Monotonicity Types

Kevin Clancy[1], Heather Miller[1,2], and Christopher Meiklejohn[3,4]

[1] Northeastern University, Boston MA 02115, USA
[2] École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[3] Université Catholique de Louvain, Louvain-le-Neuve, Belgium
[4] Instituto Superior Técnico, Lisbon, Portugal

Abstract. In the face of the increasing trend in application development to interact with more and more remote services, and cognizant of the fact that issues arising from data consistency and task coordination are core challenges in distributed programming, the systems and data management communities have taken a keen interest in developing eventually consistent coordination-free models of distributed programming. These efforts have a striking similarity; they can all be characterized by the use of monotone functions as fundamental primitives of composition as well as the monotonic evolution of data over time. Yet, ensuring that application code conforms to the monotonicity constraints of these programming models is a tricky and manual affair, without support from the underlying language or system. In this paper, we present Monotonicity Types, a language and type system for proving functions monotone, which we believe could enable the customization and extension of this class of distributed programming models. We provide a full formalization of Monotonicity Types, including a novel operational semantics oriented from the perspective inside of a function, as well as a type soundness proof using logical relations.

Keywords: CRDTs, distributed computing, concurrency, types

1 Introduction

Programming is no longer a closed-world affair. Nowadays, essentially every application must make and handle remote calls to myriad other services. Regular application developers today generally have no choice but to write distributed programs. Yet, building correct, performant, and reliable distributed systems continues to be a challenge. Those challenges associated with distribution—concurrency, partial failure, issues with availability and performance—can actually often be viewed as challenges in data management surrounding data consistency and task coordination [3]. Faced with the reality that virtually all applications must interact with multiple services distributed over nodes spread increasingly across the globe, and that issues surrounding data consistency are often at the heart of the challenges faced by developers of such systems, the systems and data management communities have identified two sets of design principles to help developers navigate these issues; techniques that require coordination, and techniques that don't.

Traditional approaches typically involve coordination: either in the form of state machine replication via a consensus algorithm such as Paxos, or by using distributed transactions. That is, both of these techniques require that concurrently executing operations either synchronously communicate or stall in order to complete. While coordination in these systems ensures strong consistency, it limits its concurrency at the cost of scalability, availability, and performance. Thus, as the need for applications to interact with more and more remote services grows, these techniques don't scale with that need.

More recently, it's become evident that coordination isn't always necessary to preserve application invariants [7,12]; for some parts of an application, weaker consistency can be tolerated in the interest of better availability (e.g., the exact number of likes on a tweet; it's likely OK most of the time to have a view of this data that is perhaps a few seconds stale). Such situations may arise from variable ordering of reads, writes, and messages in the face of a network partition or partial failure. This weaker form of consistency is known as *eventual consistency*, meaning that nodes in a distributed system will eventually agree on a definitive data value (sometimes resolving conflicts along the way) so that readers of that data will eventually all see the same value.

In this vein, several efforts have recently presented various programming models or data structures which aim to give programmers ways to reason about and reduce coordination without affecting application correctness by guaranteeing that programs are eventually consistent. These efforts include data structures designed to be replicated and distributed across a network like Conflict-Free Replicated Data Types (CRDTs) [20,19], to new programming models and languages which aim such as Bloom [3], Bloom^L [9], and Lasp [15].

A common thread across these efforts is that *monotonicity* across data types and operations is the key to providing eventual consistency across replicated objects in a distributed system. For example, the CALM theorem [4], which stands for Consistency And Logical Monotonicity, formally proves that logically monotonic programs over sets are guaranteed to be eventually consistent. Informally, this means that a block of code is logically *monotonic* if the following holds; adding things to the input can only increase the output. In contrast, a *non-monotonic* code block may need to retract a previous output if more is added to its input [1]. Said simply, eventually consistent programs without coordination can be expressed in monotonic program logic, while non-monotonic programs (those requiring destructive state modification or aggregation operations) require coordination; that is, they must be resolved via distributed coordination such as two-phase-commit or Paxos.

Bloom [3], a language that supports coordination-free distributed programming over programs whose sets grow monotonically using a Datalog-like programming abstraction, which was co-introduced alongside of the CALM theorem, only supported sets. Bloom was generalized to Bloom^L [9], which extends Bloom's assessment of monotonicity from sets to arbitrary join-semilattices, enabling the support for more interesting (monotonic) data types such as maps, or otherwise programs that "grow" according to a partial order other than set containment. Morphisms between lattices can also be defined in Bloom^L which make it possible to ensure that per-component guarantees can be extended across different lattice types. Both Bloom and Bloom^L provide montotone functions, which taken together with Bloom^L's morphisms enable monotonicity-preserving mappings between lattices.

However, Bloom^L has its limitations; (1) it remains a first-order language, where (2) morphisms and monotone functions must be user-specified to be morphisms or monotone functions. The system has no way to ensure that a monotone function annotated as such is indeed monotone. Rather, Bloom^L simply checks that user-defined morphisms and monotone functions preserve monotonicity in their arguments by merely being tagged by the user as monotonic. It assumes that the monotonicity annotations on functions are correct as it cannot analyze these functions to determine whether or not they are indeed monotonic.

Lasp [15], a functional programming model over CRDTs, improves on Bloom^L by introducing higher-order programming over lattices, with combinators such as `map` and `filter`. In particular, a main goal of Lasp is to enable the deterministic composition of CRDTs into larger composite computations that remain guaranteeably eventually consistent. Compared to Bloom^L, all of the expressible operations in Lasp are morphisms (which are by necessity monotone functions). However, while providing a higher-order programming facility and the ability to define new combinators, Lasp provides no mechanism to ensure that new user-defined combinators will exhibit the required properties of monotonicity.

Thus, while monotonicity is an essential property in the context of data structures and programming models for coordination-free distributed programming, in all cases, users are expected to understand and enforce monotonicity in their application code without support from the underlying language or system.

In this paper, we propose *Monotonicity Types*, a small language and type system with the following two overarching goals:

- **Support for higher-order programming**. e.g., the use of higher-order functions like `map` and `filter` over lattice-based replicated data structures such as CRDTs.
- **Support for tracking monotonicity with types**. e.g., prove functions monotone

Since monotone functions are used as primitive transformations across emerging coordination-free distributed and concurrent programming models and data types like Lasp, Bloom^L, LVars [13,14], amongst others, we believe that Monotonicity Types are a missing piece which could enable customization and extension of such systems by non-experts. For example, if applied in the context of Bloom^L, Monotonicity Types would be able to guarantee that first order user-defined morphisms and monotone functions preserve monotonicity in their arguments, while for Lasp, Monotonicity Types would be able to guarantee that combinators defined using higher order functions (e.g., compositional operations like `map` and `fold`) are monotonic in each of their arguments.

This paper has the following contributions:

- **A language and type system for proving functions monotone**, which:
  - is situated toward compositional programming with CRDTs, in the style of Lasp.
  - provides a novel foundation for proving relational properties with lightweight types.
- **Formal semantics**. A novel operational semantics oriented from the perspective of inside a function, which
  - describes the function's structure through a sequence of function compositions.
  - corresponds faithfully to the function's applicative behavior.
  - when characterized statically, allows tracking monotonicity across function composition.
- **Soundness proof**. A proof that the operational semantics of the calculus is soundly approximated by the type system using logical relations.

2 Conflict-Free Replicated Data Types

In this section we briefly introduce state-based Conflict-Free Replicated Data Types (CRDTs) [20,19], a core concept around which our language and type system is built (Monotonicity Types is aimed at supporting compositional programming with CRDTs), and a central concept in our running examples.

Informally, *Conflict-Free Replicated Data Types* are distributed data types that allow different replicas of a distributed CRDT instance to diverge while ensuring that, eventually, all replicas converge to the same state. State-based CRDTs achieve this through propagating updates of the local state by disseminating the entire state across replicas. The received states are then merged with remote states, leading to convergence (i.e., consistent states across all replicas).

A state-based CRDT, as defined in [2], consists of a triple (S, M, Q), where S is a join-semilattice [10], Q is a set of query functions (which return some result without modifying the state), and M is a set of mutators that perform updates; a mutator $m \in M$ takes a state $X \in S$ as input and returns a new state $X' = m(X)$. A join-semilattice is a set with a *partial order* $\sqsubseteq$ and a binary *join* operation $\sqcup$ that returns the *least upper bound* (LUB) of two elements in S; a *join* is designed to be commutative, associative, and idempotent. Mutators are defined in such a way to be *inflations*, i.e., for any mutator m and state X, the following holds: $X \sqsubseteq m(X)$.

Thus, for each replica, there is a monotonic sequence of states, defined under the lattice partial order, where each subsequent state subsumes the previous state when joined elsewhere.

We assume that for each CRDT: M is a set of user-specified inflative mutators and Q is a set of functions that derive a value from the join-semilattice. Given each mutator computes the join with the current semilattice value, mutators are guaranteed to be monotonic. Throughout the remainder of this text, when we refer to lattice or semilattice, we mean a join-semilattice.

3 Motivating Examples

To demonstrate the challenges of writing monotonic application code, we examine three seemingly simple examples, each of which that fail to preserve monotonicity through composition and function application: a unary function, an n-ary function, and a higher-order function.

3.1 Unary Function

Our unary function example uses two primitive types:

1. Nat: where S is the join-semilattice with the set of naturals as elements, 0 as bottom, $\leq$ as the ordering relation, and max as the join operation; M is the singleton set of assignment, that takes the join between a given value and the current value; and Q is the singleton set of query that returns the current value;
2. Bool: where S is the join-semilattice with the set of booleans as elements, false as bottom, false $\leq$ true as the ordering relation, and $\vee$ as the join operation; M is the singleton set of assignment, that takes the join between the given value and the current value; and Q is the singleton set of query that returns the current value.

In Figure 1, we define a function that operates on an instance of the Nat type and computes whether or not the Nat is odd using the modulo operation. Seemingly simple, while the input of this function will monotonically increase, the output of this function is neither monotone or antitone; the output of this function will alternate between true and false as the input monotonically increases.

```
fun IsOdd(x : Nat) : Bool = X % 2 != 0
```

Fig. 1: Unary function that computes whether a natural is odd or not.

3.2 n-ary Function

Our n-ary function example uses one primitive type and one type constructor:

1. NatSet: where S is the join-semilattice with all finite subsets of the natural numbers as elements, the empty set as bottom, set inclusion the ordering relation, and set union as the join operation; M is singleton set containing the operation insert, that adds an element into the set; and Q is the singleton set of the operation query, that returns the current value;
2. Record S1 $\times$ S2 $\times$ M $\times$ Q $\rightarrow$ S $\times$ M $\times$ Q: where S is the join-semilattice produced by the product of two join-semilattices S1 and S2, ordered componentwise; M is a user specified set of inflative mutators; and Q is a user specified set of query functions.

By composing NatSet's using the record data type we can implement the 2P-Set as defined by Shapiro et al. [19] The 2P-Set supports the one-time addition

```
type 2P-Set = {A : NatSet, R : NatSet}

fun 2PIntersect(a : 2P-Set, b : 2P-Set) : 2P-Set =
  { A = intersect(a.A, b.A) - union(a.R, b.R), R = empty }
```

Fig. 2: n-ary function that returns the intersection of two 2P-Sets.

and removal of an element by composing two NatSets, each of which increases monotonically.

In Figure 2, we define a function that computes the intersection of two *2P-Set*'s by taking the intersection of its addition component and the union of its removal component. Seemingly correct, this function is non-monotonic in output, if you examine the case where an element has been added to both sets and removed from only one of them.

3.3 Higher Order Functions

To demonstrate the problems with higher order programming, we use a single primitive operation on collections.

1. Filter: which operates on a NatSet by filtering the elements of the set using some predicate. Therefore, given a predicate function, filter will return a function from NatSet to NatSet.

```
fun FilterOdd(x : NatSet) : NatSet = Filter(IsOdd, x)
```

Fig. 3: Binary function that filters a NatSet given a predicate.

In our example, a predicate function filtering the odd elements will result in non-monotonic output as the naturals inside of the set increase.

4 Monotonicity Types

We saw in the previous section that it's tricky to always know whether or not some code is monotonic. For example, the unary function isOdd from Section 3.1 is useful, but non-monotone. To prevent programmers from violating the monotonicity constraints present in such scenarios, we'd like to have a type system which can prove functions monotone.

Ideally, such a system would increase the programmer's awareness and fluency with monotonicity by inferring the monotonicity of program expressions, and also allow type annotations to serve as a design language for asserting the monotonicity of a function's constituent parts.

4.1 Motivating Example

Recall the $2PIntersect$ example from Section 3.2. Wouldn't it be nice if we could annotate this function to instruct the compiler to verify that $2PIntersect$ is indeed monotonic?

In our language for monotonicity typing, we introduce a special kind of function, called an *sfun*, which is augmented with qualifiers. These qualifiers instruct the type checker to verify that the function is either monotone or antitone in its arguments.

Below we have adapted the example from Section 3.2 to use an sfun, and we have annotated its type parameters with the monotone qualifier, ↑.

```
1 type 2P-Set = { A : NatSet, R : NatSet }
2
3 sfun 2PIntersect(a : 2P-Set, b : 2P-Set) : 2P-Set[↑ a, ↑ b] =
4   { A = intersect(a.A, b.A), R = union(a.R, b.R) }
```

Fig. 4: An intersection combinator for the *2P-Set* CRDT

On line 3, the type annotation *2P-Set*[↑ a, ↑ b] indicates that the combinator produces a *2P-Set* value and is monotone separately in each of its arguments. But how do we go about proving this through type-based reasoning? As we shall see shortly, we can reason about the monotonicity of complex functions by tracking the propagation of monotonicity across primitives.

4.2 Sfuns

An sfun is a special multi-argument function abstraction for which monotonicity type-checking is applied. Sfuns have a special form of function type, written $(x_1 : B_1, x_2 : B_2, \ldots) \Rightarrow B[q_1\ x_1, q_2\ x_2, \ldots]$, called an *sfun type*, which has multiple named arguments and associates the *qualifier* q_i to each argument x_i. A qualifier expresses an argument-specific constraint placed on an sfun. Qualifying the ith argument with ↑, for example, requires that the function is separately monotone in that argument: if $c_i \le c_i'$ then $f(c_1, \ldots c_i, \ldots c_n) \le f(c_1, \ldots, c_i', \ldots, c_n)$.

We provide other qualifiers in addition to ↑. ↓ qualifies arguments that are separately antitone. Another qualifier, $\sim$ is associated with a function argument whenever changes in the argument have no effect on the function's result. The qualifier $=$ is associated with a function argument if the function simply produces the value supplied for that argument as its result. Finally, associating the qualifier ? with an argument places no restrictions on the behavior of the function with respect to changes in that argument. Since we must always qualify an sfun's arguments, ? is used as a fallback for when we can't guarantee that an argument respects any of the other qualifiers. There is an precision ordering on these qualifiers, displayed in *Figure 5*. We'll examine the formal significance of this ordering in section 5.9.

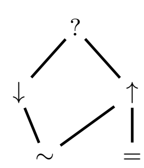

Fig. 5: A Hasse diagram of the partial order on qualifiers

4.3 Sfuns in Action

Let's examine what's happening in the sfun in the above example in Figure 4.

$2PIntersect$ utilizes two primitive binary operators provided for working with the $NatSet$ type.

$$intersect : (x : NatSet, y : NatSet) \Rightarrow NatSet[\uparrow\ x,\ \ \uparrow\ y]$$

$$union : (x : NatSet, y : NatSet) \Rightarrow NatSet[\uparrow\ x,\ \ \uparrow\ y]$$

intersect and *union* associate both of their arguments with the ↑ qualifier, meaning that each function is monotone separately in both of its arguments. The definition of a record type such as *2P-Set* automatically generates sfuns for projecting its components. For example, the definition of *2P-Set* generates projection sfuns $projA$ and $projR$, both monotone due to *2P-Set*'s componentwise ordering, with the following types:

$$projA : (x : \mathit{2P\text{-}Set}) \Rightarrow NatSet[\uparrow\ x]$$

$$projR : (x : \mathit{2P\text{-}Set}) \Rightarrow NatSet[\uparrow\ x]$$

The application of an sfun f to a list of arguments $a_1, a_2, \ldots$ is written $f(a_1, a_2, ...)$. Dot notation for field projection is syntactic sugar for the application of a projection sfun; for example, on line 4 of *Figure 4*, $a.A$ is syntactic sugar for the sfun application $projA(a)$.

As with projections, an sfun for a record type's value constructor is generated implicitly; the constructor *make-2P-Set* for the type *2P-Set* has the following type:

$$\mathit{make\text{-}2P\text{-}Set} : (A : NatSet, R : NatSet) \Rightarrow \text{2P-Set}[\uparrow\ A,\ \ \uparrow\ R]$$

Line 4 of *Figure 4* therefore desugars to a term consisting entirely of sfun applications:

$$\mathit{make\text{-}2P\text{-}Set}(intersect(projA(a),\ projA(b)),\ union(projR(a), projR(b)))$$

We would like a type system which can prove not only that this term normalizes to a *2P-Set* after a and b are instantiated with $NatSet$s, but also that when the term is treated as a multi-argument function of a and b, it is monotone separately in each argument.

4.4 Monotonicity Typing

In a standard type system, a typing derivation of the following form can be viewed as reasoning about a composition of functions.

$$\frac{\Gamma \vdash c : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash c\ s : T}$$

The left premise $\Gamma \vdash c : S \to T$ states that c is a function with domain S and codomain T. The right premise $\Gamma \vdash s : S$ states that s is a multi-argument function, the domain of which is the set of valuations of the type environment Γ, and the codomain of which is S. The conclusion states that, because the codomain of s is equal to the domain of c, we can form a composite function corresponding to $\Gamma \vdash c\ s : T$ by forwarding the result of s into c; this composite function is a multi-argument function from the valuations of Γ into T.

The above derivation is concerned only with the domains and codomains of functions. Our key idea is that, by using a richer type system, we can reason about other function properties. In particular, we develop a type system which can prove a function monotone by demonstrating it is a composition of monotone functions.

Central to our approach is acknowledging the distinction between the formal and actual arguments of a multi-argument function, and providing a way to refer directly to formal arguments. Any variable that occurs in type syntax is considered a reference to a formal argument. On the other hand, any variable which occurs in a term is considered a placeholder for an actual argument. Because the contexts in which references to formal arguments and placeholders for actual arguments occur are mutually exclusive, we can refer both to a formal argument and to its corresponding actual argument using the same identifier, unambiguously.

With the exception of sfuns, type-checking a program in our language is not fundamentally different from type checking the simply typed lambda calculus. But to type-check the body of an sfun abstraction such as *Figure 4*'s $2PIntersect$, we defer to a richer set of typing rules called the *lifted type system*, a type system oriented from the perspective of inside a function. Our type system prohibits the nesting of sfun abstractions inside other sfun abstractions; a lifted typing derivation therefore describes a term enclosed in exactly one sfun abstraction. Its type environment is split into three components Γ, Ω, and Φ. The *terminal type environment* Γ describes the context of the enclosing sfun; it remains fixed across any lifted typing derivation. The *ambient environment* Ω lists all formal arguments of the enclosing sfun (which we also call *ambient variables*); it too remains fixed across any lifted typing derivation. Finally, the *lifted type environment*, written Φ, contains all variables bound within the sfun abstraction. Importantly, types in the lifted type environment can be augmented with qualifiers constraining their dependence on the ambient variables.

As an example, we consider type checking the expression $a.A$ (desugared into $projA(a)$) on line 4 of the $2PIntersect$ sfun abstraction from *Figure 4*. Let Γ be the terminal type environment of the $2PIntersect$ abstraction, and let Ω be $2PIntersect$'s typed formal argument list taken verbatim from the program:

$$\Omega \doteq a : \mathit{2P\text{-}Set}, b : \mathit{2P\text{-}Set}$$

Finally, let Φ be the lifted type environment obtained from qualifying each of $2PIntersect$'s typed formal arguments:

$$\Phi \doteq a : \mathit{2P\text{-}Set}[=\ a,\ \sim\ b], b : \mathit{2P\text{-}Set}[\sim\ a, =\ b]$$

Then we have a typing derivation ending with a deduction of the following form:

$$\frac{\Gamma; \Omega; \Phi \vdash projA : (x : \mathit{2P\text{-}Set}) \Rightarrow NatSet[\uparrow x] \qquad \Gamma; \Omega; \Phi \vdash a : \mathit{2P\text{-}Set}[= a, \sim b]}{\Gamma; \Omega; \Phi \vdash projA(a) : NatSet[\uparrow a, \sim b]}$$

Just as with the standard typing rule for function application, we can view this as a composition of functions. The left premise tells us that $projA$ is a monotone function with domain *2P-Set* and codomain $NatSet$. The right premise tells us that a is a function, the domain of which is the ambient environment a : *2P-Set*, b : *2P-Set*, and the codomain of which is *2P-Set*. Additionally, the qualifier list $[= \ a, \sim \ b]$ tells us that this function is a selector for $2PIntersect$'s formal argument a; that is, it returns the actual argument provided to $2PIntersect$ for a and ignores the actual argument provided to $2PIntersect$ for b. The application $projA(a)$ can be viewed as a binary function, the formal arguments of which are the entries a : *2P-Set* and b : *2P-Set* of Ω, and the codomain of which is $NatSet$. At this point, how do we then determine the application's most precise qualifier for the formal argument a?

$\circ$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
?	?	?	?	?	$\sim$
$\uparrow$	?	$\uparrow$	$\downarrow$	$\uparrow$	$\sim$
$\downarrow$	?	$\downarrow$	$\uparrow$	$\downarrow$	$\sim$
$=$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$

Fig. 6: Qualifier composition $\circ$

$+$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
?	?	?	?	$=$	?
$\uparrow$	?	$\uparrow$	?	$=$	$\uparrow$
$\downarrow$	?	?	$\downarrow$	$=$	$\downarrow$
$=$	$=$	$=$	$=$	$=$	$=$
$\sim$	?	$\uparrow$	$\downarrow$	$=$	$\sim$

Fig. 7: Qualifier contraction $+$

4.5 Qualifier Composition

In general, to compute a single-argument sfun application's most precise qualifier for some $z \in \Omega$, we must draw from the information provided to us by the premises of the sfun application's typing derivation. The left premise provides a qualifier q for the sole argument of the sfun being applied. The right premise provides a qualifier p which describes the application argument as a function of z.

The table of *Figure 6* maps the pair of qualifiers p and q to the most precise qualifier that we can safely conclude for the application.

Let's look at how this strategy applies to determining the most precise qualifier for the formal argument a in the previous section. We combine a's qualifier $=$ for a with $projA$'s qualifier $\uparrow$ for its sole formal argument to determine that the application's qualifier for a is $\uparrow \circ =$ which *Figure 6* tells us is $\uparrow$. Likewise, the application's qualifier for b is $\uparrow \circ \sim$, which an examination of *Figure 6* reveals to be $\sim$.

4.6 Qualifier Contraction

Computing the qualifiers of a multi-argument sfun application requires additional care. Suppose that we have a multiplication operator *mult* on natural numbers with the following sfun type:

$$mult : (x : Nat, y : Nat) \Rightarrow Nat[\uparrow x, \uparrow y]$$

Then the following typed term-in-context represents the squaring operation on natural numbers:

$$\cdot; x : Nat; x : Nat[= \ x] \vdash mult(x, x) : Nat[\uparrow x]$$

The squaring operator is obtained simply by identifying the two formal arguments of *mult*. Performing such an identification is called *argument contraction*; along with function composition, it is one of the two function operations used to propagate monotonicity. When we know that two of a function's formal arguments x and y respect the qualifiers q_x and q_y (for example, *mult*'s formal arguments x and y respect the qualifiers $\uparrow$ and $\uparrow$), we can conclude that the formal argument resulting from the contraction of x and y respects the qualifier $q_x + q_y$, where $+$ is the *qualifier contraction* operator defined in *Figure 7*. Since *mult*'s arguments both respect the qualifier $\uparrow$, their contraction, the sole formal argument of the squaring operator, respects $\uparrow + \uparrow$, which is equal to $\uparrow$.

