\ x_2 \leftarrow S_2^s; \texttt{ret}(V_1^v, x_2))[M^c] && \text{(defn. plugging)}\\
&= (V_1, S_2^s)[M^c] && \text{(defn.)}
\end{aligned}$$

□

Finally, for proving the correctness for cast reductions, the following lemma simplifies a great deal of common reasoning about the translation of casts.

Lemma C.4 (Any Middle Type will Do)**.** *If* $A_1, A_2 \sqsubseteq A'$, *then* $\langle FA_2 \twoheadleftarrow F?\rangle\langle\langle F? \leftarrowtail FA_1\rangle\rangle M \sqsupseteq\sqsubseteq \langle FA_2 \twoheadleftarrow FA'\rangle\langle\langle FA' \leftarrowtail FA_1\rangle\rangle M$

Proof.

$$\begin{aligned}\langle FA_2 \twoheadleftarrow F?\rangle\langle\langle F? \leftarrowtail FA_1\rangle\rangle M &\sqsupseteq\sqsubseteq \langle FA_2 \twoheadleftarrow FA'\rangle\langle FA' \twoheadleftarrow F?\rangle\langle\langle F? \leftarrowtail FA'\rangle\rangle\langle\langle FA' \leftarrowtail FA_1\rangle\rangle M \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(Theorem 3.2)} \\ &\sqsupseteq\sqsubseteq \langle FA_2 \twoheadleftarrow FA'\rangle\langle\langle FA' \leftarrowtail FA_1\rangle\rangle M \qquad\qquad\qquad\qquad \text{(retraction)}\end{aligned}$$

□

Proof of Theorem 4.1.

Proof. In all cases, by Lemma C.3, congruence and $S[\mho] \sqsupseteq\sqsubseteq \mho$, it is sufficient to consider the case that $S = \bullet$.

First, we have the cases not involving casts, which are standard for the embedding of call-by-value into call-by-push-value.

- $\texttt{let}\ x = V; N \mapsto N[V/x]$

$$\begin{aligned}(\texttt{let}\ x = V; N)^c &= \texttt{bind}\ x \leftarrow V^c; N^c \\ &\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{ret}\, V^v; N^c \\ &\sqsupseteq\sqsubseteq N^c[V^v/x] \\ &\sqsupseteq\sqsubseteq (N[V/x]^c)\end{aligned}$$

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

$$\begin{aligned}((\lambda x : A.M)\ V)^c &= \texttt{bind}\ f \leftarrow (\texttt{ret}(\texttt{thunk}\ (\lambda x : A^{ty}.M^c))); \texttt{bind}\ x \leftarrow V^c; \texttt{force}\ f\ x \\ &\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow V^c; \texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\, x \\ &\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{ret}\, V^v; \texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\, x \\ &\sqsupseteq\sqsubseteq \texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\ V^v \\ &\sqsupseteq\sqsubseteq (\lambda x : A^{ty}.M^c)\ V^v \\ &\sqsupseteq\sqsubseteq M^c[V^v/x] \\ &\sqsupseteq\sqsubseteq (M[V/x])^c\end{aligned}$$

- $\texttt{split}\ ()\,\texttt{to}\ ().N \mapsto N$

$$\begin{aligned}(\texttt{split}\ ()\,\texttt{to}\ ().N)^c &= \texttt{bind}\ z \leftarrow \texttt{ret}(); \texttt{split}\ z\,\texttt{to}\ ().N^c \\ &= \texttt{split}\ ()\,\texttt{to}\ ().N^c \\ &= N^c\end{aligned}$$

- $\texttt{split}\ (V_1, V_2)\,\texttt{to}\ (x_1, x_2).N \mapsto N[V_1/x_1][V_2/x_2]$

$$\begin{aligned}(\texttt{split}\ (V_1, V_2)\,\texttt{to}\ (x_1, x_2).N)^c &= \texttt{bind}\ z \leftarrow (V_1, V_2)^c; \texttt{split}\ z\,\texttt{to}\ (x_1, x_2).N^c \\ &\sqsupseteq\sqsubseteq \texttt{bind}\ z \leftarrow \texttt{ret}(V_1^v, V_2^v); \texttt{split}\ z\,\texttt{to}\ (x_1, x_2).N^c\end{aligned}$$

$$\begin{aligned}
&\sqsupseteq\sqsubseteq \texttt{split}\ (V_1^v, V_2^v)\ \texttt{to}\ (x_1, x_2).N^c\\
&\sqsupseteq\sqsubseteq N^c[V_1^v/x_1][V_2^v/x_2]\\
&\sqsupseteq\sqsubseteq (N[V_1/x_1][V_2/x_2])^c
\end{aligned}$$

- $\texttt{case inl}\ V\{x_1.N_1 \mid x_2.N_2\} \mapsto N_1[V/x_1]$

$$\begin{aligned}
(\texttt{case inl}\ V\{x_1.N_1 \mid x_2.N_2\})^c &= \texttt{bind}\ z \leftarrow (\texttt{inl}\ V)^c; \texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\}\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ z \leftarrow \texttt{ret}(\texttt{inl}\ V^v); \texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\}\\
&\sqsupseteq\sqsubseteq \texttt{case inl}\ V^v\{x_1.N_1^c \mid x_2.N_2^c\}\\
&\sqsupseteq\sqsubseteq N_1^c[V^v/x_1]\\
&\sqsupseteq\sqsubseteq (N_1[V/x_1])^c
\end{aligned}$$

- $\texttt{case inr}\ V\{x_1.N_1 \mid x_2.N_2\} \mapsto N_2[V/x_2]$

Next, we have the interesting cases, those specific to gradual type casts/GTT.

- $\langle ? \Leftarrow ?\rangle V \mapsto V$:

$$\begin{aligned}
(\langle ? \Leftarrow ?\rangle V)^c &= \langle F? \swarrow F?\rangle \langle\!\langle F? \nwarrow F?\rangle\!\rangle[V^c]\\
&\sqsupseteq\sqsubseteq V^c && \text{(Theorem 3.2)}
\end{aligned}$$

- $\langle ? \Leftarrow A\rangle V \mapsto \langle ? \Leftarrow G\rangle\langle G \Leftarrow A\rangle V$ where $A \sqsubseteq G$

$$\begin{aligned}
\langle ? \Leftarrow A\rangle V^c &\sqsupseteq\sqsubseteq \langle F? \swarrow F?\rangle \langle\!\langle F? \nwarrow FA^{ty}\rangle\!\rangle[V^c]\\
&\sqsupseteq\sqsubseteq \langle\!\langle F? \nwarrow FA^{ty}\rangle\!\rangle[V^c] && \text{(Theorem 3.2)}\\
&\sqsupseteq\sqsubseteq \langle\!\langle F? \nwarrow FG^{ty}\rangle\!\rangle \langle\!\langle FG^{ty} \nwarrow FA^{ty}\rangle\!\rangle[V^c] && \text{(Theorem 3.2)}
\end{aligned}$$

- $\langle A \Leftarrow ?\rangle V \mapsto \langle A \Leftarrow G\rangle\langle G \Leftarrow ?\rangle V$: similar to previous case.
- $\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V \mapsto V$

$$\begin{aligned}
(\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V)^c &= \langle FG^{ty} \swarrow F?\rangle \langle\!\langle F? \nwarrow F?\rangle\!\rangle \langle F? \swarrow F?\rangle \langle\!\langle F? \nwarrow FG^{ty}\rangle\!\rangle[V^c]\\
&\sqsupseteq\sqsubseteq \langle FG^{ty} \swarrow F?\rangle \langle\!\langle F? \nwarrow FG^{ty}\rangle\!\rangle[V^c] && \text{(Theorem 3.2)}\\
&\sqsupseteq\sqsubseteq V^c && \text{(retraction)}
\end{aligned}$$

- $\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle V \mapsto \lambda x : A_1'.\langle A_2' \Leftarrow A_2\rangle(V\ (\langle A_1 \Leftarrow A_1'\rangle x))$

$$\begin{aligned}
&(\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle V)^c\\
&\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow F?\rangle \langle\!\langle F? \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle V^c\\
&\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \langle\!\langle FU(? \to F?) \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle V^c\\
&\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \langle\!\langle FU(? \to F?) \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle[\texttt{ret}\,V^v]\\
&\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \texttt{ret}\langle U(? \to F?) \nwarrow U(A_1^{ty} \to FA_2^{ty})\rangle V^v\\
&\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle\\
&\qquad \texttt{retthunk}\ (\lambda x'.\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\texttt{ret}x'); \texttt{force}\ (\langle U\underline{F}? \nwarrow U\underline{F}A_2^{ty}\rangle\\
&\qquad (\texttt{thunk}\ (\texttt{force}\ V^v\,x))))
\end{aligned}$$

$$\sqsupseteq\sqsubseteq \langle FU(A_1^{\prime ty} \to FA_2^{\prime ty}) \swarrow FU(? \to F?)\rangle$$

$$\quad \texttt{retthunk}\ (\lambda x'.\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\texttt{ret}x'); \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ ((\langle A_1^{\prime ty} \to FA_2^{\prime ty} \swarrow ? \to F?\rangle(\lambda x'.\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\texttt{ret}x');$$

$$\quad \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x)))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle((\lambda x'.\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\texttt{ret}x');$$

$$\quad \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x)))(\langle ? \nwarrow A_1^{\prime ty}\rangle y'))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle((\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\texttt{ret}(\langle ? \nwarrow A_1^{\prime ty}\rangle y');$$

$$\quad \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x))))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle((\texttt{bind}\ x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{\prime ty}\rangle\!\rangle\texttt{ret}y';$$

$$\quad \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x))))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle((\texttt{bind}\ x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^{ty};$$

$$\quad \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{force}\ V^v\ x))))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\texttt{bind}\ x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^{ty};$$

$$\quad (\texttt{force}\ V^v\ x)))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{ty}\rangle\!\rangle(\texttt{bind}\ f \leftarrow V^c;$$

$$\quad \texttt{bind}\ x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^c; (\texttt{force}\ f\ x)))$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.\langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{ty}\rangle\!\rangle(V\ (\langle A_1 \Leftarrow A_1'\rangle y'))^c)$$

$$\sqsupseteq\sqsubseteq \texttt{retthunk}\ (\lambda y'.(\langle A_2' \Leftarrow A_2\rangle(V\ (\langle A_1 \Leftarrow A_1'\rangle y')))^c)$$

$$\sqsupseteq\sqsubseteq (\lambda y'.\langle A_2' \Leftarrow A_2\rangle(V\ (\langle A_1 \Leftarrow A_1'\rangle y')))^c$$

- $\langle 1 \Leftarrow 1\rangle() \mapsto ()$

$$\begin{aligned} (\langle 1 \Leftarrow 1\rangle())^c &= \langle \underline{F}1 \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}1\rangle\!\rangle[\texttt{ret}()] \\ &\sqsupseteq\sqsubseteq \texttt{ret}() \\ &= ()^c \end{aligned}$$

- $\langle A_1' \times A_2' \Leftarrow A_1 \times A_2\rangle(V_1, V_2) \mapsto (\langle A_1' \Leftarrow A_1\rangle V_1, \langle A_2' \Leftarrow A_2\rangle V_2)$

$$(\langle A_1' \times A_2' \Leftarrow A_1 \times A_2\rangle(V_1, V_2))^c$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle(V_1, V_2)^c$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}(? \times ?)\rangle\langle\!\langle \underline{F}(? \times ?) \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle(V_1, V_2)^c$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}(? \times ?)\rangle\langle\!\langle \underline{F}(? \times ?) \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle\texttt{ret}(V_1^v, V_2^v)$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}(? \times ?)\rangle(\texttt{ret}\langle(? \times ?) \nwarrow (A_1^{ty} \times A_2^{ty})\rangle(V_1^v, V_2^v))$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}(? \times ?)\rangle$$

$$\quad (\texttt{ret}(\texttt{split}\ (V_1^v, V_2^v)\ \texttt{to}\ (x_1, x_2).(\langle ? \nwarrow A_1^{ty}\rangle x_1, \langle ? \nwarrow A_2^{ty}\rangle x_2)))$$

$$\sqsupseteq\sqsubseteq \langle \underline{F}(A_1^{\prime ty} \times A_2^{\prime ty}) \swarrow \underline{F}(? \times ?)\rangle(\texttt{ret}(\langle ? \nwarrow A_1^{ty}\rangle V_1^v, \langle ? \nwarrow A_2^{ty}\rangle x_2))$$

$$\sqsupseteq\sqsubseteq \texttt{split}\ (\langle ? \nwarrow A_1^{ty}\rangle V_1^v, \langle ? \nwarrow A_2^{ty}\rangle V_2^v)\ \texttt{to}\ (y_1, y_2).$$

$$\quad \texttt{bind}\ x_1' \leftarrow \langle \underline{F}A_1^{\prime ty} \swarrow \underline{F}?\rangle\texttt{ret}y_1;$$

$$\quad \texttt{bind}\ x_2' \leftarrow \langle \underline{F}A_2^{\prime ty} \swarrow \underline{F}?\rangle\texttt{ret}y_2; \texttt{ret}(x_1', x_2')$$

$$\begin{aligned}
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \texttt{ret}\langle ? \nwarrow A_1^{ty}\rangle V_1^v;\\
&\qquad \texttt{bind}\ x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \texttt{ret}\langle ? \nwarrow A_2^{ty}\rangle V_2^v; \texttt{ret}(x_1', x_2')\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle ? \nwarrow A_1^{ty}\rangle\!\rangle \texttt{ret}\, V_1^v;\\
&\qquad \texttt{bind}\ x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle ? \nwarrow A_2^{ty}\rangle\!\rangle \texttt{ret}\, V_2^v; \texttt{ret}(x_1', x_2')\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle ? \nwarrow A_1^{ty}\rangle\!\rangle V_1^c;\\
&\qquad \texttt{bind}\ x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle ? \nwarrow A_2^{ty}\rangle\!\rangle V_2^c; \texttt{ret}(x_1', x_2')\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x_1' \leftarrow \langle A_1' \Leftarrow A_1\rangle V_1^c; \texttt{bind}\ x_2' \leftarrow \langle A_2' \Leftarrow A_2\rangle V_2^c; \texttt{ret}(x_1', x_2')\\
&= (\langle A_1' \Leftarrow A_1\rangle V_1, \langle A_2' \Leftarrow A_2\rangle V_2)^c
\end{aligned}$$

- $\langle A_1' + A_2' \Leftarrow A_1 + A_2\rangle(\texttt{inl}\ V) \mapsto \langle A_1' \Leftarrow A_1\rangle V$

$$\begin{aligned}
&(\langle A_1' + A_2' \Leftarrow A_1 + A_2\rangle(\texttt{inl}\ V))^c\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}?\rangle \langle\!\langle \underline{F}? \nwarrow \underline{F}(A_1^{ty} + A_2^{ty})\rangle\!\rangle (\texttt{inl}\ V)^c\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}(? + ?)\rangle \langle\!\langle \underline{F}(? + ?) \nwarrow \underline{F}(A_1^{ty} + A_2^{ty})\rangle\!\rangle (\texttt{inl}\ V)^c\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}(? + ?)\rangle \langle\!\langle \underline{F}(? + ?) \nwarrow \underline{F}(A_1^{ty} + A_2^{ty})\rangle\!\rangle \texttt{ret}(\texttt{inl}\ V^v)\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}(? + ?)\rangle \texttt{ret}\langle (? + ?) \nwarrow (A_1^{ty} + A_2^{ty})\rangle (\texttt{inl}\ V^v)\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}(? + ?)\rangle\\
&\qquad \texttt{ret}(\texttt{case inl}\ V^v\{x_1.\texttt{inl}\ \langle ? \nwarrow A_1\rangle x_1 \mid x_2.\texttt{inr}\ \langle ? \nwarrow A_2\rangle x_2\})\\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} + A_2'^{ty}) \swarrow \underline{F}(? + ?)\rangle \texttt{ret}(\texttt{inl}\ \langle ? \nwarrow A_1\rangle V^v)\\
&\sqsupseteq\sqsubseteq \texttt{case}\ (\texttt{inl}\ \langle ? \nwarrow A_1\rangle V^v)\{x_1.\langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \texttt{ret}x_1 \mid x_2.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \texttt{ret}x_2\}\\
&\sqsupseteq\sqsubseteq \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \texttt{ret}\langle ? \nwarrow A_1\rangle V^v\\
&\sqsupseteq\sqsubseteq \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle \underline{F}? \nwarrow \underline{F}A_1\rangle\!\rangle \texttt{ret}\, V^v\\
&\sqsupseteq\sqsubseteq \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \langle\!\langle \underline{F}? \nwarrow \underline{F}A_1\rangle\!\rangle V^c\\
&\sqsupseteq\sqsubseteq (\langle A_1' \Leftarrow A_1\rangle V)^c
\end{aligned}$$

- $\langle A_1' + A_2' \Leftarrow A_1 + A_2\rangle(\texttt{inr}\ V) \mapsto \langle A_2' \Leftarrow A_2\rangle V$: similar to `inl` case. □

D Proofs for Section 5

Proof of Lemma 5.1. *Proof.* For the first,

$$\begin{aligned}
\texttt{bind}\ x \leftarrow S_p[\texttt{ret}x']; M[V_e/y] &\sqsupseteq\sqsubseteq \texttt{bind}\ y \leftarrow (\texttt{bind}\ x \leftarrow S_p[\texttt{ret}x']; \texttt{ret}\, V_e); M\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(comm conv, } \underline{F}\beta)\\
&\texttt{bind}\ y \leftarrow \texttt{ret}x'; M \qquad\qquad\qquad\qquad \text{(projection)}\\
&M[x'/y] \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\beta)
\end{aligned}$$

For the second,

$$\begin{aligned}
V_e[\texttt{thunk}\ S_p[M]] &\sqsupseteq\sqsubseteq V_e[\texttt{thunk}\ S_p[\texttt{force thunk}\ M]] && (U\beta)\\
&\sqsubseteq \texttt{thunk}\ M && \text{(projection)}
\end{aligned}$$

□

The proof of Lemma 5.2 relies on the following induction principle for the returner type:

Lemma D.1 ($\underline{F}(+)$ Induction Principle). *$\Gamma \mid \cdot : \underline{F}(A_1 + A_2) \vdash M_1 \sqsubseteq M_2 : \underline{B}$ holds if and only if $\Gamma, V_1 : A_1 \vdash M_1[\mathit{retinl}\ V_1] \sqsubseteq M_2[\mathit{retinl}\ V_2] : \underline{B}$ and $\Gamma, V_2 : A_2 \vdash M_2[\mathit{retinr}\ V_2] \sqsubseteq M_2[\mathit{retinr}\ V_2] : \underline{B}$*

Proof of Lemma 5.2 (cont.).

Proof. This satisfies retraction (using $\underline{F}(+)$ induction (Lemma D.1), `inr` case is the same):

$$\begin{aligned}
&\mathtt{bind}\ y \leftarrow \mathtt{inl}\ x; \mathtt{case}\ y\{\mathtt{inl}\ x.\mathtt{ret}x \mid \mathtt{inr}\ .\mho\} \sqsupseteq\sqsubseteq \mathtt{case}\ \mathtt{inl}\ x\{\mathtt{inl}\ x.\mathtt{ret}x \mid \mathtt{inr}\ .\mho\} \\
&&& (\underline{F}\beta) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \sqsupseteq\sqsubseteq \mathtt{ret}x && (+\beta)
\end{aligned}$$

and projection (similarly using $\underline{F}(+)$ induction):

$$\begin{aligned}
x' : A + A' \vdash\ & \mathtt{bind}\ (\mathtt{bind}\ y \leftarrow \mathtt{ret}x'; \mathtt{case}\ y\{\mathtt{inl}\ x.\mathtt{ret}x \mid \mathtt{inr}\ .\mho\}) \leftarrow x; \mathtt{retinl}\ x \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ (\mathtt{case}\ x'\{\mathtt{inl}\ x.\mathtt{ret}x \mid \mathtt{inr}\ .\mho\}) \leftarrow x; \mathtt{retinl}\ x && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq (\mathtt{case}\ x'\{\mathtt{inl}\ x.\mathtt{bind}\ x \leftarrow \mathtt{ret}x; \mathtt{retinl}\ x \mid \mathtt{inr}\ .\mathtt{bind}\ x \leftarrow \mho; \mathtt{retinl}\ x\}) \\
&&& (\text{commuting conversion}) \\
&\sqsupseteq\sqsubseteq (\mathtt{case}\ x'\{\mathtt{inl}\ x.\mathtt{retinl}\ x \mid \mathtt{inr}\ .\mho\}) && (\underline{F}\beta, \mho \text{ strictness}) \\
&\sqsubseteq (\mathtt{case}\ x'\{\mathtt{inl}\ x.\mathtt{retinl}\ x \mid \mathtt{inr}\ y.\mathtt{retinl}\ y\}) && (\mho \text{ bottom}) \\
&\sqsupseteq\sqsubseteq \mathtt{ret}x' && (+\eta)
\end{aligned}$$

□

Proof of Lemma 5.3. *Proof.* This satisfies retraction:

$$\begin{aligned}
\pi\mathtt{force}\ \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ z \mid \pi' \mapsto \mho\} &\sqsupseteq\sqsubseteq \pi\{\pi \mapsto \mathtt{force}\ z \mid \pi' \mapsto \mho\} && (U\beta) \\
&\sqsupseteq\sqsubseteq \mathtt{force}\ z && (\&\beta)
\end{aligned}$$

and projection:

$$\begin{aligned}
&\mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ w \mid \pi' \mapsto \mho\} \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \pi\mathtt{force}\ w \mid \pi' \mapsto \mho\} && (U\beta) \\
&\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \pi\mathtt{force}\ w \mid \pi' \mapsto \pi'\mathtt{force}\ w\} && (\mho \text{ bottom}) \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ w && (\&\eta) \\
&\sqsupseteq\sqsubseteq w && (U\eta)
\end{aligned}$$

□

Proof of Lemma 5.6. *Proof.* It is clear that the normalized system is a subset of the original: every normalized rule corresponds directly to a rule of the original system, except the normalized $A \sqsubseteq\ ?$ and $\underline{B} \sqsubseteq \underline{¿}$ rules have a subderivation that was not present originally.

For the converse, first we show by induction that reflexivity is admissible:

1. If $A \in \{?, 1, 0\}$, we use a normalized rule.
2. If $A \notin \{?, 1, 0\}$, we use the inductive hypothesis and the monotonicity rule.

3. If $\underline{B} \in \{\underline{¿}, \top\}$ use the normalized rule.
4. If $\underline{B} \notin \{\underline{¿}, \top\}$ use the inductive hypothesis and monotonicity rule.