4.7 Advanced Example

```
-- Map operations (sfuns)
getAt :: (m : NatMap, k : Nat) ⇒ Nat[↑ m, k]
joinAt :: (m : NatMap, k : Nat, n : Nat) ⇒ NatMap[↑ m, k, ↑ n]
span :: (x : NatMap) ⇒ Nat[↑ x]
emptyMap :: NatMap

-- Nat operations (sfuns)
max :: (a : Nat, b : Nat) ⇒ Nat[↑ a, ↑ b]
+ :: (x : Nat, y : Nat) ⇒ Nat[↑ x, ↑ y]
> :: (x : Nat, y : Nat) ⇒ Bool[↑ x, ↓ y]

type GCounter = { map : NatMap }

sfun sumCounters(x : GCounter, y : GCounter) : GCounter[↑ x, ↑ y] =
  let xMap : NatMap[↑ x, ↑ y] = x.map
  let yMap : NatMap[↑ x, ↑ y] = y.map
  let maxSpan : Nat[↑ x, ↑ y] = max (span xMap) (span yMap)
  fun sumCell(k : Nat, acc : NatMap[↑ x, ↑ y]) : NatMap[↑ x, ↑ y] =
    let cond : Bool[↓ x, ↓ y] = k > maxSpan
    if cond then
      acc
    else
      let acc' = joinAt acc k ((getAt xMap k) + (getAt yMap k))
      sumCell (k+1) acc'
  let initMap : NatMap[↑ x, ↑ y] = emptyMap
  { map = sumCell 0 initMap }
```

Fig. 8: A coordinatewise sum of two GCounters

We now turn our attention to a more interesting and aspirational example of a CRDT combinator that we would like our type system to support in, shown in *Figure 8*. Since correct CRDT combinators are necessarily monotone, this example demonstrates that a monotonicity type system must generally handle functions composed from a diverse collection of language constructs rather than the homogeneous chunk of sfun applications that we saw in the $2PIntersect$ example.

The combinator *sumCounters* takes two G-Counter [19] values as input, and produces a G-Counter holding the sum of the two input counters as output.

Lines 1-10 assert the types of several built-in sfun operations for working with the Nat and $NatMap$ types. $NatMap$ is the type of total maps from the natural numbers to the natural numbers, represented as finite sets of input/output pairs. $getAt(m, k)$ gets the value to which the $NatMap$ m maps the natural number k. If k is not in the domain of m, $getAt(m, k)$ returns 0. $joinAt(m, k, n)$ returns a new NatMap equal to m, except that its kth component has been replaced with the join of n with m's kth component. $span(m)$ returns the greatest natural number which m maps to a non-zero value. The $sumCounters(x, y)$ combinator iterates through the components of the two input G-Counters, placing the sum of x and y's kth components in the kth component of the result.

Importantly, this example contains a lifted function abstraction *sumCell*, aware of monotonocity of the formal arguments of its enclosing sfun abstraction. It also demonstrates the need to substitute values, like $emptyMap$ on line 25, through an sfun abstraction. This is interesting considering that any term defined outside an sfun abstraction is type-checked with the terminal type system, while terms defined inside an sfun abstraction are type-checked with the lifted type system. Noting that *sumCell* is an instance of the fold operation, we see that there's also a need to substitute polymorphic higher-order functions (like fold) through sfun abstractions.

We distill this example into two important features that a monotonicity type system should support.

- Lifted function abstractions. A lifted function abstraction is nested inside of an sfun body and is aware of the formal arguments of its enclosing sfun.
- Subtitution through sfun abstractions. A variable bound outside of an sfun abstraction should be allowed to occur inside of it. The programmer should be free to use this variable wherever it is deemed useful, without impeding monotonicity typing. Such variables may be bound to data values, higher order functions, or sfuns.

These points will motivate the design of the calculus presented in the following section.

5 Formalization

We present a calculus featuring a type system exhibiting the ability to propagate montonicity across functions composition and contraction, the need for which was demonstrated in the examples above. Proofs of stated theorems, along with a full formalization, are in the appendix.

5.1 Notational conventions

We write $1..n$ to denote the set containing the first n consecutive positive integers $\{1, 2, ..., n\}$. We often use the notation $1..n$ without first introducing the variable n; in these cases it is assumed that n is an arbitrary positive integer. We denote a vector of homogenous syntax fragments by writing a colored line over a pattern which all fragments in the vector conform to. For example, if x is the metavariable used to denote variables and T is the metavariable used to denote types then $\overline{x:T}$ denotes a vector of variables ascribed with types. We use superscript notation to specify a set used to index the elements of such a vector. For example, $\overline{x_i : T_i}^{i \in 1..n}$ denotes a vector of n typed variables indexed by the first n positive integers. We can then refer to the first variable in the vector by x_1, the second type in the vector by T_2, etc. When the index set is clear from context, it will be left explicit. When multiple syntax vectors in the same context have the same index set, we denote them using overlines of the same color. When multiple syntax vectors in the same context have distinct index sets, we denote them using overlines with distinct colors. When we write a qualified base type $B[=\ x]$, we implicitly qualify all ambient variables other than x (if any) with $\sim$.

5.2 Valuations

A *valuation* of a type environment Γ, written γ, is an assignment of each variable in the $dom(\Gamma)$ to a value of type $\Gamma(x)$. The *substitution* of a valuation γ into a term t, written γt, is the result of recursively descending into t, replacing any encountered variable x with the value $\gamma(x)$. Likewise, we use the symbol ω for valuations of ambient environments and ϕ for valuations of lifted type environments.

5.3 Lifted Base Values and Types

Consider the following lifted typing judgment, which is similar to the typing judgment for the body of the squaring function from *Section 4.4*.

$$\cdot; x : Nat; x : Nat[=\ x] \vdash mult(x, 3) : Nat[\uparrow x]$$

We interpret types as sets of values; for example, we interpret Nat as the set $\{\ldots, 0, 1, 2, \ldots\}$. This typing judgment involves the familiar type Nat, but also the novel-looking types $Nat[=\ x]$ and $Nat[\uparrow\ x]$: how should we interpret *these* types? Recall that base values are those values which describe pieces of data; 3 and 4 are base values whereas $(\lambda x : Nat.\ mult(x, 3))$ is not. Additionally, for any type T, if the values described by T are exclusively base values, we call T a *base type*; for example, Nat is a base type but $Nat \to Nat$ is not a base type. In a lifted typing judgment, all base values are viewed as constant-valued functions of the ambient environment, and so the occurrence of 3 in the above judgment is considered a function which maps any valuation of $x : Nat$ to the natural number 3.

We generalize the base values of our language to include *ambient maps*, which are essentially "lifted base values". Under an ambient environment Ω, an ambient map, which is typically written with the metavariable a, is a function from the valuations of Ω into the values of some base type. Ambient maps are purely extensional: we can only observe them by applying them to valuations of their ambient environment. Unlike lambda abstractions, ambient maps provide no description of the process used to compute the output corresponding to an input. An ambient map with whose domain is the set of valuations of ambient environment Ω is call an Ω-*ambient map*.

We're now ready to provide the interpretation of *lifted base types* such as $Nat[=\ x]$ and $Nat[\uparrow\ x]$. A qualified base type is interpreted with respect to an ambient environment Ω. For any Ω-ambient map, base type B and qualifier map Ξ with $dom(\Xi) = dom(\Omega)$, the qualified base type $B[\Xi]$ is interpreted as the set of all B-valued ambient maps which respect the qualifiers of Ξ. The way in which each qualifier constrains a lifted base type is defined formally in *Figure 15*. As a concrete example, let Ω be the ambient environment $x : Nat$ and a_x the Ω-ambient map which maps any valuation ω of $x : Nat$ to the natural number $\omega(x)$. The type $Nat[=\ x]$ is interpreted under Ω as the singleton set containing only a_x.

For the benefit of intuition, we will occasionally use the following "concrete" notations for ambient valuations and ambient maps. Let $\overline{x_i : B_i}^{i \in 1..n}$ be an ambient environment. For $i \in 1..n$, let c_i be a value of type B_i, then we write $(\overline{x_i \mapsto c_i})$ to denote the ambient valuation of $\overline{x_i : B_i}$ which maps x_i to c_i for $i \in 1..n$. Ambient maps are written concretely using standard set notation from mathematics. The ambient map a_x discussed above is written concretely as:

$$\{((x \mapsto 0), 0), ((x \mapsto 1), 1), ((x \mapsto 2), 2), \ldots\}$$

5.4 Syntax

Noting that sfun abstractions are written $(\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)$, we're now familiar with almost all of the novel syntax of our calculus, the entirety of which is listed in *Figure 9*.

$$
\begin{array}{rcll}
x & ::= & \textit{variables} & \\
a & ::= & \textit{ambient maps} & \\
p, q & ::= & \uparrow\ \mid\ \downarrow\ \mid\ \sim\ \mid\ ?\ \mid\ = & \textit{Qualifiers} \\
\Xi & ::= & \cdot \mid \Xi,\ q\ x & \textit{Qualifier maps} \\
c & ::= & \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & \textit{Terminal base-level constants} \\
d & ::= & c \mid a & \textit{Base-level constants} \\
k & ::= & + \mid - \mid\ <\ \mid\ \leq\ \mid\ =\ \mid \wedge \mid \vee & \textit{Sfun constants} \\
A, B & ::= & \mathit{Bool} \mid \mathit{Nat} & \textit{Base-level types} \\
S, T, U & ::= & B \mid B[\Xi] \mid (\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi] \mid & \textit{Types} \\
 & & S \rightarrow T \mid S \overset{\Omega}{\rightarrow} T & \\
v & ::= & k \mid d \mid a \mid (\lambda x : T.t) \mid (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) & \textit{Values} \\
s, t, u & ::= & v \mid x \mid t\ t \mid t(\overline{t_i}^{i \in 1..n}) \mid & \textit{Terms} \\
 & & \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t & \\
E & ::= & E\ t \mid v\ E \mid E(\overline{t}) \mid v(\overline{v}, E, \overline{t}) \mid & \textit{Evaluation contexts} \\
 & & \textbf{if}\ E\ \textbf{then}\ t\ \textbf{else}\ t \mid [] & \\
\Gamma & ::= & \cdot \mid \Gamma, x : T & \textit{Terminal type environments} \\
\Omega & ::= & \cdot \mid \Omega, x : B & \textit{Ambient type environments} \\
\Phi & ::= & \cdot \mid \Phi, x : T & \textit{Lifted type environments}
\end{array}
$$

Fig. 9: Syntax

A *lifted function type* $S \overset{\Omega}{\to} T$ is the type of functions which, like the *sumCell* function of *Figure 8*, inherit the ambient environment of an enclosing sfun abstraction.

5.5 Ambient Substitution

A term t for which a lifted typing judgment of the following form can be derived is called a Ω-closed term.

$$\cdot; \Omega; \cdot \vdash t : T$$

In contrast to what we would normally think of as a closed term, an Ω-closed term contains extensional fragments, the set of ambient maps occurring in t. These ambient maps share the common domain Ω. Let $\mathcal{O}[\![\Omega]\!]$ be the set of all valuations of Ω. For any $\omega \in \mathcal{O}[\![\Omega]\!]$, we can form an *ambient substitution* operator $\|\omega\|$, a unary operator on terms. An application of $\|\omega\|$ to a to a term is written $\|\omega\|t$. While defined on all terms, the restriction of $\|\omega\|$ to the Ω-closed terms is well-behaved: it is a projection from the Ω-closed terms onto the terms t for which $\cdot \vdash t : T$ for some T, called the *terminal terms*. It constructs a terminal term by recursively descending into t and replacing any ambient map $\{\overline{(\alpha, c_\alpha)}^{\alpha \in \mathcal{O}[\![\Omega]\!]}\}$ that it encounters with c_ω. To produce a terminal result, ambient substitution must also descend into type annotations occurring in t, converting every lifted type to a terminal type. *Figure 10* provides the full definition of ambient substitution.

$$
\begin{array}{lll}
\|\omega\|a & = a(\omega) & \text{when } \omega \in \mathit{dom}(a) \\
\|\omega\|a & = a & \text{when } \omega \notin \mathit{dom}(a) \\
\|\omega\|t\ t & = \|\omega\|t\ \|\omega\|t & \\
\|\omega\|t(\overline{t}) & = \|\omega\|t(\overline{\|\omega\|t_i}) & \\
\|\omega\|(\lambda x : S.t) & = (\lambda x : \|\omega\|S.\|\omega\|t) & \\
\|\omega\|\textbf{if } t \textbf{ then } t \textbf{ else } t & = \textbf{if } \|\omega\|t \textbf{ then } \|\omega\|t \textbf{ else } \|\omega\|t & \\
\|\omega\|(\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) & = (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ \|\omega\|t) & \\
\|\omega\|t & = t \quad \mathit{otherwise} & \\
 & & \\
\|\omega\|T \overset{\Omega}{\to} T & = \|\omega\|T \to \|\omega\|T & \text{when } \omega \in \mathcal{O}[\![\Omega]\!] \\
\|\omega\|B[\Xi] & = B & \text{when } \mathit{dom}(\omega) = \mathit{dom}(\Xi) \\
\|\omega\|T & = T & \text{otherwise}
\end{array}
$$

Fig. 10: Ambient substitution

5.6 Terminal Reduction and Lifted Reduction

In *Figure 11*, we define a standard small-step reduction, a binary relation on terms written $t \to t'$, called *terminal reduction*. The only novel rule is RED-SAPP, for sfun applications. To apply an sfun abstraction $(\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)$ to a vector of arguments $\overline{c_i}$, we can't merely substitute each c_i for its corresponding variable x_i. That would fail, for example, when performing the following reduction step.

$$(\tilde{\lambda}(x : Nat).\ (\lambda y : Nat[= x].y)\ x)(3) \to (\lambda y : Nat[= x].y)\ 3$$

The problem here is that we would like terminal reduction to take terminal terms to terminal terms, but the above reduction step takes a terminal term to an Ω-closed term, where Ω is the ambient environment containing the single entry $x : Nat$. Unlike the term $(\lambda y : Nat[= x].y)\ 3$, terminal terms do not contain any occurrences of qualifiers outside of sfun abstractions. To fix this, Red-SApp must first substitute a_x for x in the sfun body to obtain the Ω-closed term $(\lambda y : Nat[= x].y)\ a_x$, and then project this Ω-closed term to a terminal term by applying the ambient substitution $\|(x \mapsto 3)\|$ to obtain $(\lambda y : Nat.y)\ 3$.

$$\textsc{Red-Context}\quad \frac{t \to t'}{E[t] \to E[t']} \qquad\qquad \textsc{Red-App}\quad \frac{}{(\lambda x : T.t)\ v \to [v/x]t}$$

$$\textsc{Red-SApp}\quad \frac{\phi \in \mathcal{G}_{\overline{x_i : B_i}}[\![\overline{x_i : B_i[=\ x_i]}]\!] \qquad \omega = \cdot\overline{[x_i \mapsto c_i]}}{(\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ t)(\overline{c_i}) \to \|\omega\|\phi t} \qquad \textsc{Red-SApp-Const}\quad \frac{\delta_n(k) \text{ is defined}}{k(\overline{c_i}^{i\in 1..n}) \to \delta_n(k)(\overline{c_i})}$$

Fig. 11: Selected terminal reduction rules

Recall that the aim of our system is to prove certain functions monotone, functions such as $(\tilde{\lambda}(x : Nat).\ mult(x, x))$. A standard type system could infer the codomain of the non-sfun version of this abstraction $(\lambda x : Nat.mult(x, x))$ by deriving the following judgment:

$$x : Nat \vdash mult(x, x) : Nat$$

The above judgment describes computation which occurs after the abstraction has been applied; roughly, it says that after an actual argument of type Nat is substituted for x, the term $mult(x, x)$ normalizes to a value of type Nat. Here all occurrences of x are placeholders for the actual argument to which the abstraction is applied. This is a lost opportunity, as the following lifted typing judgment for $(\tilde{\lambda}(x : Nat).\ mult(x, x))$ allows us to reason about the compositional behavior of the sfun before it is applied.

$$\cdot; x : Nat; x : Nat[=\ x] \vdash mult(x, x) : Nat[\uparrow\ x]$$

Notably, the ambient environment of this judgment provides a name x for the abstraction's formal argument, and its derivation refers to the formal argument directly wherever the variable x occurs inside of a type. Let $\Psi = x : Nat$. Then, roughly, the above judgment says that after the selector a_x for the enclosing sfun's formal argument x is substituted for occurrences of x in $mult(x, x)$, the resulting term $mult(a_x, a_x)$ normalizes to an Ψ-ambient map of type $Nat[\uparrow\ x]$. None of our terminal reduction rules allow the reduction of $mult(a_x, a_x)$, which is an sfun applied to ambient maps. In fact, terminal reduction, which describes computation, is not the correct reduction relation for interpreting lifted typing judgments, which are intuitively about composition rather than computation. For reasoning about composition, we define a family of binary reduction relations on terms, indexed by ambient environments, for which we write $\Omega \vdash t \to t'$ and say that t Ω-*reduces* to t'. For an ambient environment Ω, we obtain the Ω-*reduction* relation by extending terminal reduction with the top-level reduction rule LRed-SApp.

$$\textsc{LRed-Term}\ \frac{t \to t'}{\Omega \vdash t \to t'} \qquad \textsc{LRed-Context}\ \frac{\Omega \vdash t \to t'}{\Omega \vdash E[t] \to E[t']} \qquad \textsc{LRed-SApp}\ \frac{\overline{\|\omega\| t(\overline{\|\omega\| d_i}) \Downarrow c_\omega}^{\,\omega \in \mathcal{O}[\![\Omega]\!]} \qquad a = \{\overline{(\omega, c_\omega)}\}}{\Omega \vdash t(\overline{d_i}^{\,i \in 1..n}) \to a}$$

Fig. 12: Ω-reduction

Intuitively, if the arguments d_i of an sfun application are Ω-ambient maps, then an sfun application can be viewed as a composition of functions. Each d_i is a function from $\mathcal{O}[\![\Omega]\!]$ into some base type matching one of the sfun's argument types; the ambient map's output is forwarded in as the value for this sfun argument. The resulting composite function is an Ω-ambient map whose codomain matches that of the sfun. The rule LRed-SApp performs such a composition. For each $\omega \in \mathcal{O}[\![\Omega]\!]$, LRed-SApp has one premise of the form $\|\omega\| t(\overline{\|\omega\| d_i}) \Downarrow c_\omega$. Ambient substitution is idempotent on well-typed sfuns: for any well-typed sfun t, we have $\|\omega\| t = t$. Thus for a well-typed sfun application, such a premise is equivalent to $t(\overline{\|\omega\| d_i}) \Downarrow c_\omega$ by substitution.

LRed-SApp is non-algorithmic for two reasons. First, $\mathcal{O}[\![\Omega]\!]$ can be infinite, and in that case there are infinitely many premises. Second, for any given $\omega \in \mathcal{O}[\![\Omega]\!]$, determining if $\|\omega\| t(\overline{\|\omega\| d_i}) \Downarrow c_\omega$ requires solving the halting problem. We could overcome these issues by representing ambient maps as syntactic abstractions rather than sets of pairs. However, for our purposes Ω-reduction need not be algorithmic. Instead, it need only satisfy the following constraints:

- It must be statically characterized by an expressive and intuitive type system.
- It must simulate terminal reduction for all valuations of Ω simultaneously.

The first point implies that an Ω-closed term of type T Ω-normalizes to a value v of type T:

$$\cdot; \Omega; \cdot \vdash t : T \Rightarrow \Omega \vdash t \Downarrow v \qquad \text{(where } v \text{ has type } T\text{)}$$

The second point is stated formally with the following theorem:

Theorem 1. *If* $\Omega \vdash t \Downarrow v$ *then for all* $\omega \in \mathcal{O}[\![\Omega]\!]$ *we have* $\|\omega\| t \Downarrow \|\omega\| v$.

This theorem implies that monotonicity results proven using the lifted reduction relation transfer to terminal reduction. For example, consider the following lifted typing judgment, for the body of the sfun $(\tilde{\lambda}(x : Nat).\ mult(x, x))$:

$$\cdot; x : Nat; x : Nat[=\ x] \vdash mult(x, x) : Nat[\uparrow\ x]$$

Together with this judgment, the fundamental theorem for lifted typing (which we'll cover in *Section 5.10*) implies that $x : Nat \vdash mult(a_x, a_x) \Downarrow d$, where d is an ambient map that is monotone with respect to the formal argument x. In concrete notation, we see that this is the case due to LRed-SApp:

$x : Nat \vdash mult(\{((x \mapsto 0), 0), ((x \mapsto 1), 1) \ldots\}, \{((x \mapsto 0), 0), ((x \mapsto 1), 1) \ldots\}) \rightarrow$
$\{((x \mapsto 0), 0), ((x \mapsto 1), 1), ((x \mapsto 2), 4), ((x \mapsto 3), 9), \ldots\}$

Formally, "d is monotone with respect to the formal argument x" means that for all $\omega_1, \omega_2 \in \mathcal{O}[\![x : Nat]\!]$ with $\omega_1(x) \leq \omega_2(x)$ we have $\|\omega_1\|d \leq \|\omega_2\|d$. But by Red-SApp, for two terminally well-typed sfun applications $(\tilde{\lambda}(x : Nat).\ mult(x, x))(c_1)$ and $(\tilde{\lambda}(x : Nat).\ mult(x, x))(c_2)$ with $c_1 \leq c_2$, we have

$$(\tilde{\lambda}(x : Nat).\ mult(x, x))(c_1) \rightarrow \|\omega_1\| mult(a_x, a_x) \Downarrow \|\omega_1\|d$$

and

$$(\tilde{\lambda}(x : Nat).\ mult(x, x))(c_2) \rightarrow \|\omega_2\| mult(a_x, a_x) \Downarrow \|\omega_2\|d$$

where $\omega_1 = (x \mapsto c_1)$ and $\omega_2 = (x \mapsto c_2)$. Since $\omega_1(x) = c_1 \leq c_2 = \omega_2(x)$, we know that $\|\omega_1\|d \leq \|\omega_2\|d$.

5.7 Type Well-Formedness

The *terminal type well-formedness* relation of *Figure 13* contains the set of types which are meaningful to the terminal typing relation.

$$\textsc{Wf-Base}\quad \frac{}{\vdash B} \qquad \textsc{Wf-Fun}\quad \frac{\vdash S \quad \vdash T}{\vdash S \rightarrow T} \qquad \textsc{Wf-SFun}\quad \frac{dom(\Xi) = \{x_i \mid i \in 1..n\}}{\vdash (\overline{x_i : B_i}^{\,i \in 1..n}) \Rightarrow A[\Xi]}$$

Fig. 13: Terminal type well-formedness

The set of types considered meaningful to the lifted typing relation under an ambient environment Ω are called Ω-*well-formed types*. The definition Ω-well-formedness is provided in *Figure 14*. For any ambient environment Ω the Ω-well-formed types form a superset of the terminally well-formed types, due to the inclusion of the ΩWF-Terminal rule.

$$\Omega\textsc{Wf-Terminal}\quad \frac{\vdash T}{\Omega \vdash T} \qquad \Omega\textsc{Wf-QualBase}\quad \frac{dom(\Omega) = dom(\Xi)}{\Omega \vdash B[\Xi]} \qquad \Omega\textsc{Wf-LFun}\quad \frac{\Omega \vdash S \quad \Omega \vdash T}{\Omega \vdash S \xrightarrow{\Omega} T}$$

Fig. 14: Lifted type well-formedness

A type environment Γ is considered well-formed if $\vdash T$ for all $x : T \in \Gamma$. A lifted type environment Φ is considered Ω-well-formed if $\Omega \vdash T$ for all $x : T \in \Phi$. Let $\omega \in \mathcal{O}[\![\Omega]\!]$. When restricted to Ω-well-formed types, $\|\omega\|$ is a projection onto the terminally well-formed types.

Theorem 2 (Projection of Well-Formed Types). *Let Ω be an ambient environment, and let S and T be types. If $\Omega \vdash T$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$, $\vdash \|\omega\|T$. Furthermore, if $\vdash S$ then $\|\omega\|S = S$.*

5.8 Logical Relations

We use the *logical relations* soundness proof method [16]. For each terminally well-formed type T, we define the set $\mathcal{V}\llbracket T\rrbracket$ of all values belonging to type T and the set $\mathcal{T}\llbracket T\rrbracket$ of all terms which normalizes to values of type T. For each terminally well-formed type environment Γ, we define $\mathcal{G}\llbracket\Gamma\rrbracket$ to be the set of all valuations γ of Γ such that for all $x \in \mathit{dom}(\Gamma)$, $\gamma(x) \in \mathcal{V}\llbracket\Gamma(x)\rrbracket$.

Likewise, for each Ω-well-formed type T, we define the set $\mathcal{V}_\Omega\llbracket T\rrbracket$ of all values beloning to type T and the set $\mathcal{T}_\Omega\llbracket T\rrbracket$ of all terms which normalize to values of type T. For each Ω-well-formed lifted type environment Φ, we define the set $\mathcal{G}_\Omega\llbracket\Phi\rrbracket$ of all valuations ϕ of Φ such that for all $x \in \mathit{dom}(\Phi)$, $\phi(x) \in \mathcal{V}_\Omega\llbracket\Phi(x)\rrbracket$. Selected logical relations are given in *Figures 15.* While our terminal logical

$$
\begin{aligned}
&\mathcal{V}\llbracket Nat\rrbracket && \doteq \{0,1,2,\ldots\}\\
&\mathcal{V}\llbracket S \to U\rrbracket && \doteq \{v \mid \forall v_s \in \mathcal{V}\llbracket S\rrbracket.\ v\ v_s \in \mathcal{T}\llbracket U\rrbracket\}\\
&\mathcal{V}\llbracket(\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi]\rrbracket && \doteq \{v \mid \overline{\forall c_i \in \mathcal{V}\llbracket B_i\rrbracket}.\ v(\overline{c_i}) \in \mathcal{T}\llbracket A\rrbracket\} \cap \mathcal{V}_*\llbracket(\overline{x_i : B_i}) \Rightarrow A[\Xi]\rrbracket
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{X}\llbracket\Omega\rrbracket && \doteq \{\Xi \mid \mathit{dom}(\Xi) = \mathit{dom}(\Omega)\}\\
&\mathcal{K}_\Omega\llbracket\uparrow x\rrbracket && \doteq \{d \mid \forall\omega,\omega' \in \mathcal{O}\llbracket\Omega\rrbracket.(\ \omega(x) \le \omega'(x) \wedge \forall y \in \mathit{dom}(\Omega) - \{x\}.\ \omega(y) = \omega'(y)\ )\\
& && \qquad \implies \|\omega\|d \le \|\omega'\|d\}\\
&\mathcal{K}_\Omega\llbracket\sim x\rrbracket && \doteq \{d \mid \forall\omega,\omega' \in \mathcal{O}\llbracket\Omega\rrbracket.(\forall y \in \mathit{dom}(\Omega) - \{x\}.\ \omega(y) = \omega'(y)) \implies \|\omega\|d = \|\omega'\|d\}\\
&\mathcal{B}_\Omega\llbracket B\rrbracket && \doteq \{d \mid \forall\omega \in \mathcal{O}\llbracket\Omega\rrbracket.\ \|\omega\|d \in \mathcal{V}\llbracket B\rrbracket\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{V}_\Omega\llbracket B\rrbracket && \doteq \mathcal{V}\llbracket B\rrbracket\\
&\mathcal{V}_\Omega\llbracket B[\Xi]\rrbracket && \doteq \{d \mid d \in \mathcal{B}_\Omega\llbracket B\rrbracket \wedge d \in \textstyle\bigcap_{z\in \mathit{dom}(\Xi)} \mathcal{K}_\Omega\llbracket\Xi(z)\ z\rrbracket\}\\
&\mathcal{V}_\Omega\llbracket S \xrightarrow{\Omega} U\rrbracket && \doteq \{v \mid \forall v_s \in \mathcal{V}_\Omega\llbracket S\rrbracket.\ v\ v_s \in \mathcal{T}_\Omega\llbracket U\rrbracket\}\\
&\mathcal{V}_\Omega\llbracket(\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi]\rrbracket && \doteq \{v \mid \overline{\forall \Xi_i \in \mathcal{X}\llbracket\Omega\rrbracket}.\ \overline{\forall d_i \in \mathcal{V}_\Omega\llbracket B_i[\Xi_i]\rrbracket}.\\
& && \qquad v(\overline{d_i}) \in \mathcal{T}_\Omega\llbracket A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi(x_i)))\ z}^{z\in \mathit{dom}(\Omega)}]\rrbracket\}\\
&\mathcal{V}_*\llbracket T\rrbracket && \doteq \textstyle\bigcap \mathcal{V}_\Omega\llbracket T\rrbracket \quad \text{where } \Omega \text{ ranges over all ambient environments}
\end{aligned}
$$

Fig. 15: Selected logical relations

relations are fairly straightforward, several aspects of the lifted logical relations deserve notice. First, a set $\mathcal{K}_\Omega\llbracket q\ x\rrbracket$ is used to define the semantics of a single qualifier map entry $q\ x$. It identifies a subset of lifted base values (i.e. constants and ambient maps) by using ambient substitution. The set $\mathcal{B}_\Omega\llbracket B\rrbracket$ describes all lifted base values which project to constants in the set $\mathcal{V}\llbracket B\rrbracket$. Together, these families of sets can be used to provide an interpretation $\mathcal{V}_\Omega\llbracket B[\Xi]\rrbracket$ of the qualified base type $B[\Xi]$: a lifted base value belongs to $\mathcal{V}_\Omega\llbracket B[\Xi]\rrbracket$ whenever it projects to an element of $\mathcal{V}\llbracket B\rrbracket$ and respects all qualifiers of Ξ. The type $S \xrightarrow{\Omega} U$ is interpreted as the set of all lambda abstractions which Ω-normalize to an value in $\mathcal{V}_\Omega\llbracket U\rrbracket$ when applied to a value in $\mathcal{V}_\Omega\llbracket S\rrbracket$. $\mathcal{V}_\Omega\llbracket(\overline{x_i : B_i}) \Rightarrow A[\Xi]\rrbracket$ describes those values which, when applied to lifted base values $\overline{d_i}$ respectively from the sets $\overline{\mathcal{B}_\Omega\llbracket B_i\rrbracket}$, have compositional behavior consistent with the qualifier map Ξ.

Recall from our discussion of *Figure 8* that within the body of a well-typed sfun, our type system allows occurrences of variables from both the terminal type environment Γ and the lifted type environment Φ. In the body of an sfun with ambient environment Ω, an occurrence of a variable $x : T \in \Gamma$ is substituted with an value in $\mathcal{V}[\![T]\!]$, but the variable occurs in a context where a value of type $\mathcal{V}_\Omega[\![T]\!]$ is expected. We therefore need the following result to allow substitution through sfun abstractions.

Lemma 1. *For all ambient environments Ω and types T with $\vdash T$, we have $\mathcal{V}[\![T]\!] \subseteq \mathcal{V}_\Omega[\![T]\!]$.*

5.9 Subtyping

Terminal subtyping, defined in *Figure 16*, is a binary relation on terminally well-formed types. The derivability of a terminal subtyping judgment $S <: U$ implies the set containment $\mathcal{V}[\![S]\!] \subseteq \mathcal{V}[\![U]\!]$. Likewise, *lifted subtyping* is defined in *Figure 17*. The derivability of a lifted subtyping judgment $\Omega \vdash S <: U$ implies the set containment $\mathcal{V}_\Omega[\![S]\!] \subseteq \mathcal{V}_\Omega[\![U]\!]$.

$$\textsc{Sub-Base}\ \frac{}{B <: B} \qquad \textsc{Sub-Fun}\ \frac{U_1 <: S_1 \qquad S_2 <: U_2}{S_1 \to S_2 <: U_1 \to U_2} \qquad \textsc{Sub-SFun}\ \frac{\overline{x_i : B_i} \vdash A[\Xi_1] <: A[\Xi_2]}{(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi_1] <: (\overline{x_i : B_i}) \Rightarrow A[\Xi_2]}$$

Fig. 16: Terminal subtyping

Sub-SFun is interesting because its premise is a lifted subtyping judgment rather than a terminal one. Intuitively, since we can substitute an sfun abstraction into any other sfun abstraction, we want a subtyping derivation for $(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi_1] <: (\overline{x_i : B_i}) \Rightarrow A[\Xi_2]$ that is independent of whichever ambient environment these types arise in. By taking an internal perspective, from within the ambient environment $\overline{x_i : B_i}$ delineated by a value of type $(\overline{x_i : B_i}) \Rightarrow A[\Xi]$, we maintain independence from the ambient environment on the outside of that value. This is not an issue for values of type $S \to T$, since $\mathcal{V}[\![S \to T]\!]$ is defined in terms of the terminal reduction relation, a subset of the Ω-reduction relation for any ambient environment Ω.

$$\textsc{LSub-Terminal}\ \frac{S <: U}{\Omega \vdash S <: U} \qquad \textsc{LSub-Base-TL}\ \frac{\overline{\sim \ \leq q_x}}{\Omega \vdash B <: B[\overline{q_x\ x}^{x \in dom(\Omega)}]} \qquad \textsc{LSub-Base-LL}\ \frac{\overline{p_x \leq q_x}}{\Omega \vdash B[\overline{p_x\ x}^{x \in dom(\Omega)}] <: B[\overline{q_x\ x}]}$$

$$\textsc{LSub-Fun-LL}\ \frac{\Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2}{\Omega \vdash S_1 \overset{\Omega}{\to} S_2 <: U_1 \overset{\Omega}{\to} U_2} \qquad \textsc{LSub-Fun-TL}\ \frac{\Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2 \qquad \vdash S_1 \to S_2}{\Omega \vdash S_1 \to S_2 <: U_1 \overset{\Omega}{\to} U_2}$$

Fig. 17: Lifted Subtyping

Some of the lifted subtyping rules utilize the partial order on qualifiers of *Figure 5*, written $p \leq q$. Their usage is justified by the following lemma.

Lemma 2 (Soundness of Qualifiers). *Suppose p and q are qualifiers with $p \leq q$. Then for all ambient environments Ω and $x \in dom(\Omega)$, we have $\mathcal{K}_\Omega[\![p\ x]\!] \subseteq \mathcal{K}_\Omega[\![q\ x]\!]$.*

LSub-Base-LL states that qualified base types are ordered componentwise by their qualifiers. The soundness of this rule is due to the monotonicity of set intersection: for all $x \in dom(\Omega)$ we have $\mathcal{K}_\Omega[\![p_x\ x]\!] \subseteq \mathcal{K}_\Omega[\![q_x\ x]\!]$, and so

$$(\mathcal{B}_\Omega[\![B]\!] \cap \bigcap_{x \in dom(\Omega)} \mathcal{K}_\Omega[\![p_x\ x]\!]) \subseteq (\mathcal{B}_\Omega[\![B]\!] \cap \bigcap_{x \in dom(\Omega)} \mathcal{K}_\Omega[\![q_x\ x]\!])$$

The definition of $\mathcal{V}_\Omega[\![B[\overline{\sim\ x}^{x \in dom(\Omega)}]]\!]$ implies that $\mathcal{V}_\Omega[\![B]\!] \subseteq \mathcal{V}_\Omega[\![B[\overline{\sim\ x}^{x \in dom(\Omega)}]]\!]$; this, combined with transitivity of set inclusion justifies LSub-Base-TL.

LSub-Fun-LL is a standard function subtyping rule, with a contravariant premise for the domain type and a covariant premise for the codomain type. Because of the LRed-Term rule in *Figure 12*, we know that for all Ω, if $t \Downarrow v$ then $\Omega \vdash t \Downarrow v$; this implies that for all types T with $\vdash T$, $\mathcal{T}[\![T]\!] \subseteq \mathcal{T}_\Omega[\![T]\!]$. Hence, for all S with $\vdash S$ we have $\mathcal{V}[\![S \to T]\!] \subseteq \mathcal{V}_\Omega[\![S \xrightarrow{\Omega} T]\!]$. Combining this observation with transitivity of set inclusion justifies LSub-Fun-TL.

5.10 Typing

Recall that our terminal type system is an ordinary type system with judgments of the form $\Gamma \vdash t : T$. It statically characterizes terms which are not enclosed in an any sfun abstraction. Our lifted type system has judgments of the form $\Gamma; \Omega; \Phi \vdash t : T$. It statically characterizes terms which are enclosed in an sfun abstraction. The type environment Γ contains all bindings formed outside of the sfun abstraction, which must be terminally well-formed. Ω containins the enclosing sfun's formal arguments, which must have base types. Φ is a lifted type environment containing bindings declared within the sfun abstraction, which must be Ω-well-formed.

For $\omega \in \Omega$, let $\|\omega\|\Phi$ denote the result of performing the ambient substitution $\|\omega\|$ on every Ω-well-formed type bound in Φ. By *Theorem 2*, $\|\omega\|\Phi$ is a terminally well-formed type environment. Writing the concatenation of $\|\omega\|\Phi$ onto Γ as $\Gamma, \|\omega\|\Phi$, we have the following theorem:

Theorem 3 (Projection of Well-Typed Terms). *If $\Gamma; \Omega; \Phi \vdash t : T$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$, we have $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|T$.*

The fundamental theorems for terminal typing and lifted typing serve as soundness criteria for these systems.

Theorem 4 (Fundamental Theorem for Terminal Typing). *If $\Gamma \vdash t : T$ then for all $\gamma \in \mathcal{G}[\![\Gamma]\!]$, $\gamma t \in \mathcal{T}[\![T]\!]$.*

Theorem 5 (Fundamental Theorem for Lifted Typing). *If $\Gamma; \Omega; \Phi \vdash t : T$ then for all $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\phi \in \mathcal{G}_\Omega[\![\Phi]\!]$ we have $\gamma\phi t \in \mathcal{T}[\![T]\!]$.*

There is a single novel terminal typing rule, for sfun abstractions:

T-SFun
$$\frac{\Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[=\ x_i]} \vdash t : A[\Xi]}{\Gamma \vdash (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) : (\overline{x_i : B_i}) \Rightarrow A[\Xi]}$$

Intuitively, this rule is justified by our discussion below *Theorem 1*: because a well-typed sfun's body normalizes to an ambient map that is extensionally equivalent to the sfun itself, the qualifier map in the premise accurately describes the sfun being type-checked.

It's worthwhile to compare the terminal and lifted abstraction typing rules, and consider their relation to LT-Terminal, a rule which implies that the terminally well-typed terms are a subset of the Ω-well-typed terms for each Ω:

$$\textsc{LT-Terminal}\ \frac{\Gamma \vdash t : T}{\Gamma; \Omega; \Phi \vdash t : T} \qquad \textsc{T-Fun}\ \frac{\vdash S \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.t) : S \to T} \qquad \textsc{LT-LFun}\ \frac{\Omega \vdash S \qquad \Gamma; \Omega; \Phi, x : S \vdash t : T}{\Gamma; \Omega; \Phi \vdash (\lambda x : S.t) : S \xrightarrow{\Omega} T}$$

LT-Terminal can be justified by applying *Lemma 1* and noting that Ω-reduction subsumes terminal reduction. Our system is non-deterministic: because the lambda abstraction is an introduction form for both terminal and lifted function types, a terminally well-typed lambda abstraction occuring inside an sfun could be typed as either. However, given that terminal function types are more precise, they could be given precedence in an algorithmic version of the system.

$$\textsc{LT-SfApp}\ \frac{\Gamma; \Omega; \Phi \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B_i[\Xi_i]}}{\Gamma; \Omega; \Phi \vdash t(\overline{s_i}^{i \in 1..n}) : A[\overline{(\Sigma_{i=1}^{n}(\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}]}$$

In LT-SfApp, note that because $(\overline{x_i : B_i}) \Rightarrow A[\Xi]$ is well-formed, $dom(\Xi) = \{x_i \mid i \in 1..n\}$, and because for $i \in 1..n$, $B_i[\Xi_i]$ is Ω-well-formed, we have $dom(\Xi_i) = dom(\Omega)$. The summation $\Sigma_{i=1}^{n}$ is a chain of applications of the qualifier contraction operation $+$ defined in *Figure 7*. $\circ$ is the qualifier composition operator defined in *Figure 6*.

Suppose we're dealing with an application of this rule with $\Omega = z : B$, $n = 2$, $x = x_1$, and $y = x_2$. Then for valuations $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\phi \in \mathcal{G}_\Omega[\![\Phi]\!]$, $\gamma\phi t$ Ω-normalizes to an sfun v of type $(x : B_1, y : B_2) \Rightarrow A[q_x\ x, q_y\ y]$, and for $i \in 1, 2$, $\gamma\phi s_i$ Ω-normalizes to an ambient map a_i of type $B_i[q_i\ z]$. Then a_1 and a_2 are both functions of Ω, and v is a function of its formal arguments x and y. An Ω-reduction step due to LRed-SApp reduces the term Ω-closed $v(a_1, a_2)$ to a composite function w, which, written using the "function notation" of elementary mathematics, looks as follows:

$$w(z) = v(a_1(z), a_2(z))$$

We would like to compute w's qualifier for z given the qualifier information we have for v, a_1, and a_2. To this end, we first compute the qualifiers of a simpler function, in which a_1 and a_2 are applied to different variables z_1 and z_2.

$$\hat{w}(z_1, z_2) = v(a_1(z_1), a_2(z_2))$$

Holding z_2 fixed we get a single-argument function which is the composition of $v' \circ a_1$ where $v_1(x) = v(x, a_2(z_2))$. We know that the qualifier of a_1 is q_1 and the qualifier of v' is q_x, and so the qualifier of the composite $v' \circ a_1$, which we'll call p_1 is $q_x \circ q_1$. All of our qualifiers denote single-argument properties: they are concerned with properties involving variance in at most one argument while all other arguments are held fixed. p_1 is therefore $\hat{w}$'s qualifier for z_1. Likewise, $\hat{w}$'s qualifier for z_2, which we'll call p_2, is $q_y \circ q_2$.

We've computed $\hat{w}$'s qualifiers p_1 and p_2 for its arguments z_1 and z_2, but our goal is to find w's qualifier for z. Fortunately, since w is the result of contracting $\hat{w}$'s two arguments z_1 and z_2 into a single argument z, the qualifier contraction operator is just what we need: w's qualifier for z is $p_1 + p_2 = (q_x \circ q_1) + (q_y \circ q_2)$. Extending this reasoning to n-ary sfun applications using induction gives us the conclusion of LT-SfApp.

6 Related Work

6.1 Type Systems

Monotonicity is a property relating multiple calls of a function. Such properties, called *relational properties*, are a blind spot of standard type systems. Nonetheless, potential applications abound, and researchers are making progress toward type systems capable of proving these properties. *Putting Differential Privacy to Work* [11,18] developed a type system for proving functions Lipschitz continuous, using a substructural type system that treats perturbation to a function's output value as a limited resource. This project achieves verifiable differential privacy for database access, and implements several realistic examples.

Barthe et. al. developed *Relational Refinement Types* [8] for the application domain of mechanisim design. Each relational refinement binds a pair of variables denoting corresponding values in separate computations. This technique is general enough to subsume the previously mentioned work on differential privacy.

Asada et al. [6] designed a system for verifying refinements containing calls to first-order functions. These refinement types, which can express monotonicity and other relational properties, are checked via translation to first-order refinements through a tupling transformation.

Datafun [5] is a typed calculus which captures the essence of Datalog, and extends it with functional programming constructs. Central to Datalog is the ability to calculate fixpoints of monotone functions, motivating the ability of Datafun's type system to prove functions monotone. The Datafun type environment is split into discrete and monotone parts, for variables bound by discrete abstractions and monotone abstractions. Monotone variables are restricted to positions which affect their contexts monotonically. A built-in type constructor for finite sets enables the expression of an iterative monotone computation as a join over a finitely indexed subset of some semilattice type. In contrast to our system, Datafun has better support for higher-order functions, but the top-down reasoning needed to program with monotone context may be less intuitive than our system, which computes monotonicity in a bottom-up manner.

6.2 Concurrent and Distributed Data Structures

LVars [13,14] are lattice-based variables designed to ensure that shared-memory parallel computations are deterministic. This is achieved by only allowing least-upper-bound writes and threshold reads of shared memory locations. LVars share many similarities with CRDTs [20,19], focused on the problem of deterministic parallel programming rather than on guaranteeing eventual consistency. A store with least-upper-bound writes increases monotonically, thus mapping a monotone function over the current state of such a store can be done in parallel. Monotonicity types could potentially assist in verifying that such functions are indeed monotone. Going one step further; if LVars were to be eventually extended to support compositions of LVar-based computations using higher-order functions, the techniques in this paper could potentially be used to prove monotonicity across composition.

FlowPools [17] are a lock-free deterministic concurrent abstraction for dataflow programming implemented in Scala. A main feature of FlowPools their support for programming higher-order functions for composing computations on FlowPools. Calls to higher-order functions like `foreach` or `fold` run concurrently across elements, and are guaranteed to be eventually applied to every element. An issue with this, however, is the fact that a user must take care to ensure that the functions passed to and applied to each element via combinators like `foreach` satisfy one of several properties, such as commutativity, idempotency, and monotonicity, to ensure that the resulting FlowPools are deterministic. Monotonicity types applied to FlowPools could guard against user errors caused by non-monotonic functions passed to FlowPool combinators.

6.3 Distributed Programming Models

Bloom [3] is a Datalog-based language for distributed programming designed to guarantee eventual consistency without coordination. This is made possible due to a focus on providing users with only monotone functions and sets, enabling all Bloom programs be order-insensitive. Bloom provides a monotonicity analysis procedure which is designed to identify *points of order*, or, program locations where the output of an asynchronously derived value is consumed by a non-monotonic operator. In this way, Bloom can identify and inform the programmer where synchronization points in a program are required. Bloom was later extended to Bloom^L [9], a lattice-based variant of Bloom able to perform Bloom's monotonicity analysis on arbitrary join-semilattices (rather than on sets only). The extension enabled Bloom^L to support more interesting (monotonic) data types such as maps, or otherwise programs that "grow" according to a partial order other than set containment. However, Bloom^L's monotonicity analysis simply checks that user-defined morphisms and monotone functions preserve monotonicity in their arguments by merely being tagged by the user as monotonic. That is, it assumes that the monotonicity annotations on functions are correct as it cannot analyze these functions to determine whether or not they are indeed monotonic. Monotonicity types could be of help here by proving functions tagged as monotone are indeed monotone.

Lasp [15] provides a functional programming model over CRDTs which focuses on providing higher-order functions (or combinators) such as `map` and `fold` to compose computations on CRDTs. Lasp's philosophy is to enable users to build up larger composite computations through functional composition which retain the properties of CRDTs over composition. However, while providing a higher-order programming facility and the ability to define new combinators, Lasp provides no mechanism to ensure that new user-defined combinators will exhibit the required properties of monotonicity, risking the correctness of Lasp programs. Monotonicity types are perfectly suited to guard against such errors by statically rejecting non-monotonic combinators.

7 Future Work

Recall the aspirational example presented in *Figure 8* illustrating the implementation of a CRDT combinator intended to take two G-Counters as input and to return the sum of the two input counters as output. This example is aspirational for a few reasons. First, we don't yet support recursion. Our model assumes that every well-typed term normalizes, so adding a standard fixpoint combinator would be problematic. Recursion in this example is particularly tricky, since the number of iterations performed is dependent upon the values of the sfun arguments; as the values of x and y get arbitrarily high, so do the number of iterations. Finally, this example involves an if expression which must be proven monotone. When the if condition switches from true to false, we need to prove that the expression in the then branch is less than or equal to the expression in the else branch.

In future work, we intend to resolve these issues with the addition of dependent refinement types to our language. In doing so, we could add a terminating fixed point combinator as described by Vazou et al. [21], which performs recursion under a well-founded, decreasing termination metric. Dependent refinement types would also allow us to type if expressions, since we could include ordering constraints in our typing rules.

8 Conclusion

In this paper, we presented a language and type system for proving functions monotone, utilizing a novel approach for propagating properties across function composition. Given that monotone functions are the fundamental primitives of composition in emerging coordination-free distributed programming models, and the relative difficulty in manually ensuring that application code satisfies these models' monotonicity constraints, Monotonicity Types could be the missing piece which enable customization and extension of such systems by non-experts. Going further, Monotonicity Types could provide a foundation for future work on extensions to practical languages, with the eventual goal of enabling safer and more flexible abstractions for these emerging distributed programming models.

References

1. Calm: consistency as logical monotonicity. `http://bloom-lang.net/calm/`. Accessed: 2017-10-01.

2. P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111(Supplement C):162–173, Jan. 2018.
3. P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR*, pages 249–260, 2011.
4. T. J. Ameloot, F. Neven, and J. Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, May 2013.
5. M. Arntzenius and N. R. Krishnaswami. Datafun: A functional datalog. *SIGPLAN Not.*, 51(9):214–227, Sept. 2016.
6. K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. *Science of Computer Programming*, 137(Supplement C):2–62, Apr. 2017.
7. P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.
8. G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-Order approximate relational refinement types for mechanism design and differential privacy. *SIGPLAN Not.*, 50(1):55–68, Jan. 2015.
9. N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM.
10. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Apr. 2002.
11. M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. *SIGPLAN Not.*, 48(1):357–370, Jan. 2013.
12. A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *SIGPLAN Not.*, 51(1):371–384, Jan. 2016.
13. L. Kuper and R. R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM.
14. L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM.
15. C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195, New York, NY, USA, 2015. ACM.
16. B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.
17. A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A Lock-Free deterministic concurrent dataflow abstraction. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 158–173. Springer, Berlin, Heidelberg, Sept. 2012.
18. J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 157–168, New York, NY, USA, 2010. ACM.
19. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

20. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, Lecture Notes in Computer Science, pages 386–400. Springer, Berlin, Heidelberg, Oct. 2011.
21. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, Aug. 2014.