Next, we show that transitivity is admissible:

1. Assume we have $A \sqsubseteq A' \sqsubseteq A''$
   a. If the left rule is $0 \sqsubseteq A'$, then either $A' = ?$ or $A' = 0$. If $A' = 0$ the right rule is $0 \sqsubseteq A''$ and we can use that proof. Otherwise, $A' = ?$ then the right rule is $? \sqsubseteq ?$ and we can use $0 \sqsubseteq ?$.
   b. If the left rule is $A \sqsubseteq A$ where $A \in \{?, 1\}$ then either $A = ?$, in which case $A'' = ?$ and we're done. Otherwise the right rule is either $1 \sqsubseteq 1$ (done) or $1 \sqsubseteq ?$ (also done).
   c. If the left rule is $A \sqsubseteq ?$ with $A \notin \{0, ?\}$ then the right rule must be $? \sqsubseteq ?$ and we're done.
   d. Otherwise the left rule is a monotonicity rule for one of $U, +, \times$ and the right rule is either monotonicity (use the inductive hypothesis) or the right rule is $A' \sqsubseteq ?$ with a sub-proof of $A' \sqsubseteq \lfloor A' \rfloor$. Since the left rule is monotonicity, $\lfloor A \rfloor = \lfloor A' \rfloor$, so we inductively use transitivity of the proof of $A \sqsubseteq A'$ with the proof of $A' \sqsubseteq \lfloor A' \rfloor$ to get a proof $A \sqsubseteq \lfloor A \rfloor$ and thus $A \sqsubseteq ?$.
2. Assume we have $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$.
   a. If the left rule is $\top \sqsubseteq \underline{B}'$ then $\underline{B}'' \in \{\underline{¿}, \top\}$ so we apply that rule.
   b. If the left rule is $\underline{¿} \sqsubseteq \underline{¿}$, the right rule must be as well.
   c. If the left rule is $\underline{B} \sqsubseteq \underline{¿}$ the right rule must be reflexivity.
   d. If the left rule is a monotonicity rule for $\&, \to, \underline{F}$ then the right rule is either also monotonicity (use the inductive hypothesis) or it's a $\underline{B} \sqsubseteq \underline{¿}$ rule and we proceed with ? above

Finally we show $A \sqsubseteq ?$, $\underline{B} \sqsubseteq \underline{¿}$ are admissible by induction on A, $\underline{B}$.

1. If $A \in \{?, 0\}$ we use the primitive rule.
2. If $A \notin \{?, 0\}$ we use the $A \sqsubseteq ?$ rule and we need to show $A \sqsubseteq \lfloor A \rfloor$. If $A = 1$, we use the $1 \sqsubseteq 1$ rule, otherwise we use the inductive hypothesis and monotonicity.
3. If $\underline{B} \in \{\underline{¿}, \top\}$ we use the primitive rule.
4. If $\underline{B} \notin \{\underline{¿}, \top\}$ we use the $\underline{B} \sqsubseteq \underline{¿}$ rule and we need to show $\underline{B} \sqsubseteq \lfloor \underline{B} \rfloor$, which follows by inductive hypothesis and monotonicity.

Every other rule in Figure 4 is a rule of the normalized system in Figure 17. □

To keep proofs high-level, we establish the following cast reductions that follow easily from β, η principles.

Lemma D.2 (Cast Reductions). *The following are all provable*

$$\llbracket \langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle \rrbracket [\texttt{inl}\ V] \sqsupseteq\sqsubseteq \texttt{inl}\ \llbracket \langle A_1' \nwarrow A_1 \rangle \rrbracket [V]$$
$$\llbracket \langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle \rrbracket [\texttt{inr}\ V] \sqsupseteq\sqsubseteq \texttt{inr}\ \llbracket \langle A_2' \nwarrow A_2 \rangle \rrbracket [V]$$
$$\llbracket \langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle \rrbracket [\mathit{ret}\texttt{inl}\ V] \sqsupseteq\sqsubseteq \texttt{bind}\ x_1 \leftarrow \llbracket \langle A_1 \swarrow A_1' \rangle \rrbracket [\mathit{ret} V];$$
$$\mathit{ret}\texttt{inl}\ x_1$$

$$\llbracket\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle\rrbracket[\texttt{retinr}\ V] \sqsupseteq\sqsubseteq \texttt{bind}\ x_2 \leftarrow \llbracket\langle A_2 \swarrow A_2'\rangle\rrbracket[\texttt{ret}V];$$
$$\texttt{retinr}\ x_2$$
$$\llbracket\langle \underline{F}1 \swarrow \underline{F}1\rangle\rrbracket \sqsupseteq\sqsubseteq \bullet$$
$$\llbracket\langle 1 \nwarrow 1\rangle\rrbracket[x] \sqsupseteq\sqsubseteq x$$
$$\llbracket\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle\rrbracket[\texttt{ret}(V_1, V_2)]$$
$$\quad \sqsupseteq\sqsubseteq \texttt{bind}\ x_1 \leftarrow \llbracket\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle\rrbracket[\texttt{ret}V_1]; \texttt{bind}\ x_2 \leftarrow \llbracket\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle\rrbracket[\texttt{ret}V_2];$$
$$\qquad \texttt{ret}(x_1, x_2)$$
$$\llbracket\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle\rrbracket[(V_1, V_2)] \sqsupseteq\sqsubseteq (\llbracket\langle A_1' \nwarrow A_1\rangle\rrbracket[V_1], \llbracket\langle A_2' \nwarrow A_2\rangle\rrbracket[V_2])$$
$$(\llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket M)\ V \sqsupseteq\sqsubseteq (\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket M)\ (\llbracket\langle A' \nwarrow A\rangle\rrbracket V)$$
$$(\texttt{force}\ (\llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket V))\ V'$$
$$\quad \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle[\texttt{ret}V']; \texttt{force}\ (\llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket(\texttt{thunk}\ (\texttt{force}\ V\ x)))$$
$$\pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket M \sqsupseteq\sqsubseteq \llbracket\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle\rrbracket \pi M$$
$$\pi'\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket M \sqsupseteq\sqsubseteq \llbracket\langle \underline{B}_2 \swarrow \underline{B}_2'\rangle\rrbracket \pi' M$$
$$\pi\texttt{force}\ (\llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket V) \sqsupseteq\sqsubseteq \texttt{force}\ \llbracket\langle U\underline{B}_1' \nwarrow U\underline{B}_1\rangle\rrbracket$$
$$\qquad \texttt{thunk}\ (\pi\texttt{force}\ V)$$
$$\pi'\texttt{force}\ (\llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket V) \sqsupseteq\sqsubseteq \texttt{force}\ \llbracket\langle U\underline{B}_2' \nwarrow U\underline{B}_2\rangle\rrbracket$$
$$\qquad \texttt{thunk}\ (\pi'\texttt{force}\ V)$$
$$\llbracket\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle\rrbracket[\texttt{ret}V] \sqsupseteq\sqsubseteq \texttt{retthunk}\ \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket\texttt{force}\ V$$
$$\texttt{force}\ \llbracket\langle U\underline{F}A' \nwarrow U\underline{F}A\rangle\rrbracket[V] \sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{force}\ V; \texttt{thunk}\ \texttt{ret}\langle A' \nwarrow A\rangle x$$

Proof of Lemma 5.8.

Proof.

1. First, retraction follows from retraction twice:

$$S_1[S_2[\texttt{ret}V_2[V_1[x]]]] \sqsupseteq\sqsubseteq S_1[\texttt{ret}[V_1[x]]] \sqsupseteq\sqsubseteq x$$

and projection follows from projection twice:

$$\begin{aligned}
\texttt{bind}\ x \leftarrow S_1[S_2[\bullet]]; \texttt{ret}V_2[V_1[x]] &\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow S_1[S_2[\bullet]]; \\
&\quad \texttt{bind}\ y \leftarrow \texttt{ret}[V_1[x]]; \texttt{ret}V_2[y] && (\underline{F}\beta)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ y \leftarrow (\texttt{bind}\ x \leftarrow S_1[S_2[\bullet]]; \texttt{ret}[V_1[x]]); \\
&\qquad \texttt{ret}V_2[y] && \text{(Commuting conversion)}\\
&\sqsubseteq \texttt{bind}\ y \leftarrow S_2[\bullet]; \texttt{ret}V_2[y] && \text{(Projection)}\\
&\sqsubseteq \bullet && \text{(Projection)}
\end{aligned}$$

2. Again retraction follows from retraction twice:

$$S_1[S_2[\texttt{force}\ V_2[V_1[z]]]] \sqsupseteq\sqsubseteq S_1[\texttt{force}\ V_1[z]] \sqsupseteq\sqsubseteq \texttt{force}\ z$$

and projection from projection twice:

$$\begin{aligned}
V_2[V_1[\texttt{thunk } S_1[S_2[\texttt{force } w]]]] \sqsupseteq\sqsubseteq{} & V_2[V_1[\texttt{thunk } S_1[\texttt{force thunk } S_2[\texttt{force } w]]]] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\beta) \\
\sqsubseteq{} & V_2[\texttt{thunk } S_2[\texttt{force } w]] && \text{(Projection)} \\
\sqsubseteq{} & w && \text{(Projection)}
\end{aligned}$$

□

Proof of Lemma 5.10.

Proof. By induction on normalized type precision derivations.

1. $A \sqsubseteq A$ $(A \in \{?, 1\})$, because identity is an ep pair.
2. $0 \sqsubseteq A$ (that $A \in \{?, 0\}$ is not important):

   a. Retraction is

   $$x : 0 \vdash \texttt{ret} x \sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow \texttt{retabsurd } x; \mho : \underline{F}A$$

   which holds by 0η

   b. Projection is

   $$\bullet : \underline{F}A \vdash \texttt{bind } x \leftarrow (\texttt{bind } y \leftarrow \bullet; \mho); \texttt{retabsurd } x \sqsubseteq \bullet : \underline{F}A$$

   Which we calculate:

   $$\begin{aligned}
   & \texttt{bind } x \leftarrow (\texttt{bind } y \leftarrow \bullet; \mho); \texttt{retabsurd } x \\
   & \sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow \bullet; \texttt{bind } x \leftarrow \mho; \texttt{retabsurd } x && \text{(comm conv)} \\
   & \sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow \bullet; \mho && \text{(Strictness of Stacks)} \\
   & \sqsubseteq \texttt{bind } y \leftarrow \bullet; \texttt{ret} y && (\mho \text{ is } \bot) \\
   & \sqsupseteq\sqsubseteq \bullet && (\underline{F}\eta)
   \end{aligned}$$

3. $+$:

   a. Retraction is

   $$\begin{aligned}
   & x : A_1 + A_2 \vdash \\
   & [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle]\!][\texttt{ret}[\![\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle]\!][x]] \\
   & = [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle]\!] \\
   & \qquad [\texttt{retcase } x\{x_1.\texttt{inl } [\![\langle A_1' \nwarrow A_1\rangle]\!][x_1] \mid x_1.\texttt{inr } [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2]\}] \\
   & \sqsupseteq\sqsubseteq \texttt{case } x && \text{(commuting conversion)} \\
   & \quad \{x_1.[\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle]\!][\texttt{retinl } [\![\langle A_1' \nwarrow A_1\rangle]\!][x_1]] \\
   & \quad \mid x_2.[\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle]\!][\texttt{retinr } [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2]]\} \\
   & \sqsupseteq\sqsubseteq \texttt{case } x && \text{(cast computation)} \\
   & \quad \{x_1.\texttt{bind } x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle]\!][\texttt{ret}[\![\langle A_1' \nwarrow A_1\rangle]\!]x_1]; \texttt{retinl } x_1 \\
   & \quad \mid x_2.\texttt{bind } x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle]\!][\texttt{ret}[\![\langle A_2' \nwarrow A_2\rangle]\!]x_2]; \texttt{retinr } x_2\} \\
   & \sqsupseteq\sqsubseteq \texttt{case } x\{x_1.\texttt{retinl } x_1 \mid x_2.\texttt{retinr } x_2\} && \text{(IH retraction)} \\
   & \sqsupseteq\sqsubseteq \texttt{ret} x && (+\eta)
   \end{aligned}$$

b. For Projection:

$$\bullet : A_1' + A_2' \vdash$$

$$\mathtt{bind}\ x \leftarrow \llbracket\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle\rrbracket; \llbracket\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle\rrbracket[x]$$

$$= \mathtt{bind}\ x \leftarrow (\mathtt{bind}\ x' \leftarrow \bullet; \mathtt{case}\ x'\{x_1'.\mathtt{bind}\ x_1 \leftarrow \llbracket\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle\rrbracket[\mathtt{ret}\ x_1'];$$

$$\quad \mathtt{ret}\ \mathtt{inl}\ x_1 \mid x_2'.\cdots\});$$

$$\llbracket\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle\rrbracket$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \bullet;'\ \mathtt{case}\ x' \qquad \text{(Commuting Conversion)}$$

$$\{x_1'.\mathtt{bind}\ x_1 \leftarrow \llbracket\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle\rrbracket[\mathtt{ret}x_1']; \llbracket\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle\rrbracket\mathtt{retinl}\ x_1$$

$$\mid x_2'.\mathtt{bind}\ x_2 \leftarrow \llbracket\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle\rrbracket[\mathtt{ret}x_2']; \llbracket\langle A_1' + A_2' \nwarrow A_1 + A_2\rangle\rrbracket\mathtt{retinr}\ x_2\}$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \bullet;'\ \mathtt{case}\ x' \qquad \text{(Cast Computation)}$$

$$\{x_1'.\mathtt{bind}\ x_1 \leftarrow \llbracket\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle\rrbracket[\mathtt{ret}x_1']; \mathtt{retinl}\ \llbracket\langle A_1' \nwarrow A_1\rangle\rrbracket x_1$$

$$\mid x_2'.\mathtt{bind}\ x_2 \leftarrow \llbracket\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle\rrbracket[\mathtt{ret}x_2']; \mathtt{retinr}\ \llbracket\langle A_2' \nwarrow A_2\rangle\rrbracket x_2\}$$

$$\sqsubseteq \mathtt{bind}\ x \leftarrow \bullet;'\ \mathtt{case}\ x'\{x_1'.\mathtt{retinl}\ x_1' \mid x_2'.\mathtt{retinr}\ x_2'\} \qquad \text{(IH projection)}$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \bullet;'\ \mathtt{ret}x' \qquad (+\eta)$$

$$\sqsupseteq\sqsubseteq \bullet \qquad (\underline{F}\eta)$$

4. $\times$:

a. First, Retraction:

$$x : A_1 \times A_2 \vdash$$

$$\llbracket\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle\rrbracket[\mathtt{ret}\llbracket\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle\rrbracket[x]]$$

$$= \llbracket\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle\rrbracket[\mathtt{retsplit}\ x\ \mathtt{to}\ (x_1, x_2).(\llbracket\langle A_1' \nwarrow A_1\rangle\rrbracket[x_1],$$

$$\llbracket\langle A_2' \nwarrow A_2\rangle\rrbracket[x_2])]$$

$$\sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (x_1, x_2).\llbracket\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle\rrbracket[\mathtt{ret}(\llbracket\langle A_1' \nwarrow A_1\rangle\rrbracket[x_1],$$

$$\llbracket\langle A_2' \nwarrow A_2\rangle\rrbracket[x_2])] \qquad \text{(commuting conversion)}$$

$$\sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (x_1, x_2). \qquad \text{(cast reduction)}$$

$$\mathtt{bind}\ y_1 \leftarrow \llbracket\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle\rrbracket[\mathtt{ret}\llbracket\langle A_1' \nwarrow A_1\rangle\rrbracket[x_1]];$$

$$\mathtt{bind}\ y_2 \leftarrow \llbracket\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle\rrbracket[\mathtt{ret}\llbracket\langle A_2' \nwarrow A_2\rangle\rrbracket[x_2]];$$

$$\mathtt{ret}(y_1, y_2)$$

$$\sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (x_1, x_2).\mathtt{bind}\ y_1 \leftarrow \mathtt{ret}x_1; \mathtt{bind}\ y_2 \leftarrow \mathtt{ret}x_2; \mathtt{ret}(y_1, y_2)$$

$$\text{(IH retraction)}$$

$$\sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (x_1, x_2).\mathtt{ret}(x_1, x_2) \qquad (\underline{F}\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{ret}x \qquad (\times\eta)$$

b. Next, Projection:

$$\bullet : \underline{F}A' \vdash$$

$$\mathtt{bind}\ x \leftarrow \llbracket\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle\rrbracket[\bullet]; \mathtt{ret}\llbracket\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle\rrbracket[x]$$

$$
\begin{aligned}
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && (\underline{F}\eta, \times\eta)\\
&\quad \texttt{bind}\ x \leftarrow [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A'_1 \times A'_2)\rangle]\!][\texttt{ret}(x'_1, x'_2)];\\
&\quad \texttt{ret}[\![\langle A'_1 \times A'_2 \nwarrow A_1 \times A_2\rangle]\!][x]\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && \text{(cast reduction)}\\
&\quad \texttt{bind}\ x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!][\texttt{ret}x'_1];\\
&\quad \texttt{bind}\ x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A'_2\rangle]\!][\texttt{ret}x'_2];\\
&\quad \texttt{ret}[\![\langle A'_1 \times A'_2 \nwarrow A_1 \times A_2\rangle]\!][(x_1, x_2)]\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && \text{(cast reduction)}\\
&\quad \texttt{bind}\ x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!][\texttt{ret}x'_1];\\
&\quad \texttt{bind}\ x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A'_2\rangle]\!][\texttt{ret}x'_2];\\
&\quad \texttt{ret}([\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1], [\![\langle A'_2 \nwarrow A_2\rangle]\!][x_2])\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && (\underline{F}\beta, \text{twice})\\
&\quad \texttt{bind}\ x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!][\texttt{ret}x'_1];\\
&\quad \texttt{bind}\ x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A'_2\rangle]\!][\texttt{ret}x'_2];\\
&\quad \texttt{bind}\ y'_2 \leftarrow \texttt{ret}[\![\langle A'_2 \nwarrow A_2\rangle]\!][x_2];\\
&\quad \texttt{bind}\ y'_1 \leftarrow \texttt{ret}[\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1];\\
&\quad \texttt{ret}(y'_1, y'_2)\\
&\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && \text{(IH Projection)}\\
&\quad \texttt{bind}\ x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!][\texttt{ret}x'_1];\\
&\quad \texttt{bind}\ y'_2 \leftarrow \texttt{ret}x'_2;\\
&\quad \texttt{bind}\ y'_1 \leftarrow \texttt{ret}[\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1];\\
&\quad \texttt{ret}(y'_1, y'_2)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && (\underline{F}\beta)\\
&\quad \texttt{bind}\ x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!][\texttt{ret}x'_1];\\
&\quad \texttt{bind}\ y'_1 \leftarrow \texttt{ret}[\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1];\\
&\quad \texttt{ret}(x'_1, y'_2)\\
&\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2). && \text{(IH Projection)}\\
&\quad \texttt{bind}\ y'_1 \leftarrow \texttt{ret}x'_1;\\
&\quad \texttt{ret}(x'_1, y'_2)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{split}\ x'\ \texttt{to}\ (x'_1, x'_2).\texttt{ret}(x'_1, x'_2) && (\underline{F}\beta)\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{ret}x' && (\times\eta)\\
&\sqsupseteq\sqsubseteq \bullet && (\underline{F}\eta)
\end{aligned}
$$

5. U: By inductive hypothesis, $(x.[\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!], \langle \underline{B} \swarrow \underline{B}'\rangle)$ is a computation ep pair

   a. To show retraction we need to prove:

$$x : U\underline{B} \vdash \texttt{ret}x \sqsupseteq\sqsubseteq \texttt{bind}\ y \leftarrow (\texttt{retthunk}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!]);$$

$$\texttt{retthunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\texttt{force}\ y] : \underline{F}U\underline{B}'$$

Which we calculate as follows:

$$
\begin{array}{ll}
x : U\underline{B} \vdash & \\
\llbracket\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle\rrbracket[(\texttt{ret}\llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[x])] & \\
\quad \sqsupseteq\sqsubseteq \texttt{retthunk}\ (\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[x]]) & \text{(Cast Reduction)} \\
\quad \sqsupseteq\sqsubseteq \texttt{retthunk force}\ x & \text{(IH Retraction)} \\
\quad \sqsupseteq\sqsubseteq \texttt{ret}x & (U\eta)
\end{array}
$$

b. To show projection we calculate:

$$
\begin{array}{ll}
\texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle\rrbracket[\bullet]; \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[x] & \\
\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle\rrbracket[\texttt{ret}x']; \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[x] & \\
 & (\underline{F}\eta) \\
\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \texttt{bind}\ x \leftarrow \texttt{retthunk}\ (\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{force}\ x']); & \\
\qquad \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[x] & \text{(Cast Reduction)} \\
\sqsupseteq\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[\texttt{thunk}\ (\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{force}\ x'])] & (\underline{F}\beta) \\
\sqsubseteq \texttt{bind}\ x' \leftarrow \bullet; x' & \text{(IH Projection)} \\
\sqsupseteq\sqsubseteq \bullet & (\underline{F}\eta)
\end{array}
$$

1. There's a few base cases about the dynamic computation type, then
2. $\top$:

   a. Retraction is by $\top\eta$:

$$z : U\top \vdash \texttt{force}\ z \sqsupseteq\sqsubseteq \{\} : \top$$

   b. Projection is

$$
\begin{array}{rll}
\texttt{thunk}\ \mho & \sqsubseteq \texttt{thunk force}\ w & (\mho \text{ is } \bot) \\
 & \sqsupseteq\sqsubseteq w & (U\eta)
\end{array}
$$

3. &:

   a. Retraction

$$
\begin{array}{ll}
z : U(\underline{B}_1 \mathbin{\&} \underline{B}_2) \vdash & \\
\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[z]] & \\
\ \sqsupseteq\sqsubseteq \{\pi \mapsto \pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[z]] & \\
 & (\&\eta) \\
\qquad \mid \pi' \mapsto \pi'\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[z]]\} & \\
\ \sqsupseteq\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle\rrbracket[\pi\texttt{force}\ \llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[z]] & \\
 & \text{(Cast reduction)} \\
\qquad \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \swarrow \underline{B}_2'\rangle\rrbracket[\pi'\texttt{force}\ \llbracket\langle U(\underline{B}_1' \mathbin{\&} \underline{B}_2') \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[z]]\} & \\
\ \sqsupseteq\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U\underline{B}_1' \nwarrow U\underline{B}_1\rangle\rrbracket[\texttt{thunk}\ \pi\texttt{force}\ z]] & \\
 & \text{(Cast reduction)} \\
\qquad \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \swarrow \underline{B}_2'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U\underline{B}_2' \nwarrow U\underline{B}_2\rangle\rrbracket[\texttt{thunk}\ \pi'\texttt{force}\ z]]\} &
\end{array}
$$

$$\begin{array}{lr}
\sqsupseteq\sqsubseteq \{\pi \mapsto \mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ z \mid \pi' \mapsto \mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ z\} & \\
 & \text{(IH retraction)} \\
\sqsupseteq\sqsubseteq \{\pi \mapsto \pi\mathtt{force}\ z \mid \pi' \mapsto \pi'\mathtt{force}\ z\} & (U\beta) \\
\sqsupseteq\sqsubseteq \mathtt{force}\ z & (\&\eta)
\end{array}$$

b. Projection

$$\begin{array}{lr}
w : U\underline{B}'_1 \mathbin{\&} \underline{B}'_2 \vdash & \\
\llbracket\langle U(\underline{B}'_1 \mathbin{\&} \underline{B}'_2) \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket[\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ \llbracket\langle U(\underline{B}'_1 \mathbin{\&} \underline{B}'_2) \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket & \\
[\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & (U\eta) \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \pi\mathtt{force}\ \llbracket\langle U(\underline{B}'_1 \mathbin{\&} \underline{B}'_2) \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket & \\
[\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & \\
\qquad\qquad \mid \pi' \mapsto \pi'\mathtt{force}\ \llbracket\langle U(\underline{B}'_1 \mathbin{\&} \underline{B}'_2) \nwarrow U(\underline{B}_1 \mathbin{\&} \underline{B}_2)\rangle\rrbracket\} & \\
[\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & (\&\eta) \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_1 \nwarrow U\underline{B}_1\rangle\rrbracket & \\
[\mathtt{thunk}\ \pi\mathtt{force}\ \mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & \\
\qquad\qquad \mid \pi' \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_2 \nwarrow U\underline{B}_2\rangle\rrbracket\} & \\
[\mathtt{thunk}\ \pi'\mathtt{force}\ \mathtt{thunk}\ \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & \text{(cast reduction)} \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_1 \nwarrow U\underline{B}_1\rangle\rrbracket & \\
\quad [\mathtt{thunk}\ \pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & (U\beta) \\
\qquad\qquad \mid \pi' \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_2 \nwarrow U\underline{B}_2\rangle\rrbracket\} & \\
\quad [\mathtt{thunk}\ \pi'\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \swarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\mathtt{force}\ w]] & \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_1 \nwarrow U\underline{B}_1\rangle\rrbracket[\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \swarrow \underline{B}'_1\rangle\rrbracket[\pi\mathtt{force}\ w]] & \\
 & \text{(cast reduction)} \\
\qquad\qquad \mid \pi' \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_2 \nwarrow U\underline{B}_2\rangle\rrbracket[\mathtt{thunk}\ \llbracket\langle \underline{B}_2 \swarrow \underline{B}'_2\rangle\rrbracket[\pi'\mathtt{force}\ w]]\} & \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_1 \nwarrow U\underline{B}_1\rangle\rrbracket & \\
\quad [\mathtt{thunk}\ \llbracket\langle \underline{B}_1 \swarrow \underline{B}'_1\rangle\rrbracket[\mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ w]] & (U\beta) \\
\qquad\qquad \mid \pi' \mapsto \mathtt{force}\ \llbracket\langle U\underline{B}'_2 \nwarrow U\underline{B}_2\rangle\rrbracket[\mathtt{thunk}\ \llbracket\langle \underline{B}_2 \swarrow \underline{B}'_2\rangle\rrbracket\} & \\
\qquad [\mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ w]] & \\
\ \sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ w \mid \pi' \mapsto \mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ w\} & \\
 & \text{(IH projection)} \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \pi\mathtt{force}\ w \mid \pi' \mapsto \pi'\mathtt{force}\ w\} & (U\beta) \\
\ \sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ w & (\&\eta) \\
\ \sqsupseteq\sqsubseteq w & (U\eta)
\end{array}$$