Appendix A Language

We explain that the metavariable a is used to denote sets of pairs which satisfy the function property: $(x, c) \in a \wedge (x, d) \in a \Rightarrow c = d$.

$$
\begin{array}{rcll}
x & ::= & \mathit{variables} & \\
p, q & ::= & \uparrow \;|\; \downarrow \;|\; \sim \;|\; ? \;|\; = & \mathit{Qualifiers} \\
\Xi & ::= & \cdot \mid \Xi,\ q\ x & \mathit{Qualifier\ maps} \\
c & ::= & \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & \mathit{Terminal\ base\text{-}level\ constants} \\
d & ::= & c \mid a & \mathit{Base\text{-}level\ constants} \\
k & ::= & + \mid - \mid \; < \; \mid \; \leq \; \mid \; = \; \mid \wedge \mid \vee & \mathit{Sfun\ constants} \\
A, B & ::= & \mathit{Bool} \mid \mathit{Int} & \mathit{Base\text{-}level\ types} \\
S, T, U & ::= & B \mid B[\Xi] \mid (\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi] \mid & \mathit{Types} \\
 & & S \rightarrow T \mid S \xrightarrow{\Omega} T & \\
v & ::= & k \mid d \mid a \mid (\lambda x : T.t) \mid (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) & \mathit{Values} \\
s, t, u & ::= & v \mid x \mid t\ t \mid t\ [\ \overline{t_i}^{i \in 1..n}\ ] \mid & \mathit{Terms} \\
 & & \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t \mid \mathbf{let}\ x = t\ \mathbf{in}\ t & \\
E & ::= & E\ t \mid v\ E \mid E\ [\overline{t}] \mid v\ [\overline{v}\ E\ \overline{t}] \mid & \mathit{Evaluation\ contexts} \\
 & & \mathbf{if}\ E\ \mathbf{then}\ t\ \mathbf{else}\ t \mid \mathbf{let}\ x = E\ \mathbf{in}\ t \mid [] &
\end{array}
$$

Fig. 18: Syntax

$$kty : k \rightarrow T$$

$$
\begin{array}{ll}
ty(+) & = (x : Int, y : Int) \Rightarrow Int[\uparrow\ x,\ \uparrow\ y] \\
ty(-) & = (x : Int, y : Int) \Rightarrow Int[\uparrow\ x,\ \downarrow\ y] \\
ty(<) & = (x : Int, y : Int) \Rightarrow Int[\downarrow\ x,\ \uparrow\ y] \\
ty(\leq) & = (x : Int, y : Int) \Rightarrow Int[\downarrow\ x,\ \uparrow\ y] \\
ty(=) & = (x : Int, y : Int) \Rightarrow Int[?\ x, ?\ y] \\
ty(\wedge) & = (x : Int, y : Int) \Rightarrow Int[\uparrow\ x,\ \uparrow\ y] \\
ty(\vee) & = (x : Int, y : Int) \Rightarrow Int[\uparrow\ x,\ \uparrow\ y]
\end{array}
$$

Fig. 19: Sfun constant types

$$
\begin{array}{ll}
cty : c \to B & \\
& \\
ty(true) & = Bool \\
ty(false) & = Bool \\
ty(0) & = Int \\
ty(1) & = Int \\
& \vdots
\end{array}
$$

Fig. 20: Base constant types

$\circ$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
?	?	?	?	?	$\sim$
$\uparrow$	?	$\uparrow$	$\downarrow$	$\uparrow$	$\sim$
$\downarrow$	?	$\downarrow$	$\uparrow$	$\downarrow$	$\sim$
$=$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$

Fig. 21: Qualifier composition $\circ$

$+$	?	$\uparrow$	$\downarrow$	$=$	$\sim$
?	?	?	?	$=$	?
$\uparrow$	?	$\uparrow$	?	$=$	$\uparrow$
$\downarrow$	?	?	$\downarrow$	$=$	$\downarrow$
$=$	$=$	$=$	$=$	$=$	$=$
$\sim$	?	$\uparrow$	$\downarrow$	$=$	$\sim$

Fig. 22: Qualifier contraction $+$

$$
\textsc{Wf-Base}\quad \frac{}{\vdash B}
\qquad
\textsc{Wf-Fun}\quad \frac{\vdash S \qquad \vdash T}{\vdash S \to T}
\qquad
\textsc{Wf-SFun}\quad \frac{dom(\Xi) = \{x_i \mid i \in 1..n\}}{\vdash (\overline{x_i : B_i}^{\,i \in 1..n}) \Rightarrow A[\Xi]}
$$

Fig. 23: Terminal type well-formedness

$$
\Omega\textsc{Wf-Terminal}\quad \frac{\vdash T}{\Omega \vdash T}
\qquad
\Omega\textsc{Wf-QualBase}\quad \frac{dom(\Omega) = dom(\Xi)}{\Omega \vdash B[\Xi]}
\qquad
\Omega\textsc{Wf-LFun}\quad \frac{\Omega \vdash S \qquad \Omega \vdash T}{\Omega \vdash S \xrightarrow{\Omega} T}
$$

Fig. 24: Lifted type well-formedness

$$
\textsc{Sub-Base}\quad \frac{}{B <: B}
\qquad
\textsc{Sub-Fun}\quad \frac{U_1 <: S_1 \qquad S_2 <: U_2}{S_1 \to S_2 <: U_1 \to U_2}
\qquad
\textsc{Sub-SFun}\quad \frac{\overline{x_i : B_i} \vdash A[\Xi_1] <: A[\Xi_2]}{(\overline{x_i : B_i}^{\,i \in 1..n}) \Rightarrow A[\Xi_1] <: (\overline{x_i : B_i}) \Rightarrow A[\Xi_2]}
$$

Fig. 25: Terminal subtyping

$$\textsc{LSub-Terminal}\ \frac{S <: U}{\Omega \vdash S <: U} \qquad \textsc{LSub-Base-TL}\ \frac{\overline{\sim \quad \leq q_x}}{\Omega \vdash B <: B[\overline{q_x\ x}^{x \in dom(\Omega)}]} \qquad \textsc{LSub-Base-LL}\ \frac{\overline{p_x \leq q_x}}{\Omega \vdash B[\overline{p_x\ x}^{x \in dom(\Omega)}] <: B[\overline{q_x\ x}]}$$

$$\textsc{LSub-Fun-LL}\ \frac{\Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2}{\Omega \vdash S_1 \xrightarrow{\Omega} S_2 <: U_1 \xrightarrow{\Omega} U_2} \qquad \textsc{LSub-Fun-TL}\ \frac{\Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2 \qquad \vdash S_1 \to S_2}{\Omega \vdash S_1 \to S_2 <: U_1 \xrightarrow{\Omega} U_2}$$

Fig. 26: Lifted Subtyping

$$\textsc{T-Constant}\ \frac{ty(c) = B}{\Gamma \vdash c : B} \qquad \textsc{T-SfConstant}\ \frac{ty(k) = T}{\Gamma \vdash k : T} \qquad \textsc{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{T-Fun}\ \frac{\vdash S \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.t) : S \to T}$$

$$\textsc{T-SFun}\ \frac{\Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[= \ \ x_i]} \vdash t : A[\Xi]}{\Gamma \vdash (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) : (\overline{x_i : B_i}) \Rightarrow A[\Xi]} \qquad \textsc{T-App}\ \frac{\Gamma \vdash t : S \to U \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : U}$$

$$\textsc{T-SfApp}\ \frac{\Gamma \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma \vdash s_i : B_i}}{\Gamma \vdash t\ [\overline{s_i}^{i \in 1..n}] : A} \qquad \textsc{T-IfThenElse}\ \frac{\Gamma \vdash s_1 : Bool \qquad \Gamma \vdash s_2 : S \qquad \Gamma \vdash s_3 : S}{\Gamma \vdash \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 : S}$$

$$\textsc{T-Sub}\ \frac{\Gamma \vdash t : S \qquad \vdash U \qquad S <: U}{\Gamma \vdash t : U}$$

Fig. 27: Terminal typing

$$\frac{\Gamma \vdash t : T}{\Gamma; \Omega; \Phi \vdash t : T}\ \textsc{LT-Terminal} \qquad \frac{x : T \in \Phi}{\Gamma; \Omega; \Phi \vdash x : T}\ \textsc{LT-LVar} \qquad \frac{\Omega \vdash S \qquad \Gamma; \Omega; \Phi, x : S \vdash t : T}{\Gamma; \Omega; \Phi \vdash (\lambda x : S.t) : S \xrightarrow{\Omega} T}\ \textsc{LT-LFun}$$

$$\frac{\Gamma; \Omega; \Phi \vdash t : S \xrightarrow{\Omega} U \qquad \Gamma; \Omega; \Phi \vdash s : S}{\Gamma; \Omega; \Phi \vdash t\ s : U}\ \textsc{LT-LApp}$$

$$\frac{\Gamma; \Omega; \Phi \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B_i[\Xi_i]}}{\Gamma; \Omega; \Phi \vdash t\ [\overline{s_i}^{i \in 1..n}] : A[\overline{(+_{i=1}^{n}(\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}]}\ \textsc{LT-SfApp}$$

$$\frac{\Gamma; \Omega; \Phi \vdash s_1 : \mathit{Bool} \qquad \Gamma; \Omega; \Phi \vdash s_2 : S \qquad \Gamma; \Omega; \Phi \vdash s_3 : S}{\Gamma; \Omega; \Phi \vdash \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 : S}\ \textsc{LT-IfThenElse}$$

$$\frac{\Gamma; \Omega; \Phi \vdash t : S \qquad \Omega \vdash U \qquad \Omega \vdash S <: U}{\Gamma; \Omega; \Phi \vdash t : U}\ \textsc{LT-Sub}$$

Fig. 28: Lifted typing

$$\begin{array}{lll}
\|\omega\|a & = a(\omega) & \text{when } \omega \in \mathit{dom}(a) \\
\|\omega\|a & = a & \text{when } \omega \notin \mathit{dom}(a) \\
\|\omega\|t\ t & = \|\omega\|t\ \|\omega\|t & \\
\|\omega\|t\ [\overline{t}] & = \|\omega\|t\ [\overline{\|\omega\|t_i}] & \\
\|\omega\|(\lambda x : S.t) & = (\lambda x : \|\omega\|S.\|\omega\|t) & \\
\|\omega\|\textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t & = \textbf{if}\ \|\omega\|t\ \textbf{then}\ \|\omega\|t\ \textbf{else}\ \|\omega\|t & \\
\|\omega\|(\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t) & = (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ \|\omega\|t) & \\
\|\omega\|t & = t \quad \mathit{otherwise} & \\
 & & \\
\|\omega\|T \xrightarrow{\Omega} T & = \|\omega\|T \to \|\omega\|T & \text{when } \omega \in \mathcal{O}[\![\Omega]\!] \\
\|\omega\|B[\Xi] & = B & \text{when } \mathit{dom}(\omega) = \mathit{dom}(\Xi) \\
\|\omega\|T & = T & \text{otherwise}
\end{array}$$

Fig. 29: Ambient substitution

$$\textsc{Red-Context}\quad \frac{t \to t'}{E[t] \to E[t']} \qquad \textsc{Red-Let}\quad \frac{}{\textbf{let } x = v \textbf{ in } t \to [v/x]t} \qquad \textsc{Red-App}\quad \frac{}{(\lambda x : T.t)\ v \to [v/x]t}$$

$$\textsc{Red-SApp}\quad \frac{\phi \in \mathcal{G}_{\overline{x_i:B_i}}[\![\overline{x_i : B_i[=\ x_i]}]\!] \qquad \omega = \cdot\overline{[x_i \mapsto c_i]}}{(\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ t)\ [\overline{c_i}] \to \|\omega\|\phi t} \qquad \textsc{Red-SApp-Const}\quad \frac{\delta_n(k) \text{ is defined}}{k\ [\overline{c_i}^{i\in 1..n}] \to \delta_n(k)(\overline{c_i})}$$

$$\textsc{Red-IfTrue}\quad \frac{}{\textbf{if } \mathit{true} \textbf{ then } t_1 \textbf{ else } t_2 \to t_1} \qquad \textsc{Red-IfFalse}\quad \frac{}{\textbf{if } \mathit{false} \textbf{ then } t_1 \textbf{ else } t_2 \to t_2}$$

Fig. 30: Terminal reduction

$$\textsc{LRed-Context}\quad \frac{\Omega \vdash t \to t'}{\Omega \vdash E[t] \to E[t']} \qquad \textsc{LRed-Let}\quad \frac{}{\Omega \vdash \textbf{let } x = v \textbf{ in } t \to [v/x]t} \qquad \textsc{LRed-App}\quad \frac{}{\Omega \vdash (\lambda x : T.t)\ v \to [v/x]t}$$

$$\textsc{LRed-SApp-Lift}\quad \frac{\overline{(\tilde{\lambda}(\overline{x_i : B_i}).\ t)\ [\overline{\|\omega\|d_i}] \Downarrow c_\omega}^{\omega\in\mathcal{O}[\![\Omega]\!]} \qquad a = \{\overline{(\omega, c_\omega)}\}}{\Omega \vdash (\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ t)\ [\overline{d_i}] \to a}$$

$$\textsc{LRed-SApp}\quad \frac{\phi \in \mathcal{G}_{\overline{x_i:B_i}}[\![\overline{x_i : B_i[=\ x_i]}]\!] \qquad \omega = \cdot\overline{[x_i \mapsto c_i]}}{\Omega \vdash (\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ t)\ [\overline{c_i}] \to \|\omega\|\phi t} \qquad \textsc{LRed-SApp-Const}\quad \frac{\delta_n(k) \text{ is defined}}{\Omega \vdash k\ [\overline{c_i}^{i\in 1..n}] \to \delta(k)(\overline{c_i})}$$

$$\textsc{LRed-IfTrue}\quad \frac{}{\Omega \vdash \textbf{if } \mathit{true} \textbf{ then } t_1 \textbf{ else } t_2 \to t_1} \qquad \textsc{LRed-IfFalse}\quad \frac{}{\Omega \vdash \textbf{if } \mathit{false} \textbf{ then } t_1 \textbf{ else } t_2 \to t_2}$$

Fig. 31: Lifted reduction

$$
\begin{aligned}
&\mathcal{C}[\![Int]\!] && \doteq \{0, 1, -1, 2, -2, \ldots\}\\
&\mathcal{C}[\![Bool]\!] && \doteq \{false, true\}\\
\\
&\mathcal{V}[\![B]\!] && \doteq \mathcal{C}[\![B]\!]\\
&\mathcal{V}[\![S \to U]\!] && \doteq \{v \mid \forall v_s \in \mathcal{V}[\![S]\!].\ v\ v_s \in \mathcal{T}[\![U]\!]\} \cap \mathcal{V}_*[\![S \to U]\!]\\
&\mathcal{V}[\![(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]]\!] && \doteq \{v \mid \overline{\forall c_i \in \mathcal{V}[\![B_i]\!]}.\ v\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]\} \cap \mathcal{V}_*[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]\\
\\
&\mathcal{T}[\![T]\!] && \doteq \{t \mid t \Downarrow v \wedge v \in \mathcal{V}[\![T]\!]\}\\
\\
&\mathcal{G}[\![\cdot]\!] && \doteq \{\emptyset\}\\
&\mathcal{G}[\![\Gamma, x : T]\!] && \doteq \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![T]\!]\}\\
\\
&\mathcal{O}[\![\cdot]\!] && \doteq \{\emptyset\}\\
&\mathcal{O}[\![\Omega, x : B]\!] && \doteq \{\omega \mid \omega[x \mapsto c] \wedge c \in \mathcal{C}[\![B]\!]\}
\end{aligned}
$$

Fig. 32: Terminal logical relations

$$
\begin{aligned}
&\mathcal{X}[\![\Omega]\!] &&\doteq \{\Xi \mid \mathit{dom}(\Xi) = \mathit{dom}(\Omega)\} \\[1em]
&\mathcal{K}_\Omega[\![\uparrow x]\!] &&\doteq \{d \mid \forall \omega, \omega' \in \mathcal{O}[\![\Omega]\!].(\ \omega(x) \leq \omega'(x) \wedge \forall y \in \mathit{dom}(\Omega) - \{x\}.\ \omega(y) = \omega'(y)\ ) \\
& && \qquad \implies \|\omega\|d \leq \|\omega'\|d\} \\
&\mathcal{K}_\Omega[\![\downarrow x]\!] &&\doteq \{d \mid \forall \omega, \omega' \in \mathcal{O}[\![\Omega]\!].(\ \omega(x) \leq \omega'(x) \wedge \forall y \in \mathit{dom}(\Omega) - \{x\}.\ \omega(y) = \omega'(y)\ ) \\
& && \qquad \implies \|\omega\|d \geq \|\omega'\|d\} \\
&\mathcal{K}_\Omega[\![\sim x]\!] &&\doteq \{d \mid \forall \omega, \omega' \in \mathcal{O}[\![\Omega]\!].(\forall y \in \mathit{dom}(\Omega) - \{x\}.\ \omega(y) = \omega'(y)) \implies \|\omega\|d = \|\omega'\|d\} \\
&\mathcal{K}_\Omega[\![= x]\!] &&\doteq \{d \mid \forall \omega \in \mathcal{O}[\![\Omega]\!].\ \|\omega\|d = \omega(x)\} \\
&\mathcal{K}_\Omega[\![?\ x]\!] &&\doteq \{d \mid \mathit{true}\} \\[1em]
&\mathcal{B}_\Omega[\![B]\!] &&\doteq \{d \mid \forall \omega \in \mathcal{O}[\![\Omega]\!].\ \|\omega\|d \in \mathcal{V}[\![B]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{V}_\Omega[\![B]\!] &&\doteq \mathcal{V}[\![B]\!] \\
&\mathcal{V}_\Omega[\![B[\Xi]]\!] &&\doteq \{d \mid d \in \mathcal{B}_\Omega[\![B]\!] \wedge d \in \textstyle\bigcap_{z \in \mathit{dom}(\Xi)} \mathcal{K}_\Omega[\![\Xi(z)\ z]\!]\} \\
&\mathcal{V}_\Omega[\![S \to U]\!] &&\doteq \mathcal{V}[\![S \to T]\!] \\
&\mathcal{V}_\Omega[\![S \xrightarrow{\Omega} U]\!] &&\doteq \{v \mid \forall v_s \in \mathcal{V}_\Omega[\![S]\!].\ v\ v_s \in \mathcal{T}_\Omega[\![U]\!]\} \\
&\mathcal{V}_\Omega[\![(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]]\!] &&\doteq \{v \mid \overline{\forall \Xi_i \in \mathcal{X}[\![\Omega]\!]}.\ \overline{\forall d_i \in \mathcal{V}_\Omega[\![B_i[\Xi_i]]\!]}. \\
& && \qquad v\ [\overline{d_i}] \in \mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi(x_i)))\ z}^{z \in \mathit{dom}(\Omega)}]]\!]\} \\[1em]
&\mathcal{T}_\Omega[\![T]\!] &&\doteq \{t \mid \Omega \vdash t \Downarrow v \wedge v \in \mathcal{V}_\Omega[\![T]\!]\} \\[1em]
&\mathcal{G}_\Omega[\![\cdot]\!] &&\doteq \{\emptyset\} \\
&\mathcal{G}_\Omega[\![\Phi, x : T]\!] &&\doteq \{\phi[x \mapsto v] \mid \phi \in \mathcal{G}_\Omega[\![\Phi]\!] \wedge v \in \mathcal{V}_\Omega[\![T]\!]\} \\[1em]
&\mathcal{G}_*[\![\Phi]\!] &&\doteq \textstyle\bigcap \mathcal{G}_\Omega[\![\Phi]\!] \quad \text{where } \Omega \text{ ranges over all ambient environments} \\
&\mathcal{V}_*[\![T]\!] &&\doteq \textstyle\bigcap \mathcal{V}_\Omega[\![T]\!] \quad \text{where } \Omega \text{ ranges over all ambient environments} \\
&\mathcal{T}_*[\![T]\!] &&\doteq \textstyle\bigcap \mathcal{T}_\Omega[\![T]\!] \quad \text{where } \Omega \text{ ranges over all ambient environments}
\end{aligned}
$$

Fig. 33: Lifted logical relations

Appendix B Theorems

Lemma 3. *For all values v and all ambient valuations ω, $\|\omega\|v$ is a value.*

Proof. Clear from a brief examination of *Figure 29*.

Theorem 6. *If $\Omega \vdash u \to u'$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$, $\|\omega\|u \to^* \|\omega\|u'$.*

Proof. By induction on the derivation of $\Omega \vdash u \to u'$.

LRed-Context: $u = E[t] \qquad u' = E[t'] \qquad \Omega \vdash t \to t'$

By the IH, we know that $\forall\omega \in \mathcal{O}[\![\Omega]\!].\ \|\omega\|t \to^* \|\omega\|t'$. Let $\omega \in \mathcal{O}[\![\Omega]\!]$. We need to show $\|\omega\|E[t] \to^* \|\omega\|E[t']$. The proof proceeds by induction on the structure of $E[t]$, applying *Lemma 3* implicitly in several places.

Case $E = \cdot$:

$\|\omega\|E[t] = \|\omega\|t \to^* \|\omega\|t' = \|\omega\|E[t']$ by the IH.

Case $E[t] = E'[t]\ s$:

$\|\omega\|E[t] = \|\omega\|E'[t]\ \|\omega\|s \to^* \|\omega\|E'[t']\ \|\omega\|s = \|\omega\|E[t']$

Case $E[t] = v\ E'[t]$:

$\|\omega\|E[t] = \|\omega\|v\ \|\omega\|E'[t] \to^* \|\omega\|v\ \|\omega\|E'[t'] = \|\omega\|E[t']$

Case $E[t] = v\ [\overline{v}\ E'[t]\ \overline{s}]$:

$\|\omega\|E[t] = \|\omega\|v\ [\overline{\|\omega\|v}\ \|\omega\|E'[t]\ \overline{\|\omega\|s}]$

$\to^* \|\omega\|v\ [\overline{\|\omega\|v}\ \|\omega\|E'[t']\ \overline{\|\omega\|s}] = \|\omega\|E[t']$

Case $E[t] = E'[t]\ [\ \overline{s}\ ]$:

$\|\omega\|E[t] = \|\omega\|E'[t]\ [\ \overline{\|\omega\|s}\ ] \to^* \|\omega\|E'[t']\ [\ \overline{\|\omega\|s}\ ]$

Case $E[t] = \textbf{if}\ E'[t]\ \textbf{then}\ s_1\ \textbf{else}\ s_2$:

$\|\omega\|E[t] = \textbf{if}\ \|\omega\|E'[t]\ \textbf{then}\ \|\omega\|s_1\ \textbf{else}\ \|\omega\|s_2 \to^*$
$\textbf{if}\ \|\omega\|E'[t']\ \textbf{then}\ \|\omega\|s_1\ \textbf{else}\ \|\omega\|s_2 = \|\omega\|E[t']$

Case $E[t] = \textbf{let}\ x = E'[t]\ \textbf{in}\ s$:

$\|\omega\|E[t] = \textbf{let}\ x = \|\omega\|E'[t]\ \textbf{in}\ \|\omega\|s \to^* \textbf{let}\ x = \|\omega\|E'[t']\ \textbf{in}\ \|\omega\|s$
$= \|\omega\|E[t']$

Case LRed-App: $u = (\lambda x : S.\ t)\ v \qquad u' = [v/x]t$

$\|\omega\|u = \|\omega\|(\lambda x : S.t)\ \|\omega\|v \to [\|\omega\|v/x]\|\omega\|t = \|\omega\|[v/x]t = \|\omega\|u'$.