4. $\to$:

a. Retraction

$$\begin{array}{l}
z : U(A \to \underline{B}) \vdash \\
\llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket[\mathtt{force}\ \llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[z]]
\end{array}$$

$\sqsupseteq\sqsubseteq \lambda x : A.(\llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket[\texttt{force}\ \llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[z]])\, x$ $(\to \eta)$

$\sqsupseteq\sqsubseteq \lambda x : A.\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[(\texttt{force}\ \llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[z])(\llbracket\langle A' \nwarrow A\rangle\rrbracket[x])]$ (cast reduction)

$\sqsupseteq\sqsubseteq \lambda x : A.$ (cast reduction)
$\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{bind}\ y \leftarrow \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket[\texttt{ret}\langle A' \nwarrow A\rangle[x]];$
$\texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle[\texttt{thunk}\ ((\texttt{force}\ z)\, y)]]$

$\sqsupseteq\sqsubseteq \lambda x : A.\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{bind}\ y \leftarrow \texttt{ret} x; \texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle$
$[\texttt{thunk}\ ((\texttt{force}\ z)\, y)]]$ (IH Retraction)

$\sqsupseteq\sqsubseteq \lambda x : A.\llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B}\rangle[\texttt{thunk}\ ((\texttt{force}\ z)\, x)]]$ $(\underline{F}\beta)$

$\sqsupseteq\sqsubseteq \lambda x : A.\texttt{force thunk}\ ((\texttt{force}\ z)\, x)$ (IH retraction)

$\sqsupseteq\sqsubseteq \lambda x : A.(\texttt{force}\ z)\, x$ $(U\beta)$

$\sqsupseteq\sqsubseteq \texttt{force}\ z$ $(\to \eta)$

b. Projection

$w : U(A' \to \underline{B}') \vdash$
$\llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[\texttt{thunk}\ \llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket[\texttt{force}\ w]]$

$\sqsupseteq\sqsubseteq \texttt{thunk force}\ \llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[\texttt{thunk}\ \llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket$
$[\texttt{force}\ w]]$ $(U\eta)$

$\sqsupseteq\sqsubseteq \texttt{thunk}\ \lambda x' : A'.$
$(\texttt{force}\ \llbracket\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle\rrbracket[\texttt{thunk}\ \llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket$
$[\texttt{force}\ w]])\, x'$ $(\to \eta)$

$\sqsupseteq\sqsubseteq \texttt{thunk}\ \lambda x' : A'.$
$\texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket[\texttt{ret} x'];$ (cast reduction)
$\texttt{force}\ \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[\texttt{thunk}\ ((\texttt{force thunk}\ \llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket$
$[\texttt{force}\ w])\, x)]$

$\sqsupseteq\sqsubseteq \texttt{thunk}\ \lambda x' : A'.$
$\texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket[\texttt{ret} x'];$ $(U\beta)$
$\texttt{force}\ \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[\texttt{thunk}\ ((\llbracket\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\rrbracket[\texttt{force}\ w])\, x)]$

$\sqsupseteq\sqsubseteq \texttt{thunk}\ \lambda x' : A'.$
$\texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket[\texttt{ret} x'];$ (cast reduction)
$\texttt{force}\ \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[\texttt{thunk}\ \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[(\texttt{force}\ w)\, (\langle A' \nwarrow A\rangle[x])]]$

$\sqsupseteq\sqsubseteq \texttt{thunk}\ \lambda x' : A'.$
$\texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \swarrow \underline{F}A'\rangle\rrbracket[\texttt{ret} x'];$ $(\underline{F}\beta)$
$\texttt{bind}\ x' \leftarrow \texttt{ret}\langle A' \nwarrow A\rangle[x];$
$\texttt{force}\ \llbracket\langle U\underline{B}' \nwarrow U\underline{B}\rangle\rrbracket[\texttt{thunk}\ \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[(\texttt{force}\ w)\, x']]$

$$
\begin{array}{ll}
\sqsubseteq \mathtt{thunk}\ \lambda x' : A'. & \text{(IH projection)} \\
\quad \mathtt{bind}\ x' \leftarrow \mathtt{ret} x'; & \\
\quad \mathtt{force}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\, x']] & \\
\sqsupseteq\sqsubseteq \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\, x']] & \\
 & (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!] & \\
\quad [\mathtt{force}\ \mathtt{thunk}\ ((\mathtt{force}\ w)\, x')]] & (\underline{F}\beta) \\
\sqsubseteq \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ \mathtt{thunk}\ ((\mathtt{force}\ w)\, x') & \text{(IH projection)} \\
\sqsupseteq\sqsubseteq \mathtt{thunk}\ \lambda x' : A'.((\mathtt{force}\ w)\, x') & (U\beta) \\
\sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ w & (\to \eta) \\
\sqsupseteq\sqsubseteq w & (U\eta)
\end{array}
$$

5. $\underline{F}$:

   a. To show retraction we need to show

$$
z : U\underline{F}A \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ \mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z;
$$

$$
\mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!])]
$$

   We calculate:

$$
\begin{array}{ll}
[\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ \mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!])] & \\
\quad \sqsupseteq\sqsubseteq [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][(\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!])] & (U\beta) \\
\quad \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!]] & \text{(comm conv)} \\
\quad \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{ret} x & \text{(IH value retraction)} \\
\quad \sqsupseteq\sqsubseteq \mathtt{force}\ z & (\underline{F}\eta)
\end{array}
$$

   b. To show projection we need to show

$$
w : U\underline{F}A' \vdash \mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ \mathtt{thunk}\ [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ w];
$$

$$
\mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!]) \sqsubseteq w : U\underline{B}'
$$

   We calculate as follows

$$
\begin{array}{ll}
\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ \mathtt{thunk}\ [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ w]; \mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!]) & \\
\quad \sqsupseteq\sqsubseteq \mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ w]; \mathtt{ret}[\![\langle A' \nwarrow A\rangle]\!]) & (U\beta) \\
\quad \sqsubseteq \mathtt{thunk}\ \mathtt{force}\ w & \text{(IH value projection)} \\
\quad \sqsupseteq\sqsubseteq w & (U\eta)
\end{array}
$$

□

Proof of Lemma 5.11.

Proof. By symmetry it is sufficient to show $S_1 \sqsubseteq S_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_1 \sqsubseteq S_1}{\texttt{bind}\ x \leftarrow S_1; \texttt{ret}x \sqsubseteq \texttt{bind}\ x \leftarrow \bullet; S_1[\texttt{ret}x]}}{\texttt{bind}\ x \leftarrow S_1; \texttt{ret}V_e \sqsubseteq \texttt{bind}\ x \leftarrow \bullet; \texttt{ret}x}}{\texttt{bind}\ x \leftarrow S_1; \texttt{ret}x \sqsubseteq \texttt{bind}\ x \leftarrow \bullet; S_2[\texttt{ret}x]}}{\bullet : \underline{F}A' \vdash S_1 \sqsubseteq S_2 : \underline{F}A}$$

similarly to show $V_1 \sqsubseteq V_2$:

$$\dfrac{\dfrac{\dfrac{x : U\underline{B} \vdash \texttt{thunk force}\ V_2 \sqsubseteq \texttt{thunk force}\ V_2 : U\underline{B}'}{x : U\underline{B} \vdash \texttt{thunk force}\ x \sqsubseteq \texttt{thunk}\ S_p[\texttt{force}\ V_2]}}{x : U\underline{B} \vdash \texttt{thunk force}\ V_1 \sqsubseteq \texttt{thunk force}\ V_2 : U\underline{B}'}}{x : U\underline{B} \vdash V_1 \sqsubseteq V_2 : U\underline{B}'}$$

□

Proof of Lemma 5.12.

Proof. We proceed by induction on $A, \underline{B}$, following the proof that reflexivity is admissible given in Lemma 5.6.

1. If $A \in \{1, ?\}$, then $[\![\langle A \nwarrow A\rangle]\!][x] = x$.
2. If $A = 0$, then $\texttt{absurd}\ x \sqsupseteq\sqsubseteq x$ by 0η.
3. If $A = U\underline{B}$, then by inductive hypothesis $[\![\langle \underline{B} \swarrow \underline{B}\rangle]\!] \sqsupseteq\sqsubseteq \bullet$. By Lemma 5.9, $(x.x, \bullet)$ is a computation ep pair from $\underline{B}$ to itself. But by Lemma 5.10, $([\![\langle U\underline{B} \nwarrow U\underline{B}\rangle]\!][x], \bullet)$ is also a computation ep pair so the result follows by uniqueness of embeddings from computation projections Lemma 5.11.
4. If $A = A_1 \times A_2$ or $A = A_1 + A_2$, the result follows by the η principle and inductive hypothesis.
5. If $\underline{B} = \underline{¿}$, $[\![\langle \underline{¿} \swarrow \underline{¿}\rangle]\!] = \bullet$.
6. For $\underline{B} = \top$, the result follows by $\top\eta$.
7. For $\underline{B} = \underline{B}_1 \mathbin{\&} \underline{B}_2$ or $\underline{B} = A \to \underline{B}'$, the result follows by inductive hypothesis and η.
8. For $\underline{B} = \underline{F}A$, by inductive hypothesis, the downcast is a projection for the value embedding $x.x$, so the result follows by identity ep pair and uniqueness of projections from value embeddings. □

Proof of Lemma 5.13.

Proof. By mutual induction on $A, \underline{B}$.

1. $A \sqsubseteq A' \sqsubseteq A''$

   a. If $A = 0$, we need to show $x : 0 \vdash [\![\langle A'' \nwarrow 0\rangle]\!][x] \sqsupseteq\sqsubseteq [\![\langle A'' \nwarrow A'\rangle]\!][[\![\langle A' \nwarrow 0\rangle]\!][x]] : A''$ which follows by 0η.

   b. If $A = ?$, then $A' = A'' = ?$, and both casts are the identity.

   c. If $A \notin \{?, 0\}$ and $A' = ?$, then $A'' = ?$ and $[\![\langle ? \nwarrow ?\rangle]\!][[\![\langle ? \nwarrow A\rangle]\!]] = [\![\langle ? \nwarrow A\rangle]\!]$ by definition.

   d. If $A, A' \notin \{?, 0\}$ and $A'' = ?$, then $\lfloor A \rfloor = \lfloor A' \rfloor$, which we call G and

$$[\![\langle ? \nwarrow A\rangle]\!] = [\![\langle ? \nwarrow G\rangle]\!][[\![\langle G \nwarrow A\rangle]\!]]$$

and

$$\llbracket\langle ? \leftarrowtail A'\rangle\rrbracket[\llbracket\langle A' \leftarrowtail A\rangle\rrbracket] = \llbracket\langle ? \leftarrowtail G\rangle\rrbracket[\llbracket\langle G \leftarrowtail A'\rangle\rrbracket[\llbracket\langle A' \leftarrowtail A\rangle\rrbracket]]$$

so this reduces to the case for $A \sqsubseteq A' \sqsubseteq G$, below.

e. If $A, A', A'' \not\in \{?, 0\}$, then they all have the same top-level constructor:

i.$+$: We need to show for $A_1 \sqsubseteq A_1' \sqsubseteq A_1''$ and $A_2 \sqsubseteq A_2' \sqsubseteq A_2''$:

$$\begin{aligned} x : \llbracket A_1\rrbracket + \llbracket A_2\rrbracket \vdash\ & \llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\llbracket\langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle\rrbracket[x]] \\ & \sqsupseteq\sqsubseteq \llbracket\langle A_1'' + A_2'' \leftarrowtail A_1 + A_2\rangle\rrbracket[x] : \llbracket A_1''\rrbracket + \llbracket A_2''\rrbracket \end{aligned}$$

We proceed as follows:

$$\begin{aligned} & \llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\llbracket\langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle\rrbracket[x]] \\ & \sqsupseteq\sqsubseteq \mathtt{case}\ x && (+\eta) \\ & \qquad \{x_1.\llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\llbracket\langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle\rrbracket[\mathtt{inl}\ x_1]] \\ & \qquad \mid x_2.\llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\llbracket\langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle\rrbracket[\mathtt{inr}\ x_2]]\} \\ & \sqsupseteq\sqsubseteq \mathtt{case}\ x && \text{(cast reduction)} \\ & \qquad \{x_1.\llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\mathtt{inl}\ \llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x_1]] \\ & \qquad \mid x_2.\llbracket\langle A_1'' + A_2'' \leftarrowtail A_1' + A_2'\rangle\rrbracket[\mathtt{inr}\ \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[x_2]]\} \\ & \sqsupseteq\sqsubseteq \mathtt{case}\ x && \text{(cast reduction)} \\ & \qquad \{x_1.\mathtt{inl}\ \llbracket\langle A_1'' \leftarrowtail A_1'\rangle\rrbracket[\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x_1]] \\ & \qquad \mid x_2.\mathtt{inr}\ \llbracket\langle A_2'' \leftarrowtail A_2'\rangle\rrbracket[\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[x_2]]\} \\ & \sqsupseteq\sqsubseteq \mathtt{case}\ x && \text{(IH)} \\ & \qquad \{x_1.\mathtt{inl}\ \llbracket\langle A_1'' \leftarrowtail A_1\rangle\rrbracket[x_1] \\ & \qquad \mid x_2.\mathtt{inr}\ \llbracket\langle A_2'' \leftarrowtail A_2\rangle\rrbracket[x_2]\} \\ & = \llbracket\langle A_1'' + A_2'' \leftarrowtail A_1 + A_2\rangle\rrbracket[x] && \text{(definition)} \end{aligned}$$

ii.1: By definition both sides are the identity.

iii.$\times$: We need to show for $A_1 \sqsubseteq A_1' \sqsubseteq A_1''$ and $A_2 \sqsubseteq A_2' \sqsubseteq A_2''$:

$$\begin{aligned} x : \llbracket A_1\rrbracket \times \llbracket A_2\rrbracket \vdash\ & \llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1' \times A_2'\rangle\rrbracket[\llbracket\langle A_1' \times A_2' \leftarrowtail A_1 \times A_2\rangle\rrbracket[x]] \\ & \sqsupseteq\sqsubseteq \llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1 \times A_2\rangle\rrbracket[x] : \llbracket A_1''\rrbracket \times \llbracket A_2''\rrbracket. \end{aligned}$$

We proceed as follows:

$$\begin{aligned} & \llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1' \times A_2'\rangle\rrbracket[\llbracket\langle A_1' \times A_2' \leftarrowtail A_1 \times A_2\rangle\rrbracket[x]] \\ & \sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (y,z).\llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1' \times A_2'\rangle\rrbracket[\llbracket\langle A_1' \times A_2' \leftarrowtail A_1 \times A_2\rangle\rrbracket[(y,z)]] \\ & && (\times\eta) \\ & \sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (y,z).\llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1' \times A_2'\rangle\rrbracket[(\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[y], \\ & \qquad \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[z])] && \text{(cast reduction)} \\ & \sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (y,z).(\llbracket\langle A_1'' \leftarrowtail A_1'\rangle\rrbracket[\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[y]], \llbracket\langle A_2'' \leftarrowtail A_2'\rangle\rrbracket \\ & \qquad [\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[z]]) && \text{(cast reduction)} \\ & \sqsupseteq\sqsubseteq \mathtt{split}\ x\ \mathtt{to}\ (y,z).(\llbracket\langle A_1'' \leftarrowtail A_1\rangle\rrbracket[y], \llbracket\langle A_2'' \leftarrowtail A_2\rangle\rrbracket[z]) && \text{(IH)} \\ & = \llbracket\langle A_1'' \times A_2'' \leftarrowtail A_1 \times A_2\rangle\rrbracket[x] && \text{(definition)} \end{aligned}$$

iv.$U\underline{B} \sqsubseteq U\underline{B}' \sqsubseteq U\underline{B}''$. We need to show

$$x : U\underline{B} \vdash \llbracket\langle U\underline{B}'' \rightarrowtail U\underline{B}'\rangle\rrbracket[\llbracket\langle U\underline{B}' \rightarrowtail U\underline{B}\rangle\rrbracket[x]] \sqsupseteq\sqsubseteq \llbracket\langle U\underline{B}'' \rightarrowtail U\underline{B}\rangle\rrbracket[x] : U\underline{B}''$$

By composition of ep pairs, we know

$$(x.\llbracket\langle U\underline{B}'' \rightarrowtail U\underline{B}'\rangle\rrbracket[\llbracket\langle U\underline{B}' \rightarrowtail U\underline{B}\rangle\rrbracket[x]], \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket\langle \underline{B}' \twoheadleftarrow \underline{B}''\rangle\rrbracket])$$

is a computation ep pair. Furthermore, by inductive hypothesis, we know

$$\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket\langle \underline{B}' \twoheadleftarrow \underline{B}''\rangle\rrbracket] \sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}''\rangle\rrbracket$$

so then both sides form ep pairs paired with $\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}''\rangle\rrbracket$, so it follows because computation projections determine embeddings Lemma 5.11.

2. $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$

a. If $\underline{B} = \top$, then the result is immediate by $\eta\top$.
b. If $\underline{B} = \underline{¿}$, then $\underline{B}' = \underline{B}'' = \underline{¿}$ then both sides are just $\bullet$.
c. If $\underline{B} \not\in \{\underline{¿}, \top\}$, and $\underline{B}' = \underline{¿}$, then $\underline{B}'' = \underline{¿}$

$$\llbracket\langle \underline{B} \twoheadleftarrow \underline{¿}\rangle\rrbracket[\llbracket\langle \underline{¿} \twoheadleftarrow \underline{¿}\rangle\rrbracket] = \llbracket\langle \underline{B} \twoheadleftarrow \underline{¿}\rangle\rrbracket$$

d. If $\underline{B}, \underline{B}' \not\in \{\underline{¿}, \top\}$, and $\underline{B}'' = \underline{¿}$, and $\lfloor\underline{B}\rfloor = \lfloor\underline{B}'\rfloor$, which we call $\underline{G}$. Then we need to show

$$\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket\langle \underline{B}' \twoheadleftarrow \underline{G}\rangle\rrbracket[\llbracket\langle \underline{G} \twoheadleftarrow \underline{¿}\rangle\rrbracket]] \sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{G}\rangle\rrbracket[\llbracket\langle \underline{G} \twoheadleftarrow [\rangle\rrbracket\underline{¿}]]$$

so the result follows from the case $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{G}$, which is handled below.

e. If $\underline{B}, \underline{B}', \underline{B}'' \not\in \{\underline{¿}, \top\}$, then they all have the same top-level constructor:

i.& We are given $\underline{B}_1 \sqsubseteq \underline{B}'_1 \sqsubseteq \underline{B}''_1$ and $\underline{B}_2 \sqsubseteq \underline{B}'_2 \sqsubseteq \underline{B}''_2$ and we need to show

$$\bullet : \underline{B}''_1 \mathbin{\&} \underline{B}''_2 \vdash \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket] : \underline{B}_1 \mathbin{\&} \underline{B}_2$$

We proceed as follows:

$$\begin{aligned}
&\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket] \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket] && (\&\eta) \\
&\quad \mid \pi' \mapsto \pi'\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}'_1 \mathbin{\&} \underline{B}'_2\rangle\rrbracket[\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}'_1\rangle\rrbracket[\pi\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket] && \text{(cast reduction)} \\
&\quad \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \twoheadleftarrow \underline{B}'_2\rangle\rrbracket[\pi'\llbracket\langle \underline{B}'_1 \mathbin{\&} \underline{B}'_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}'_1\rangle\rrbracket[\llbracket\langle \underline{B}'_1 \twoheadleftarrow \underline{B}''_1\rangle\rrbracket[\pi\bullet]] && \text{(cast reduction)} \\
&\quad \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \twoheadleftarrow \underline{B}'_2\rangle\rrbracket\llbracket\langle \underline{B}'_2 \twoheadleftarrow \underline{B}''_2\rangle\rrbracket[\pi'\bullet]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}''_1\rangle\rrbracket[\pi\bullet] \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \twoheadleftarrow \underline{B}''_2\rangle\rrbracket[\pi'\bullet]\} && \text{(IH)} \\
&= \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}''_1 \mathbin{\&} \underline{B}''_2\rangle\rrbracket && \text{(definition)}
\end{aligned}$$

ii.→, assume we are given $A \sqsubseteq A' \sqsubseteq A''$ and $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$, then we proceed:

$$\begin{aligned}
&\llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\llbracket\langle A' \to \underline{B}' \twoheadleftarrow A'' \to \underline{B}''\rangle\rrbracket] \\
&\sqsupseteq\sqsubseteq \lambda x : A.(\llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\llbracket\langle A' \to \underline{B}' \twoheadleftarrow A'' \to \underline{B}''\rangle\rrbracket[\bullet])\, x && (\to \eta)
\end{aligned}$$

$$\sqsupseteq\sqsubseteq \lambda x : A.⟦\langle \underline{B} \twoheadleftarrow \underline{B}' \rangle⟧[(⟦\langle A' \to \underline{B}' \twoheadleftarrow A'' \to \underline{B}'' \rangle⟧[\bullet])\; ⟦\langle A' \leftarrowtail A \rangle⟧[x]]$$

(cast reduction)

$$\sqsupseteq\sqsubseteq \lambda x : A.⟦\langle \underline{B} \twoheadleftarrow \underline{B}' \rangle⟧[⟦\langle \underline{B}' \twoheadleftarrow \underline{B}'' \rangle⟧[\bullet\; ⟦\langle A'' \leftarrowtail A' \rangle⟧[⟦\langle A' \leftarrowtail A \rangle⟧[x]]]]$$

(cast reduction)

$$\sqsupseteq\sqsubseteq \lambda x : A.⟦\langle \underline{B} \twoheadleftarrow \underline{B}'' \rangle⟧[\bullet\; ⟦\langle A'' \leftarrowtail A \rangle⟧[x]]$$

$$= ⟦\langle A \to \underline{B} \twoheadleftarrow A \to \underline{B}'' \rangle⟧[\bullet] \qquad \text{(definition)}$$

iii. $\underline{F}A \sqsubseteq \underline{F}A' \sqsubseteq \underline{F}A''$. First, by composition of ep pairs, we know

$$(x.⟦\langle A'' \leftarrowtail A' \rangle⟧[⟦\langle A' \leftarrowtail A \rangle⟧[x]], ⟦\langle \underline{F}A \twoheadleftarrow \underline{F}A' \rangle⟧)[⟦\langle \underline{F}A' \twoheadleftarrow \underline{F}A'' \rangle⟧]$$

form a value ep pair. Furthermore, by inductive hypothesis, we know

$$x : A \vdash ⟦\langle A'' \leftarrowtail A' \rangle⟧[⟦\langle A' \leftarrowtail A \rangle⟧[x]] \sqsupseteq\sqsubseteq ⟦\langle A'' \leftarrowtail A \rangle⟧[x]$$

so the two sides of our equation are both projections with the same value embedding, so the equation follows from uniqueness of projections from value embeddings. □

Proof of Lemma 5.14.

Proof.

1. Assume $\texttt{ret}\, V_e[V] \sqsubseteq M : \underline{F}A'$. Then by retraction, $\texttt{ret}\, V \sqsubseteq S_p[\texttt{ret}\, V_e[V]]$ so by transitivity, the result follows by substitution:

$$\frac{S_p \sqsubseteq S_p \qquad \texttt{ret}\, V_e[V] \sqsubseteq M}{S_p[\texttt{ret}\, V_e[V]] \sqsubseteq M}$$

2. Assume $\texttt{ret}\, V \sqsubseteq S_p[M] : \underline{F}A$. Then by projection, $\texttt{bind}\ x \leftarrow S_p[M]; \texttt{ret}\, V_e[x] \sqsubseteq M$, so it is sufficient to show

$$\texttt{ret}\, V_e[V] \sqsubseteq \texttt{bind}\ x \leftarrow S_p[M]; \texttt{ret}\, V_e[x]$$

but again by substitution we have

$$\texttt{bind}\ x \leftarrow \texttt{ret}\, V; \texttt{ret}\, V_e[x] \sqsubseteq \texttt{bind}\ x \leftarrow S_p[M]; \texttt{ret}\, V_e[x]$$

and by $\underline{F}\beta$, the LHS is equivalent to $\texttt{ret}\, V_e[V]$.

3. Assume $z' : U\underline{B}' \vdash M \sqsubseteq S[S_p[\texttt{force}\ z']]$, then by projection, $S[S_p[\texttt{force}\ V_e]] \sqsubseteq S[\texttt{force}\ z]$ and by substitution:

$$\frac{M \sqsubseteq S[S_p[\texttt{force}\ z']] \qquad V_e \sqsubseteq V_e \qquad S[S_p[\texttt{force}\ V_e]] = (S[S_p[\texttt{force}\ z']])[V_e/z']}{M[V_e/z'] \sqsubseteq S[S_p[\texttt{force}\ V_e]]}$$

4. Assume $z : U\underline{B} \vdash M[V_e/z'] \sqsubseteq S[\texttt{force}\ z]$. Then by retraction, $M \sqsubseteq M[V_e[\texttt{thunk}\ S_p[\texttt{force}\ z]]]$ and by substitution:

$$M[V_e[\texttt{thunk}\ S_p[\texttt{force}\ z]]] \sqsubseteq S[\texttt{force}\ \texttt{thunk}\ S_p[\texttt{force}\ z]]$$

and the right is equivalent to $S[S_p[\texttt{force}\ z]]$ by $U\beta$. □

Proof of Theorem 5.8 (Axiomatic Graduality).

Proof. By mutual induction over term precision derivations. For the β, η and reflexivity rules, we use the identity expansion lemma and the corresponding β, η rule of CBPV* Lemma 5.12.

For compatibility rules a pattern emerges. Universal rules (positive intro, negative elim) are easy, we don't need to reason about casts at all. For "(co)-pattern matching rules" (positive elim, negative intro), we need to invoke the η principle (or commuting conversion, which is derived from the η principle). In all compatibility cases, the cast reduction lemma keeps the proof straightforward.

Fortunately, all reasoning about "shifted" casts is handled in lemmas, and here we only deal with the "nice" value upcasts/stack downcasts.

1. Transitivity for values: The GTT rule is

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi' : \Gamma' \sqsubseteq \Gamma'' \qquad \Phi'' : \Gamma \sqsubseteq \Gamma'' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \qquad \Phi' \vdash V' \sqsubseteq V'' : A' \sqsubseteq A''}{\Phi'' \vdash V \sqsubseteq V'' : A \sqsubseteq A''}$$

Which under translation (and the same assumptions about the contexts) is

$$\frac{\llbracket\Gamma\rrbracket \vdash \llbracket\langle A' \nwarrow A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] : \llbracket A'\rrbracket \qquad \llbracket\Gamma'\rrbracket \vdash \llbracket\langle A' \nwarrow A'\rangle\rrbracket[\llbracket V'\rrbracket] \sqsubseteq \llbracket V''\rrbracket[\llbracket\Phi'\rrbracket] : \llbracket A''\rrbracket}{\llbracket\Gamma\rrbracket \vdash \llbracket\langle A'' \nwarrow A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V''\rrbracket[\llbracket\Phi''\rrbracket] : \llbracket A''\rrbracket}$$

We proceed as follows, the key lemma here is the cast decomposition lemma:

$$\begin{aligned}
\llbracket\langle A'' \nwarrow A\rangle\rrbracket[\llbracket V\rrbracket] &\sqsupseteq\sqsubseteq \llbracket\langle A'' \nwarrow A'\rangle\rrbracket[\llbracket\langle A' \nwarrow A\rangle\rrbracket[\llbracket V\rrbracket]] && \text{(cast decomposition)}\\
&\sqsubseteq \llbracket\langle A'' \nwarrow A'\rangle\rrbracket[\llbracket V'\rrbracket[\llbracket\Phi\rrbracket]] && \text{(IH)}\\
&\sqsubseteq \llbracket V''\rrbracket[\llbracket\Phi'\rrbracket][\llbracket\Phi\rrbracket] && \text{(IH)}\\
&\sqsupseteq\sqsubseteq \llbracket V''\rrbracket[\llbracket\Phi''\rrbracket] && \text{(cast decomposition)}
\end{aligned}$$

2. Transitivity for terms: The GTT rule is

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \quad \Phi' : \Gamma' \sqsubseteq \Gamma'' \quad \Phi'' : \Gamma \sqsubseteq \Gamma'' \quad \Psi : \Delta \sqsubseteq \Delta' \quad \Psi : \Delta' \sqsubseteq \Delta'' \quad \Psi'' : \Delta \sqsubseteq \Delta'' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}' \qquad \Phi' \mid \Psi' \vdash M' \sqsubseteq M'' : \underline{B}' \sqsubseteq \underline{B}''}{\Phi'' \mid \Psi'' \vdash M \sqsubseteq M'' : \underline{B} \sqsubseteq \underline{B}''}$$

Which under translation (and the same assumptions about the contexts) is

$$\frac{\llbracket\Gamma\rrbracket \mid \llbracket\Delta'\rrbracket \vdash \llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{B}\rrbracket \qquad [5pt]\llbracket\Gamma'\rrbracket \mid \llbracket\Delta''\rrbracket \vdash \llbracket M'\rrbracket[\llbracket\Psi'\rrbracket] \sqsubseteq \llbracket\langle \underline{B}' \swarrow \underline{B}''\rangle\rrbracket[\llbracket M''\rrbracket[\llbracket\Phi'\rrbracket]] : \llbracket\underline{B}'\rrbracket}{\llbracket\Gamma\rrbracket \mid \llbracket\Delta''\rrbracket \vdash \llbracket M\rrbracket[\llbracket\Psi''\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}''\rangle\rrbracket[\llbracket M''\rrbracket[\llbracket\Phi''\rrbracket]] : \llbracket\underline{B}\rrbracket}$$

We proceed as follows, the key lemma here is the cast decomposition lemma:

$$\begin{aligned}
\llbracket M\rrbracket[\llbracket\Psi''\rrbracket] &\sqsupseteq\sqsubseteq \llbracket M\rrbracket[\llbracket\Psi\rrbracket][\llbracket\Psi'\rrbracket] && \text{(Cast decomposition)}\\
&\sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Psi'\rrbracket][\llbracket\Phi\rrbracket]] && \text{(IH)}\\
&\sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}'\rangle\rrbracket[\llbracket\langle \underline{B}' \swarrow \underline{B}''\rangle\rrbracket[\llbracket M''\rrbracket[\llbracket\Phi'\rrbracket][\llbracket\Phi\rrbracket]]] && \text{(IH)}\\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \swarrow \underline{B}''\rangle\rrbracket[\llbracket M''\rrbracket[\llbracket\Phi''\rrbracket]] && \text{(Cast decomposition)}
\end{aligned}$$

3. Substitution of a value in a value: The GTT rule is

$$\frac{\Phi, x \sqsubseteq x' : A_1 \sqsubseteq A_1' \vdash V_2 \sqsubseteq V_2' : A_2 \sqsubseteq A_2' \qquad \Phi \vdash V_1 \sqsubseteq V_1' : A_1 \sqsubseteq A_1'}{\Phi \vdash V_2[V_1/x] \sqsubseteq V_2'[V_1'/x'] : A_2 \sqsubseteq A_2'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$. Under translation, we need to show

$$\frac{\begin{array}{c}\llbracket\Gamma\rrbracket, x : \llbracket A_1\rrbracket \vdash \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[\llbracket V_2\rrbracket] \sqsubseteq \llbracket V_2'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x]/x'] : \llbracket A_2'\rrbracket \\ \llbracket\Gamma\rrbracket \vdash \llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[\llbracket V_1\rrbracket] \sqsubseteq \llbracket V_1'\rrbracket[\llbracket\Phi\rrbracket] : \llbracket A_1'\rrbracket\end{array}}{\llbracket\Gamma\rrbracket \vdash \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[\llbracket V_2[V_1/x]\rrbracket] \sqsubseteq \llbracket V_2'[V_1'/x']\rrbracket[\llbracket\Phi\rrbracket] : \llbracket A_2'\rrbracket}$$

Which follows by compositionality:

$$\begin{aligned}
\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[\llbracket V_2[V_1/x]\rrbracket] &= (\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[\llbracket V_2\rrbracket])[\llbracket V_1\rrbracket/x] && \text{(Compositionality)}\\
&\sqsubseteq \llbracket V_2'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x]/x'][\llbracket V_1\rrbracket/x] && \text{(IH)}\\
&= \llbracket V_2'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[\llbracket V_1\rrbracket]/x']\\
&\sqsubseteq \llbracket V_2'\rrbracket[\llbracket\Phi\rrbracket][\llbracket V_1'\rrbracket[\llbracket\Phi\rrbracket]/x'] && \text{(IH)}\\
&= \llbracket V_2'[V_1'/x']\rrbracket[\llbracket\Phi\rrbracket]
\end{aligned}$$

4. Substitution of a value in a term: The GTT rule is

$$\frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \vdash M[V/x] \sqsubseteq M'[V'/x'] : \underline{B} \sqsubseteq \underline{B}'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$ and $\Psi : \Delta \sqsubseteq \Delta'$. Under translation this is:

$$\frac{\begin{array}{c}\llbracket\Gamma\rrbracket, x : \llbracket A\rrbracket \mid \llbracket\Delta\rrbracket \vdash \llbracket M\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket[x]/x']] : \llbracket\underline{B}\rrbracket \\ \llbracket\Gamma\rrbracket \vdash \llbracket\langle A' \leftarrowtail A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] : \llbracket A'\rrbracket\end{array}}{\llbracket\Gamma\rrbracket \mid \llbracket\Delta\rrbracket \vdash \llbracket M[V/x]\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'[V'/x']\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{B}\rrbracket}$$

Which follows from compositionality of the translation:

$$\begin{aligned}
\llbracket M[V/x]\rrbracket &= \llbracket M\rrbracket[\llbracket V\rrbracket/x] && \text{(Compositionality)}\\
&\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket[x]/x']][\llbracket V\rrbracket/x] && \text{(IH)}\\
&= \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket[\llbracket V\rrbracket]/x']]\\
&\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket V'\rrbracket[\llbracket\Phi\rrbracket]/x']] && \text{(IH)}\\
&= \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'[V'/x']\rrbracket[\llbracket\Phi\rrbracket]] && \text{(Compositionality)}
\end{aligned}$$

5. Substitution of a term in a stack: The GTT rule is

$$\frac{\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash S \sqsubseteq S' : \underline{C} \sqsubseteq \underline{C}' \qquad \Phi \mid \cdot \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \cdot \vdash S[M] \sqsubseteq S'[M'] : \underline{C} \sqsubseteq \underline{C}'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$. Under translation this is

$$\frac{\begin{array}{c}\llbracket\Gamma\rrbracket \mid \bullet : \llbracket\underline{B}'\rrbracket \vdash \llbracket S\rrbracket[\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\bullet]] \sqsubseteq \llbracket\langle \underline{C} \twoheadleftarrow \underline{C}'\rangle\rrbracket[\llbracket S'\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{C}\rrbracket \\ \llbracket\Gamma\rrbracket \mid \cdot \vdash \llbracket M\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{B}\rrbracket\end{array}}{\llbracket\Gamma\rrbracket \mid \cdot \vdash \llbracket S[M]\rrbracket \sqsubseteq \llbracket\langle \underline{C} \twoheadleftarrow \underline{C}'\rangle\rrbracket[\llbracket S'[M']\rrbracket[\llbracket\Phi\rrbracket]] : \llbracket\underline{C}\rrbracket}$$

We follows easily using compositionality of the translation:

$$\begin{aligned}
[\![S[M]]\!] &= [\![S]\!][[\![M]\!]] && \text{(Compositionality)}\\
&\sqsubseteq [\![S]\!][[\![\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]]]] && \text{(IH)}\\
&\sqsubseteq [\![\langle \underline{C} \twoheadleftarrow \underline{C}'\rangle]\!][[\![S']\!][[\![\Phi]\!]][[\![M']\!][[\![\Phi]\!]]]] && \text{(IH)}\\
&= [\![\langle \underline{C} \twoheadleftarrow \underline{C}'\rangle]\!][[\![S'[M']]\!][[\![\Phi]\!]]] && \text{(Compositionality)}
\end{aligned}$$

6. Variables: The GTT rule is

$$\Gamma_1 \sqsubseteq \Gamma_1', x \sqsubseteq x' : A \sqsubseteq A', \Gamma_2 \sqsubseteq \Gamma_2' \vdash x \sqsubseteq x' : A \sqsubseteq A'$$

which under translation is

$$[\![\Gamma_1]\!], x : [\![A]\!], [\![\Gamma_2]\!] \vdash [\![\langle A' \leftarrowtail A\rangle]\!][x] \sqsubseteq [\![\langle A' \leftarrowtail A\rangle]\!][x] : [\![A']\!]$$

which is an instance of reflexivity.

7. Hole: The GTT rule is

$$\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$$

which under translation is

$$[\![\Gamma]\!] \mid \bullet : \underline{B}' \vdash [\![\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle]\!][\bullet] \sqsubseteq [\![\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle]\!][\bullet] : \underline{B}$$

which is an instance of reflexivity.

8. Error is bottom: The GTT axiom is

$$\Phi \vdash \mho \sqsubseteq M : \underline{B}$$

where $\Phi : \Gamma \sqsubseteq \Gamma'$, so we need to show

$$[\![\Gamma]\!] \vdash \mho \sqsubseteq [\![\langle \underline{B} \twoheadleftarrow \underline{B}\rangle]\!][[\![M]\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]$$

which is an instance of the error is bottom axiom of CBPV.

9. Error strictness: The GTT axiom is

$$\Phi \vdash S[\mho] \sqsubseteq \mho : \underline{B}$$

where $\Phi : \Gamma \sqsubseteq \Gamma'$, which under translation is

$$[\![\Gamma]\!] \vdash [\![S]\!][\mho] \sqsubseteq [\![\langle \underline{B} \twoheadleftarrow \underline{B}\rangle]\!][\mho] : [\![\underline{B}]\!]$$

By strictness of stacks in CBPV, both sides are equivalent to $\mho$, so it follows by reflexivity.

10. UpCast-L: The GTT axiom is

$$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \leftarrowtail A\rangle x \sqsubseteq x' : A'$$

which under translation is

$$x : [\![A]\!] \vdash [\![\langle A' \leftarrowtail A'\rangle]\!][[\![\langle A' \leftarrowtail A\rangle]\!][x]] \sqsubseteq [\![\langle A' \leftarrowtail A\rangle]\!][x] : A'$$

Which follows by identity expansion and reflexivity.

11. UpCast-R: The GTT axiom is

$$x : A \vdash x \sqsubseteq \langle A' \leftarrowtail A\rangle x : A \sqsubseteq A'$$

which under translation is

$$x : \llbracket A \rrbracket \vdash \llbracket \langle A' \leftarrowtail A\rangle \rrbracket[x] \sqsubseteq \llbracket \langle A' \leftarrowtail A\rangle \rrbracket[\llbracket \langle A \leftarrowtail A\rangle \rrbracket[x]] : \llbracket A' \rrbracket$$

which follows by identity expansion and reflexivity.

12. DnCast-R: The GTT axiom is

$$\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle : \underline{B}$$

Which under translation is

$$\bullet : \llbracket \underline{B}' \rrbracket \vdash \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\bullet] \sqsubseteq \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}\rangle \rrbracket[\llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\bullet]] : \llbracket \underline{B} \rrbracket$$

Which follows by identity expansion and reflexivity.

13. DnCast-L: The GTT axiom is

$$\bullet : \underline{B}' \vdash \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$$

So under translation we need to show

$$\bullet : \llbracket \underline{B}' \rrbracket \vdash \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\llbracket \langle \underline{B}' \twoheadleftarrow \underline{B}'\rangle \rrbracket[\bullet]] \sqsubseteq \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket \bullet : \llbracket \underline{B} \rrbracket$$

Which follows immediately by reflexivity and the lemma that identity casts are identities.

14. 0 elim, we do the term case, the value case is similar

$$\frac{\langle 0 \leftarrowtail 0\rangle[\llbracket V \rrbracket] \sqsubseteq \llbracket V' \rrbracket[\llbracket \Phi \rrbracket]}{\texttt{absurd}\ \llbracket V \rrbracket \sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \texttt{absurd}\ \llbracket V' \rrbracket[\llbracket \Phi \rrbracket]}$$

Immediate by 0η.

15. $+$ intro, we do the `inl` case, the `inr` case is the same:

$$\frac{\llbracket \langle A_1' \leftarrowtail A_1\rangle \rrbracket[\llbracket V \rrbracket] \sqsubseteq \llbracket V' \rrbracket[\llbracket \Phi \rrbracket]}{\llbracket \langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle \rrbracket[\texttt{inl}\ \llbracket V \rrbracket] \sqsubseteq \texttt{inl}\ \llbracket V' \rrbracket[\llbracket \Phi \rrbracket]}$$

Which follows easily:

$$\begin{aligned} \llbracket \langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle \rrbracket[\texttt{inl}\ \llbracket V \rrbracket] &\sqsupseteq\sqsubseteq \texttt{inl}\ \llbracket \langle A_1' \leftarrowtail A_1\rangle \rrbracket \llbracket V \rrbracket && \text{(cast reduction)} \\ &\sqsubseteq \texttt{inl}\ \llbracket V' \rrbracket[\llbracket \Phi \rrbracket] && \text{(IH)} \end{aligned}$$

16. $+$ elim, we do just the cases where the continuations are terms:

$$\frac{\begin{array}{c} \llbracket \langle A_1' + A_2' \leftarrowtail A_1 + A_2\rangle \rrbracket[\llbracket V \rrbracket] \sqsubseteq \llbracket V' \rrbracket[\llbracket \Phi \rrbracket] \\ \llbracket M_1 \rrbracket[\llbracket \Psi \rrbracket] \sqsubseteq \llbracket M_1' \rrbracket[\llbracket \Phi \rrbracket][\llbracket \langle A_1' \leftarrowtail A_1\rangle \rrbracket[x_1]/x_1'] \qquad \llbracket M_2 \rrbracket[\llbracket \Psi \rrbracket] \sqsubseteq \llbracket M_2' \rrbracket[\llbracket \Phi \rrbracket][\llbracket \langle A_2' \leftarrowtail A_2\rangle \rrbracket[x_2]/x_2'] \end{array}}{\texttt{case}\ \llbracket V \rrbracket\{x_1.\llbracket M_1 \rrbracket[\llbracket \Psi \rrbracket] \mid x_2.\llbracket M_2 \rrbracket[\llbracket \Psi \rrbracket]\} \sqsubseteq \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\texttt{case}\ \llbracket V \rrbracket'[\llbracket \Phi \rrbracket]\{x_1'.\llbracket M_1' \rrbracket[\llbracket \Phi \rrbracket] \mid x_2'.\llbracket M_2' \rrbracket[\llbracket \Phi \rrbracket]\}]}$$

$$\begin{aligned} &\texttt{case}\ \llbracket V \rrbracket\{x_1.\llbracket M_1 \rrbracket[\llbracket \Psi \rrbracket] \mid x_2.\llbracket M_2 \rrbracket[\llbracket \Psi \rrbracket]\} \\ &\sqsubseteq \llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\texttt{case}\ \llbracket V \rrbracket\{x_1.\llbracket M_1' \rrbracket[\llbracket \Phi \rrbracket][\llbracket \langle A_1' \leftarrowtail A_1\rangle \rrbracket[x_1]/x_1'] \mid x_2.\llbracket M_2' \rrbracket \\ &\qquad [\llbracket \Phi \rrbracket][\llbracket \langle A_2' \leftarrowtail A_2\rangle \rrbracket[x_2]/x_2']\}] && \text{(IH)} \\ &\sqsupseteq\sqsubseteq \texttt{case}\ \llbracket V \rrbracket && \text{(comm conv)} \\ &\qquad \{x_1.\llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\llbracket M_1' \rrbracket[\llbracket \Phi \rrbracket][\llbracket \langle A_1' \leftarrowtail A_1\rangle \rrbracket[x_1]/x_1']] \\ &\qquad \mid x_2.\llbracket \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle \rrbracket[\llbracket M_2' \rrbracket[\llbracket \Phi \rrbracket][\llbracket \langle A_2' \leftarrowtail A_2\rangle \rrbracket[x_2]/x_2']]\} \end{aligned}$$

$$\begin{aligned}
&\sqsupseteq\sqsubseteq \texttt{case}\ \llbracket V\rrbracket && (+\beta)\\
&\qquad \{x_1.\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case inl}\ \llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket x_1\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\\
&\qquad \mid x_2.\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case inr}\ \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket x_2\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\}\\
&\sqsupseteq\sqsubseteq \texttt{case}\ \llbracket V\rrbracket && \text{(cast reduction)}\\
&\{x_1.\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case}\ \llbracket\langle A_1'+A_2' \leftarrowtail A_1+A_2\rangle\rrbracket\texttt{inl}\ x_1\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\\
&\mid x_2.\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case}\ \llbracket\langle A_1'+A_2' \leftarrowtail A_1+A_2\rangle\rrbracket\texttt{inr}\ x_2\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\}\\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case}\ \llbracket\langle A_1'+A_2' \leftarrowtail A_1+A_2\rangle\rrbracket[\llbracket V\rrbracket]\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\\
&\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{case}\ \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]\{x_1'.\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid x_2'.\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}] && \text{(IH)}
\end{aligned}$$

17. 1 intro:

$$\llbracket\langle 1 \leftarrowtail 1\rangle\rrbracket[()] \sqsubseteq ()$$

Immediate by cast reduction.