Case LRed-SApp-Lift:

$u = (\tilde{\lambda}(\overline{x_i : B_i}).\ s)\ [\ \overline{d_i}\ ]$

$u' = \{\overline{\omega \mapsto c_\omega}^{\omega \in \mathcal{O}[\![\Omega]\!]}\}$

where $\forall\omega \in \mathcal{O}[\![\Omega]\!].\ (\tilde{\lambda}(\overline{x_i : B_i}).\ s)\ [\ \overline{\|\omega\|d_i}\ ] \Downarrow c_\omega$

$\|\omega\|u = (\tilde{\lambda}(\overline{x_i : B_i}).\ s)\ [\ \overline{\|\omega\|d_i}\ ] \to^* c_\omega$
$= \|\omega\|\{\overline{\omega \mapsto c_\omega}^{\omega \in \mathcal{O}[\![\Omega]\!]}\} = \|\omega\|u'$

Case LRed-SApp:

$u = (\tilde{\lambda}(\overline{x_i : B_i}).\ s)\ [\overline{c_i}]$

$\phi \in \mathcal{G}_{\overline{x_i : B_i}}[\![\overline{x_i : B_i[=\ x_i]}]\!]$

$\psi = \cdot\overline{[x_i \mapsto c_i]}$

$u' = \|\psi\|\phi s$

$\|\omega\|u = (\tilde{\lambda}(\overline{x_i : B_i}).\ \|\omega\|s)\ [\overline{\|\omega\|c_i}] = (\tilde{\lambda}(\overline{x_i : B_i}).\ \|\omega\|s)\ [\overline{c_i}] \to \|\psi\|\phi\|\omega\|s = \|\omega\|\|\psi\|\phi s = \|\omega\|u'$

Case LRed-SApp-Const:

$u = k\ [\overline{c_i}^{i \in 1..n}]$
$u' = \delta(k)(\overline{c_i})$

$\|\omega\|u = u \to \delta(k)(\overline{c_i}) = \|\omega\|\delta(k)(\overline{c_i}) = \|\omega\|u'$

Case LRed-If-True:

$u = \textbf{if}\ true\ \textbf{then}\ t_1\ \textbf{else}\ t_2$
$u' = t_1$

$\|\omega\|u = \textbf{if}\ true\ \textbf{then}\ \|\omega\|t_1\ \textbf{else}\ \|\omega\|t_2$
$\to \|\omega\|t_1 = \|\omega\|u'$

Case LRed-If-False:

Symmetric to LRed-If-True case.

Case LRed-Let:

$u = \textbf{let}\ x = v\ \textbf{in}\ s$
$u' = [v/x]s$

$\|\omega\|u = \textbf{let}\ x = \|\omega\|v\ \textbf{in}\ \|\omega\|s \to [\|\omega\|v/x]\|\omega\|s = \|\omega\|u'$

Corollary 1. *For all $\omega \in \mathcal{O}[\![\Omega]\!]$, $\Omega \vdash u \Downarrow v$ implies $\|\omega\|u \Downarrow \|\omega\|v$.*

Proof. Since $\Omega \vdash u \Downarrow v$, there is a sequence of reduction steps $\Omega \vdash u \to u_1,\ \Omega \vdash u_1 \to u_2,\ \ldots, \Omega \vdash u_i \to v$. By the preceding lemma $\|\omega\|u \to^* \|\omega\|u_1,\ \|\omega\|u_1 \to^* \|\omega\|u_2,\ \ldots$, and $\|\omega\|u_i \to^* \|\omega\|v$. Stitching these together gives $\|\omega\|u \Downarrow \|\omega\|v$.

Lemma 4 (Monotonicity of $\circ$ and $+$). *The qualifier operators $\circ$ and $+$ are both monotone in each of their arguments separately.*

Proof. Tedious but straightforward proof omitted.

Lemma 5 (Qualifier order soundness). *Let Ω be an ambient environment, and let p and q be qualifiers such that $p \leq q$. Then for all $z \in dom(\Omega)$, $\mathcal{K}_\Omega[\![p\ z]\!] \subseteq \mathcal{K}_\Omega[\![q\ z]\!]$*

Proof. $\subseteq$ is a transitive relation, so it will be sufficient to show that containment holds for each covering pair $p \prec q$ in the qualifier order. We say that p covers q, or $p \prec q$, whenever $p \leq q$ and for all r such that $p \leq r$ we have $q \leq r$. It can be seen from a cursory glance of *Figure 32* that $p \prec q$ implies $\mathcal{K}_\Omega[\![p\ z]\!] \subseteq \mathcal{K}_\Omega[\![q\ z]\!]$.

Lemma 6. *If $\Omega \vdash A[\Xi_1] <: A[\Xi_2]$ then $\mathcal{V}_\Omega[\![A[\Xi_1]]\!] \subseteq \mathcal{V}_\Omega[\![A[\Xi_2]]\!]$.*

Proof. For each $z \in dom(\Omega)$ we have $\mathcal{K}_\Omega[\![\Xi_1(z)\ z]\!] \subseteq \mathcal{K}_\Omega[\![\Xi_2(z)\ z]\!]$ by *Lemma 5*. Since the intersection operator is monotone with respect to each of its constituent sets, we have

$$\mathcal{B}_\Omega[\![A]\!] \cap \bigcap_{z \in dom(\Omega)} \mathcal{K}_\Omega[\![\Xi_1(z)\ z]\!] \subseteq \mathcal{B}_\Omega[\![A]\!] \cap \bigcap_{z \in dom(\Omega)} \mathcal{K}_\Omega[\![\Xi_2(z)\ z]\!]$$

i.e.

$$\mathcal{V}_\Omega[\![A[\Xi_1]]\!] \subseteq \mathcal{V}_\Omega[\![A[\Xi_2]]\!]$$

Assumption 1 (Types of Constants) *For all constants c we have $c \in \mathcal{V}[\![ty(c)]\!]$ and $\vdash ty(c)$. Also, for all sfun constants k we have $k \in \mathcal{V}[\![ty(k)]\!]$ and $\vdash ty(k)$.*

Lemma 7. *If $t \Downarrow v$ then for all ambient environments Ω we have $\Omega \vdash t \Downarrow v$.*

Proof. This is a simple consequence of the lifted reduction rules being a superset of the terminal reduction rules.

Lemma 8. *If $\vdash T$ then $\mathcal{V}[\![T]\!] \subseteq \mathcal{V}_*[\![T]\!]$.*

Proof. Let Ω be an ambient environment. If $\vdash T$ then T must have the form B (from WF-BASE), $S \to T$ (from WF-FUN), or $(\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi]$. In the first two cases, we see that $\mathcal{V}_\Omega[\![T]\!] = \mathcal{V}[\![T]\!]$ by definition. In the third case, $\mathcal{V}[\![T]\!]$ is defined as the intersection of a set with $\mathcal{V}_*[\![T]\!]$, and so $\mathcal{V}[\![T]\!] \subseteq \mathcal{V}_*[\![T]\!] \subseteq \mathcal{V}_\Omega[\![T]\!]$.

Lemma 9. *If $\vdash T$ then $\mathcal{T}[\![T]\!] \subseteq \mathcal{T}_*[\![T]\!]$.*

Proof. Assume $\vdash T$ and let $t \in \mathcal{T}[\![T]\!]$. Then there is some v_t such that $t \Downarrow v_t$ and $v_t \in \mathcal{V}[\![T]\!]$. But $\mathcal{V}[\![T]\!] \subseteq \mathcal{V}_*[\![T]\!]$ and so $v_t \in \mathcal{V}_*[\![T]\!]$. The lifted reduction rules are a superset of the terminal reduction rules, hence $\Omega \vdash t \Downarrow v_t$, hence $t \in \mathcal{T}_*[\![T]\!]$.

Appendix C Fundamental subtyping theorems

Theorem 7. *If $\vdash S$, $\vdash U$, and $S <: U$ then $\mathcal{V}[\![S]\!] \subseteq \mathcal{V}[\![U]\!]$*

Proof. By induction on the derivation of $S <: U$.

Case: SUB-BASE

$S = B \qquad U = B$

By the reflexivity of set inclusion, we have $\mathcal{V}[\![B]\!] \subseteq \mathcal{V}[\![B]\!]$

Case: SUB-FUN

$S = S_1 \to S_2 \qquad U = U_1 \to U_2 \qquad U_1 <: S_1 \qquad S_2 <: U_2$

Applying the IH gives $\mathcal{V}[\![U_1]\!] \subseteq \mathcal{V}[\![S_1]\!]$ and $\mathcal{V}[\![S_2]\!] \subseteq \mathcal{V}[\![U_2]\!]$
Let $v \in \mathcal{V}[\![S_1 \to S_2]\!]$. Let $v_1 \in \mathcal{V}[\![U_1]\!]$. Since $\mathcal{V}[\![U_1]\!] \subseteq \mathcal{V}[\![S_1]\!]$, we have $v_1 \in \mathcal{V}[\![S_1]\!]$. Unfolding the definition of $\mathcal{V}[\![S_1 \to S_2]\!]$, we see that $v\ v_1 \in \mathcal{T}[\![S_2]\!]$; i.e., $v\ v_1 \Downarrow v_2 \in \mathcal{V}[\![S_2]\!]\ subseteq \mathcal{V}[\![U_2]\!]$. Therefore $v\ v_1 \in \mathcal{T}[\![U_2]\!]$, and so $v \in \mathcal{V}[\![U_1 \to U_2]\!]$. Since $v \in \mathcal{V}[\![S_1 \to S_2]\!]$ implies $v \in \mathcal{V}[\![U_1 \to U_2]\!]$, we have

$$\mathcal{V}[\![S_1 \to S_2]\!] \subseteq \mathcal{V}[\![U_1 \to U_2]\!]$$

Case: SUB-SFUN

$S = (\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi_S] \qquad U = (\overline{x_i : B_i}) \Rightarrow A[\Xi_U] \qquad \overline{x_i : B_i} \vdash \underline{A[\Xi_S]} <: A[\Xi_U]$

$x_i : B_i \vdash A[\Xi_1] <: A[\Xi_2]$ must have been proven by the rule LSUB-BASE-LL, and therefore for $i \in 1..n$ we have $\Xi_1(x_i) \leq \Xi_2(x_i)$. Let Ω be an ambient environment. By monotonicity of qualifier composition and contraction we have

$$\Omega \vdash A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_S(x_i))\ z}^{z \in dom\Omega}] <: A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_U(x_i))\ z}^{z \in dom\Omega}]$$

Hence by *Lemma 6* we have

$$\mathcal{V}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_S(x_i))\ z}^{z \in dom\Omega}]]\!] \subseteq \mathcal{V}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_U(x_i))\ z}^{z \in dom\Omega}]]\!]$$

We then have

$$\mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_S(x_i))\ z}^{z \in dom\Omega}]]\!] \subseteq \mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_U(x_i))\ z}^{z \in dom\Omega}]]\!]$$

Which leads to

$$\mathcal{V}_\Omega[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_S]]\!] \subseteq \mathcal{V}_\Omega[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_U]]\!]$$

Finally, since $\cap$ is monotone separately in each argument, we have:
$\mathcal{V}[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_S]]\!] =$
$\{v \mid \overline{\forall c_i \in \mathcal{V}[\![B_i]\!]}.\ v\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]\} \cap \mathcal{V}_*[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_S]]\!] \subseteq$
$\{v \mid \overline{\forall c_i \in \mathcal{V}[\![B_i]\!]}.\ v\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]\} \cap \mathcal{V}_*[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_U]]\!] =$
$\mathcal{V}[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_U]]\!]$

Theorem 8. *For all ambient environments Ω, if $\Omega \vdash S$, $\Omega \vdash U$, and $\Omega \vdash S <: U$ then $\mathcal{V}_\Omega[\![S]\!] \subseteq \mathcal{V}_\Omega[\![U]\!]$.*

Proof. By induction on the structure of $\Omega \vdash S <: T$.

Case: LSub-Terminal

$S <: U$ could have been proven using one of the rules Sub-Base, Sub-Fun, or Sub-SFun. We consider the three cases separately.

Case: Sub-Base

$S = B \qquad U = B$

Due to the reflexivity of $\subseteq$ we have $\mathcal{V}_\Omega[\![S]\!] = \mathcal{V}_\Omega[\![B]\!] \subseteq \mathcal{V}_\Omega[\![B]\!] = \mathcal{V}_\Omega[\![U]\!]$

Case: Sub-Fun

$S = S_1 \to S_2 \qquad U = U_1 \to U_2$

$\mathcal{V}_\Omega[\![S_1 \to S_2]\!] = \mathcal{V}[\![S_1 \to S_2]\!] \subseteq \mathcal{V}[\![U_1 \to U_2]\!] = \mathcal{V}_\Omega[\![U_1 \to U_2]\!]$

Case: Sub-SFun

$S = (\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi_S] \qquad U = (\overline{x_i : B_i}) \Rightarrow A[\Xi_U] \qquad \overline{x_i : B_i} \vdash A[\Xi_S] <: A[\Xi_U]$

$\overline{x_i : B_i} \vdash A[\Xi_S] <: A[\Xi_U]$ could only have been proven by LSub-Base-LL, and so for each $i \in 1..n$ we must have $\Xi_S(x_i) \leq \Xi_U(x_i)$. By monotonicity of qualifier composition and contraction we have

$$\Omega \vdash A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_S(x_i))\ z}^{z \in dom\Omega}] <: A[\overline{(+_{i=1}^{n}(\Xi_i(z) \circ \Xi_U(x_i))\ z}^{z \in dom\Omega}]$$

Hence by *Lemma 6* we have

$$\mathcal{V}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z)\circ\Xi_S(x_i))\ z}^{z\in dom\Omega}]]\!] \subseteq \mathcal{V}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z)\circ\Xi_U(x_i))\ z}^{z\in dom\Omega}]]\!]$$

We then have

$$\mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z)\circ\Xi_S(x_i))\ z}^{z\in dom\Omega}]]\!] \subseteq \mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n}(\Xi_i(z)\circ\Xi_U(x_i))\ z}^{z\in dom\Omega}]]\!]$$

Which finally leads to

$$\mathcal{V}_\Omega[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_S]]\!] \subseteq \mathcal{V}_\Omega[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi_U]]\!]$$

Case: LSub-Base-TL

$S = B \qquad U = B[\overline{q_z\ z}^{z\in dom(\Omega)}] \qquad \overline{\sim\leq q_z}$

Let $c \in \mathcal{V}_\Omega[\![B]\!] = \mathcal{C}[\![B]\!]$. Since for all $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\|\omega\|c = c$, we see that $c \in \mathcal{B}_\Omega[\![B]\!]$. Letting $z \in dom(\Omega)$ and $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\forall x \in dom(\Omega) - \{z\}.\ \omega(x) = \omega'(x)$, we have that $\|\omega\|c = c = \|\omega'\|c$. Hence, $c \in \mathcal{K}_\Omega[\![\sim\ \ z]\!]$. By *Lemma 5* we then have $c \in \mathcal{K}_\Omega[\![\sim\ \ z]\!] \subseteq \mathcal{K}_\Omega[\![q_z\ z]\!]$. Therefore

$$c \in \mathcal{B}_\Omega[\![B]\!] \cap \bigcap_{z\in dom(\Omega)} \mathcal{K}_\Omega[\![q_z\ z]\!] = \mathcal{V}_\Omega[\![B[\overline{q_z\ z}]]\!]$$

Case: LSub-Base-LL

$S = A[\overline{p_x\ x}^{x\in dom(\Omega)}] \qquad U = A[\overline{q_x\ x}^{x\in dom(\Omega)}] \qquad \overline{p_x \leq q_x}$

This case just amounts to *Lemma 6*.

Case: LSub-Fun-LL

$S = S_1 \xrightarrow{\Omega} S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$

Applying the IH gives us $\mathcal{V}_\Omega[\![U_1]\!] \subseteq \mathcal{V}_\Omega[\![S_1]\!]$ and $\mathcal{V}_\Omega[\![S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_2]\!]$. Let $v \in \mathcal{V}_\Omega[\![S_1 \to S_2]\!]$. Let $v_1 \in \mathcal{V}_\Omega[\![U_1]\!]$. Since $\mathcal{V}_\Omega[\![U_1]\!] \subseteq \mathcal{V}_\Omega[\![S_1]\!]$ we have $v_1 \in \mathcal{V}_\Omega[\![S_1]\!]$. Unfolding the definition of $\mathcal{V}_\Omega[\![S_1 \xrightarrow{\Omega} S_2]\!]$, we see that $v\ v_1 \in \mathcal{T}_\Omega[\![S_2]\!]$. Since $\mathcal{V}_\Omega[\![S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_2]\!]$, we have $\mathcal{T}_\Omega[\![S_2]\!] \subseteq \mathcal{T}_\Omega[\![U_2]\!]$ and therefore $v\ v_1 \in \mathcal{T}_\Omega[\![U_2]\!]$. This proves that $v \in \mathcal{V}_\Omega[\![U_1 \to U_2]\!]$. Since $v \in \mathcal{V}_\Omega[\![S_1 \xrightarrow{\Omega} S_2]\!]$ implies $v \in \mathcal{V}_\Omega[\![U_1 \xrightarrow{\Omega} U_2]\!]$, we have

$$\mathcal{V}_\Omega[\![S_1 \xrightarrow{\Omega} S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_1 \xrightarrow{\Omega} U_2]\!]$$

Case: LSub-FunTL

$S = S_1 \to S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$

Applying the IH gives $\mathcal{V}_\Omega[\![U_1]\!] \subseteq \mathcal{V}_\Omega[\![S_1]\!]$ and $\mathcal{V}_\Omega[\![S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_2]\!]$.
Let $v \in \mathcal{V}_\Omega[\![S_1 \to S_2]\!]$. Let $v_1 \in \mathcal{V}_\Omega[\![U_1]\!]$. Then since $\mathcal{V}_\Omega[\![U_1]\!] \subseteq \mathcal{V}_\Omega[\![S_1]\!]$, we have $v_1 \in \mathcal{V}_\Omega[\![S_1]\!]$. Hence, unfolding the definition of $\mathcal{V}_\Omega[\![S_1 \to S_2]\!]$ we have

$v\ v_1 \in \mathcal{T}[\![S_2]\!]$. By *Lemma 9* we have $v\ v_1 \in \mathcal{T}_\Omega[\![S_2]\!]$. Since $\mathcal{V}_\Omega[\![S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_2]\!]$, we have $v\ v_1 \in \mathcal{T}_\Omega[\![U_2]\!]$. Hence $v \in \mathcal{V}_\Omega[\![U_1 \xrightarrow{\Omega} U_2]\!]$. Since $v \in \mathcal{V}_\Omega[\![S_1 \to S_2]\!]$ implies $v \in \mathcal{V}_\Omega[\![U_1 \xrightarrow{\Omega} U_2]\!]$, we have

$$\mathcal{V}_\Omega[\![S_1 \to S_2]\!] \subseteq \mathcal{V}_\Omega[\![U_1 \xrightarrow{\Omega} U_2]\!]$$

Appendix D SFun Abstraction Typing

T-SFun is the thorniest case involved in our proof of the fundamental theorem for terminal typing. As such, I've split the proof into several lemmas.

Lemma 10. *Let Ω be an ambient environment, $z \in dom(\Omega)$, and $\overline{d_i \in \mathcal{B}_\Omega[\![B_i]\!] \cap \mathcal{K}_\Omega[\![p_i\ z]\!]}^{i \in 1..n}$. Further, let $\Psi = \overline{x_i : B_i}$ and $\phi \in \mathcal{G}_\Psi[\![\overline{x_i : B_i[=\ x_i]}]\!]$. Let t be a term such that $\Psi \vdash \phi t \Downarrow d_t$, where d_t is a base-level value such that $\overline{d_t \in \mathcal{K}_\Psi[\![q_i\ x_i]\!]}$.*

For $k \in 1..n$ and $\pi = \{\overline{\pi_i}^{i \in 1..n - \{k\}}\}$ where $\overline{\pi_i \in \mathcal{O}[\![\Omega]\!]}$, define the term $s_{\pi k}$ as

$$s_{\pi k} \doteq (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)\ [\overline{\|\pi_j\| d_j}^{j \in 1..(k-1)}\ \ d_k\ \ \overline{\|\pi_j\| d_j}^{j \in (k+1)..n}]$$

Then for all $s_{\pi k}$ we have $\Omega \vdash s_{\pi k} \Downarrow d_{\pi k}$ for some value $d_{\pi k} \in \mathcal{K}_\Omega[\![(p_k \circ q_k)\ z]\!]$

Proof.
Let $k \in 1..n$ and $\omega \in \mathcal{O}[\![\Omega]\!]$. By Red-SFun-App we have $\|\omega\| s_{\pi k} \to \|\psi_{\pi\omega}\| \phi t$ where

$$\psi_{\pi\omega} \doteq \overline{[x_j \mapsto \|\pi_j\| d_j]}[x_k \mapsto \|\omega\| d_k]\overline{[x_j \mapsto \|\pi_j\| d_j]}$$

Since $\Psi \vdash \phi t \Downarrow d_t$ and $\psi_{\pi\omega} \in \mathcal{O}[\![\overline{x_i : B_i}]\!]$, we can apply *Corollary 1* to get

$$\|\omega\| s_{\pi k} \to \|\psi_{\pi\omega}\| \phi t \Downarrow \|\psi_{\pi\omega}\| d_t$$

Then by LRed-SFun-App we know that in Ω-lifted reduction, $s_{\pi k}$ normalizes to a lifted constant $d_{\pi k}$ such that $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t$. To show $d_{\pi k} \in \mathcal{K}_\Omega[\![(p_k \circ q_k)\ z]\!]$, we proceed by we case splitting on the qualifier pair (p_k, q_k), letting _ denote a wildcard that matches any qualifier.

Case p_k is $\sim$, q_k is _ :
Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(x) = \omega'(x)$ for all $x \in dom(\Omega)$ with $x \neq z$. Since $d_k \in \mathcal{K}_\Omega[\![\sim\ z]\!]$, we have $\|\omega\| d_k = \|\omega'\| d_k$. We therefore have $\|\psi_{\pi\omega}\| = \|\psi_{\pi\omega'}\|$, and so $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t = \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\sim\ z]\!] = \mathcal{K}_\Omega[\![(p_k \circ q_k)\ z]\!]$.

Case p_k is _, q_k is $\sim$:
Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(x) = \omega'(x)$ for all $x \in dom(\Omega)$ with $x \neq z$. For $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Since $d_t \in \mathcal{K}_\Psi[\![\sim\ x_i]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t = \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\sim\ z]\!] = \mathcal{K}_\Omega[\![(p_k \circ q_k)\ z]\!]$.

Case p_k **is** $?$ **or** q_k **is** $?$ **:**
We've already established that $\Omega \vdash s_{\pi k} \Downarrow d_{\pi k}$ where $d_{\pi k}$ is some lifted constant. Because we're trying to prove that this constant belongs to $\mathcal{K}_\Omega[\![?\ z]\!] = \{c \mid \mathit{true}\}$, that's all we need to establish.

Case p_k **is** $=$, $q_k \in \{\uparrow, \downarrow, =\}$**:**
For $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k = \omega(z)$. Since

$$(p_k \circ q_k) = (= \circ\ q_k) = q_k$$

we just need to prove $d_{\pi k} \in \mathcal{K}_\Omega[\![q_k\ z]\!]$.

Subcase q_k **is** $\uparrow$:
Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and $\omega(x) = \omega'(x)$ for $x \in \mathit{dom}(\Omega) - \{z\}$. Since $d_k \in \mathcal{K}_\Omega[\![=\ z]\!]$ we have $\psi_{\pi\omega}(x_k) = \omega(z)$. Then $\psi_{\pi\omega}(x_k) = \omega(z) \leq \omega'(z) = \psi_{\pi\omega'}(x_k)$ and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![\uparrow x_k]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t \leq \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$.

Subcase q_k **is** $\downarrow$:
This is symmetric to the above subcase. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and $\omega(x) = \omega'(x)$ for $x \in \mathit{dom}(\Omega) - \{z\}$. Then $\psi_{\pi\omega}(x_k) = \omega(z) \leq \omega'(z) = \psi_{\pi\omega'}(x_k)$ and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$.
Because $d_t \in \mathcal{K}_\Omega[\![\downarrow\ x_k]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t \geq \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow z]\!]$.

Subcase q_k **is** $=$:
Let $\omega \in \mathcal{O}[\![\Omega]\!]$. Then $\psi_{\pi\omega}(x_k) = \omega(z)$.
Because $d_t \in \mathcal{K}_\Psi[\![= x_k]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t = \psi_{\pi\omega}(x_k) = \omega(z)$.
Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![= z]\!]$.

Case $p_k \in \{\uparrow, \downarrow\}$, q_k **is** $=$:
Since

$$(p_k \circ q_k) = (p_k \circ =) = p_k$$

we just need to prove $d_{\pi k} \in \mathcal{K}_\Omega[\![p_k\ z]\!]$.

Subcase p_k **is** $\uparrow$:
Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for all $x \in \mathit{dom}(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![p_k\ z]\!]$ we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \leq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![= x_k]\!]$ we have

$$\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t = \psi_{\pi\omega}(x_k) \leq \psi_{\pi\omega'}(x_k) = \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$$

Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow\ z]\!]$.

Subcase p_k **is** $\downarrow$:
This is symmetric to the above subcase. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for all $x \in \mathit{dom}(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![p_k\ z]\!]$ we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \geq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for

$i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![= x_k]\!]$ we have

$$\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t = \psi_{\pi\omega}(x_k) \geq \psi_{\pi\omega'}(x_k) = \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$$

Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow\ z]\!]$.

Case p_k is $\uparrow$, q_k is $\uparrow$:
Since $\uparrow \circ \uparrow = \uparrow$, we must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for $x \in dom(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![\uparrow\ z]\!]$, we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \leq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![\uparrow x_k]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t \leq \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow\ z]\!]$.

Case p_k is $\uparrow$, q_k is $\downarrow$:
Since $\uparrow \circ \downarrow = \downarrow$, we must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow z]\!]$. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for $x \in dom(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![\uparrow\ z]\!]$, we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \leq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![\downarrow x_k]\!]$ we have $\|\omega\| d_{\pi k} = \|\psi_{\pi\omega}\| d_t \geq \|\psi_{\pi\omega'}\| d_t = \|\omega'\| d_{\pi k}$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow\ z]\!]$.

Case p_k is $\downarrow$, q_k is $\uparrow$:
Since $\downarrow \circ \uparrow = \downarrow$, we must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow z]\!]$. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for $x \in dom(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![\downarrow\ z]\!]$, we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \geq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![\downarrow x_k]\!]$ we have $\|\omega'\| d_{\pi k} = \|\psi_{\pi\omega'}\| d_t \leq \|\psi_{\pi\omega}\| d_t = \|\omega\| d_{\pi k}$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\downarrow\ z]\!]$.

Case p_k is $\downarrow$, q_k is $\downarrow$:
Since $\downarrow \circ \downarrow = \uparrow$, we must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$. Let $\omega, \omega' \in \mathcal{O}[\![\Omega]\!]$ such that $\omega(z) \leq \omega'(z)$ and for $x \in dom(\Omega) - \{z\}$, $\omega(x) = \omega'(x)$. Then since $d_k \in \mathcal{K}_\Omega[\![\downarrow\ z]\!]$, we have $\psi_{\pi\omega}(x_k) = \|\omega\| d_k \geq \|\omega'\| d_k = \psi_{\pi\omega'}(x_k)$, and for $i \neq k$ we have $\psi_{\pi\omega}(x_i) = \|\pi_i\| d_i = \psi_{\pi\omega'}(x_i)$. Because $d_t \in \mathcal{K}_\Psi[\![\downarrow x_k]\!]$ we have $\|\omega'\| d_{\pi k} = \|\psi_{\pi\omega'}\| d_t \geq \|\psi_{\pi\omega}\| d_t = \|\omega\| d_{\pi k}$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow\ z]\!]$.

Lemma 11. *Let Ω be an ambient environment, $z \in dom(\Omega)$, $\Psi \doteq \overline{x_i : B_i}^{i \in 1..n}$, and $\overline{d_i \in \mathcal{B}_\Omega[\![B_i]\!] \cap \mathcal{K}_\Omega[\![p_i\ z]\!]}$. Let $\phi \in \mathcal{G}_\Psi[\![\overline{x_i : B_i[=\ x_i]}]\!]$. Let t be a term such that $\Psi \vdash \phi t \Downarrow d_t$ and $d_t \in \mathcal{K}_\Psi[\![q_i\ x_i]\!]$.*

For $k \in 1..n$ and $\pi = \{\overline{\pi_i}^{i \in (k+1)..n}\}$ where $\overline{\pi_i \in \mathcal{O}[\![\Omega]\!]}$, we define the term $s_{\pi k}$ as

$$s_{\pi k} \doteq (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)\ [\overline{d_j}^{j \in 1..k}\ \ \overline{\|\pi_j\| d_j}^{j \in (k+1)..n}]$$

Additionally, letting $a_k = p_k \circ q_k$ and defining r_k with

- $r_1 \doteq a_1$
- $r_k \doteq r_{k-1} + a_k$ *for* $k > 1$.

Then for all $s_{\pi k}$ we have $\Omega \vdash s_{\pi k} \Downarrow d_{\pi k}$ for some value $d_{\pi k}$ in $\mathcal{K}_\Omega[\![r_k\ z]\!]$.

Proof.
We proceed by induction on k.

Case $k = 1$**:**
This is a straightforward corollary of *Lemma 10*.

Case $k > 1$**:**
Let $\pi \in (\mathcal{O}[\![\Omega]\!])^{(k+1)..n}$. By Red-SFun-App we have $\|\omega\|s_{\pi k} \to \|\psi_{\pi\omega}\|\phi t$ where

$$\psi_{\pi\omega} \doteq \overline{[x_j \mapsto \|\omega\|d_j]}^{j\in 1..k} \ \overline{[x_j \mapsto \|\pi_j\|d_j]}^{j\in(k+1)..n}$$

Since $\Psi \vdash \phi t \Downarrow d_t$ and $\psi_{\pi\omega} \in \mathcal{O}[\![\Psi]\!]$, we can apply *Corollary 1* to get

$$\|\omega\|s_{\pi k} \to \|\psi_{\pi\omega}\|\phi t \Downarrow \|\psi_{\pi\omega}\|d_t$$

Then by LRed-SFun-App we know that $s_{\pi k}$ Ω-normalizes to a lifted constant $d_{\pi k}$ such that $\|\omega\|d_{\pi k} = \|\psi_{\pi\omega}\|d_t$. We proceed to prove $d_{\pi k} \in \mathcal{K}_\Omega[\![r_k\ z]\!]$ by case splitting on the qualifier pair (r_{k-1}, a_k), letting $\_$ denote a wildcard that matches any qualifier.

Case r_{k-1} **is** $=$, a_k **is** $\_$ **:**
For all qualifiers q, $(= + q)$ is equal to $=$. We therefore must prove that $d_{\pi k} \in \mathcal{K}_\Omega[\![= z]\!]$. Let $\omega \in \mathcal{O}[\![\Omega]\!]$ and $\pi' = \pi[k \mapsto \omega]$. Since by the IH $d_{\pi'(k-1)} \in \mathcal{K}_\Omega[\![= z]\!]$ we have $\|\omega\|s_{\pi k} = \|\omega\|s_{\pi'(k-1)} \Downarrow \|\omega\|d_{\pi'(k-1)} = \omega(z)$. Since $\|\omega\|s_{\pi k} \Downarrow \omega(z)$, $\|\omega\|s_{\pi k} \Downarrow \|\omega\|d_{\pi k}$, and our reduction relation is deterministic, we have $\|\omega\|d_{\pi k} = \omega(z)$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![= z]\!]$.

Case r_{k-1} **is** $\_$, a_k **is** $=$**:**
For all qualifiers q, $(q + =)$ is equal to $=$. We therefore must prove that $d_{\pi k} \in \mathcal{K}_\Omega[\![= z]\!]$. Let $\omega \in \mathcal{O}[\![\Omega]\!]$. We can apply *Lemma 10* to the term

$$u \doteq (\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ t)\ [\overline{\|\omega\|d_j}^{j\in 1..(k-1)}\ \ d_k\ \ \overline{\|\pi_j\|d_j}^{j\in(k+1)..n}]$$

to get $\Omega \vdash u \Downarrow d_u \in \mathcal{K}_\Omega[\![= z]\!]$. We therefore have

$$\|\omega\|s_{\pi k} \Downarrow \|\omega\|d_{\pi k}$$

and

$$\|\omega\|s_{\pi k} = \|\omega\|u \Downarrow \|\omega\|d_u = \omega(z)$$

Therefore $\|\omega\|d_{\pi k} = \omega(z)$. Hence $d_{\pi k} \in \mathcal{K}_\Omega[\![= z]\!]$.

Case r_{k-1} **is** $?, a_k \in \{\uparrow, \downarrow, \sim\}$ **:**
Here $r_{k-1} + a_k = \ ?$. Since $\mathcal{K}_\Omega[\![?\ z]\!] = \{c \mid \mathit{true}\}$ and we've already established that $d_{\pi k}$ is a lifted constant (at the top of the outer case k > 1), we are done.

Case $r_{k-1} \in \{\uparrow, \downarrow, \sim\}, a_k$ **is** $?$ **:**
Similar to the above case.

Case r_{k-1} **is** $\sim$, a_k **is** $\sim$**:**
We must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\sim\ z]\!]$. Let $\omega_1, \omega_2 \in \mathcal{O}[\![\Omega]\!]$ with $\omega_1(x) = \omega_2(x)$ for all $x \in \mathit{dom}(\Omega) - \{z\}$. Let $\pi'_1 = \pi[k \mapsto \omega_1]$ and $\pi'_2 = \pi[k \mapsto \omega_2]$. Then by the IH we have $\|\omega_1\|d_{\pi k} = \|\omega_1\|d_{\pi'_1(k-1)} = \|\omega_2\|d_{\pi'_1(k-1)}$. By *Lemma 10* we have $\|\omega_2\|d_{\pi'_1(k-1)} = \|\omega_2\|d_{\pi'_2(k-1)} = \|\omega_2\|d_{\pi k}$. Hence $\|\omega_1\|d_{\pi k} = \|\omega_2\|d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\sim\ z]\!]$.

Case r_{k-1} **is** $\sim$, a_k **is** $\uparrow$:

We must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$. Let $\omega_1, \omega_2 \in \mathcal{O}[\![\Omega]\!]$ with $\omega_1(z) \leq \omega_2(z)$ and $\omega_1(x) = \omega_2(x)$ for $x \in \mathit{dom}(\Omega) - \{z\}$. Let $\pi_1 \doteq \pi[k \mapsto \omega_1]$ and $\pi_2 \doteq \pi[k \mapsto \omega_2]$. Then $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)}$. By the IH, $\|\omega_1\| d_{\pi_1(k-1)} = \|\omega_2\| d_{\pi_1(k-1)}$. By *Lemma 10*, $\|\omega_2\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_2(k-1)}$.

Chaining these together, we get $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)} = \|\omega_2\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_2(k-1)} = \|\omega_2\| d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$.

Case r_{k-1} **is** $\sim$, a_k **is** $\downarrow$:

Symmetric to the above case.

Case r_{k-1} **is** $\uparrow$, a_k **is** $\sim$:

We must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$. Let $\omega_1, \omega_2 \in \mathcal{O}[\![\Omega]\!]$ with $\omega_1(z) \leq \omega_2(z)$ and $\omega_1(x) = \omega_2(x)$ for $x \in \mathit{dom}(\Omega) - \{z\}$. Let $\pi_1 \doteq \pi[k \mapsto \omega_1]$ and $\pi_2 \doteq \pi[k \mapsto \omega_2]$. Then $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)}$. By the IH, $\|\omega_1\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_1(k-1)}$. By *Lemma 10*, $\|\omega_2\| d_{\pi_1(k-1)} = \|\omega_2\| d_{\pi_2(k-1)}$.

Chaining these together gives $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_1(k-1)} = \|\omega_2\| d_{\pi_2(k-1)} = \|\omega_2\| d_{\pi k}$, and so $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$.

Case r_{k-1} **is** $\downarrow$, a_k **is** $\sim$:

Symmetric to the above case.

Case r_{k-1} **is** $\uparrow$, a_k **is** $\uparrow$:

Since $\uparrow + \uparrow \;=\; \uparrow$, we must show that $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$. Let $\omega_1, \omega_2 \in \mathcal{O}[\![\Omega]\!]$ such that $\omega_1(z) \leq \omega_2(z)$ and $\omega_1(x) = \omega_2(x)$ for all $x \in \mathit{dom}(\Omega) - \{z\}$. Let $\pi_1 \doteq \pi[k \mapsto \omega_1]$ and $\pi_2 \doteq \pi[k \mapsto \omega_2]$. Then $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)}$ and $\|\omega_2\| d_{\pi k} = \|\omega_2\| d_{\pi_2(k-1)}$. By the IH, we have $\|\omega_1\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_1(k-1)}$. By *Lemma 10* we have $\|\omega_2\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_2(k-1)}$.

Chaining these together gives $\|\omega_1\| d_{\pi k} = \|\omega_1\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_1(k-1)} \leq \|\omega_2\| d_{\pi_2(k-1)} = \|\omega_2\| d_{\pi k}$, and hence $d_{\pi k} \in \mathcal{K}_\Omega[\![\uparrow z]\!]$.

Case r_{k-1} **is** $\downarrow$, a_k **is** $\downarrow$:

Symmetric to the above case.

Case r_{k-1} **is** $\uparrow$, a_k **is** $\downarrow$:

$\uparrow + \downarrow = ?$, so this case is trivial.

Case r_{k-1} **is** $\downarrow$, a_k **is** $\uparrow$:

$\downarrow + \uparrow = ?$, so this case is trivial.

Corollary 2. *Let Ω be an ambient environment and $z \in \mathit{dom}(\Omega)$. Let $\Psi \doteq \overline{x_i : B_i}^{i \in 1..n}$, $\overline{d_i \in \mathcal{B}_\Omega[\![B_i]\!] \cap \mathcal{K}_\Omega[\![p_i\ z]\!]}$, and $\phi \in \mathcal{G}_\Psi[\![\overline{x_i : B_i[=\ x_i]}]\!]$. Let t be a term such that $\Psi \vdash \phi t \Downarrow d_t$ where $\overline{d_t \in \mathcal{K}_\Psi[\![q_i\ x_i]\!]}$.*

Then defining $s \doteq (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)\ [\overline{d_i}]$ we have

$$\Omega \vdash s \Downarrow d_s \in \mathcal{K}_\Omega[\![(+_{i=1}^{n}\ p_i \circ q_i)\ z]\!]$$

.

Proof. Apply *Lemma 11* with $\pi = \emptyset$ and $k = n$.

Corollary 3. *Let Ω be an ambient environment and $z \in dom(\Omega)$. Let $\Psi \doteq \overline{x_i : B_i}^{i \in 1..n}$, $\overline{d_i \in \mathcal{V}_\Omega[\![B_i[\overline{p_{iz}\ z}^{z \in dom(\Omega)}]]\!]}$, and $\phi \in \mathcal{G}_\Psi[\![\overline{x_i : B_i[=\ x_i]}]\!]$. Let t be a term such that $\Psi \vdash \phi t \Downarrow d_t$ where $d_t \in \mathcal{V}_\Psi[\![A[\overline{q_i\ x_i}]]\!]$.*

Then defining $s \doteq (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ t)\ [\overline{d_i}]$ we have

$$\Omega \vdash s \Downarrow d_s \in \mathcal{V}_\Omega[\![A[\overline{(+_{i=1}^{n}\ p_{iz} \circ q_i)\ z}^{z \in dom(\Omega)}]]\!]$$

Proof. Examining the definition of $\mathcal{V}_\Omega[\![A[\Xi]]\!]$, we see that we must prove two things:

1. For each $z \in dom(\Omega)$, $d_s \in \mathcal{K}_\Omega[\![(+_{i=1}^{n} p_{iz} \circ q_i)\ z]\!]$.
2. $d_s \in \mathcal{B}_\Omega[\![A]\!]$

(1) is a simple consequence of *Corollary 2*. Letting $\omega \in \mathcal{O}[\![\Omega]\!]$. (2) can be shown as follows. Define $\psi_\omega \doteq [\overline{x_i \mapsto \|\omega\| d_i}]$. We then have $\psi \in \mathcal{O}[\![\Psi]\!]$.

Since $\|\omega\| s \to \|\psi\| \phi t$ and $\Psi \vdash \phi t \Downarrow d_t \in \mathcal{V}_\Psi[\![A[\overline{q_i\ x_i}]]\!]$, we have $\|\omega\| s \to \|\psi\| \phi t \Downarrow \|\psi\| d_t$. Since $d_t \in \mathcal{V}_\Psi[\![A[\overline{q_i\ x_i}]]\!]$, we know $\|\psi\| d_t \in \mathcal{C}[\![A]\!]$. By LRed-SFun-App we then know $\Omega \vdash s \to a$, where a is an ambient map, mapping each $\omega \in \mathcal{O}[\![\Omega]\!]$ to $\|\psi_\omega\| d_t \in \mathcal{C}[\![A]\!]$. In other words, $d_s = a \in \mathcal{B}_\Omega[\![A]\!]$.

Appendix E Fundamental typing theorems

Theorem 9 (Fundamental theorem for terminal typing). *If $\Gamma \vdash t : T$ then for all $\gamma \in \mathcal{G}[\![\Gamma]\!]$, $\gamma t \in \mathcal{T}[\![T]\!]$.*

Theorem 10 (Fundamental theorem for lifted typing). *If $\Gamma; \Omega; \Phi \vdash t : T$ then for all $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\phi \in \mathcal{G}_\Omega[\![\Phi]\!]$ we have $\gamma \phi t \in \mathcal{T}[\![T]\!]$.*

Proof. The above two theorems are proven by mutual induction on the derivation of $\Gamma \vdash t : T$ and $\Gamma; \Omega; \Phi \vdash t : T$.

Case: T-Constant

$t = c \qquad T = ty(c)$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. By the *Types of Constants* assumption, $\gamma c = c \in \mathcal{V}[\![ty(c)]\!]$.

Case: T-SfConstant

$t = k \qquad T = ty(k)$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. By the *Types of Constants* assumption, $\gamma k = k \in \mathcal{V}[\![ty(k)]\!]$.

Case: T-Var

$t = x \qquad x : T \in \Gamma$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Then by the definition of $\mathcal{G}[\![\Gamma]\!]$, $\gamma x \in \mathcal{V}[\![T]\!]$ and hence $\gamma x \in \mathcal{T}[\![T]\!]$.

Case: T-SFun

$t = (\tilde{\lambda}(\overline{x_i : B_i}^{i\in 1..n}).\ s) \qquad T = (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[=\ x_i]} \vdash s : A[\Xi]$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. For this case, we must prove that $\gamma t = (\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s)$ satisfies two properties:

- $(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s) \in \{v \mid \overline{c_i \in \mathcal{V}[\![B_i]\!]}.v\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]\}$
- $(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s) \in \mathcal{V}_*[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]$

To prove the first point, let $c_i \in \mathcal{V}[\![B_i]\!]$, $\phi \in \mathcal{G}[\![\overline{x_i : B_i[=\ x_i]}]\!]$, and $\Psi = \overline{x_i : B_i}$. Then $(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s)\ [\overline{c_i}] \to \|\psi\|\phi\gamma s$ where $\psi = \overline{[x_i \mapsto c_i]} \in \mathcal{O}[\![\Psi]\!]$. By the IH, we know that $\gamma\phi s \in \mathcal{T}_\Psi[\![A[\Xi]]\!]$ and hence $\phi\gamma s = \gamma\phi s \in \mathcal{T}_\Psi[\![A[\Xi]]\!]$. Unfolding the definition of $\mathcal{T}_\Psi[\![A[\Xi]]\!]$ we get the existence of a value d_s such that $\Psi \vdash \phi\gamma s \Downarrow d_s \in \mathcal{B}_\Psi[\![A]\!]$. Applying *Corollary 1* then gives $\|\psi\|\phi\gamma s \Downarrow \|\psi\|d_s$. Since $d_s \in \mathcal{B}_\Psi[\![A]\!]$, we know that $\|\psi\|d_s \in \mathcal{V}[\![A]\!]$. Chaining these facts together gives

$$(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s)\ [\overline{c_i}] \to \|\psi\|\phi\gamma s \Downarrow \|\psi\|d_s \in \mathcal{V}[\![A]\!]$$

We then have

$$(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s) \in \{v \mid \overline{c_i \in \mathcal{V}[\![B_i]\!]}.v\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]\}$$

To prove the second point, letting Ω be an arbitrary ambient environment, we must prove that

$$(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s) \in \mathcal{V}_\Omega[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]$$

To this end, let $\overline{\Xi_i \in \mathcal{X}[\![\Omega]\!]}$ and $\overline{d_i \in \mathcal{V}_\Omega[\![B_i[\Xi_i]]\!]}$ for $i \in 1..n$. We then have $t\ [\overline{d_i}] \in \mathcal{T}_\Omega[\![A[\overline{(+_{i=1}^{n} \Xi_i(z) \circ \Xi(x_i))\ z}^{z\in dom(\Omega)}]]\!]$ by *Corollary 3*. Therefore $(\tilde{\lambda}(\overline{x_i : B_i}).\ \gamma s) \in \mathcal{V}_\Omega[\![T]\!]$.

Case: T-Fun

$t = (\lambda x : S.u) \qquad T = S \to U \qquad \Gamma, x : S \vdash u : U$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Then $\gamma t = (\lambda x : S.\gamma u)$. Since $(\lambda x : S.\gamma u)$ is a value, we must show that $(\lambda x : S.\gamma u) \in \mathcal{V}[\![S \to U]\!]$. Let $v_s \in \mathcal{V}[\![S]\!]$. Defining $\gamma' \doteq \gamma[x \mapsto v_s]$, we see from the definition of $\mathcal{G}[\![\Gamma, x : T]\!]$ in *Figure 32* that $\gamma' \in \mathcal{G}[\![\Gamma, x : S]\!]$. We then have $(\lambda x : S.\gamma u)\ v_s \to [x \mapsto v_s]\gamma u = \gamma' u$. Applying the IH to the premise $\Gamma, x : S \vdash u : U$ tells us that $\gamma' u \Downarrow v_u \in \mathcal{V}[\![U]\!]$. Hence $(\lambda x : S.\gamma u)\ v_s \to^* \gamma' u \Downarrow v_u \in \mathcal{V}[\![U]\!]$; i.e., $\gamma t \in \mathcal{T}[\![U]\!]$.

Case: T-App

$t = u\ s \qquad T = U \qquad \Gamma \vdash u : S \to U \qquad \Gamma \vdash s : S$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Applying the IH, we get $\gamma u \Downarrow v_u \mathcal{V}[\![S \to U]\!]$ and $\gamma s \Downarrow v_s \mathcal{V}[\![S]\!]$. Hence we have $\gamma t = \gamma u\ \gamma s \to^* v_u\ \gamma s \to^* v_u\ v_s$. By the definition of $\mathcal{V}[\![S \to U]\!]$, we have $v_u\ v_s \Downarrow v_t \in \mathcal{V}[\![U]\!]$, and so $\gamma t \Downarrow v_t \in \mathcal{V}[\![U]\!]$; i.e., $\gamma t \in \mathcal{T}[\![U]\!]$.

Case: T-SfApp

$t = u\ [\overline{s_i}^{i\in 1..n}]$ $\qquad T = A \qquad \Gamma \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma \vdash s_i : B_i}$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Applying the IH to $\Gamma \vdash u : (\overline{x_i : B_i}) \Rightarrow A[\Xi]$ gives $\gamma u \Downarrow v_u \in \mathcal{V}[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]$. Applying the IH to each of the hypotheses $\overline{\Gamma \vdash s_i : B_i}$ gives $\overline{\gamma s_i \Downarrow c_i \in \mathcal{V}[\![B_i]\!]}$. We then have $\gamma t = \gamma u\ [\overline{\gamma s_i}] \rightarrow^* v_u\ [\overline{c_i}]$. Since $v_u \in \mathcal{V}[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]$ and $\overline{c_i \in \mathcal{V}[\![B_i]\!]}$, the definition of $\mathcal{V}[\![(\overline{x_i : B_i}) \Rightarrow A[\Xi]]\!]$ in *Figure 32* tells us that $v_u\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]$. Since $\gamma t \rightarrow^* v_u\ [\overline{c_i}]$ and $v_u\ [\overline{c_i}] \in \mathcal{T}[\![A]\!]$ we have $\gamma t \in \mathcal{T}[\![A]\!]$.

Case: T-IfThenElse

$t = \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 \qquad T = S \qquad \Gamma \vdash s_1 : \mathit{Bool} \qquad \Gamma \vdash s_2 : S$
$\Gamma \vdash s_3 : S$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Applying the IH to $\Gamma \vdash s_1 : \mathit{Bool}$ gives $\gamma s_1 \in \mathcal{T}[\![\mathit{Bool}]\!]$, and so $\gamma s_1 \Downarrow v_1$ where either $v_1 = \mathit{true}$ or $v_1 = \mathit{false}$.

Case: $v_1 = \mathit{true}$

Then $\gamma t = \textbf{if } \gamma s_1 \textbf{ then } \gamma s_2 \textbf{ else } \gamma s_3 \rightarrow^* \textbf{if } \mathit{true} \textbf{ then } \gamma s_2 \textbf{ else } \gamma s_3 \rightarrow \gamma s_2$. Applying the IH to $\Gamma \vdash s_2 : S$ gives $\gamma s_2 \in \mathcal{T}[\![S]\!]$, and so $\gamma t \rightarrow^* \gamma s_2 \in \mathcal{T}[\![S]\!]$.

Case: $v_1 = \mathit{false}$

Then $\gamma t = \textbf{if } \gamma s_1 \textbf{ then } \gamma s_2 \textbf{ else } \gamma s_3 \rightarrow^* \textbf{if } \mathit{false} \textbf{ then } \gamma s_2 \textbf{ else } \gamma s_3 \rightarrow \gamma s_3$. Applying the IH to $\Gamma \vdash s_3 : S$ gives $\gamma s_3 \in \mathcal{T}[\![S]\!]$, and so $\gamma t \rightarrow^* \gamma s_3 \in \mathcal{T}[\![S]\!]$.