18. 1 elim (continuations are terms case):

$$\frac{\llbracket\langle 1 \leftarrowtail 1\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] \qquad \llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}{\texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ ().\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle[\texttt{split}\ \llbracket V\rrbracket'[\llbracket\Phi\rrbracket]\ \texttt{to}\ ().\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}$$

which follows by identity expansion Lemma 5.12.

19. × intro:

$$\frac{\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket\llbracket V_1\rrbracket \sqsubseteq \llbracket V_1'[\llbracket\Phi\rrbracket]\rrbracket \qquad \llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket\llbracket V_2\rrbracket \sqsubseteq \llbracket V_2'[\llbracket\Phi\rrbracket]\rrbracket}{\llbracket\langle A_1'\times A_2' \leftarrowtail A_1\times A_2\rangle\rrbracket[(\llbracket V_1\rrbracket,\llbracket V_2\rrbracket)] \sqsubseteq (\llbracket V_1'[\llbracket\Phi\rrbracket]\rrbracket,\llbracket V_2'[\llbracket\Phi\rrbracket]\rrbracket)}$$

We proceed:

$$\begin{aligned}
\llbracket\langle A_1'\times A_2' \leftarrowtail A_1\times A_2\rangle\rrbracket[(\llbracket V_1\rrbracket,\llbracket V_2\rrbracket)] &\sqsupseteq\sqsubseteq (\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket\llbracket V_1\rrbracket,\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket\llbracket V_2\rrbracket)\\
&\qquad\qquad \text{(cast reduction)}\\
&\sqsubseteq (\llbracket V_1'[\llbracket\Phi\rrbracket]\rrbracket,\llbracket V_2'[\llbracket\Phi\rrbracket]\rrbracket) \qquad \text{(IH)}
\end{aligned}$$

20. × elim: We show the case where the continuations are terms, the value continuations are no different:

$$\frac{\begin{gathered}\llbracket\langle A_1'\times A_2' \leftarrowtail A_1\times A_2\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]\\ \llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x]/x'][\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[y]/y']]\end{gathered}}{\texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ (x,y).\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \langle \underline{B} \twoheadleftarrow \underline{B}'\rangle[\texttt{split}\ \llbracket V\rrbracket'[\llbracket\Phi\rrbracket]\ \texttt{to}\ (x',y').\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}$$

We proceed as follows:

$$\begin{aligned}
&\texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ (x,y).\llbracket M\rrbracket[\llbracket\Psi\rrbracket]\\
&\sqsubseteq \texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ (x,y).\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x]/x']\\
&[\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[y]/y']] && \text{(IH)}\\
&\sqsupseteq\sqsubseteq \texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ (x,y). && (\times\beta)\\
&\qquad \texttt{split}\ (\llbracket\langle A_1' \leftarrowtail A_1\rangle\rrbracket[x],\llbracket\langle A_2' \leftarrowtail A_2\rangle\rrbracket[y])\ \texttt{to}\ (x',y').\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]\\
&\sqsupseteq\sqsubseteq \texttt{split}\ \llbracket V\rrbracket\ \texttt{to}\ (x,y). && \text{(cast reduction)}
\end{aligned}$$

$$\begin{aligned}
&\texttt{split } \llbracket\langle A_1' \times A_2' \leftarrowtail A_1 \times A_2'\rangle\rrbracket[(x,y)] \texttt{ to } (x',y').\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]\\
&\sqsupseteq\sqsubseteq \texttt{split } \llbracket\langle A_1' \times A_2' \leftarrowtail A_1 \times A_2\rangle\rrbracket[\llbracket V\rrbracket] \texttt{ to } (x',y').\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]\\
&\hfill(\times\eta)\\
&\sqsubseteq \texttt{split } \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] \texttt{ to } (x',y').\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] \qquad \text{(IH)}\\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{split } \llbracket V'\rrbracket[\llbracket\Phi\rrbracket] \texttt{ to } (x',y').\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]\\
&\hfill\text{(commuting conversion)}
\end{aligned}$$

21. U intro:

$$\frac{\llbracket M\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}{\llbracket\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle\rrbracket[\texttt{thunk } \llbracket M\rrbracket] \sqsubseteq \texttt{thunk } \llbracket M'\rrbracket[\llbracket\Phi\rrbracket]}$$

We proceed as follows:

$$\begin{aligned}
\llbracket\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle\rrbracket[\texttt{thunk } \llbracket M\rrbracket] &\sqsubseteq \llbracket\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle\rrbracket[\texttt{thunk } \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]]\\
&\hfill\text{(IH)}\\
&\sqsubseteq \texttt{thunk } \llbracket M'\rrbracket[\llbracket\Phi\rrbracket] \qquad \text{(alt projection)}
\end{aligned}$$

22. U elim:

$$\frac{\llbracket\langle U\underline{B}' \leftarrowtail U\underline{B}\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]}{\texttt{force } \llbracket V\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket\texttt{force } \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]}$$

By hom-set formulation of adjunction Lemma 5.14.

23. $\top$ intro:

$$\{\} \sqsubseteq \llbracket\langle \top \twoheadleftarrow \top\rangle\rrbracket[\{\}]$$

Immediate by $\top\eta$

24. & intro:

$$\frac{\llbracket M_1\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}_1'\rangle\rrbracket[\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket]] \qquad \llbracket M_2\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B}_2 \twoheadleftarrow \underline{B}_2'\rangle\rrbracket[\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]]}{\{\pi \mapsto \llbracket M_1\rrbracket[\llbracket\Psi\rrbracket] \mid \pi' \mapsto \llbracket M_2\rrbracket[\llbracket\Psi\rrbracket]\} \sqsubseteq \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\{\pi \mapsto \llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid \pi' \mapsto \llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]}$$

We proceed as follows:

$$\begin{aligned}
&\{\pi \mapsto \llbracket M_1\rrbracket[\llbracket\Psi\rrbracket] \mid \pi' \mapsto \llbracket M_2\rrbracket[\llbracket\Psi\rrbracket]\}\\
&\sqsubseteq \{\pi \mapsto \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}_1'\rangle\rrbracket[\llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket]] \mid \pi' \mapsto \llbracket\langle \underline{B}_2 \twoheadleftarrow \underline{B}_2'\rangle\rrbracket[\llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]]\} \qquad \text{(IH)}\\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\{\pi \mapsto \llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid \pi' \mapsto \llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\\
&\hfill\text{(cast reduction)}\\
&\quad \mid \pi' \mapsto \pi'\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\{\pi \mapsto \llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid \pi' \mapsto \llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}]\}\\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\{\pi \mapsto \llbracket M_1'\rrbracket[\llbracket\Phi\rrbracket] \mid \pi' \mapsto \llbracket M_2'\rrbracket[\llbracket\Phi\rrbracket]\}] \qquad (\&\eta)
\end{aligned}$$

25. & elim, we show the π case, π' is symmetric:

$$\frac{\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}{\pi\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}_1'\rangle\rrbracket[\pi\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}$$

We proceed as follows:

$$\begin{aligned}
\pi\llbracket M\rrbracket[\llbracket\Psi\rrbracket] &\sqsubseteq \pi\llbracket\langle \underline{B}_1 \mathbin{\&} \underline{B}_2 \twoheadleftarrow \underline{B}_1' \mathbin{\&} \underline{B}_2'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] \qquad \text{(IH)}\\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B}_1 \twoheadleftarrow \underline{B}_1'\rangle\rrbracket[\pi\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] \qquad \text{(cast reduction)}
\end{aligned}$$

26.

$$\frac{\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket x/x']]}{\lambda x : A.\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\lambda x' : A'.\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]]}$$

We proceed as follows:

$$\begin{aligned}
&\lambda x : A.\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \\
&\sqsubseteq \lambda x : A.\llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket x/x']] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq \lambda x : A.(\llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\lambda x'.\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]])\, x && \text{(cast reduction)} \\
&\sqsupseteq\sqsubseteq \llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\lambda x'.\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] && (\to \eta)
\end{aligned}$$

27. We need to show

$$\frac{\begin{array}{c}\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]] \\ \llbracket\langle A' \leftarrowtail A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]\end{array}}{\llbracket M\rrbracket[\llbracket\Psi\rrbracket]\ \llbracket V\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]\ \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]]}$$

We proceed:

$$\begin{aligned}
&\llbracket M\rrbracket[\llbracket\Psi\rrbracket]\ \llbracket V\rrbracket \\
&\sqsubseteq (\llbracket\langle A \to \underline{B} \twoheadleftarrow A' \to \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]])\ \llbracket V\rrbracket && \text{(IH)} \\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]\ (\llbracket\langle A' \leftarrowtail A\rangle\rrbracket\llbracket V\rrbracket)] && \text{(cast reduction)} \\
&\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket M'\rrbracket[\llbracket\Phi\rrbracket]\ \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]] && \text{(IH)}
\end{aligned}$$

28. We need to show

$$\frac{\llbracket\langle A' \leftarrowtail A\rangle\rrbracket[\llbracket V\rrbracket] \sqsubseteq \llbracket V'\rrbracket[\llbracket\Phi\rrbracket]}{\texttt{ret}\llbracket V\rrbracket \sqsubseteq \llbracket\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\rrbracket[\texttt{ret}\llbracket V'\rrbracket[\llbracket\Phi\rrbracket]]}$$

By hom-set definition of adjunction Lemma 5.14

29. We need to show

$$\frac{\begin{array}{c}\llbracket M\rrbracket[\llbracket\Psi\rrbracket] \sqsubseteq \llbracket\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\rrbracket[\llbracket M'\rrbracket[\Phi]] \\ \llbracket N\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket N\rrbracket[\Phi][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket x/x']]\end{array}}{\texttt{bind}\ x \leftarrow \llbracket M\rrbracket[\llbracket\Psi\rrbracket];\ \llbracket N\rrbracket \sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{bind}\ x' \leftarrow \llbracket M'\rrbracket[\llbracket\Phi\rrbracket];\ \llbracket N'\rrbracket[\llbracket\Phi\rrbracket]]}$$

We proceed:

$$\begin{aligned}
&\texttt{bind}\ x \leftarrow \llbracket M\rrbracket[\llbracket\Psi\rrbracket];\ \llbracket N\rrbracket \\
&\sqsubseteq \texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\rrbracket[\llbracket M'\rrbracket[\Phi]];\ \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket N\rrbracket[\Phi][\llbracket\langle A' \leftarrowtail A\rangle\rrbracket x/x']] \\
& && \text{(IH, congruence)} \\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \llbracket\langle \underline{F}A \twoheadleftarrow \underline{F}A'\rangle\rrbracket[\llbracket M'\rrbracket[\Phi]]; \\
&\qquad \texttt{bind}\ x' \leftarrow \texttt{ret}\llbracket\langle A' \leftarrowtail A\rangle\rrbracket[x];\ \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket N\rrbracket[\Phi]] && (\underline{F}\beta) \\
&\sqsubseteq \texttt{bind}\ x' \leftarrow \llbracket M'\rrbracket[\Phi];\ \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\llbracket N\rrbracket[\Phi]] && \text{(Projection)} \\
&\sqsupseteq\sqsubseteq \llbracket\langle \underline{B} \twoheadleftarrow \underline{B}'\rangle\rrbracket[\texttt{bind}\ x' \leftarrow \llbracket M'\rrbracket[\Phi];\ \llbracket N\rrbracket[\Phi]] && \text{(commuting conversion)}
\end{aligned}$$

□

$$\Gamma, x : A, \Gamma' \vdash x \sqsubseteq x : A \qquad \Gamma \mid \bullet : \underline{B} \vdash \bullet \sqsubseteq \bullet : \underline{B} \qquad \Gamma \vdash \mho \sqsubseteq \mho : \underline{B}$$

$$\dfrac{\Gamma \vdash V \sqsubseteq V' : A \qquad \Gamma, x : A \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \texttt{let}\ x = V; M \sqsubseteq \texttt{let}\ x = V'; M' : \underline{B}} \qquad \dfrac{\Gamma \vdash V \sqsubseteq V' : 0}{\Gamma \vdash \texttt{abort}\ V \sqsubseteq \texttt{abort}\ V' : \underline{B}}$$

$$\dfrac{\Gamma \vdash V \sqsubseteq V' : A_1}{\Gamma \vdash \texttt{inl}\ V \sqsubseteq \texttt{inl}\ V' : A_1 + A_2} \qquad \dfrac{\Gamma \vdash V \sqsubseteq V' : A_2}{\Gamma \vdash \texttt{inr}\ V \sqsubseteq \texttt{inr}\ V' : A_1 + A_2}$$

$$\dfrac{\Gamma \vdash V \sqsubseteq V' : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash M_1 \sqsubseteq M_1' : \underline{B} \qquad \Gamma, x_2 : A_2 \vdash M_2 \sqsubseteq M_2' : \underline{B}}{\Gamma \vdash \texttt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \sqsubseteq \texttt{case}\ V'\{x_1.M_1' \mid x_2.M_2'\} : \underline{B}} \qquad \Gamma \vdash () \sqsubseteq () : 1$$

$$\dfrac{\Gamma \vdash V_1 \sqsubseteq V_1' : A_1 \qquad \Gamma \vdash V_2 \sqsubseteq V_2' : A_2}{\Gamma \vdash (V_1, V_2) \sqsubseteq (V_1', V_2') : A_1 \times A_2} \qquad \dfrac{\Gamma \vdash V \sqsubseteq V' : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \texttt{split}\ V\ \texttt{to}\ (x, y).M \sqsubseteq \texttt{split}\ V'\ \texttt{to}\ (x, y).M' : \underline{B}}$$

$$\dfrac{\Gamma \vdash V \sqsubseteq V' : A[\mu X.A/X]}{\Gamma \vdash \texttt{roll}_{\mu X.A}\ V \sqsubseteq \texttt{roll}_{\mu X.A}\ V' : \mu X.A}$$

$$\dfrac{\Gamma \vdash V \sqsubseteq V' : \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \texttt{unroll}\ V\ \texttt{to roll}\ x.M \sqsubseteq \texttt{unroll}\ V'\ \texttt{to roll}\ x.M' : \underline{B}}$$

$$\dfrac{\Gamma \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \texttt{thunk}\ M \sqsubseteq \texttt{thunk}\ M' : U\underline{B}} \qquad \dfrac{\Gamma \vdash V \sqsubseteq V' : U\underline{B}}{\Gamma \vdash \texttt{force}\ V \sqsubseteq \texttt{force}\ V' : \underline{B}} \qquad \dfrac{\Gamma \vdash V \sqsubseteq V' : A}{\Gamma \vdash \texttt{ret} V \sqsubseteq \texttt{ret} V' : \underline{F}A}$$

$$\dfrac{\Gamma \vdash M \sqsubseteq M' : \underline{F}A \qquad \Gamma, x : A \vdash N \sqsubseteq N' : \underline{B}}{\Gamma \vdash \texttt{bind}\ x \leftarrow M; N \sqsubseteq \texttt{bind}\ x \leftarrow M'; N' : \underline{B}} \qquad \dfrac{\Gamma, x : A \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \lambda x : A.M \sqsubseteq \lambda x : A.M' : A \to \underline{B}}$$

$$\dfrac{\Gamma \vdash M \sqsubseteq M' : A \to \underline{B} \qquad \Gamma \vdash V \sqsubseteq V' : A}{\Gamma \vdash M\ V \sqsubseteq M'\ V' : \underline{B}}$$

$$\dfrac{\Gamma \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \qquad \Gamma \vdash M_2 \sqsubseteq M_2' : \underline{B}_2}{\Gamma \vdash \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} \sqsubseteq \{\pi \mapsto M_1' \mid \pi' \mapsto M_2'\} : \underline{B}_1 \mathbin{\&} \underline{B}_2} \qquad \dfrac{\Gamma \vdash M \sqsubseteq M' : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \vdash \pi M \sqsubseteq \pi M' : \underline{B}_1}$$

$$\dfrac{\Gamma \vdash M \sqsubseteq M' : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \vdash \pi' M \sqsubseteq \pi' M' : \underline{B}_2} \qquad \dfrac{\Gamma \vdash M \sqsubseteq M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \vdash \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M \sqsubseteq \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M' : \nu \underline{Y}.\underline{B}}$$

$$\dfrac{\Gamma \vdash M \sqsubseteq M' : \nu \underline{Y}.\underline{B}}{\Gamma \vdash \texttt{unroll}\ M \sqsubseteq \texttt{unroll}\ M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}$$

Fig. E.1. CBPV inequational theory (congruence rules).

E Proofs for Section 6

Proof of Lemma 6.3.

Proof.

$$\begin{aligned}
&\texttt{bind}\ x \leftarrow M; \texttt{bind}\ y \leftarrow N; N' \\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow M; \texttt{bind}\ y \leftarrow N; \texttt{bind}\ x \leftarrow \texttt{force thunk ret} x; N' && (U\beta, \underline{F}\beta)
\end{aligned}$$

$$\texttt{case inl } V\{x_1.M_1 \mid x_2.M_2\} \sqsupseteq\sqsubseteq M_1[V/x_1] \qquad \texttt{case inr } V\{x_1.M_1 \mid x_2.M_2\} \sqsupseteq\sqsubseteq M_2[V/x_2]$$

$$\frac{\Gamma, x : A_1 + A_2 \vdash M : \underline{B}}{\Gamma, x : A_1 + A_2 \vdash M \sqsupseteq\sqsubseteq \texttt{case } x\{x_1.M[\texttt{inl } x_1/x] \mid x_2.M[\texttt{inr } x_2/x]\} : \underline{B}}$$

$$\texttt{split } (V_1, V_2)\,\texttt{to } (x_1, x_2).M \sqsupseteq\sqsubseteq M[V_1/x_1, V_2/x_2]$$

$$\frac{\Gamma, x : A_1 \times A_2 \vdash M : \underline{B}}{\Gamma, x : A_1 \times A_2 \vdash M \sqsupseteq\sqsubseteq \texttt{split } x\,\texttt{to } (x_1, x_2).M[(x_1, x_2)/x] : \underline{B}} \qquad \frac{\Gamma, x : 1 \vdash M : \underline{B}}{\Gamma, x : 1 \vdash M \sqsupseteq\sqsubseteq M[()/x] : \underline{B}}$$

$$\texttt{unroll roll}_A\ V\,\texttt{to roll } x.M \sqsupseteq\sqsubseteq M[V/x]$$

$$\frac{\Gamma, x : \mu X.A \vdash M : \underline{B}}{\Gamma, x : \mu X.A \vdash M \sqsupseteq\sqsubseteq \texttt{unroll } x\,\texttt{to roll } y.M[\texttt{roll}_{\mu X.A}\ y/x] : \underline{B}} \qquad \texttt{force thunk } M \sqsupseteq\sqsubseteq M$$

$$\frac{\Gamma \vdash V : U\underline{B}}{\Gamma \vdash V \sqsupseteq\sqsubseteq \texttt{thunk force } V : U\underline{B}} \qquad \texttt{let } x = V; M \sqsupseteq\sqsubseteq M[V/x]$$

$$\texttt{bind } x \leftarrow \texttt{ret}V; M \sqsupseteq\sqsubseteq M[V/x] \qquad \Gamma \mid \bullet : \underline{F}A \vdash \bullet \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \bullet; \texttt{ret}x : \underline{F}A$$

$$(\lambda x : A.M)\ V \sqsupseteq\sqsubseteq M[V/x] \qquad \frac{\Gamma \vdash M : A \to \underline{B}}{\Gamma \vdash M \sqsupseteq\sqsubseteq \lambda x : A.M\,x : A \to \underline{B}} \qquad \pi\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M$$

$$\pi'\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M' \qquad \frac{\Gamma \vdash M : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \vdash M \sqsupseteq\sqsubseteq \{\pi \mapsto \pi M \mid \pi' \mapsto \pi' M\} : \underline{B}_1 \mathbin{\&} \underline{B}_2}$$

$$\frac{\Gamma \vdash M : \top}{\Gamma \vdash M \sqsupseteq\sqsubseteq \{\} : \top} \qquad \texttt{unroll roll}_{\underline{B}}\ M \sqsupseteq\sqsubseteq M \qquad \frac{\Gamma \vdash M : \nu\underline{Y}.\underline{B}}{\Gamma \vdash M \sqsupseteq\sqsubseteq \texttt{roll}_{\nu\underline{Y}.\underline{B}}\ \texttt{unroll } M : \nu\underline{Y}.\underline{B}}$$

Fig. E.2. CBPV β, η rules.

$$\Gamma \vdash \mho \sqsubseteq M : \underline{B} \qquad \Gamma \vdash S[\mho] \sqsupseteq\sqsubseteq \mho : \underline{B} \qquad \Gamma \vdash M \sqsubseteq M : \underline{B} \qquad \Gamma \vdash V \sqsubseteq V : A \qquad \Gamma \mid \underline{B} \vdash S \sqsubseteq S : \underline{B}'$$

$$\frac{\Gamma \vdash M_1 \sqsubseteq M_2 : \underline{B} \qquad \Gamma \vdash M_2 \sqsubseteq M_3 : \underline{B}}{\Gamma \vdash M_1 \sqsubseteq M_3 : \underline{B}} \qquad \frac{\Gamma \vdash V_1 \sqsubseteq V_2 : A \qquad \Gamma \vdash V_2 \sqsubseteq V_3 : A}{\Gamma \vdash V_1 \sqsubseteq V_3 : A}$$

$$\frac{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \mid \underline{B} \vdash S_2 \sqsubseteq S_3 : \underline{B}'}{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_3 : \underline{B}'} \qquad \frac{\Gamma, x : A \vdash M_1 \sqsubseteq M_2 : \underline{B} \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \vdash M_1[V_1/x] \sqsubseteq M_2[V_2/x] : \underline{B}}$$

$$\frac{\Gamma, x : A \vdash V_1' \sqsubseteq V_2' : A' \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \vdash V_1'[V_1/x] \sqsubseteq V_2'[V_2/x] : A'} \qquad \frac{\Gamma, x : A \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \mid \underline{B} \vdash S_1[V_1/x] \sqsubseteq S_2[V_2/x] : \underline{B}'}$$

$$\frac{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \vdash M_1 \sqsubseteq M_2 : \underline{B}}{\Gamma \vdash S_1[M_1] \sqsubseteq S_2[M_2] : \underline{B}'} \qquad \frac{\Gamma \mid \underline{B}' \vdash S_1' \sqsubseteq S_2' : \underline{B}'' \qquad \Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}'}{\Gamma \mid \underline{B} \vdash S_1'[S_1] \sqsubseteq S_2'[S_2] : \underline{B}''}$$

Fig. E.3. CBPV logical and error rules.

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{bind}\ w \leftarrow \mathtt{retthunk}\ \mathtt{ret}x; \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N' \qquad (\underline{F}\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ w \leftarrow (\mathtt{bind}\ x \leftarrow M; \mathtt{retthunk}\ \mathtt{ret}x); \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N' \qquad (\underline{F}\eta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ w \leftarrow \mathtt{retthunk}\ M; \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N' \qquad (M \text{ thunkable})$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow \mathtt{force}\ \mathtt{thunk}\ M; N' \qquad (\underline{F}\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow M; N' \qquad (U\beta)$$

□

Proof of Lemma 6.4.

Proof.

$$\mathtt{bind}\ y \leftarrow (\mathtt{bind}\ x \leftarrow M; N); \mathtt{retthunk}\ \mathtt{ret}y$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{bind}\ y \leftarrow N; \mathtt{retthunk}\ \mathtt{ret}y \qquad (\underline{F}\eta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{retthunk}\ N \qquad (N \text{ thunkable})$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{ret}x; N) \qquad (\underline{F}\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{bind}\ w \leftarrow \mathtt{retthunk}\ \mathtt{ret}x; \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N) \qquad (\underline{F}\beta, U\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ w \leftarrow (\mathtt{bind}\ x \leftarrow M; \mathtt{retthunk}\ \mathtt{ret}x); \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N) \qquad (\underline{F}\eta)$$

$$\sqsupseteq\sqsubseteq \mathtt{bind}\ w \leftarrow \mathtt{retthunk}\ M; \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ w; N) \qquad (M \text{ thunkable})$$

$$\sqsupseteq\sqsubseteq \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ \mathtt{thunk}\ M; N) \qquad (\underline{F}\beta)$$

$$\sqsupseteq\sqsubseteq \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow M; N) \qquad (U\beta)$$

□

Proof of Lemma 6.6.

Proof. Introduction forms follow from return is thunkable and thunkables compose. For elimination forms it is sufficient to show that when the branches of pattern matching are thunkable, the pattern match is thunkable.

1. x: We need to show $x^{\dagger} = \mathtt{ret}x$ is thunkable, which we proved as a lemma above.
2. 0 elim, we need to show

   $$\mathtt{bind}\ y \leftarrow \mathtt{absurd}\ V; \mathtt{retthunk}\ \mathtt{ret}y \sqsupseteq\sqsubseteq \mathtt{retthunk}\ \mathtt{absurd}\ V$$

   but by $\eta 0$ both sides are equivalent to $\mathtt{absurd}\ V$.
3. $+$ elim, we need to show

   $$\mathtt{retthunk}\ (\mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\}) \sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow (\mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\});$$
   $$\mathtt{retthunk}\ \mathtt{ret}y$$

$$\begin{aligned}
&\texttt{retthunk (case } V\{x_1.M_1 \mid x_2.M_2\})\\
&\sqsupseteq\sqsubseteq \texttt{case } V && (+\eta)\\
&\qquad \{x_1.\texttt{retthunk (case inl } x_1\{x_1.M_1 \mid x_2.M_2\})\\
&\qquad\ \mid x_2.\texttt{retthunk (case inr } x_2\{x_1.M_1 \mid x_2.M_2\})\}\\
&\sqsupseteq\sqsubseteq \texttt{case } V && (+\beta)\\
&\qquad \{x_1.\texttt{retthunk } M_1\\
&\qquad\ \mid x_2.\texttt{retthunk } M_2\}\\
&\sqsupseteq\sqsubseteq \texttt{case } V && (M_1, M_2 \text{ thunkable})\\
&\qquad \{x_1.\texttt{bind } y \leftarrow M_1; \texttt{retthunk ret} y\\
&\qquad\ \mid x_2.\texttt{bind } y \leftarrow M_2; \texttt{retthunk ret} y\}\\
&\sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow (\texttt{case } V\{x_1.M_1 \mid x_2.M_2\}); \texttt{retthunk ret} y\\
& && (\text{commuting conversion})
\end{aligned}$$

4. $\times$ elim

$$\begin{aligned}
&\texttt{retthunk (split } V \texttt{ to } (x,y).M)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } (x,y).\texttt{retthunk split } (x,y) \texttt{ to } (x,y).M && (\times\eta)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } (x,y).\texttt{retthunk } M && (\times\beta)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } (x,y).\texttt{bind } z \leftarrow M; \texttt{retthunk ret} z && (M \text{ thunkable})\\
&\sqsupseteq\sqsubseteq \texttt{bind } z \leftarrow (\texttt{split } V \texttt{ to } (x,y).M); \texttt{retthunk ret} z\\
& && (\text{commuting conversion})
\end{aligned}$$

5. 1 elim

$$\begin{aligned}
&\texttt{retthunk (split } V \texttt{ to } ().xyM)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } ().\texttt{retthunk split } () \texttt{ to } ().M && (1\eta)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } ().\texttt{retthunk } M && (1\beta)\\
&\sqsupseteq\sqsubseteq \texttt{split } V \texttt{ to } ().\texttt{bind } z \leftarrow M; \texttt{retthunk ret} z && (M \text{ thunkable})\\
&\sqsupseteq\sqsubseteq \texttt{bind } z \leftarrow (\texttt{split } V \texttt{ to } ().M); \texttt{retthunk ret} z && (\text{commuting conversion})
\end{aligned}$$

6. μ elim

$$\begin{aligned}
&\texttt{retthunk (unroll } V \texttt{ to roll } x.M)\\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.\texttt{retthunk unroll roll } x \texttt{ to roll } x.M && (\mu\eta)\\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.\texttt{retthunk } M && (\mu\beta)\\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.\texttt{bind } y \leftarrow M; \texttt{retthunk ret} y && (M \text{ thunkable})\\
&\sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow (\texttt{unroll } V \texttt{ to roll } x.M); \texttt{retthunk ret} y\\
& && (\text{commuting conversion})
\end{aligned}$$

□

Proof of Lemma 6.8.

Proof.

$$\begin{aligned}
&N[\texttt{thunk } M/y][\texttt{thunk (bind } x \leftarrow \texttt{force } z; \texttt{force } x)/x]\\
&= N[\texttt{thunk } (M[\texttt{thunk (bind } x \leftarrow \texttt{force } z; \texttt{force } x)])/y]
\end{aligned}$$

$$\begin{aligned}
&\sqsupseteq\sqsubseteq N[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M)/y] && (M \text{ linear})\\
&\sqsupseteq\sqsubseteq N[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ \mathtt{thunk}\ M)/y] && (U\beta)\\
&\sqsupseteq\sqsubseteq N[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{bind}\ y \leftarrow \mathtt{retthunk}\ M; \mathtt{force}\ y)/y] && (\underline{F}\beta)\\
&\sqsupseteq\sqsubseteq N[\mathtt{thunk}\ (\mathtt{bind}\ y \leftarrow (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{retthunk}\ M); \mathtt{force}\ y)/y] && (\underline{F}\eta)\\
&\sqsupseteq\sqsubseteq N[\mathtt{thunk}\ (\mathtt{bind}\ y \leftarrow \mathtt{force}\ w; \mathtt{force}\ y)/y][\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z;\\
&\qquad \mathtt{retthunk}\ M)/w] && (U\beta)\\
&\sqsupseteq\sqsubseteq (\mathtt{bind}\ y \leftarrow \mathtt{force}\ w; N)[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{retthunk}\ M)/w]\\
& && (N \text{ linear})\\
&\sqsupseteq\sqsubseteq (\mathtt{bind}\ y \leftarrow (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{retthunk}\ M); N) && (U\beta)\\
&\sqsupseteq\sqsubseteq (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{bind}\ y \leftarrow \mathtt{retthunk}\ M; N && (\underline{F}\eta)\\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; N[\mathtt{thunk}\ M/y]
\end{aligned}$$

□

Proof of Lemma 6.9.

Proof. There are 4 classes of rules for complex stacks: those that are rules for simple stacks (•, computation type elimination forms), introduction rules for negative computation types where the subterms are complex stacks, elimination of positive value types where the continuations are complex stacks and finally application to a complex value.

The rules for simple stacks are easy: they follow immediately from the fact that forcing to a stack is linear and that complex stacks compose. For the negative introduction forms, we have to show that binding commutes with introduction forms. For pattern matching forms, we just need commuting conversions. For function application, we use the lemma that binding a thunkable in a linear term is linear.

1. •: This is just saying that $\mathtt{force}\ z$ is linear, which we showed above.
2. → elim We need to show, assuming that $\Gamma, x : \underline{B} \vdash M : \underline{C}$ is linear in x and $\Gamma \vdash N : \underline{F}A$ is thunkable, that

   $$\mathtt{bind}\ y \leftarrow N; M\, y$$

   is linear in x.

   $$\begin{aligned}
   &\mathtt{bind}\ y \leftarrow N; (M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)/x])\, y\\
   &\sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow N; (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M)\, y && (M \text{ linear in } x)\\
   &\sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow N; \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M\, y && (\underline{F}\eta)\\
   &\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{bind}\ y \leftarrow N; M\, y && (\text{thunkables are central})
   \end{aligned}$$

3. → intro

   $$\begin{aligned}
   &\lambda y : A.M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)/x]\\
   &\sqsupseteq\sqsubseteq \lambda y : A.\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M && (M \text{ is linear})\\
   &\sqsupseteq\sqsubseteq \lambda y : A.\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; (\lambda y : A.M)\, y && (\to \beta)\\
   &\sqsupseteq\sqsubseteq \lambda y : A.(\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; (\lambda y : A.M))\, y && (\underline{F}\eta)\\
   &\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; (\lambda y : A.M) && (\to \eta)
   \end{aligned}$$

4. $\top$ intro We need to show

$$\mathtt{bind}\ w \leftarrow \mathtt{force}\ z; \{\} \sqsupseteq\sqsubseteq \{\}$$

Which is immediate by $\top\eta$

5. & intro

$$\begin{aligned}
&\{\pi \mapsto M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)]/x \\
&\mid \pi' \mapsto N[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)/x]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M && (M, N \text{ linear}) \\
&\qquad \mid \pi' \mapsto \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; N\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \pi\{\pi \mapsto M \mid \pi' \mapsto N\} && (\&\beta) \\
&\qquad \mid \pi' \mapsto \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \pi'\{\pi \mapsto M \mid \pi' \mapsto N\}\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi(\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \{\pi \mapsto M \mid \pi' \mapsto N\}) && (\underline{F}\eta) \\
&\qquad \mid \pi' \mapsto \pi'(\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \{\pi \mapsto M \mid \pi' \mapsto N\})\} \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \{\pi \mapsto M \mid \pi' \mapsto N\} && (\&\eta)
\end{aligned}$$

6. ν intro

$$\begin{aligned}
&\mathtt{roll}\ M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)/x] \\
&\sqsupseteq\sqsubseteq \mathtt{roll}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M) && (M \text{ is linear}) \\
&\sqsupseteq\sqsubseteq \mathtt{roll}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{unroll}\ \mathtt{roll}\ M) && (\nu\beta) \\
&\sqsupseteq\sqsubseteq \mathtt{roll}\ \mathtt{unroll}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{roll}\ M) && (\underline{F}\eta) \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; (\mathtt{roll}\ M) && (\nu\eta)
\end{aligned}$$

7. $\underline{F}$ elim: Assume $\Gamma, x : A \vdash M : \underline{F}A'$ and $\Gamma, y : A' \vdash N : \underline{B}$, then we need to show

$$\mathtt{bind}\ y \leftarrow M; N$$

is linear in M.

$$\begin{aligned}
&\mathtt{bind}\ y \leftarrow M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{force}\ x)/x]; N \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow (\mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M); N && (M \text{ is linear}) \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{bind}\ y \leftarrow M; N && (\underline{F}\eta)
\end{aligned}$$

8. 0 elim: We want to show $\Gamma, x : U\underline{B} \vdash \mathtt{absurd}\ V : \underline{C}$ is linear in x, which means showing:

$$\mathtt{absurd}\ V \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{absurd}\ V$$

which follows from 0η

9. $+$ elim: Assuming $\Gamma, x : U\underline{B}, y_1 : A_1 \vdash M_1 : \underline{C}$ and $\Gamma, x : U\underline{B}, y_2 : A_2 \vdash M_2 : \underline{C}$ are linear in x, and $\Gamma \vdash V : A_1 + A_2$, we need to show

$$\mathtt{case}\ V\{y_1.M_1 \mid y_2.M_2\}$$

is linear in x.

$$\begin{aligned}
&\texttt{case}\ V\\
&\ \{y_1.M_1[\texttt{thunk}\ (\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]\\
&\ \ \mid y_2.M_2[\texttt{thunk}\ (\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]\}\\
&\sqsupseteq\sqsubseteq \texttt{case}\ V\{y_1.\texttt{bind}\ x \leftarrow \texttt{force}\ z; M_1 \mid y_2.\texttt{bind}\ x \leftarrow \texttt{force}\ z; M_2\}\\
&\hfill (M_1, M_2 \text{ linear})\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{case}\ V\{y_1.M_1 \mid y_2.M_2\}
\end{aligned}$$

10. $\times$ elim: Assuming $\Gamma, x : U\underline{B}, y_1 : A_1, y_2 : A_2 \vdash M : \underline{B}$ is linear in x and $\Gamma \vdash V : A_1 \times A_2$, we need to show

$$\texttt{split}\ V\ \texttt{to}\ (y_1, y_2).M$$

is linear in x.

$$\begin{aligned}
&\texttt{split}\ V\ \texttt{to}\ (y_1, y_2).M[[\texttt{thunk}\ (\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]]\\
&\sqsupseteq\sqsubseteq \texttt{split}\ V\ \texttt{to}\ (y_1, y_2).\texttt{bind}\ x \leftarrow \texttt{force}\ z; M && (M \text{ linear})\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{split}\ V\ \texttt{to}\ (y_1, y_2).M && (\text{comm. conv})
\end{aligned}$$

11. μ elim: Assuming $\Gamma, x : U\underline{B}, y : A[\mu X.A/X] \vdash M : \underline{C}$ is linear in x and $\Gamma \vdash V : \mu X.A$, we need to show

$$\texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M$$

is linear in x.

$$\begin{aligned}
&\texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M[\texttt{thunk}\ (\texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]\\
&\sqsupseteq\sqsubseteq \texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.\texttt{bind}\ x \leftarrow \texttt{force}\ z; M && (M \text{ linear})\\
&\sqsupseteq\sqsubseteq \texttt{bind}\ x \leftarrow \texttt{force}\ z; \texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M && (\text{commuting conversion})
\end{aligned}$$

□

Proof of Lemma 6.10. *Proof.*

1. First, note that every occurrence of a variable in $E^{\dagger}$ is of the form $\texttt{ret}x$ for some variable x. This means we can define substitution of a *term* for a variable in a simplified term by defining $E^{\dagger}[N/\texttt{ret}x]$ to replace every $\texttt{ret}x : \underline{F}A$ with $N : \underline{F}A$. Then it is an easy observation that simplification is compositional on the nose with respect to this notion of substitution:

$$(E[V/x])^{\dagger} = E^{\dagger}[V^{\dagger}/\texttt{ret}x]$$

Next by repeated invocation of $U\beta$,

$$E^{\dagger}[V^{\dagger}/\texttt{ret}x] \sqsupseteq\sqsubseteq E^{\dagger}[\texttt{force}\ \texttt{thunk}\ V^{\dagger}/\texttt{ret}x]$$

Then we can lift the definition of the thunk to the top-level by $\underline{F}\beta$:

$$E^{\dagger}[\texttt{force}\ \texttt{thunk}\ V^{\dagger}/\texttt{ret}x] \sqsupseteq\sqsubseteq \texttt{bind}\ \texttt{thunk}\ \ \leftarrow \texttt{ret}; V^{\dagger}wE^{\dagger}[\texttt{force}\ w/\texttt{ret}x]$$

Then because $V^\dagger$ is thunkable, we can bind it at the top-level and reduce an administrative redex away to get our desired result:

$$\begin{aligned}
&\texttt{bind thunk } \leftarrow \texttt{ret}; V^\dagger w E^\dagger[\texttt{force } w/\texttt{ret}x] \\
&\sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow V^\dagger; \texttt{bind } w \leftarrow \texttt{retthunk ret}x; E^\dagger[\texttt{force } w/\texttt{ret}x] \\
&&& (V \text{ thunkable}) \\
&\sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow V^\dagger; E^\dagger[\texttt{force thunk ret}x/\texttt{ret}x] && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow V^\dagger; E^\dagger[\texttt{ret}x/\texttt{ret}x] && (U\beta) \\
&\sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow V^\dagger; E^\dagger
\end{aligned}$$

2. Note that every occurrence of z in $S^\dagger$ is of the form $\texttt{force } z$. This means we can define substitution of a *term* $M : \underline{B}$ for $\texttt{force } z$ in $S^\dagger$ by replacing $\texttt{force } z$ with M. It is an easy observation that simplification is compositional on the nose with respect to this notion of substitution:

$$(S[M/\bullet])^\dagger = S^\dagger[M^\dagger/\texttt{force } z]$$

Then by repeated $U\beta$, we can replace $M^\dagger$ with a forced thunk:

$$S^\dagger[M^\dagger/\texttt{force } z] \sqsupseteq\sqsubseteq S^\dagger[\texttt{force thunk } M^\dagger/\texttt{force } z]$$

which since we are now substituting a force for a force is the same as substituting the thunk for the variable:

$$S^\dagger[\texttt{force thunk } M^\dagger/\texttt{force } z] \sqsupseteq\sqsubseteq S^\dagger[\texttt{thunk } M^\dagger/z]$$

□

Proof of Theorem 6.1.

Proof.

1. Reflexivity is translated to reflexivity.
2. Transitivity is translated to transitivity.
3. Compatibility rules are translated to compatibility rules.
4. Substitution of a Value

$$\frac{\Gamma, x : A, \Delta^\dagger \vdash E^\dagger \sqsubseteq E'^\dagger : T^\dagger \qquad \Gamma \vdash V^\dagger \sqsubseteq V'^\dagger : \underline{F}A}{\Gamma, \Delta^\dagger \vdash E[V/x]^\dagger \sqsubseteq E'[V'/x]^\dagger : T^\dagger}$$

By the compositionality lemma, it is sufficient to show:

$$\texttt{bind } x \leftarrow V^\dagger; E^\dagger \sqsubseteq \texttt{bind } x \leftarrow V'^\dagger; E'$$

which follows by bind compatibility.

5. Plugging a term into a hole:

$$\frac{\Gamma, z : U\underline{C} \vdash S^\dagger \sqsubseteq S'^\dagger : \underline{B} \qquad \Gamma, \Delta^\dagger \vdash M^\dagger \sqsubseteq M'^\dagger : \underline{C}}{\Gamma, \Delta^\dagger \vdash S[M]^\dagger \sqsubseteq S'[M']^\dagger : \underline{B}}$$

By compositionality, it is sufficient to show

$$S^\dagger[\texttt{thunk } M^\dagger/z] \sqsubseteq S'^\dagger[\texttt{thunk } M'^\dagger/z]$$

which follows by thunk compatibility and the simple substitution rule.

6. Stack strictness We need to show for S a complex stack, that

$$(S[\mho])^\dagger \sqsupseteq\sqsubseteq \mho$$

By stack compositionality we know

$$(S[\mho])^\dagger \sqsupseteq\sqsubseteq S^\dagger[\texttt{thunk } \mho/z]$$

$$\begin{aligned} [\![S]\!][\texttt{thunk } \mho/z] &\sqsupseteq\sqsubseteq S^\dagger[\texttt{thunk } (\texttt{bind } y \leftarrow \mho; \mho)/z] && \text{(Stacks preserve } \mho\text{)} \\ &\sqsupseteq\sqsubseteq \texttt{bind } y \leftarrow \mho; S^\dagger[\texttt{thunk } \mho/z] && (S^\dagger \text{ is linear in } z) \\ &\sqsupseteq\sqsubseteq \mho && \text{(Stacks preserve } \mho\text{)} \end{aligned}$$

7. 1β By compositionality it is sufficient to show

$$\texttt{bind } x \leftarrow \texttt{ret}(); \texttt{split } x \texttt{ to } ().E^\dagger \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \texttt{ret}(); E^\dagger$$

which follows by $\underline{F}\beta, 1\beta$.

8. 1η We need to show for $\Gamma, x : 1 \mid \Delta \vdash E : T$

$$E^\dagger \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \texttt{ret}x; \texttt{split } x \texttt{ to } ().(E[()/x])^\dagger$$

after a $\underline{F}\beta$, it is sufficient using 1η to prove:

$$(E[()/x])^\dagger \sqsupseteq\sqsubseteq E^\dagger[()/x]$$

which follows by compositionality and $\underline{F}\beta$:

$$(E[()/x])^\dagger \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \texttt{ret}(); E^\dagger \sqsupseteq\sqsubseteq E^\dagger[()/x]$$

9. $\times\beta$ By compositionality it is sufficient to show

$$\begin{aligned} &\texttt{bind } x \leftarrow (\texttt{bind } x_1 \leftarrow {V_1}^\dagger; \texttt{bind } x_2 \leftarrow {V_2}^\dagger; \texttt{ret}(x_1, x_2)); \texttt{split } x \texttt{ to } (x_1, x_2).E^\dagger \\ &\sqsupseteq\sqsubseteq \texttt{bind } x_1 \leftarrow {V_1}^\dagger; \texttt{bind } x_2 \leftarrow {V_2}^\dagger; E^\dagger \end{aligned}$$

which follows by $\underline{F}\eta, \underline{F}\beta, \times\beta$.