Case: T-Sub

$T = U \qquad \Gamma \vdash t : S \qquad \vdash U \qquad S <: U$

By *Lemma 18* we have $\vdash S$. Since $\vdash S$, $\vdash U$, and $S <: U$ we have $\mathcal{V}[\![S]\!] <: \mathcal{V}[\![U]\!]$. Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Applying the IH to $\Gamma \vdash t : S$ gives $\gamma t \Downarrow v_t \in \mathcal{V}[\![S]\!] \subset \mathcal{V}[\![U]\!]$, and therefore $\gamma t \in \mathcal{T}[\![U]\!]$.

Case: LT-Terminal

$\Gamma \vdash t : T$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. By the IH, we have $\gamma t \downarrow v_t \in \mathcal{V}[\![T]\!]$. Hence, letting $\phi \in \mathcal{G}_\Omega[\![\Phi]\!]$ $\gamma\phi t = \gamma t \Downarrow v_t \in \mathcal{V}[\![T]\!] \subseteq \mathcal{V}_\Omega[\![T]\!]$. The set inclusion $\mathcal{V}[\![T]\!] \subseteq \mathcal{V}_\Omega[\![T]\!]$ is a consequence of *Lemma 8*.

Case: LT-LVar

$t = x \qquad x : T \in \Phi$

Let $\phi \in \mathcal{G}_\Omega[\![\Gamma]\!]$. Then by the definition of $\mathcal{G}_\Omega[\![\Phi]\!]$, $\phi x \in \mathcal{V}[\![T]\!]$ and hence $\phi x \in \mathcal{T}[\![T]\!]$.

Case: LT-LFun

$t = (\lambda x : S.u) \qquad T = S \xrightarrow{\Omega} U \qquad \vdash S \qquad \Gamma; \Omega; \Phi, x : S \vdash u : U$

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\phi \in \mathcal{G}_\Omega[\![\Phi]\!]$. Let $v_s \in \mathcal{V}_\Omega[\![S]\!]$ and $\phi' \doteq \phi[x \mapsto v_s]$. Then $\phi' \in \mathcal{G}_\Omega[\![\Phi, x : S]\!]$. Applying the IH to the premise $\Gamma; \Omega; \Phi, x : S \vdash u : U$ then gives $\gamma\phi' u \mathcal{T}_\Omega\llbracket U\rrbracket$.

$\gamma\phi t = (\lambda x : S.\gamma\phi u)$, and so finally we have

$$(\lambda x : S.\gamma\phi u)\ v_s \to [x \mapsto v_s]\gamma\phi u = \gamma\phi' u \in \mathcal{T}_\Omega\llbracket U\rrbracket$$

Proving that

$$\gamma\phi t = (\lambda x : S.\gamma\phi u) \in \mathcal{V}_\Omega\llbracket S \xrightarrow{\Omega} U\rrbracket \subseteq \mathcal{T}_\Omega\llbracket S \xrightarrow{\Omega} U\rrbracket$$

.

Case: LT-LApp
$t = s\ u \qquad T = S \qquad \Gamma; \Omega; \Phi \vdash s : U \xrightarrow{\Omega} S \qquad \Gamma \vdash u : U$

Let $\gamma \in \mathcal{G}\llbracket\Gamma\rrbracket$ and $\phi \in \mathcal{G}_\Omega\llbracket\Phi\rrbracket$. Then $\gamma\phi t = \gamma\phi s\ \gamma\phi u$. By the IH, $\gamma\phi s \Downarrow v_s \in \mathcal{V}_\Omega\llbracket U \xrightarrow{\Omega} S\rrbracket$ and $\gamma\phi u \Downarrow v_u \in \mathcal{V}_\Omega\llbracket U\rrbracket$. Unfolding the definition of $\mathcal{V}_\Omega\llbracket U \xrightarrow{\Omega} S\rrbracket$ we see that $\Omega \vdash \gamma\phi s\ \gamma\phi u \to^* v_s\ \gamma\phi u \to^* v_s\ v_u \in \mathcal{T}_\Omega\llbracket S\rrbracket$. Hence $\gamma\phi t = \gamma\phi s\ \gamma\phi u \in \mathcal{T}_\Omega\llbracket S\rrbracket$.

Case: LT-SfApp
$t = u\ [\overline{s_i}^{i\in 1..n}] \qquad T = A[\overline{+_{i=1}^{n}(\Xi_i(z) \circ \Xi(x_i))}^{z\in dom(\Omega)}] \qquad \Gamma; \Omega; \Phi \vdash u : \overline{(x_i : B_i)} \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B_i[\Xi_i]}$

Let $\gamma \in \mathcal{G}\llbracket\Gamma\rrbracket$ and $\Phi \in \mathcal{G}_\Omega\llbracket\Phi\rrbracket$. Applying the IH, $\Omega \vdash \gamma\phi u \Downarrow v_u \in \mathcal{V}_\Omega\llbracket\overline{(x_i : B_i)} \Rightarrow A[\Xi]\rrbracket$ and $\Omega \vdash \gamma\phi s_i \Downarrow d_i \in \mathcal{V}\llbracket B_i[\Xi_i]\rrbracket$. Unfolding the definition of $\mathcal{V}_\Omega\llbracket\overline{(x_i : B_i)} \Rightarrow A[\Xi]\rrbracket$ gives $v_u\ [\overline{d_i}] \in \mathcal{T}_\Omega\llbracket A[\overline{+_{i=1}^{n}(\Xi_i(z) \circ \Xi(x_i))}^{z\in dom(\Omega)}]\rrbracket$. Therefore

$$\gamma\phi t = \gamma\phi u\ [\overline{\gamma\phi s_i}] \to^* v_u\ [\overline{d_i}] \in \mathcal{T}_\Omega\llbracket A[\overline{+_{i=1}^{n}(\Xi_i(z) \circ \Xi(x_i))}^{z\in dom(\Omega)}]\rrbracket$$

Case: LT-IfThenElse
$t = \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 \qquad T = S \qquad \Gamma; \Omega; \Phi \vdash s_1 : \mathit{Bool} \qquad \Gamma; \Omega; \Phi \vdash s_2 : S \qquad \Gamma; \Omega; \Phi \vdash s_3 : S$

Let $\gamma \in \mathcal{G}\llbracket\Gamma\rrbracket$ and $\phi \in \mathcal{G}_\Omega\llbracket\Phi\rrbracket$. Applying the IH to $\Gamma; \Omega; \Phi \vdash s_1 : \mathit{Bool}$ gives $\gamma\phi s_1 \in \mathcal{T}_\Omega\llbracket\mathit{Bool}\rrbracket$, and so $\Omega \vdash \gamma\phi s_1 \Downarrow v_1$ where either $v_1 = \mathit{true}$ or $v_1 = \mathit{false}$.

Case: $v_1 = \mathit{true}$
Then $\gamma\phi t = \textbf{if}\ \gamma\phi s_1\ \textbf{then}\ \gamma\phi s_2\ \textbf{else}\ \gamma\phi s_3 \to^* \textbf{if}\ \mathit{true}\ \textbf{then}\ \gamma\phi s_2\ \textbf{else}\ \gamma\phi s_3 \to \gamma\phi s_2$. Applying the IH to $\Gamma; \Omega; \Phi \vdash s_2 : S$ gives $\gamma s_2 \in \mathcal{T}\llbracket S\rrbracket$, and so $\gamma\phi t \to^* \gamma\phi s_2 \in \mathcal{T}\llbracket S\rrbracket$.

Case: $v_1 = \mathit{false}$
Then $\gamma\phi t = \textbf{if}\ \gamma\phi s_1\ \textbf{then}\ \gamma\phi s_2\ \textbf{else}\ \gamma\phi s_3 \to^* \textbf{if}\ \mathit{false}\ \textbf{then}\ \gamma\phi s_2\ \textbf{else}\ \gamma\phi s_3 \to \gamma\phi s_3$.
Applying the IH to $\Gamma; \Omega; \Phi \vdash s_3 : S$ gives $\gamma\phi s_3 \in \mathcal{T}\llbracket S\rrbracket$, and so $\gamma\phi t \to^* \gamma\phi s_3 \in \mathcal{T}\llbracket S\rrbracket$.

Case: LT-SUB
$T = U \qquad \Gamma; \Omega; \Phi \vdash t : S \qquad \Omega \vdash U \qquad \Omega \vdash S <: U$

By *Lemma 19* we have $\Omega \vdash S$. Since $\Omega \vdash S$, $\Omega \vdash U$, and $\Omega \vdash S <: U$ we have by the fundamental theorem for lifted subtyping that $\Omega \vdash \mathcal{V}_\Omega[\![S]\!] <: \mathcal{V}_\Omega[\![U]\!]$. Let $\gamma \in \mathcal{G}_\Omega[\![\Gamma]\!]$. Applying the IH to $\Gamma; \Omega; \Phi \vdash t : S$ gives $\gamma\phi t \Downarrow v_t \in \mathcal{V}_\Omega[\![S]\!] \subset \mathcal{V}_\Omega[\![U]\!]$, and therefore, since $\mathcal{V}_\Omega[\![U]\!] \subseteq \mathcal{T}_\Omega[\![U]\!]$, $\gamma\phi t \in \mathcal{T}_\Omega[\![U]\!]$.

Appendix F Projection Theorem and Friends

Lemma 12. *If $\vdash T$ then for all ambient environments Ω we have $\Omega \vdash T$.*

Proof. This is a simple consequence of the ΩWF-TERMINAL rule.

Lemma 13. *If $\vdash T$ then for all ambient environments Ω and $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\|\omega\|T = T$*

Proof. By induction on the structure of $\vdash T$.

Case Wf-Base:
$T = B$
Trivial.
Case Wf-Fun:
$T = S \to T$
Trivial.
Case Wf-SFun:
$T = (\overline{x_i : B_i}) \Rightarrow A[\Xi]$
Trivial.

Lemma 14. *Suppose $\Psi \vdash T$, Ω is an ambient environment such that $dom(\Omega) \neq dom(\Psi)$, and $\omega \in \mathcal{O}[\![\Omega]\!]$. Then $\|\omega\|T = T$.*

Proof. **Case ΩWf-Base:**
$T = B$
Trivial
Case ΩWf-QualBase:
$T = B[\Xi]$
We know that $dom(\omega) = dom(\Omega) \neq dom(\Psi) = dom(\Xi)$. Hence $\|\omega\|B[\Xi] = B[\Xi]$.
Case ΩWf-LFun:
$T = S \xrightarrow{\Psi} U$
Since $\Psi \neq \Omega$, we have $\|\omega\|T = T$.
Case ΩWf-SFun:
$T = (\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]$
Trivial.

Lemma 15 (Well-formed Type Projection). *If $\Omega \vdash T$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\vdash \|\omega\|T$.*

Proof. By induction on the derivation of $\Omega \vdash T$.

Case ΩWf-Terminal

Applying *Lemma 13* to the premise $\vdash T$ gives $\|\omega\|T = T$. Since $\vdash T$ and $\|\omega\|T = T$, we have $\vdash \|\omega\|T$.

Case ΩWf-QualBase

$T = B[\Xi] \qquad dom(\Xi) = dom(\Omega)$

Since $dom(\Omega) = dom(\Xi)$ we have $\|\omega\|B[\Xi] = B$. We then apply Wf-Base to get $\vdash B$. Since $\vdash B$ and $\|\omega\|T = \|\omega\|B[\Xi] = B$, we have $\vdash \|\omega\|T$.

Case ΩWf-LFun

$T = S \xrightarrow{\Omega} U \qquad \Omega \vdash S \qquad \Omega \vdash U$

Applying the IH to both premises gives $\vdash \|\omega\|S$ and $\vdash \|\omega\|U$. An application of the rule Wf-Fun then gives $\vdash \|\omega\|S \to \|\omega\|U$. Since $\|\omega\|T = \|\omega\|(S \xrightarrow{\Omega} U) = \|\omega\|S \to \|\omega\|U$. Since $\vdash \|\omega\|S \to \|\omega\|U$ and $\|\omega\|T = \|\omega\|S \to \|\omega\|U$, we have $\vdash \|\omega\|T$.

Lemma 16. *Suppose $\Gamma; \Psi; \Phi \vdash t : T$. Let Ω be an ambient environment whose variables are distinct from those of $dom(\Psi)$ and those bound by sfun abstractions occuring in t. Let $\omega \in \mathcal{O}[\![\Omega]\!]$. Then we have $\|\omega\|t = t$.*

Lemma 17. *If $\Gamma \vdash t : T$ then for all ambient environments Ω whose variables are distinct from those bound by sfun abstractions occuring in t, and all $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\|\omega\|t = t$.*

Proof. We prove the above two lemmas by simultaneous induction on the dervation of $\Gamma; \Psi; \Phi \vdash t : T$ and $\Gamma \vdash t : T$.

Case LT-Terminal:

Applying the IH gives $\|\omega\|t = t$.

Case LT-LVar:

Trivial.

Case LT-LFun:

$t = (\lambda x : S.u) \qquad \Psi \vdash S \qquad \Gamma; \Psi; \Phi, x : S \vdash u : U$

Applying *Lemma 14* to the left premise gives $\|\omega\|S = S$. Applying the IH to the right premise gives $\|\omega\|u = u$. Hence $\|\omega\|t = \|\omega\|(\lambda x : \|\omega\|S.\|\omega\|u) = (\lambda x : S.u) = t$.

Case LT-LApp:

$t = u\ s \qquad \Gamma; \Psi; \Phi \vdash t : S \xrightarrow{\Omega} U \qquad \Gamma; \Omega; \Phi \vdash s : S$

Applying the IH to both premises gives $\|\omega\|u = u$ and $\|\omega\|s = s$. Hence we have $\|\omega\|t = \|\omega\|(u\ s) = \|\omega\|u\ \|\omega\|s = u\ s = t$

Case LT-SfApp:

$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad \Gamma; \Psi; \Phi \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Psi; \Phi \vdash s_i : B_i[\Xi_i]}$

Applying the IH to the premises gives $\|\omega\|u = u$ and for all $i \in 1..n$, $\|\omega\|s_i = s_i$. Then $\|\omega\|t = \|\omega\|(u\ [\overline{s_i}]) = \|\omega\|u\ [\overline{\|\omega\|s_i}] = u\ [\overline{s_i}] = t$

Case LT-IfThenElse:

$t = \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 \qquad \Gamma; \Psi; \Phi \vdash s_1 : \mathit{Bool} \qquad \Gamma; \Psi; \Phi \vdash s_2 \qquad \Gamma; \Psi; \Phi \vdash s_3$

Applying the IH to the premises gives $\|\omega\|s_1 = s_1$, $\|\omega\|s_2 = s_2$, and $\|\omega\|s_3 = s_3$. Then $\|\omega\|t = \|\omega\|\textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 = \textbf{if } \|\omega\|s_1 \textbf{ then } \|\omega\|s_2 \textbf{ else } \|\omega\|s_3 = \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 = t$.

Case Lt-Sub:

$\Gamma; \Psi; \Phi \vdash t : S \qquad \Psi \vdash U \qquad \Omega \vdash S <: U$

Applying the IH to the left premise gives $\|\omega\|t = t$.

Case T-Constant:

$t = c$

Trivial.

Case T-SfConstant:

$t = k$

Trivial.

Case T-Var:

$t = x$

Trivial.

Case T-Fun:

$t = (\lambda x : S.u) \qquad T = S \to U \qquad \vdash S \qquad \Gamma, x : S \vdash u : U$

Applying *Lemma 13* to $\vdash S$, we see that $\|\omega\|S = S$. Applying the IH to $\Gamma, x : S \vdash u : U$ gives $\|\omega\|u = u$. Therefore $\|\omega\|(\lambda x : S.u) = (\lambda x : \|\omega\|S.\|\omega\|u) = (\lambda x : S.u)$

Case T-SFun:

$t = (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ u) \qquad \Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[= \ x_i]} \vdash t : A[\Xi]$

Since the domain of Ω is distinct from set of variables bound by sfun abstractions in t, we can apply the IH to get $\|\omega\|u = u$. Therefore $\|\omega\|t = \|\omega\|(\tilde{\lambda}(\overline{x_i : B_i}).\ u) = (\tilde{\lambda}(\overline{x_i : B_i}).\ \|\omega\|u) = (\tilde{\lambda}(\overline{x_i : B_i}).\ u) = t$.

Case T-App:

$t = u\ s \qquad \Gamma \vdash u : S \to U \qquad \Gamma \vdash s : S$

Applying the IH to both premises gives $\|\omega\|u = u$ and $\|\omega\|s = s$. Then $\|\omega\|t = \|\omega\|(u\ s) = \|\omega\|u\ \|\omega\|s = u\ s = t$.

Case T-SfApp:

$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad \Gamma u : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma \vdash s_i}$

Case T-IfThenElse

$t = \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 \qquad \Gamma \vdash s_1 : Bool \qquad \Gamma \vdash s_2 : S \qquad \Gamma s_3 \vdash : S$

Applying the IH to the premises gives $\|\omega\|s_1 = s_1$, $\|\omega\|s_2 = s_2$, and $\|\omega\|s_3 = s_3$. We then have $\|\omega\|t = \|\omega\|\textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 = \textbf{if } \|\omega\|s_1 \textbf{ then } \|\omega\|s_2 \textbf{ else } \|\omega\|s_3 = \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 = t$.

Case T-Sub:

$\Gamma \vdash t : S$

Applying the IH to the premise $\Gamma \vdash t : S$ gives $\|\omega\|t = t$.

Lemma 18. *If* $\vdash \Gamma$ *and* $\Gamma \vdash t : T$ *then* $\vdash T$.

Lemma 19. *If* $\Omega \vdash \Phi$ *and* $\Gamma; \Omega; \Phi \vdash t : T$ *then* $\Omega \vdash T$.

Proof. By mutual induction on the structure of $\Gamma \vdash t : T$ and $\Gamma; \Omega; \Phi \vdash t : T$.

Case T-Constant:
$t = c \qquad T = ty(c) = B$
This case is a corollary of the types of constants assumption.
Case T-SfConstant:
This case is a corollary of the types of constants assumption.
Case T-Var:
Since $\vdash \Gamma$ and $x : T \in \Gamma$, we know that $\vdash T$.
Case T-Fun:
$t = (\lambda x : S.u) \qquad \vdash S \qquad \Gamma, x : S \vdash u : U$
Applying the IH to the premise $\Gamma, x : S \vdash u : U$ gives $\vdash U$. Since $\vdash S$ and $\vdash U$ Wf-Fun gives $\vdash S \to U$.
Case T-SFun:
$t = (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ u) \qquad T = (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[=\ x_i]} \vdash A[\Xi]$
Applying the IH to the premise gives $\overline{x_i : B_i} \vdash A[\Xi]$. This can only be derived from ΩWf-QualBase, the premise of which is $dom(\overline{x_i : B_i}) = dom(\Xi)$; i.e., $dom(\Xi) = \{x_i \mid i \in 1..n\}$. Using this fact, we apply Wf-SFun to get $\vdash (\overline{x_i : B_i}) \Rightarrow A[\Xi]$ i.e. $\vdash T$.
Case T-App:
$t = u\ s \qquad T = U \qquad \Gamma \vdash u : S \to U \qquad \Gamma \vdash s : S$
Applying the IH to the left premise gives $\vdash S \to U$, which must have been derived using Wf-Fun. The premises of Wf-Fun give us $\vdash S$ and $\vdash U$. Hence $\vdash U$; i.e., $\vdash T$.
Case T-SfApp:
$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad T = A \qquad \overline{\Gamma \vdash s_i : B_i}$
Applying Wf-Base gives $\vdash A$.
Case T-IfThenElse:
$t = \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 \qquad T = S \qquad \Gamma \vdash s_1 : Bool \qquad \Gamma \vdash s_2 : S \qquad \Gamma \vdash s_3 : S$
Applying the IH to $\Gamma \vdash s_2 : S$ gives $\vdash S$. Since $T = S$, we have $\vdash T$.
Case T-Sub:
$T = U \qquad \vdash U \qquad \ldots$
$\vdash U$, which is what we are trying to prove, is a premise to this rule.
Case LT-Terminal:
Applying the IH gives $\vdash T$. Applying ΩWf-Terminal then gives $\Omega \vdash T$.
Case LT-Var:
By assumption, $\Omega \vdash \Phi$; because $x : T \in \Phi$ we have $\Omega \vdash T$.
Case LT-LFun:
$t = (\tilde{\lambda}(x : S).\ u) \qquad T = S \xrightarrow{\Omega} U \qquad \Omega \vdash S \qquad \Gamma; \Omega; \Phi, x : S \vdash u : U$
Applying the IH to $\Gamma; \Omega; \Phi, x : S \vdash u : U$ gives $\Omega \vdash U$. Because $\Omega \vdash S$ and $\Omega \vdash U$, ΩWf-LFun gives $\Omega \vdash S \xrightarrow{\Omega} U$. Since $T = S \xrightarrow{\Omega} U$ we have $\Omega \vdash T$.
Case LT-LApp:
$t = u\ s \qquad T = U \qquad \Gamma; \Omega; \Phi \vdash t : S \xrightarrow{\Omega} U \qquad \Gamma; \Omega; \Phi \vdash s : S$
Applying the IH to $\Gamma; \Omega; \Phi \vdash t : S \xrightarrow{\Omega} U$ gives $\Omega \vdash S \xrightarrow{\Omega} U$. This judgment must have been derived from an application of ΩWf-LFun, the premises of which are $\Omega \vdash S$ and $\Omega \vdash U$. Since $\Omega \vdash U$ and $T = U$, we have $\Omega \vdash T$.

Case LT-SfApp:
$t = u\ [\overline{s_i}^{i\in 1..n}]$ $\qquad T = A[\overline{(+_{i=1}^{n}(\Xi(x_i)\circ \Xi_i(z)))\ z}^{z\in dom(\Omega)}]$
Since clearly $dom(\overline{(+_{i=1}^{n}(\Xi(x_i)\circ \Xi_i(z)))\ z}^{z\in dom(\Omega)}) = dom(\Omega)$, we can apply ΩWf-QualBase to get $\Omega \vdash A[\overline{(+_{i=1}^{n}(\Xi(x_i)\circ \Xi_i(z)))\ z}^{z\in dom(\Omega)}]$.

Case LT-IfThenElse:
$t = \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 \qquad T = S \qquad \Gamma;\Omega;\Phi \vdash s_1 : \mathit{Bool} \qquad \Gamma;\Omega;\Phi \vdash s_2 : S \qquad \Gamma;\Omega;\Phi \vdash s_3 : S$
Applying the IH to $\Gamma;\Omega;\Phi \vdash s_2 : S$ gives $\Omega \vdash S$. Since $T = S$ we have $\Omega \vdash T$.

Case LT-Sub:
$\Gamma;\Omega;\Phi \vdash t : S \qquad \Omega \vdash U \qquad \Omega \vdash S <: U$
The premise $\Omega \vdash U$ is what we are trying to prove.

Lemma 20. *If $S <: U$ then $\vdash S$ and $\vdash U$.*

Lemma 21. *If $\Omega \vdash S <: U$ then $\Omega \vdash S$ and $\Omega \vdash U$.*

Proof. We prove the above two lemmas by mutual induction on the derivation of $S <: U$ and $\Omega \vdash S <: U$.

Case Sub-Base:
$S = B \qquad T = B$
We have $\vdash B$ by Wf-Base, and so $\vdash S$ and $\vdash T$.

Case Sub-Fun
$S = S_1 \to S_2 \qquad U = U_1 \to U_2 \qquad U_1 <: S_1 \qquad S_2 <: U_2$
Applying the IH to both premises gives $\vdash U_1, \vdash S_1, \vdash S_2$, and $\vdash U_2$. Applying Wf-Fun then gives $\vdash S_1 \to S_2$ and $\vdash U_1 \to U_2$.

Case Sub-SFun:
$S = (\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi_1] \qquad U = (\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi_2]$ Applying the IH to the premise gives $\overline{x_i : B_i} \vdash A[\Xi_1]$ and $\overline{x_i : B_i} \vdash A[\Xi_2]$. These judgments must have been derived with applications of ΩWf-QualBase, the premise of which tells us that $dom(\overline{x_i : B_i}) = dom(\Xi_1)$ and $dom(\overline{x_i : B_i}) = dom(\Xi_2)$. Since $\{x_i \mid i \in 1..n\} = dom(\overline{x_i : B_i}) = dom(\Xi_j)$ for $j = 1, 2$, we can apply Wf-SFun to get

$$\vdash (\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi_1]$$

and

$$\vdash (\overline{x_i : B_i}^{i\in 1..n}) \Rightarrow A[\Xi_2]$$

Case LSub-Terminal:
$S <: U$
Applying the IH to premise yields $\vdash S$ and $\vdash U$. By *Lemma 12* we then have $\Omega \vdash S$ and $\Omega \vdash U$.

Case LSub-Base-TL:
$S = B \qquad U = B[\overline{q_x\ x}^{x\in dom(\Omega)}]$
By Wf-Base we have $\vdash B$, so applying *Lemma 12* gives $\Omega \vdash B$. Since $dom(\overline{q_x\ x}^{x\in dom(\Omega)}) = \Omega$, applying ΩWf-QualBase gives $\Omega \vdash B[\overline{q_x\ x}]$.

Case LSub-Base-LL:

$S = B[\overline{p_x\ x}^{x \in dom(\Omega)}] \qquad U = B[\overline{p_x\ x}^{x \in dom(\Omega)}]$

Since $dom(\overline{p_x\ x}^{x \in dom(\Omega)}) = \Omega$ and $dom(\overline{q_x\ x}^{x \in dom(\Omega)}) = \Omega$, we can apply ΩWF-QUALBASE to get $\Omega \vdash B[\overline{p_x\ x}]$ and $\Omega \vdash B[\overline{q_x\ x}]$.

Case LSub-FunLL:

$S = S_1 \xrightarrow{\Omega} S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$

Applying the IH to the left premise gives $\Omega \vdash U_1$ and $\Omega \vdash S_1$. Applying the IH to the right premise gives $\Omega \vdash S_2$ and $\Omega \vdash U_2$. With this, we can apply ΩWF-LFUN to get $\Omega \vdash S_1 \xrightarrow{\Omega} S_2$ and $\Omega \vdash U_1 \xrightarrow{\Omega} U_2$.

Case LSub-FunTL:

$S = S_1 \to S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$
$\vdash S_1 \to S_2$

Applying the IH to our premises gives $\Omega \vdash U_1$, $\Omega \vdash S_1$, $\Omega \vdash S_2$, and $\Omega \vdash U_2$. Applying ΩWF-LFUN gives $\Omega \vdash U_1 \to U_2$. Applying *Lemma 12* to $\vdash S_1 \to S_2$ gives $\Omega \vdash S_1 \to S_2$.

Lemma 22 (Subtyping Projection). *If $\Omega \vdash S$, $\Omega \vdash U$, and $\Omega \vdash S <: U$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$, we have $\|\omega\|S <: \|\omega\|U$.*

Proof. Let $\omega \in \mathcal{O}[\![\Omega]\!]$. We proceed by induction on the derivation of $\Omega \vdash S <: U$.

Case LSub-Terminal:

$S <: U$

Applying *Lemma 20* to $S <: U$ gives $\vdash S$ and $\vdash U$. By idempotency of ambient substitution on terminally well-formed types, we have $\|\omega\|S = S$ and $\|\omega\|U = U$, and hence since $S <: T$ we have $\|\omega\|S <: \|\omega\|U$.

Case LSub-Base-TL:

$S = B \qquad U = B[\overline{q_x\ x}^{x \in dom(\Omega)}] \qquad \overline{\leq q_x}$

$\|\omega\|B = B$ and $\|\omega\|B[\overline{q_x\ x}] = B$. Applying SUB-BASE gives $\vdash B <: B$, and so by substitution we have $\vdash \|\omega\|B <: \|\omega\|B[\overline{q_x\ x}]$.

Case LSub-Base-LL:

$S = B[\overline{p_x\ x}^{x \in dom(\Omega)}] \qquad U = B[\overline{q_x\ x}^{x \in dom(\Omega)}] \qquad \overline{p_x \leq q_x}$

$\|\omega\|B[\overline{p_x\ x}^{x \in dom(\Omega)}] = B$ and $\|\omega\|B[\overline{q_x\ x}^{x \in dom(\Omega)}] = B$. Applying WF-BASE gives $\vdash B <: B$. By substitution, $\vdash \|\omega\|B[\overline{p_x\ x}^{x \in dom(\Omega)}] <: \|\omega\|B[\overline{q_x\ x}^{x \in dom(\Omega)}]$.

Case LSub-Fun-LL:

$S = S_1 \xrightarrow{\Omega} S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$

Applying the IH to the premises gives $\|\omega\|U_1 <: \|\omega\|S_1$ and $\|\omega\|S_2 <: \|\omega\|U_2$. Applying SUB-FUN then gives $\|\omega\|S_1 \to \|\omega\|S_2 <: \|\omega\|U_1 \to \|\omega\|U_2$. Since $\|\omega\|(S_1 \xrightarrow{\Omega} S_2) = \|\omega\|S_1 \to \|\omega\|S_2$ and $\|\omega\|(U_1 \xrightarrow{\Omega} U_2) = \|\omega\|U_1 \to \|\omega\|U_2$, we have by substitution $\|\omega\|(S_1 \xrightarrow{\Omega} S_2) <: \|\omega\|(U_1 \xrightarrow{\Omega} U_2)$.

Case LSub-Fun-TL:

$S = S_1 \to S_2 \qquad U = U_1 \xrightarrow{\Omega} U_2 \qquad \Omega \vdash U_1 <: S_1 \qquad \Omega \vdash S_2 <: U_2$
$\vdash S_1 \to S_2$

Applying the IH to the premises gives $\|\omega\|U_1 <: \|\omega\|S_1$ and $\|\omega\|S_2 <: \|\omega\|U_2$. Hence by SUB-FUN we have $\|\omega\|S_1 \to \|\omega\|S_2 <: \|\omega\|U_1 \to \|\omega\|U_2$. Inverting

$\vdash S_1 \to S_2$ gives $\vdash S_1$ and $\vdash S_2$. By *Lemma 13* we have $\|\omega\|S_1 = S_1$ and $\|\omega\|S_2 = S_2$. Then $\|\omega\|S_1 \to S_2 = S_1 \to S_1 = \|\omega\|S_1 \to \|\omega\|S_2$. Furthermore, $\|\omega\|(U_1 \overset{\Omega}{\to} U_2) = \|\omega\|U_1 \to \|\omega\|U_2$. By substitution we then have $\|\omega\|(S_1 \to S_2) <: \|\omega\|(U_1 \overset{\Omega}{\to} U_2)$.

Lemma 23 (Terminal Permutation). *If $\Gamma \vdash t : T$ has a height-n derivation and Δ is a permutation of Γ then then $\Delta \vdash t : T$ has a height-n derivation.*

Lemma 24 (Lifted Permutation of Terminal Environment). *If $\Gamma; \Omega; \Phi \vdash t : T$ has a height-n derivation and Δ is a permutation of Γ then then $\Delta; \Omega; \Phi \vdash t : T$ has a height-n derivation.*

Proof. By simultaneous induction on the structure of $\Gamma \vdash t : T$ and $\Gamma; \Omega; \Phi \vdash t : T$.

Case T-Constant:

$t = c \qquad T = B \qquad ty(c) = B$

The premise gives $ty(c) = B$, so we can apply T-Constant to conclude $\Delta \vdash c : B$, which has the same height (1).

Case T-SfConstant:

Similar to the above case.

Case T-Var

$t = z \qquad z : T \in \Gamma$

If $z : T \in \Gamma$ then $z : T \in \Delta$, because Δ is a permutation of Γ. Applying T-Var gives $\Delta, x : S \vdash z : T$.

Case T-Fun:

$t = (\lambda y : T_1.u) \qquad \vdash T_1 \qquad T = T_1 \to T_2 \qquad \Gamma, y : T_1 \vdash u : T_2$

Since Δ is a permutation of Γ, $\Delta, y : T_1$ is a permutation of $\Gamma, y : T_1$. Applying the IH to the right premise gives gives a same-height derivation of $\Delta, y : T_1 \vdash u : T_2$. Applying T-Fun gives a same-height derivation $\Delta \vdash (\lambda y : T_1.u) : T_1 \to T_2$.

Case T-SFun:

$t = (\tilde{\lambda}(\overline{x_i : B_i}).\ u) \qquad T = (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[= \ x_i]}$

Applying the IH gives a same-height derivation of $\Delta; \overline{x_i : B_i}; \overline{x_i : B[= \ x_i]} \vdash t : A[\Xi]$. Applying T-SFun gives a same-height derivation of $\Delta \vdash (\tilde{\lambda}(\overline{x_i : B_i}).\ t) : T = (\overline{x_i : B_i}) \Rightarrow A[\Xi]$.

Case T-App:

$t = u\ s \qquad T = U \qquad \Gamma \vdash u : S \to U \qquad \Gamma \vdash s : S$

Applying the IH to the premises gives $\Delta \vdash u : S \to U$ of height n_1 and $\Delta \vdash s : S$ of height n_2. Applying T-App gives a same-height derivation of $\Delta \vdash u\ s : U$.

Case T-SfApp:

Apply the IH to the premises and then apply T-SfApp.

Case T-IfThenElse:

Apply the IH to the premises and then apply T-IfThenElse.

Case T-Sub:
Apply the IH to the left premise and the apply T-Sub.

Case LT-Terminal:
Applying the IH gives a same-height derivation of $\Delta \vdash t : T$. Applying LT-Terminal gives a same-height-derivation of $\Gamma, x : S; \Omega; \Phi \vdash t : T$.

Case LT-LVar:
$t = y \qquad y : T \in \phi$
Since $y : T \in \Phi$, we can apply LT-LVar to conclude $\Delta; \Omega; \Phi \vdash y : T$.

Case LT-LFun:
$t = (\lambda y : T_1.u) \qquad T = T_1 \xrightarrow{\Omega} T_2 \qquad \Omega \vdash T_1 \qquad \Gamma; \Omega; \Phi, y : T_1 \vdash u : T_2$
Applying the IH to the right premise gives a same-height derivation of $\Delta; \Omega; \Phi, y : T_1 \vdash u : T_2$. Applying LT-LFun gives a same-height derivation of $\Delta; \Omega; \Phi \vdash (\tilde{\lambda}(y : T_1).\ u) : T_1 \xrightarrow{\Omega} T_2$.

Case LT-LApp:
$t = u\ s \qquad T = T_2 \qquad \Gamma; \Omega; \Phi \vdash u : T_1 \xrightarrow{\Omega} T_2 \qquad \Gamma; \Omega; \Phi \vdash s : T_1$

Apply the IH to the premises and apply LT-LApp to the results.

Case LT-SfApp:
$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad T = A[\overline{(+_{i=1}^{n}(\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}] \qquad \Gamma; \Omega; \Phi \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B[\Xi]}$

Apply the IH to the premises and apply LT-SfApp to the results.

Case LT-IfThenElse:
$t = \textbf{if}\ u_1\ \textbf{then}\ u_2\ \textbf{else}\ u_3 \qquad T = U \qquad \Gamma; \Omega; \Phi \vdash u_1 : Bool \qquad \Gamma; \Omega; \Phi \vdash u_2 : U \qquad \Gamma; \Omega; \Phi \vdash u_3 : U$

Apply the IH to the premises and apply LT-IfThenElse to the results.

Case LT-Sub:
$T = U_2 \qquad \Gamma; \Omega; \Phi \vdash t : U_1 \qquad \Omega \vdash U_2 \qquad Omega \vdash U_1 <: U_2$
Applying the IH to the left premise yields a same-height derivation of $\Delta; \Omega; \Phi \vdash t : U_1$. Applying LT-Sub yields $\Delta; \Omega; \Phi \vdash t : U_2$.

Lemma 25 (Weakening for Terminal Typing). *If $\Gamma \vdash t : T$ then for all variables $x \notin dom(\Gamma)$ which do not occur as bindings in t, and terminally well-formed types S we have $\Gamma, x : S \vdash t : T$.*

Lemma 26 (Weakening of Terminal Type Environment in Lifted Typing). *If $\Gamma; \Omega; \Phi \vdash t : T$ then for all variables $x \notin dom(\Gamma)$ which do not occur as bindings in t, and all terminally well-formed types S we have $\Gamma, x : S; \Omega; \Phi \vdash t : T$.*

Proof. We prove the above two lemmas by simultaneous induction on the structure of. $\Gamma \vdash x : T$ and $\Gamma; \Omega; \Phi \vdash t : T$.

Case T-Constant:
$t = c \qquad T = B \qquad ty(c) = B$
The premise gives $ty(c) = B$, so we can apply T-Constant to conclude $\Gamma, x : S \vdash c : B$.

Case T-SfConstant:
Similar to the above case.
Case T-Var
$t = z \qquad z : T \in \Gamma$
Since $z : T \in \Gamma$ we have $z : T \in \Gamma, x : S$. Applying T-Var gives $\Gamma, x : S \vdash z : T$.
Case T-Fun:
$t = (\lambda y : T_1.u) \qquad \vdash T_1 \qquad T = T_1 \to T_2 \qquad \Gamma, y : T_1 \vdash u : T_2$
Applying the IH to the right premise gives gives $\Gamma, y : T_1, x : S \vdash u : T_2$. Applying the permutation lemma gives $\Gamma, x : S, y : T_1 \vdash u : T_2$. Applying T-Fun gives $\Gamma, x : S \vdash (\lambda y : T_1.u)$.
Case T-SFun:
$t = (\tilde{\lambda}(\overline{x_i : B_i}).\ u) \qquad T = (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[=\ x_i]}$
Applying the IH gives $\Gamma, x : S; \overline{x_i : B_i}; \overline{x_i : B[=\ x_i]} \vdash t : A[\Xi]$. Applying T-SFun gives $\Gamma, x : S \vdash (\tilde{\lambda}(\overline{x_i : B_i}).\ t) : T = (\overline{x_i : B_i}) \Rightarrow A[\Xi]$.
Case T-App:
$t = u\ s \qquad T = U \qquad \Gamma \vdash u : S \to U \qquad \Gamma \vdash s : S$
Applying the IH to the premises gives $\Gamma, x : S \vdash u : S \to U$ and $\Gamma, x : S \vdash s : S$. Applying T-App gives $\Gamma, x : S \vdash u\ s : U$.
Case T-SfApp:
Apply the IH to the premises and then apply T-SfApp.
Case T-IfThenElse:
Apply the IH to the premises and then apply T-IfThenElse.
Case T-Sub:
Apply the IH to the left premise and the apply T-Sub.
Case LT-Terminal:
Applying the IH gives $\Gamma, x : S \vdash t : T$. Applying LT-Terminal gives $\Gamma, x : S; \Omega; \Phi \vdash t : T$.
Case LT-LVar:
$t = y \qquad y : T \in \phi$
Since $y : T \in \Phi$, we can apply LT-LVar to conclude $\Gamma, x : S; \Omega; \Phi \vdash y : T$.
Case LT-LFun:
$t = (\lambda y : T_1.u) \qquad T = T_1 \xrightarrow{\Omega} T_2 \qquad \Omega \vdash T_1 \qquad \Gamma; \Omega; \Phi, y : T_1 \vdash u : T_2$
Applying the IH to the right premise gives $\Gamma, x : S; \Omega; \Phi, y : T_1 \vdash u : T_2$. Applying LT-LFun gives $\Gamma, x : S; \Omega; \Phi \vdash (\tilde{\lambda}(y : T_1).\ u) : T_1 \xrightarrow{\Omega} T_2$
Case LT-LApp:
$t = u\ s \qquad T = T_2 \qquad \Gamma; \Omega; \Phi \vdash u : T_1 \xrightarrow{\Omega} T_2 \qquad \Gamma; \Omega; \Phi \vdash s : T_1$
Applying the IH to the premises gives $\Gamma, x : S; \Omega; \Phi \vdash u : T_1 \xrightarrow{\Omega} T_2$. and $\Gamma, x : S; \Omega; \Phi \vdash s : T_2$. Applying LT-LApp gives $\Gamma, x : S; \Omega; \Phi \vdash u\ s : T_2$.
Case LT-SfApp:
$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad T = A[\overline{(+_{i=1}^{n}(\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}] \qquad \Gamma; \Omega; \Phi \vdash t : (\overline{x_i : B_i}) \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B[\Xi]}$

Apply the IH to all premises and then apply LT-SfApp to the results.
Case LT-IfThenElse:
$t = \textbf{if}\ u_1\ \textbf{then}\ u_2\ \textbf{else}\ u_3 \qquad T = U \qquad \Gamma; \Omega; \Phi \vdash u_1 : \mathit{Bool} \qquad \Gamma; \Omega; \Phi \vdash$

$u_2 : U \qquad \Gamma; \Omega; \Phi \vdash u_3 : U$

Applying the IH to the premises yields $\Gamma, x : S; \Omega; \Phi \vdash u_1 : \mathit{Bool}$, $\Gamma, x : S; \Omega; \Phi \vdash u_2 : U$, and $\Gamma, x : S; \Omega; \Phi \vdash u_3 : U$. Applying LT-SfApp yeilds $\Gamma, x : S; \Omega; \Phi \vdash \textbf{if } u_1 \textbf{ then } u_2 \textbf{ else } u_3 : U$

Case LT-Sub:

$T = U_2 \qquad \Gamma; \Omega; \Phi \vdash t : U_1 \qquad \Omega \vdash U_2 \qquad \mathit{Omega} \vdash U_1 <: U_2$

Applying the IH to the left premise yields $\Gamma, x : S; \Omega; \Phi \vdash t : U_1$. Applying LT-Sub yields $\Gamma, x : S; \Omega; \Phi \vdash t : U_2$.

Theorem 11 (Well-typed Term Projection). *Let Γ_1, Γ_2 denote the concatenation of two terminal type environments Γ_1 and Γ_2. If $\vdash \Gamma$, $\Omega \vdash \Phi$, and $\Gamma; \Omega; \Phi \vdash t : T$ then for all $\omega \in \mathcal{O}[\![\Omega]\!]$ we have $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|T$.*

Proof. By induction on the derivation of $\Gamma; \Omega; \Phi \vdash t : T$.

Case LT-Terminal:

$\Gamma \vdash t : T$

Since $\Gamma \vdash t : T$, *Lemma 17* gives $\|\omega\|t = t$ and *Lemma 18* gives $\vdash T$. Since $\vdash T$, *Lemma 13* gives $\|\omega\|T = T$. Weakening the terminal typing judgment $\Gamma \vdash t : T$, we get $\Gamma, \|\omega\|\Phi \vdash t : T$. Hence $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|T$.

Case LT-Var:

Since $x : T \in \Phi$ we have $x : \|\omega\|T \in \|\omega\|\Phi$, and so we can apply T-Var to get $\Gamma, \|\omega\|\Phi \vdash x : \|\omega\|T$. Since $\|\omega\|x = x$, we have $\Gamma, \|\omega\|\Phi \vdash \|\omega\|x : \|\omega\|T$.

Case LT-LFun:

$t = (\lambda x : S.u) \qquad T = S \xrightarrow{\Omega} U \qquad \Omega \vdash S \qquad \Gamma; \Omega; \Phi, x : S \vdash u : U$

Applying the IH to $\Gamma; \Omega; \Phi, x : S \vdash u : U$ gives $\Gamma, \|\omega\|\Phi, x : \|\omega\|S \vdash \|\omega\|u : \|\omega\|U$. Applying well-formed type projection to $\Omega \vdash S$ gives $\vdash \|\omega\|S$.

We can the apply T-Fun to get $\Gamma, \|\omega\|\Phi \vdash (\lambda x : \|\omega\|S.\|\omega\|u) : \|\omega\|S \to \|\omega\|U$. Since $\|\omega\|t = \|\omega\|(\lambda x : S.u) = (\lambda x : \|\omega\|S.\|\omega\|u)$ and $\|\omega\|T = \|\omega\|S \xrightarrow{\Omega} U = \|\omega\|S \to \|\omega\|U$, we have $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|T$.

Case LT-LApp:

$t = u\ s \qquad T = U \qquad \Gamma; \Omega; \Phi \vdash u : S \xrightarrow{\Omega} U \qquad \Gamma; \Omega; \Phi \vdash s : S$

Applying the IH to both premises gives $\Gamma, \|\omega\|\Phi \vdash \|\omega\|u : \|\omega\|S \to \|\omega\|U$ and $\Gamma, \|\omega\|\Phi \vdash \|\omega\|s : \|\omega\|S$. Applying T-App then gives $\Gamma, \|\omega\|\Phi \vdash \|\omega\|u\ \|\omega\|s : \|\omega\|U$. Since $T = U$ and $\|\omega\|t = \|\omega\|(u\ s) = \|\omega\|u\ \|\omega\|s$, we have $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|T$.

Case LT-SfApp

$t = u\ [\overline{s_i}^{i \in 1..n}] \qquad T = A[\overline{(+_{i=1}^{n} (\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}] \qquad \Gamma; \Omega; \Phi \vdash t : \overline{(x_i : B_i)} \Rightarrow A[\Xi] \qquad \overline{\Gamma; \Omega; \Phi \vdash s_i : B[\Xi]}$

For $i \in 1..n$, since $\Gamma; \Omega; \Phi \vdash s_i : B_i[\Xi_i]$, we know that $\Omega \vdash B[\Xi_i]$ by *Lemma 19*. This must have been proven from ΩWf-Base, the premise of which is $\mathit{dom}(\Omega) = \mathit{dom}\Xi$. Hence $\|\omega\|B_i[\Xi_i] = B_i$. Obviously, $\mathit{dom}(\Omega) = \mathit{dom}(\overline{(+_{i=1}^{n} (\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)})$, and so $\|\omega\|T = \|\omega\|A[\overline{(+_{i=1}^{n} (\Xi(x_i) \circ \Xi_i(z)))\ z}^{z \in dom(\Omega)}] = A$.

Applying the IH to the left premises gives $\Gamma, \|\omega\|\Phi \vdash \|\omega\|u : \|\omega\|(\overline{x_i : B_i}) \Rightarrow A[\Xi]$. and hence $\Gamma, \|\omega\|\Phi \vdash \|\omega\|u : (\overline{x_i : B_i}) \Rightarrow A[\Xi]$. Applying the IH to the premises on the right gives, for $i \in 1..n$, $\Gamma, \|\omega\|\Phi \vdash \|\omega\|s_i : \|\omega\|B_i[\Xi_i]$ and hence $\Gamma, \|\omega\|\Phi \vdash \|\omega\|s_i : B_i$. We then apply T-SfApp to get $\Gamma, \|\omega\|\Psi \vdash \|\omega\|u\ [\overline{\|\omega\|s_i}] : A$. Since $\|\omega\|t = \|\omega\|(u\ [\overline{s_i}]) = \|\omega\|u\ [\overline{\|\omega\|s_i}]$ and $\|\omega\|T = A$, we have $\Gamma, \|\omega\|\Psi \vdash \|\omega\|t : \|\omega\|T$.

Case LT-IfThenElse:

$t = \textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 \qquad T = S \qquad \Gamma; \Omega; \Phi \vdash s_1 : \mathit{Bool} \qquad \Gamma; \Omega; \Phi \vdash s_2 : S \qquad \Gamma; \Omega; \Phi \vdash s_3 : S$

Applying the IH to the first premise gives $\Gamma, \|\omega\|\Psi \vdash \|\omega\|s_1 : \|\omega\|\mathit{Bool}$; i.e., $\Gamma, \|\omega\|\Psi \vdash \|\omega\|s_1 : \mathit{Bool}$. Applying the IH to the second and third premises gives $\Gamma, \|\omega\|\Psi \vdash \|\omega\|s_2 : \|\omega\|S$ and $\Gamma, \|\omega\|\Psi \vdash \|\omega\|s_2 : \|\omega\|S$.
Applying T-IfThenElse then gives $\Gamma, \|\omega\|\Psi \vdash \textbf{if}\ \|\omega\|s_1\ \textbf{then}\ \|\omega\|s_2\ \textbf{else}\ \|\omega\|s_3 : \|\omega\|S$. Since $\|\omega\|t = \|\omega\|\textbf{if}\ s_1\ \textbf{then}\ s_2\ \textbf{else}\ s_3 = \textbf{if}\ \|\omega\|s_1\ \textbf{then}\ \|\omega\|s_2\ \textbf{else}\ \|\omega\|s_3$ and $T = S$, we have $\Gamma, \|\omega\|\Psi \vdash \|\omega\|t : \|\omega\|T$.

Case LT-Sub:

$T = U \qquad \Gamma; \Omega; \Phi \vdash t : S \qquad \Omega \vdash U \qquad \mathit{Omega} \vdash S <: U$

Applying the IH to $\Gamma; \Omega; \Phi \vdash t : S$ gives $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|S$. Applying well-formed type projection to $\Omega \vdash U$ gives $\vdash \|\omega\|U$. Applying *Lemma 22* $\mathit{Omega} \vdash S <: U$ gives $\|\omega\|S <: \|\omega\|U$. With this, we apply T-Sub to get $\Gamma, \|\omega\|\Phi \vdash \|\omega\|t : \|\omega\|U$.