10. $\times\eta$ We need to show for $\Gamma, x : A_1 \times A_2 \mid \Delta \vdash E : T$ that

$$E^\dagger \sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \texttt{ret}x; \texttt{split } x \texttt{ to } (x_1, x_2).(E[(x_1, x_2)/x])^\dagger$$

by $\underline{F}\beta$, $\times\eta$ it is sufficient to show

$$E[(x_1, x_2)/x]^\dagger \sqsupseteq\sqsubseteq E^\dagger[(x_1, x_2)/x]$$

Which follows by compositionality:

$$\begin{aligned} &E[(x_1, x_2)/x]^\dagger \\ &\sqsupseteq\sqsubseteq \texttt{bind } x_1 \leftarrow x_1; \texttt{bind } x_2 \leftarrow x_2; \texttt{bind } x \leftarrow \texttt{ret}(x_1, x_2); E^\dagger && \text{(compositionality)} \\ &\sqsupseteq\sqsubseteq \texttt{bind } x \leftarrow \texttt{ret}(x_1, x_2); E^\dagger && (\underline{F}\beta) \\ &\sqsupseteq\sqsubseteq E^\dagger[(x_1, x_2)/x] \end{aligned}$$

11. 0η We need to show for any $\Gamma, x : 0 \mid \Delta \vdash E : T$ that

    $$E^\dagger \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{ret} x;\ \mathtt{absurd}\ x$$

    which follows by 0η
12. $+\beta$ Without loss of generality, we do the $\mathtt{inl}$ case By compositionality it is sufficient to show

    $$\mathtt{bind}\ x \leftarrow (\mathtt{bind}\ x \leftarrow V^\dagger; \mathtt{inl}\ x);\ \mathtt{case}\ x\{x_1.E_1^\dagger \mid x_2.E_2^\dagger\} \sqsupseteq\sqsubseteq E_1[V/x_1]^\dagger$$

    which holds by $\underline{F}\eta, \underline{F}\beta, +\beta$
13. $+\eta$ We need to show for any $\Gamma, x : A_1 + A_2 \mid \Delta \vdash E : T$ that

    $$E^\dagger \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{ret} x;\ \mathtt{case}\ x\{x_1.(E[\mathtt{inl}\ x_1/x])^\dagger \mid x_2.(E[\mathtt{inl}\ x_2/x])^\dagger\}$$

    $$\begin{aligned}
    & E^\dagger \\
    & \sqsupseteq\sqsubseteq \mathtt{case}\ x\{x_1.E^\dagger[\mathtt{inl}\ x_1/x] \mid x_2.E^\dagger[\mathtt{inl}\ x_2/x]\} && (+\eta) \\
    & \sqsupseteq\sqsubseteq \mathtt{case}\ x\{x_1.\mathtt{bind}\ x \leftarrow \mathtt{retinl}\ x_1; E^\dagger \mid x_2.\mathtt{bind}\ x \leftarrow \mathtt{retinl}\ x_2; E^\dagger\} && (\underline{F}\beta) \\
    & \sqsupseteq\sqsubseteq \mathtt{case}\ x\{x_1.E[\mathtt{inl}\ x_1]/x^\dagger \mid x_2.E[\mathtt{inl}\ x_2]/x^\dagger\} && (\text{compositionality}) \\
    & \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{ret} x;\ \mathtt{case}\ x\{x_1.E[\mathtt{inl}\ x_1]/x^\dagger \mid x_2.E[\mathtt{inl}\ x_2]/x^\dagger\} && (\underline{F}\beta)
    \end{aligned}$$

14. $\mu\beta$ By compositionality it is sufficient to show

    $$\begin{aligned}
    & \mathtt{bind}\ x \leftarrow (\mathtt{bind}\ y \leftarrow V^\dagger; \mathtt{retroll}\ y); \mathtt{unroll}\ x\,\mathtt{to}\ \mathtt{roll}\ y.E \\
    & \sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow V^\dagger; E^\dagger
    \end{aligned}$$

    which follows by $\underline{F}\eta, \underline{F}\beta, \mu\beta$.
15. $\mu\eta$ We need to show for $\Gamma, x : \mu X.A \mid \Delta \vdash E : T$ that

    $$E^\dagger \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{ret} x; \mathtt{unroll}\ x\,\mathtt{to}\ \mathtt{roll}\ y.(E[\mathtt{roll}\ y/x])^\dagger$$

    by $\underline{F}\beta$, $\times\eta$ it is sufficient to show

    $$E[\mathtt{roll}\ y/x]^\dagger \sqsupseteq\sqsubseteq E^\dagger[\mathtt{roll}\ y/x]$$

    Which follows by compositionality:

    $$\begin{aligned}
    & E[\mathtt{roll}\ y/x]^\dagger \\
    & \sqsupseteq\sqsubseteq \mathtt{bind}\ y \leftarrow \mathtt{ret} y; \mathtt{bind}\ x \leftarrow \mathtt{retroll}\ y; E^\dagger && (\text{compositionality}) \\
    & \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \mathtt{retroll}\ y; E^\dagger && (\underline{F}\beta) \\
    & \sqsupseteq\sqsubseteq E^\dagger[\mathtt{roll}\ y/x] && (\underline{F}\beta)
    \end{aligned}$$

16. $U\beta$ We need to show

    $$\mathtt{bind}\ x \leftarrow \mathtt{ret} M^\dagger; \mathtt{force}\ x \sqsupseteq\sqsubseteq M^\dagger$$

    which follows by $\underline{F}\beta, U\beta$
17. $U\eta$ We need to show for any $\Gamma \vdash V : U\underline{B}$ that

    $$V^\dagger \sqsupseteq\sqsubseteq \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow V^\dagger; \mathtt{force}\ x)$$

    By compositionality it is sufficient to show

    $$V^\dagger \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow V^\dagger; \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{ret} x; \mathtt{force}\ x)$$

which follows by $U\eta$ and some simple reductions:

$$\begin{aligned}
&\mathtt{bind}\ x \leftarrow V^{\dagger}; \mathtt{retthunk}\ (\mathtt{bind}\ x \leftarrow \mathtt{ret}x; \mathtt{force}\ x) && \\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow V^{\dagger}; \mathtt{retthunk}\ \mathtt{force}\ x && (\underline{F}\beta)\\
&\sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow V^{\dagger}; \mathtt{ret}x && (U\eta)\\
&\sqsupseteq\sqsubseteq V^{\dagger} && (\underline{F}\eta)
\end{aligned}$$

18. $\to \beta$ By compositionality it is sufficient to show

$$\mathtt{bind}\ x \leftarrow V^{\dagger}; (\lambda x : A.M^{\dagger})\, x \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow V^{\dagger}; M^{\dagger}$$

which follows by $\to \beta$

19. $\to \eta$ We need to show

$$z : U(A \to \underline{B}) \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq \lambda x : A.\mathtt{bind}\ x \leftarrow \mathtt{ret}x; (\mathtt{force}\ z)\, x$$

which follows by $\underline{F}\beta, \to \eta$

20. $\top\eta$ We need to show

$$z : U\top \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq \{\}$$

which is exactly $\top\eta$.

21. $\&\beta$ Immediate by simple $\&\beta$.

22. $\&\eta$ We need to show

$$z : U(\underline{B}_1 \,\&\, \underline{B}_2) \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq \{\pi \mapsto \pi\mathtt{force}\ z \mid \pi' \mapsto \pi'\mathtt{force}\ z\}$$

which is exactly $\&\eta$

23. $\nu\beta$ Immediate by simple $\nu\beta$

24. $\nu\eta$ We need to show

$$z : U(\nu \underline{Y}.\underline{B}) \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq \mathtt{roll}\ \mathtt{unroll}\ z$$

which is exactly $\nu\eta$

25. $\underline{F}\beta$ We need to show

$$\mathtt{bind}\ x \leftarrow V^{\dagger}; M^{\dagger} \sqsupseteq\sqsubseteq M[V/x]^{\dagger}$$

which is exactly the compositionality lemma.

26. $\underline{F}\eta$ We need to show

$$z : U(\underline{F}A)\mathtt{force}\ z \vdash \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; \mathtt{bind}\ x \leftarrow \mathtt{ret}x; \mathtt{ret}x$$

which follows by $\underline{F}\beta, \underline{F}\eta$ □

F Proofs for Section 7

To prove Lemma 7.5, we develop a few lemmas about the interaction between contextual lifting and operations on relations.

In the following, we write $\sim^{\circ}$ for the opposite of a relation ($x \sim^{\circ} y$ iff $y \sim x$), $\Rightarrow$ for containment/implication ($\sim \Rightarrow \sim'$ iff $x \sim y$ implies $x \sim' y$), $\Leftrightarrow$ for bicontainment/equality, $\vee$ for union ($x(\sim \vee \sim')y$ iff $x \sim y$ or $x \sim' y$), and $\wedge$ for intersection ($x(\sim \wedge \sim')y$ iff $x \sim y$ and $x \sim' y$).

Lemma F.1 (Contextual Lift commutes with Conjunction)**.**

$$(\sim_1 \wedge \sim_2)^{ctx} \Leftrightarrow \sim_1^{ctx} \wedge \sim_2^{ctx}$$

Lemma F.2 (Contextual Lift commutes with Dualization)**.**

$$\sim^{\circ ctx} \Leftrightarrow \sim^{ctx \circ}$$

Lemma F.3 (Contextual Decomposition Lemma)**.** *Let* $\sim$ *be a reflexive relation* $(= \Rightarrow \sim)$, *and* $\leqslant$ *be a reflexive, antisymmetric relation* $(= \Rightarrow \leqslant$ *and* $(\leqslant \wedge \leqslant^\circ) \Leftrightarrow =)$. *Then*

$$\sim^{ctx} \Leftrightarrow (\sim \vee \leqslant)^{ctx} \wedge ((\sim^\circ \vee \leqslant)^{ctx})^\circ$$

Proof. Note that despite the notation, $\leqslant$ need not be assumed to be transitive. Reflexive relations form a lattice with $\wedge$ and $\vee$ with $=$ as $\perp$ and the total relation as $\top$ (e.g., $(= \vee \sim) \Leftrightarrow \sim$ because $\sim$ is reflexive, and $(= \wedge \sim) \Leftrightarrow =$). So we have

$$\sim \Leftrightarrow (\sim \vee \leqslant) \wedge (\sim \vee \leqslant^\circ)$$

because FOILing the right-hand side gives

$$(\sim \wedge \sim) \vee (\leqslant \wedge \sim) \vee (\sim \wedge \leqslant^\circ) \vee (\leqslant \wedge \leqslant^\circ)$$

By antisymmetry, $(\leqslant \wedge \leqslant^\circ)$ is $=$, which is the unit of $\vee$, so it cancels. By idempotence, $(\sim \wedge \sim)$ is $\sim$. Then by absorption, the whole thing is $\sim$.

Opposite is *not* de Morgan: $(P \vee Q)^\circ = P^\circ \vee Q^\circ$, and similarly for $\wedge$. But it is involutive: $(P^\circ)^\circ \Leftrightarrow P$.

So using Lemmas F.1, F.2 we can calculate as follows:

$$\begin{array}{rcl}
\sim^{\mathrm{ctx}} & \Leftrightarrow & ((\sim \vee \leqslant) \wedge (\sim \vee \leqslant^\circ))^{\mathrm{ctx}} \\
 & \Leftrightarrow & (\sim \vee \leqslant)^{\mathrm{ctx}} \wedge (\sim \vee \leqslant^\circ)^{\mathrm{ctx}} \\
 & \Leftrightarrow & (\sim \vee \leqslant)^{\mathrm{ctx}} \wedge ((\sim \vee \leqslant^\circ)^\circ)^{\circ \mathrm{ctx}} \\
 & \Leftrightarrow & (\sim \vee \leqslant)^{\mathrm{ctx}} \wedge ((\sim^\circ \vee (\leqslant^\circ)^\circ)^\circ)^{\mathrm{ctx}} \\
 & \Leftrightarrow & (\sim \vee \leqslant)^{\mathrm{ctx}} \wedge (\sim^\circ \vee \leqslant)^{\circ \mathrm{ctx}} \\
 & \Leftrightarrow & (\sim \vee \leqslant)^{\mathrm{ctx}} \wedge (\sim^\circ \vee \leqslant)^{\mathrm{ctx} \circ}
\end{array}$$

□

As a corollary, the decomposition of contextual equivalence into diverge approximation in Ahmed (2006) and the decomposition of precision in New & Ahmed (2018) are really the same trick:

Proof of Corollary 7.2.

Proof.

For part 1 (though we will not use this below), applying Lemma F.3 with $\sim$ taken to be $=$ (which is reflexive) and $\leqslant$ taken to be $\preceq$ (which is reflexive and antisymmetric) gives that contextual equivalence is symmetric contextual divergence approximation:

$$=^{\mathrm{ctx}} \Leftrightarrow (= \vee \preceq)^{\mathrm{ctx}} \wedge ((=^\circ \vee \preceq)^{\mathrm{ctx}})^\circ \Leftrightarrow \preceq^{\mathrm{ctx}} \wedge ((\preceq)^{\mathrm{ctx}})^\circ$$

For part (2), the same argument with $\sim$ taken to be $=$ and $\leqslant$ taken to be $\sqsubseteq$ (which is also antisymmetric) gives that contextual equivalence is symmetric contextual precision:

$$=^{\mathrm{ctx}} \Leftrightarrow \sqsubseteq^{\mathrm{ctx}} \wedge ((\sqsubseteq)^{\mathrm{ctx}})^{\circ}$$

For part (3), applying Lemma F.3 with $\sim$ taken to be $\sqsubseteq$ and $\leqslant$ taken to be $\preceq$ gives that precision decomposes as

$$\sqsubseteq^{\mathrm{ctx}} \Leftrightarrow (\sqsubseteq \vee \preceq)^{\mathrm{ctx}} \wedge ((\sqsubseteq^{\circ} \vee \preceq)^{\mathrm{ctx}})^{\circ} \Leftrightarrow \preceq\sqsubseteq^{\mathrm{ctx}} \wedge ((\preceq\sqsupseteq)^{\mathrm{ctx}})^{\circ}$$

Since both $\preceq\sqsubseteq$ and $\preceq\sqsupseteq$ are of the form $- \vee \preceq$, both are divergence preorders. Thus, it suffices to develop logical relations for divergence preorders below. □

Proof of Theorem 7.1.

Proof. For each congruence rule

$$\frac{\Gamma \mid \Delta \vdash E_1 \sqsubseteq E_1' : T_1 \cdots}{\Gamma' \mid \Delta' \vdash E_c \sqsubseteq E_c' : T_c}$$

we prove for every $i \in \mathbb{N}$ the validity of the rule

$$\frac{\Gamma \mid \Delta \vDash E_1 \trianglelefteq_i^{\log} E_1' \in T_1 \cdots}{\Gamma \mid \Delta \vDash E_c \trianglelefteq_i^{\log} E_c' \in T_c}$$

1. $\Gamma, x : A, \Gamma' \vDash x \trianglelefteq_i^{\log} x \in A$. Given $\gamma_1 \trianglelefteq_{\Gamma,x:A,\Gamma',i}^{\log} \gamma_2$, then by definition $\gamma_1(x) \trianglelefteq_{A,i}^{\log} \gamma_2(x)$.
2. $\Gamma \vDash \mho \trianglelefteq_i^{\log} \mho \in \underline{B}$ We need to show $S_1[\mho] \trianglelefteq^i \mathrm{result}(S_2[\mho])$. By anti-reduction and strictness of stacks, it is sufficient to show $\mho \trianglelefteq_i^{\log} \mho$. If $i = 0$ there is nothing to show, otherwise, it follows by reflexivity of $\trianglelefteq$.
3. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A \qquad \Gamma, x : A \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}}{\Gamma \vDash \texttt{let } x = V; M \trianglelefteq_i^{\log} \texttt{let } x = V'; M' \in \underline{B}}$

   Each side takes a 0-cost step, so by anti-reduction, this reduces to

   $$S_1[M[\gamma_1, V/x]] \trianglelefteq^i \mathrm{result}(S_2[M'[\gamma_2, V'/x]])$$

   which follows by the assumption $\Gamma, x : A \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}$
4. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in 0}{\Gamma \vDash \texttt{abort } V \trianglelefteq_i^{\log} \texttt{abort } V' \in \underline{B}}$. By assumption, we get $V[\gamma_1] \trianglelefteq_{0,i}^{\log} V'[\gamma_2]$, but this is a contradiction.
5. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A_1}{\Gamma \vDash \texttt{inl } V \trianglelefteq_i^{\log} \texttt{inl } V' \in A_1 + A_2}$. Direct from assumption, rule for sums.
6. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A_2}{\Gamma \vDash \texttt{inr } V \trianglelefteq_i^{\log} \texttt{inr } V' \in A_1 + A_2}$ Direct from assumption, rule for sums.
7. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vDash M_1 \trianglelefteq_i^{\log} M_1' \in \underline{B} \qquad \Gamma, x_2 : A_2 \vDash M_2 \trianglelefteq_i^{\log} M_2' \in \underline{B}}{\Gamma \vDash \texttt{case } V\{x_1.M_1 \mid x_2.M_2\} \trianglelefteq_i^{\log} \texttt{case } V'\{x_1.M_1' \mid x_2.M_2'\} \in \underline{B}}$

   By case analysis of $V[\gamma_1] \trianglelefteq_i^{\log} V'[\gamma_2]$.

a. If $V[\gamma_1] = \texttt{inl}\ V_1, V'[\gamma_2] = \texttt{inl}\ V_1'$ with $V_1 \trianglelefteq^{\log}_{A_1,i} V_1'$, then taking 0 steps, by anti-reduction the problem reduces to

$$S_1[M_1[\gamma_1, V_1/x_1]] \trianglelefteq^{i} \mathrm{result}(S_1[M_1[\gamma_1, V_1/x_1]])$$

which follows by assumption.

b. For $\texttt{inr}$, the same argument.

8. $\Gamma \vDash () \trianglelefteq^{\log}_{i} () \in 1$ Immediate by unit rule.

9. $\dfrac{\Gamma \vDash V_1 \trianglelefteq^{\log}_{i} V_1' \in A_1 \qquad \Gamma \vDash V_2 \trianglelefteq^{\log}_{i} V_2' \in A_2}{\Gamma \vDash (V_1, V_2) \trianglelefteq^{\log}_{i} (V_1', V_2') \in A_1 \times A_2}$ Immediate by pair rule.

10. $\dfrac{\Gamma \vDash V \trianglelefteq^{\log}_{i} V' \in A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vDash M \trianglelefteq^{\log}_{i} M' \in \underline{B}}{\Gamma \vDash \texttt{split}\ V\ \texttt{to}\ (x, y).M \trianglelefteq^{\log}_{i} \texttt{split}\ V'\ \texttt{to}\ (x, y).M' \in \underline{B}}$ By $V \trianglelefteq^{\log}_{A_1 \times A_2, i} V'$, we know $V[\gamma_1] = (V_1, V_2)$ and $V'[\gamma_2] = (V_1', V_2')$ with $V_1 \trianglelefteq^{\log}_{A_1,i} V_1'$ and $V_2 \trianglelefteq^{\log}_{A_2,i} V_2'$. Then by anti-reduction, the problem reduces to

$$S_1[M[\gamma_1, V_1/x, V_2/y]] \trianglelefteq^{i} \mathrm{result}(S_1[M'[\gamma_1, V_1'/x, V_2'/y]])$$

which follows by assumption.

11. $\dfrac{\Gamma \vDash V \trianglelefteq^{\log}_{i} V' \in A[\mu X.A/X]}{\Gamma \vDash \texttt{roll}_{\mu X.A}\ V \trianglelefteq^{\log}_{i} \texttt{roll}_{\mu X.A}\ V' \in \mu X.A}$ If $i = 0$, we're done. Otherwise $i = j + 1$, and our assumption is that $V[\gamma_1] \trianglelefteq^{\log}_{A[\mu X.A/X], j+1} V'[\gamma_2]$ and we need to show that $\texttt{roll}\ V[\gamma_1] \trianglelefteq^{\log}_{\mu X.A, j+1} \texttt{roll}\ V'[\gamma_2]$. By definition, we need to show $V[\gamma_1] \trianglelefteq^{\log}_{A[\mu X.A/X], j} V'[\gamma_2]$, which follows by downward closure.

12. $\dfrac{\Gamma \vDash V \trianglelefteq^{\log}_{i} V' \in \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \vDash M \trianglelefteq^{\log}_{i} M' \in \underline{B}}{\Gamma \vDash \texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ x.M \trianglelefteq^{\log}_{i} \texttt{unroll}\ V'\ \texttt{to}\ \texttt{roll}\ x.M' \in \underline{B}}$ If $i = 0$, then by triviality at 0, we're done. Otherwise, $V[\gamma_1] \trianglelefteq^{\log}_{\mu X.A, j+1} V'[\gamma_2]$ so $V[\gamma_1] = \texttt{roll}\ V_\mu, V'[\gamma_2] = \texttt{roll}\ V_\mu'$ with $V_\mu \trianglelefteq^{\log}_{A[\mu X.A/X], j} V_\mu'$. Then each side takes 1 step, so by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1, V_\mu/x]] \trianglelefteq^{j} \mathrm{result}(S_2[M'[\gamma_2, V_\mu'/x]])$$

which follows by assumption and downward closure of the stack, value relations.

13. $\dfrac{\Gamma \vDash M \trianglelefteq^{\log}_{i} M' \in \underline{B}}{\Gamma \vDash \texttt{thunk}\ M \trianglelefteq^{\log}_{i} \texttt{thunk}\ M' \in U\underline{B}}$. We need to show $\texttt{thunk}\ M[\gamma_1] \trianglelefteq^{\log}_{U\underline{B}, i} \texttt{thunk}\ M'[\gamma_2]$, so let $S_1 \trianglelefteq^{\log}_{\underline{B}, j} S_2$ for some $j \leq i$, and we need to show

$$S_1[\texttt{force thunk}\ M_1[\gamma_1]] \trianglelefteq^{j} \mathrm{result}(S_2[\texttt{force thunk}\ M_2[\gamma_2]])$$

Then each side reduces in a 0-cost step and it is sufficient to show

$$S_1[M_1[\gamma_1]] \trianglelefteq^{j} \mathrm{result}(S_2[M_2[\gamma_2]])$$

Which follows by downward closure for terms and substitutions.

14. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\mathrm{log}} V' \in U\underline{B}}{\Gamma \vDash \texttt{force}\ V \trianglelefteq_i^{\mathrm{log}} \texttt{force}\ V' \in \underline{B}}$.
We need to show $S_1[\texttt{force}\ V[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[\texttt{force}\ V'[\gamma_2]])$, which follows by the definition of $V[\gamma_1] \trianglelefteq_{U\underline{B},i}^{\mathrm{log}} V'[\gamma_2]$.

15. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\mathrm{log}} V' \in A}{\Gamma \vDash \texttt{ret}\,V \trianglelefteq_i^{\mathrm{log}} \texttt{ret}\,V' \in \underline{F}A}$
We need to show $S_1[\texttt{ret}\,V[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[\texttt{ret}\,V'[\gamma_2]])$, which follows by the orthogonality definition of $S_1 \trianglelefteq_{\underline{F}A,i}^{\mathrm{log}} S_2$.

16. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\mathrm{log}} M' \in \underline{F}A \qquad \Gamma, x : A \vDash N \trianglelefteq_i^{\mathrm{log}} N' \in \underline{B}}{\Gamma \vDash \texttt{bind}\ x \leftarrow M; N \trianglelefteq_i^{\mathrm{log}} \texttt{bind}\ x \leftarrow M'; N' \in \underline{B}}$.
We need to show $\texttt{bind}\ x \leftarrow M[\gamma_1]; N[\gamma_2] \trianglelefteq^i \mathrm{result}(\texttt{bind}\ x \leftarrow M'[\gamma_2]; N'[\gamma_2])$. By $M \trianglelefteq_i^{\mathrm{log}} M' \in \underline{F}A$, it is sufficient to show that

$$\texttt{bind}\ x \leftarrow \bullet; N[\gamma_1] \trianglelefteq_{\underline{F}A,i}^{\mathrm{log}} \texttt{bind}\ x \leftarrow \bullet; N'[\gamma_2]$$

So let $j \leq i$ and $V \trianglelefteq_{A,j}^{\mathrm{log}} V'$, then we need to show

$$\texttt{bind}\ x \leftarrow \texttt{ret}\,V; N[\gamma_1] \trianglelefteq_{\underline{F}A,j}^{\mathrm{log}} \texttt{bind}\ x \leftarrow \texttt{ret}\,V'; N'[\gamma_2]$$

By anti-reduction, it is sufficient to show

$$N[\gamma_1, V/x] \trianglelefteq^j \mathrm{result}(N'[\gamma_2, V'/x])$$

which follows by anti-reduction for $\gamma_1 \trianglelefteq_{\Gamma,i}^{\mathrm{log}} \gamma_2$ and $N \trianglelefteq_i^{\mathrm{log}} N'$.

17. $\dfrac{\Gamma, x : A \vDash M \trianglelefteq_i^{\mathrm{log}} M' \in \underline{B}}{\Gamma \vDash \lambda x : A.M \trianglelefteq_i^{\mathrm{log}} \lambda x : A.M' \in A \to \underline{B}}$ We need to show

$$S_1[\lambda x : A.M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[\lambda x : A.M'[\gamma_2]]).$$

By $S_1 \trianglelefteq_{A \to \underline{B},i}^{\mathrm{log}} S_2$, we know $S_1 = S_1'[\bullet V_1]$, $S_2 = S_2'[\bullet V_2]$ with $S_1' \trianglelefteq_{\underline{B},i}^{\mathrm{log}} S_2'$ and $V_1 \trianglelefteq_{A,i}^{\mathrm{log}} V_2$. Then by anti-reduction it is sufficient to show

$$S_1'[M[\gamma_1, V_1/x]] \trianglelefteq^i \mathrm{result}(S_2'[M'[\gamma_2, V_2/x]])$$

which follows by $M \trianglelefteq_i^{\mathrm{log}} M'$.

18. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\mathrm{log}} M' \in A \to \underline{B} \qquad \Gamma \vDash V \trianglelefteq_i^{\mathrm{log}} V' \in A}{\Gamma \vDash M\ V \trianglelefteq_i^{\mathrm{log}} M'\ V' \in \underline{B}}$ We need to show

$$S_1[M[\gamma_1]\ V[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M'[\gamma_2]\ V'[\gamma_2]])$$

so by $M \trianglelefteq_i^{\mathrm{log}} M'$ it is sufficient to show $S_1[\bullet V[\gamma_1]] \trianglelefteq_{A \to \underline{B},i}^{\mathrm{log}} S_2[\bullet V'[\gamma_2]]$ which follows by definition and assumption that $V \trianglelefteq_i^{\mathrm{log}} V'$.

19. $\Gamma \vdash \{\} : \top$ We assume we are given $S_1 \trianglelefteq_{\top,i}^{\mathrm{log}} S_2$, but this is a contradiction.

20. $\dfrac{\Gamma \vDash M_1 \trianglelefteq_i^{\log} M_1' \in \underline{B}_1 \qquad \Gamma \vDash M_2 \trianglelefteq_i^{\log} M_2' \in \underline{B}_2}{\Gamma \vDash \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} \trianglelefteq_i^{\log} \{\pi \mapsto M_1' \mid \pi' \mapsto M_2'\} \in \underline{B}_1 \mathbin{\&} \underline{B}_2}$ We need to show

    $$S_1[\{\pi \mapsto M_1[\gamma_1] \mid \pi' \mapsto M_2[\gamma_1]\}] \trianglelefteq^i \text{result}(S_2[\{\pi \mapsto M_1'[\gamma_1] \mid \pi' \mapsto M_2'[\gamma_2]\}]).$$

    We proceed by case analysis of $S_1 \trianglelefteq_{\underline{B}_1 \& \underline{B}_2, i}^{\log} S_2$

    a. In the first possibility $S_1 = S_1'[\pi \bullet]$, $S_2 = S_2'[\pi \bullet]$ and $S_1' \trianglelefteq_{\underline{B}_1, i}^{\log} S_2'$. Then by anti-reduction, it is sufficient to show

       $$S_1'[M_1[\gamma_1]] \trianglelefteq^i \text{result}(S_2'[M_1'[\gamma_2]])$$

       which follows by $M_1 \trianglelefteq_i^{\log} M_1'$.
    b. Same as previous case.
21. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \vDash \pi M \trianglelefteq_i^{\log} \pi M' \in \underline{B}_1}$ We need to show $S_1[\pi M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\pi M'[\gamma_2]])$, which follows by $S_1[\pi \bullet] \trianglelefteq_{\underline{B}_1 \& \underline{B}_2, i}^{\log} S_2[\pi \bullet]$ and $M \trianglelefteq_i^{\log} M'$.

22. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}_1 \mathbin{\&} \underline{B}_2}{\Gamma \vDash \pi' M \trianglelefteq_i^{\log} \pi' M' \in \underline{B}_2}$ Similar to previous case.
23. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \vDash \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M \trianglelefteq_i^{\log} \texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M' \in \nu \underline{Y}.\underline{B}}$ We need to show that

    $$S_1[\texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{roll}_{\nu \underline{Y}.\underline{B}}\ M'[\gamma_2]])$$

    If $i = 0$, we invoke triviality at 0. Otherwise, $i = j + 1$ and we know by $S_1 \trianglelefteq_{\nu \underline{Y}.\underline{B}, j+1}^{\log} S_2$ that $S_1 = S_1'[\texttt{unroll}\ \bullet]$ and $S_2 = S_2'[\texttt{unroll}\ \bullet]$ with $S_1' \trianglelefteq_{\underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}], j}^{\log} S_2'$, so by anti-reduction it is sufficient to show

    $$S_1'[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2'[M'[\gamma_2]])$$

    which follows by $M \trianglelefteq_i^{\log} M'$ and downward closure.
24. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \nu \underline{Y}.\underline{B}}{\Gamma \vDash \texttt{unroll}\ M \trianglelefteq_i^{\log} \texttt{unroll}\ M' \in \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}$ We need to show

    $$S_1[\texttt{unroll}\ M] \trianglelefteq^i \text{result}(S_2[\texttt{unroll}\ M']),$$

    which follows because $S_1[\texttt{unroll}\ \bullet] \trianglelefteq_{\nu \underline{Y}.\underline{B}, i}^{\log} S_2[\texttt{unroll}\ \bullet]$ and $M \trianglelefteq_i^{\log} M'$.

□

Proof of Corollary 7.5.

Proof. Two cases

1. If $\text{result}(M) \trianglelefteq R$ then we need to show for every $i \in \mathbb{N}$, $M \trianglelefteq^i R$. By the unary model lemma, $M \trianglelefteq^i \text{result}(M)$, so the result follows by the module Lemma 7.6.
2. If $M \trianglelefteq^i R$ for every i, then there are two possibilities: M is always related to R because it takes i steps, or at some point M terminates.

a. If $M \Mapsto^i M_i$ for every $i \in \mathbb{N}$, then $\mathrm{result}(M) = \Omega$, so $\mathrm{result}(M) \trianglelefteq R$ because $\trianglelefteq$ is a divergence preorder.
b. Otherwise there exists some $i \in \mathbb{M}$ such that $M \Mapsto^i \mathrm{result}(M)$, so it follows by the module Lemma 7.6. □

Proof of Lemma 7.12.

Proof. Proof is by mutual lexicographic induction on the pair (i, A) or $(i, \underline{B})$. All cases are straightforward uses of the inductive hypotheses except the shifts $U, \underline{F}$.

1. If $V_1 \trianglelefteq^{\log}_{U\underline{B},i} V_2$ and $V_2 \trianglelefteq^{\log}_{U\underline{B},\omega} V_3$, then we need to show that for any $S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2$ with $j \leq i$,

$$S_1[\texttt{force}\ V_1] \trianglelefteq^j \mathrm{result}(S_2[\texttt{force}\ V_3])$$

   By reflexivity, we know $S_2 \trianglelefteq^{\log}_{\underline{B},\omega} S_2$, so by assumption

$$S_2[\texttt{force}\ V_2] \trianglelefteq^\omega \mathrm{result}(S_2[\texttt{force}\ V_3])$$

   which by the limiting Lemma 7.5 is equivalent to

$$\mathrm{result}(S_2[\texttt{force}\ V_2]) \trianglelefteq \mathrm{result}(S_2[\texttt{force}\ V_3])$$

   so then by the module Lemma 7.6, it is sufficient to show

$$S_1[\texttt{force}\ V_1] \trianglelefteq^j \mathrm{result}(S_2[\texttt{force}\ V_2])$$

   which holds by assumption.
2. If $S_1 \trianglelefteq^{\log}_{\underline{F}A,i} S_2$ and $S_2 \trianglelefteq^{\log}_{\underline{F}A,\omega} S_3$, then we need to show that for any $V_1 \trianglelefteq^{\log}_{j,A} V_2$ with $j \leq i$ that

$$S_1[\texttt{ret}\,V_1] \trianglelefteq^j \mathrm{result}(S_3[\texttt{ret}\,V_2])$$

   First by reflexivity, we know $V_2 \trianglelefteq^{\log}_{A,\omega} V_2$, so by assumption,

$$S_2[\texttt{ret}\,V_2] \trianglelefteq^\omega \mathrm{result}(S_3[\texttt{ret}\,V_2])$$

   Which by the limit Lemma 7.5 is equivalent to

$$\mathrm{result}(S_2[\texttt{ret}\,V_2]) \trianglelefteq^\omega \mathrm{result}(S_3[\texttt{ret}\,V_2])$$

   So by the module Lemma 7.6, it is sufficient to show

$$S_1[\texttt{ret}\,V_1] \trianglelefteq^j \mathrm{result}(S_2[\texttt{ret}\,V_2])$$

   which holds by assumption. □

Proof of Lemma 7.13.

Proof.

1. By induction on the length of the context, follows from closed value case.
2. Assume $\gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2$ and $S_1 \trianglelefteq^{\log}_{\underline{B},i} S_2$. We need to show

$$S_1[M_1[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M_3[\gamma_2]])$$

by reflexivity and assumption, we know

$$S_2[M_2[\gamma_2]] \trianglelefteq^{\omega} \text{result}(S_2[M_3[\gamma_2]])$$

and by limit Lemma 7.5, this is equivalent to

$$\text{result}(S_2[M_2[\gamma_2]]) \trianglelefteq \text{result}(S_2[M_3[\gamma_2]])$$

so by the module Lemma 7.6 it is sufficient to show

$$S_1[M_1[\gamma_1]] \trianglelefteq^{i} \text{result}(S_2[M_2[\gamma_2]])$$

which follows by assumption.

3. Assume $\gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2$. Then $V_1[\gamma_1] \trianglelefteq^{\log}_{A,i} V_2[\gamma_2]$ and by reflexivity $\gamma_2 \trianglelefteq^{\log}_{\Gamma,\omega} \gamma_2$ so $V_2[\gamma_2] \trianglelefteq^{\log}_{A,\omega} V_3[\gamma_2]$ so the result holds by the closed case.
4. Stack case is essentially the same as the value case. □

Proof of Lemma 7.14.

Proof. The β rules for all cases except recursive types are direct from anti-reduction.

1. $\mu X.A - \beta$:

   a. We need to show

   $$S_1[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_1]\ \texttt{to roll}\ x.M[\gamma_1]] \trianglelefteq^{\log}_{i} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   The left side takes 1 step to $S_1[M[\gamma_1, V[\gamma_1]/x]]$ and we know

   $$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq^{\log}_{i} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   by assumption and reflexivity, so by anti-reduction we have

   $$S_1[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_1]\ \texttt{to roll}\ x.M[\gamma_1]] \trianglelefteq^{\log}_{i+1} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   so the result follows by downward closure.

   b. For the other direction we need to show

   $$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq^{\log}_{i} \text{result}(S_2[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_2]\ \texttt{to roll}\ x.M[\gamma_2]])$$

   Since results are invariant under steps, this is the same as

   $$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq^{\log}_{i} \text{result}(S_2[M[\gamma_2, V[\gamma_2/x]]])$$

   which follows by reflexivity and assumptions about the stacks and substitutions.

2. $\mu X.A - \eta$:

   a. We need to show for any $\Gamma, x : \mu X.A \vdash M : \underline{B}$, and appropriate substitutions and stacks,

   $$S_1[\texttt{unroll roll}_{\mu X.A}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_1]] \trianglelefteq^{\log}_{i}$$

   $$\text{result}(S_2[M[\gamma_2]])$$

   By assumption, $\gamma_1(x) \trianglelefteq^{\log}_{\mu X.A,i} \gamma_2(x)$, so we know

   $$\gamma_1(x) = \texttt{roll}_{\mu X.A}\ V_1$$

and

$$\gamma_2(x) = \texttt{roll}_{\mu X.A}\ V_2$$

so the left side takes a step:

$$\begin{aligned} &S_1[\texttt{unroll roll}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}\ y/x][\gamma_1]] \\ &\quad \mapsto^1 S_1[M[\texttt{roll}\ y/x][\gamma_1][V_1/y]] \\ &\quad = S_1[M[\texttt{roll}\ V_1/x][\gamma_1]] \\ &\quad = S_1[M[\gamma_1]] \end{aligned}$$

and by reflexivity and assumptions we know

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \mathrm{result}(S_2[M[\gamma_2]])$$

so by anti-reduction we know

$$\begin{aligned} &S_1[\texttt{unroll roll}_{\mu X.A}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_1]] \\ &\qquad \trianglelefteq_{i+1}^{\log} \mathrm{result}(S_2[M[\gamma_2]]) \end{aligned}$$

so the result follows by downward closure.

b. Similarly, to show

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \mathrm{result}(S_2[\texttt{unroll roll}_{\mu X.A}\ \gamma_2(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_2]])$$

by the same reasoning as above, $\gamma_2(x) = \texttt{roll}_{\mu X.A}\ V_2$, so because result is invariant under reduction we need to show

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \mathrm{result}(S_2[M[\gamma_2]])$$

which follows by assumption and reflexivity.

3. $\nu \underline{Y}.\underline{B} - \beta$

a. We need to show

$$S_1[\texttt{unroll roll}_{\nu \underline{Y}.\underline{B}}\ M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

By the operational semantics,

$$S_1[\texttt{unroll roll}_{\nu \underline{Y}.\underline{B}}\ M[\gamma_1]] \mapsto^1 S_1[M[\gamma_1]]$$

and by reflexivity and assumptions

$$S_1[M[\gamma_1]] \trianglelefteq^i S_2[M[\gamma_2]]$$

so the result follows by anti-reduction and downward closure.

b. We need to show

$$S_1[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[\texttt{unroll roll}_{\nu \underline{Y}.\underline{B}}\ M[\gamma_2]])$$

By the operational semantics and invariance of result under reduction this is equivalent to

$$S_1[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

which follows by assumption.

4. $\nu \underline{Y}.\underline{B} - \eta$

   a. We need to show

$$S_1[\texttt{roll unroll } M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   by assumption, $S_1 \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B},i} S_2$, so

$$S_1 = S_1'[\texttt{unroll } \bullet]$$

   and therefore the left side reduces:

$$\begin{aligned} S_1[\texttt{roll unroll } M[\gamma_1]] &= S_1'[\texttt{unroll roll unroll } M[\gamma_1]] \\ &\mapsto^1 S_1'[\texttt{unroll } M[\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

   and by assumption and reflexivity,

$$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   so the result holds by anti-reduction and downward closure.

   b. Similarly, we need to show

$$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{roll unroll } M[\gamma_2]])$$

   as above, $S_1 \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B},i} S_2$, so we know

$$S_2 = S_2'[\texttt{unroll } \bullet]$$

   so

$$\text{result}(S_2[\texttt{roll unroll } M[\gamma_2]]) = \text{result}(S_2[M[\gamma_2]])$$

   and the result follows by reflexivity, anti-reduction and downward closure.

5. 0η Let $\Gamma, x : 0 \vdash M : \underline{B}$.

   a. We need to show

$$S_1[\texttt{absurd } \gamma_1(x)] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   By assumption $\gamma_1(x) \trianglelefteq^{\log}_{0,i} \gamma_2(x)$ but this is a contradiction

   b. Other direction is the same contradiction.

6. $+\eta$. Let $\Gamma, x : A_1 + A_2 \vdash M : \underline{B}$

   a. We need to show

$$S_1[\texttt{case } \gamma_1(x)\{x_1.M[\texttt{inl } x_1/x][\gamma_1] \mid x_2.M[\texttt{inr } x_2/x][\gamma_1]\}] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   by assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1+A_2,i} \gamma_2(x)$, so either it's an `inl` or *inr*. The cases are symmetric so assume $\gamma_1(x) = \texttt{inl } V_1$. Then

$$\begin{aligned} &S_1[\texttt{case } \gamma_1(x)\{x_1.M[\texttt{inl } x_1/x][\gamma_1] \mid x_2.M[\texttt{inr } x_2/x][\gamma_1]\}] \\ = &S_1[\texttt{case } (\texttt{inl } V_1)\{x_1.M[\texttt{inl } x_1/x][\gamma_1] \mid x_2.M[\texttt{inr } x_2/x][\gamma_1]\}] \\ &\mapsto^0 S_1[M[\texttt{inl } V_1/x][\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

and so by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^i S_2[M[\gamma_2]]$$

which follows by reflexivity and assumptions.

b. Similarly, We need to show

$$\begin{aligned}&\mathrm{result}(S_1[M[\gamma_1]]) \trianglelefteq^i\\ &\qquad \mathrm{result}(S_2[\texttt{case}\ \gamma_2(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_2] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_2]\}])\end{aligned}$$

and by assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1+A_2,i} \gamma_2(x)$, so either it's an $\texttt{inl}$ or *inr*. The cases are symmetric so assume $\gamma_2(x) = \texttt{inl}\ V_2$. Then

$$S_2[\texttt{case}\ \gamma_2(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_2] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_2]\}] \Mapsto^0 S_2[M[\gamma_2]]$$

So the result holds by invariance of result under reduction, reflexivity and assumptions.

7. 1η Let $\Gamma, x : 1 \vdash M : \underline{B}$

a. We need to show

$$S_1[M[()/x][\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

By assumption $\gamma_1(x) \trianglelefteq^{\log}_{1,i} \gamma_2(x)$ so $\gamma_1(x) = ()$, so this is equivalent to

$$S_1[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

which follows by reflexivity, assumption.

b. Opposite case is similar.

8. $\times\eta$ Let $\Gamma, x : A_1 \times A_2 \vdash M : \underline{B}$

a. We need to show

$$S_1[\texttt{split}\ x\ \texttt{to}\ (x_1, y_1).M[(x_1, y_1)/x][\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

By assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1\times A_2,i} \gamma_2(x)$, so $\gamma_1(x) = (V_1, V_2)$, so

$$\begin{aligned}&S_1[\texttt{split}\ x\ \texttt{to}\ (x_1, y_1).M[(x_1, y_1)/x][\gamma_1]]\\ &= S_1[\texttt{split}\ (V_1, V_2)\ \texttt{to}\ (x_1, y_1).M[(x_1, y_1)/x][\gamma_1]]\\ &\Mapsto^0 S_1[M[(V_1, V_2)/x][\gamma_1]]\\ &= S_1[M[\gamma_1]]\end{aligned}$$

So by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

which follows by reflexivity, assumption.

b. Opposite case is similar.

9. $U\eta$ Let $\Gamma \vdash V : U\underline{B}$

a. We need to show that

$$\texttt{thunk force}\ V[\gamma_1] \trianglelefteq^{\log}_{U\underline{B},i} V[\gamma_2]$$

So assume $S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2$ for some $j \leq i$, then we need to show

$$S_1[\texttt{force thunk force } V[\gamma_1]] \trianglelefteq^{j} \text{result}(S_2[\texttt{force } V[\gamma_2]])$$

The left side takes a step:

$$S_1[\texttt{force thunk force } V[\gamma_1]] \mapsto^{0} S_1[\texttt{force } V[\gamma_1]]$$

so by anti-reduction it is sufficient to show

$$S_1[\texttt{force } V[\gamma_1]] \trianglelefteq^{j} \text{result}(S_2[\texttt{force } V[\gamma_2]])$$

which follows by assumption.

b. Opposite case is similar.

10. $F\eta$

a. We need to show that given $S_1 \trianglelefteq^{\log}_{\underline{F}A,i} S_2$,

$$S_1[\texttt{bind } x \leftarrow \bullet; \texttt{ret} x] \trianglelefteq^{\log}_{\underline{F}A,i} S_2$$

So assume $V_1 \trianglelefteq^{\log}_{A,j} V_2$ for some $j \leq i$, then we need to show

$$S_1[\texttt{bind ret} V_1 \leftarrow \bullet; \texttt{ret} x] \trianglelefteq^{j} \text{result}(S_2[\texttt{ret} V_2])$$

The left side takes a step:

$$S_1[\texttt{bind ret} V_1 \leftarrow \bullet; \texttt{ret} x] \mapsto^{0} S_1[\texttt{ret} V_1]$$

so by anti-reduction it is sufficient to show

$$S_1[\texttt{ret} V_1] \trianglelefteq^{j} \text{result}(S_2[\texttt{ret} V_2])$$

which follows by assumption

b. Opposite case is similar.

11. $\to \eta$ Let $\Gamma \vdash M : A \to \underline{B}$

a. We need to show

$$S_1[(\lambda x : A.M[\gamma_1]\, x)] \trianglelefteq^{i} \text{result}(S_2[M[\gamma_2]])$$

by assumption that $S_1 \trianglelefteq^{\log}_{A \to \underline{B},i} S_2$, we know

$$S_1 = S_1'[\bullet\ V_1]$$

so the left side takes a step:

$$\begin{aligned} S_1[(\lambda x : A.M[\gamma_1]\, x)] &= S_1'[(\lambda x : A.M[\gamma_1]\, x)\ V_1] \\ &\mapsto^{0} S_1'[M[\gamma_1]\ V_1] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

So by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^{i} \text{result}(S_2[M[\gamma_2]])$$

which follows by reflexivity, assumption.

b. Opposite case is similar.

12. $\&\eta$ Let $\Gamma \vdash M : \underline{B}_1 \,\&\, \underline{B}_2$

    a. We need to show

    $$S_1[\{\pi \mapsto \pi M[\gamma_1] \mid \pi' \mapsto \pi' M[\gamma_1]\}] \trianglelefteq^i \mathrm{result}(S_1[M[\gamma_2]])$$

    by assumption, $S_1 \trianglelefteq^{\log}_{\underline{B}_1 \& \underline{B}_2, i} S_2$ so either it starts with a π or π' so assume that $S_1 = S_1'[\pi \bullet]$ (π' case is similar). Then the left side reduces

    $$\begin{aligned} S_1[\{\pi \mapsto \pi M[\gamma_1] \mid \pi' \mapsto \pi' M[\gamma_1]\}] &= S_1'[\pi \{\pi \mapsto \pi M[\gamma_1] \mid \pi' \mapsto \pi' M[\gamma_1]\}] \\ &\Mapsto^0 S_1'[\pi M[\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

    So by anti-reduction it is sufficient to show

    $$S_1[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[M[\gamma_2]])$$

    which follows by reflexivity, assumption.

    b. Opposite case is similar.

13. $\top\eta$ Let $\Gamma \vdash M : \top$

    a. In either case, we assume we are given $S_1 \trianglelefteq^{\log}_{\top, i} S_2$, but this is a contradiction.

□

Proof of Lemma 7.15.

Proof. We do the term case, the value case is similar. Given $\gamma_1 \trianglelefteq^{\log}_{\Gamma, i} \gamma_2$, we have $V_1[\gamma_1] \trianglelefteq^{\log}_{A, i} V_2[\gamma_2]$ so

$$\gamma_1, V_1[\gamma_1]/x \trianglelefteq^{\log}_{\Gamma, x:A, i} \gamma_2, V_2[\gamma_2]/x$$

and by associativity of substitution

$$M_1[V_1/x][\gamma_1] = M_1[\gamma_1, V_1[\gamma_1]/x]$$

and similarly for M_2, so if $S_1 \trianglelefteq^{\log}_{\underline{B}, i} S_2$ then

$$S_1[M_1[\gamma_1, V_1[\gamma_1]/x]] \trianglelefteq^i \mathrm{result}(S_2[M_2[\gamma_2, V_2[\gamma_2]/x]])$$

□

Proof of Lemma 7.16.

Proof.

1. It is sufficient by the limit lemma to show $\mathrm{result}(S[\mho]) \trianglelefteq \mho$ which holds by reflexivity because $S[\mho] \Mapsto^0 \mho$.
2. We need to show $S[\mho] \preceq\sqsubseteq^i R$ for arbitrary R, so by the limit lemma it is sufficient to show $\mho \preceq\sqsubseteq R$, which is true by definition.
3. By the limit lemma it is sufficient to show $R \preceq\sqsupseteq \mho$ which is true by definition.

□

Proof of Theorem 7.2.

Proof.

For the first part, from Lemma 7.17, we have $E \preceq\sqsubseteq^{\omega} E'$ and $E' \preceq\sqsupseteq^{\omega} E$. By Lemma 7.6, we then have $E \preceq\sqsubseteq^{\mathrm{ctx}} E'$ and $E' \preceq\sqsupseteq^{\mathrm{ctx}} E$. Finally, by Corollary 7.2, $E \sqsubseteq^{\mathrm{ctx}} E'$ iff $E \preceq\sqsubseteq^{\mathrm{ctx}}$ $E'$and $E((\preceq\sqsupseteq)^{\mathrm{ctx}})^{\circ}E'$, so we have the result.

For the second part, applying the first part twice gives $E \sqsubseteq^{\mathrm{ctx}} E'$ and $E' \sqsubseteq^{\mathrm{ctx}} E$, and we concluded in Corollary 7.2 that this coincides with contextual equivalence. □